THESE

présentée par

Pascal RAYMOND

pour obtenir le grade de DOCTEUR
de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE
(arrêté ministériel du 23 novembre 1988)

(Spécialité : Informatique)

Compilation efficace d'un langage déclaratif synchrone : le générateur de code LUSTRE-V3

Date de soutenance : 20 novembre 1991

Composition du jury :	Président	J. Mossière
	Rapporteurs	A. Benveniste
		G. Berry
	Examinateurs	N. Halbwachs
		P. Le Guernic

Thèse préparée au sein du Laboratoire de Génie Informatique

Table des matières

Introduction

Le langage Lustre

Le domaine d'application du langage Lustre [CPHP87, HCRP91b, HCRP91a] est celui des systèmes réactifs. Contrairement aux systèmes classiques (ou transformationnels), le but de ceux-ci n'est pas de délivrer un résultat, mais de réagir continûment à leur environnement [HP85, Ber89].

Pour programmer de tels systèmes, le langage Lustre adopte un point de vue *flot de données* [Kah74]. Dans ce modèle, un système est constitué d'un réseau d'opérateurs agissant en parallèle au rythme de leurs entrées. Un programme Lustre peut être vu comme la transcription textuelle d'un tel réseau : des identificateurs sont associés à certains "fils" du réseau et les relations entre ces identificateurs sont représentées par des équations (figure 0.1).

```
X = 2*Y + Z

W = X + 1
```

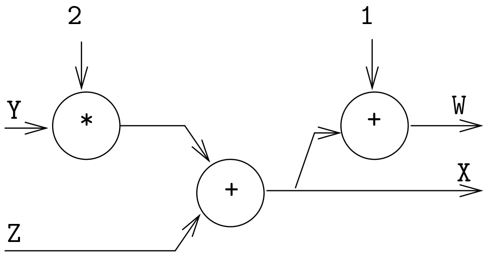


Fig. 0.1 – *Flot de données : forme équationnelle et réseau*

Pour décrire des systèmes réactifs à partir du modèle flot de données, Lustre adopte une approche synchrone : les opérateurs du réseau sont des primitives idéales dont la "traversée" par le flot des données prend un temps nul. Cette approche permet d'introduire de façon très simple une dimension temporelle. Les entrées du réseau sont interprétées comme des fonctions d'un même "temps" de référence ; puisque la traversée des opérateurs prend par hypothèse un temps nul, la valeur des sorties est instantanément disponible. Les sorties sont donc des fonctions du même temps de référence que les entrées. Dans notre exemple (figure 0.1), cela revient à interpréter chaque variable comme une fonction du temps :

$$\begin{array}{rrcl} \forall t & \mathtt{X}(t) & = & 2\mathtt{Y}(t) + \mathtt{Z}(t) \\ \textit{et} & \mathtt{W}(t) & = & \mathtt{X}(t) + 1 \end{array}$$

L'approche flot de données n'est intéressante que si on peut faire référence au passé, c'est-à-dire introduire des "retards" dans le réseau (l'équivalent des bascules dans un circuit logique). Ceci est possible en Lustre grâce à l'opérateur de mémorisation `pre`. En interprétant son

temps de base	0	1	2	3	4	5	6	7
C	*vrai*	*faux*	*vrai*	*vrai*	*faux*	*vrai*	*faux*	*vrai*
temps sur C	0		1	2		3		4
C′	*faux*		*vrai*	*faux*		*vrai*		*vrai*
temps sur C′			0			1		2

FIG. 0.2 – *Flots booléens et horloges*

argument comme une fonction du temps, “`Y = pre X`” peut être vu comme la fonction :

$$\forall t \neq 0 \quad \mathtt{Y}(t) \quad = \quad \mathtt{X}(t-1)$$

L'introduction de retards pose le problème de la valeur d'un flot à l'instant initial. Pour initialiser correctement les flots, le programmeur dispose de l'opérateur binaire `->` ; par exemple, pour “`Z = X -> Y`”, on a :

$$\begin{array}{rrcl} & \mathtt{Z}(0) & = & \mathtt{X}(0) \\ \forall t \neq 0 & \mathtt{Z}(t) & = & \mathtt{Y}(t) \end{array}$$

Grâce à ces deux opérateurs on peut alors écrire des définitions récurrentes, comme par exemple la suite des entiers naturels : `N = 0 -> pre N + 1`.

Une autre caractéristique de LUSTRE est la possibilité de définir plusieurs notions de temps dans un même programme. En effet, un flot n'est pas simplement une suite de valeurs : il possède aussi une *horloge* qui représente la suite d'instants où ces valeurs apparaissent. Les différentes horloges d'un programme ne sont pas indépendantes les unes des autres : il existe une échelle de temps de référence (appelée horloge de base) de laquelle on extrait par échantillonnage des horloges plus “lentes”. L'horloge de base d'un programme LUSTRE est la notion de temps la plus fine qu'il connaisse : ses instants sont définis par la séquence des entrées. Les autres horloges sont définies grâce aux flots booléens : les instants de l'horloge définie par le flot booléen C sont ceux où la valeur de C est *vrai*. Par exemple, la figure 0.2 représente les échelles de temps définies par le flot C sur l'horloge de base, et par le flot C′ sur l'horloge C.

Pour manipuler les horloges, le langage LUSTRE dispose de deux opérateurs. L'opérateur binaire `when` permet d'*échantillonner* un flot sur une horloge plus lente. De manière duale, l'opérateur `current` permet de *projeter* un flot (déjà échantillonné) sur l'horloge immédiatement plus rapide. Ces opérateurs sont présentés plus en détail dans le chapitre 1.

La notion d'horloge est très stricte en LUSTRE : il n'est pas question d'opérer sur des flots qui n'ont pas la même notion de temps. Pour que le programme soit correct, il faut pouvoir associer une horloge à chaque opérateur. Pour ce faire, on suppose que les constantes, comme les entrées, sont sur l'horloge de base, et on cherche à vérifier que tout opérateur s'applique à des opérandes de même horloge.

Enfin, LUSTRE est un langage modulaire. Un programme, appelé *nœud*, est constitué d'une spécification d'interface (noms et types des paramètres formels d'entrée et de sortie), et d'un système d'équations définissant les sorties (et d'éventuelles variables locales) en fonction des entrées. Tout nœud LUSTRE peut être réutilisé dans la définition d'un autre nœud.

Le compilateur LUSTRE-V2

La base de notre travail est le compilateur LUSTRE-V2 écrit par J. Plaice [Pla88]. De façon tout à fait classique, la compilation d'un programme y est réalisée en deux temps : analyse statique puis génération de code. Au cours de la première phase, le compilateur effectue les vérifications habituelles (syntaxe et types) et des opérations particulières au langage comme le calcul des horloges et l'expansion des nœuds internes. L'expansion consiste en quelque sorte à réaliser l'inclusion textuelle des sous-programmes dans le programme principal. Cette opération est nécessaire car le compilateur produit un code purement séquentiel ; or il est généralement impossible de produire du code séquentiel de façon modulaire à partir d'un programme parallèle.

La structure de contrôle du code produit est un automate fini. La synthèse d'un automate de contrôle est une adaptation de la méthode employée pour la compilation du langage ESTEREL [1]. Il s'agit en premier lieu de choisir dans le programme un sous-ensemble d'expressions booléennes que l'on considère comme le "noyau" de contrôle du programme. Ce noyau peut être vu comme un circuit logique, composé d'opérateurs classiques (`and`, `or`, `not` etc...) et de retards (mémoires booléennes) ; la figure 0.3 représente le circuit associé à l'équation "`n = e and not pre n`". Le comportement de ce sous-réseau booléen est parfaitement déterminé par un certain nombre d'entrées logiques, qu'on appelle les *conditions* du programme ; dans notre exemple la seule condition est le booléen `e` qui est par exemple une entrée du programme. Ce circuit ne peut être que dans un nombre fini d'états, caractérisés par la valeur associée à chaque mémoire (appelée variable d'état). Dans notre exemple, si on est dans l'état où "`pre n` = *true*", la sortie `n` est fausse, indépendamment de l'entrée `e` ; on en déduit que l'état successeur est "`pre n` = *false*". Dans cet état on peut déterminer que la sortie porte la même valeur que `e` ; l'état successeur est donc, suivant la valeur de cette entrée, l'état "`pre n` = *true*" ou l'état "`pre n` = *false*". Les comportements possibles du circuit peuvent donc être synthétisés sur un automate fini (figure 0.3). A chaque état de l'automate de contrôle, on peut associer un programme simplifié :

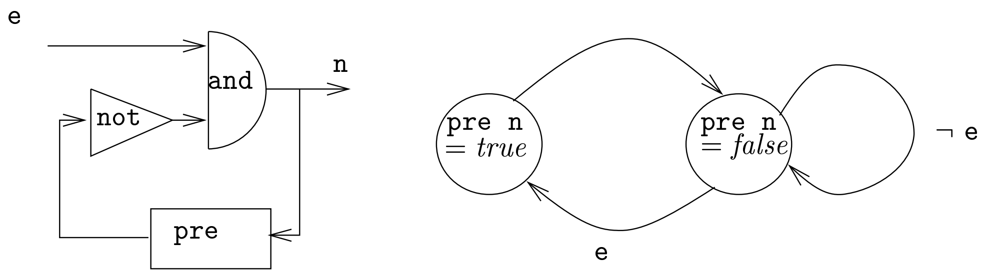


FIG. 0.3 – *Le circuit et l'automate associés à "*`n = e and not pre n`*"*

les variables d'état sont remplacées par leur valeur (vrai ou faux), ce qui permet de simplifier les expressions booléennes où elles apparaissent.

Cette méthode permet de produire un code très rapide : un grand nombre de calculs booléens est effectué une fois pour toutes dès la compilation. Malheureusement, l'aspect exhaustif de cette génération provoque souvent une explosion combinatoire de la taille des automates, et donc du code produit (dans le pire des cas, un automate modélisant le comportement de n

1. les équipes travaillant à la compilation de Lustre et d'Esterel ont d'ailleurs défini un code portable commun [PS87]

variables booléennes a 2^n états, et 2^{2n} transitions). L'explosion du nombre d'états peut avoir deux origines :

- Le programme considéré (ou plutôt son "noyau" booléen) est intrinsèquement complexe. Une solution consiste alors à réduire le noyau de contrôle, en traitant certaines expressions booléennes comme de simples données. Cette solution existe déjà dans le compilateur Lustre-V2, qui propose trois options pour le choix du noyau de contrôle. Parmi ces options il faut citer la solution "extrême" qui consiste à considérer toutes les expressions booléennes comme des données. On obtient alors un code dit en *boucle simple* (un seul état), dont la taille est toujours linéaire par rapport à celle du programme initial.
- L'autre cause d'explosion n'est pas forcément liée à la complexité du programme, mais à la méthode de génération. En effet, l'algorithme utilisé dans le compilateur Lustre-V2 produit généralement des automates non minimaux : cet algorithme est *dirigé par les données*, c'est-à-dire qu'il calcule exhaustivement le comportement des mémoires booléennes sans tenir compte de leur influence effective sur les sorties du programme. Dans ce cas, la solution consiste à réduire a posteriori l'automate obtenu [Fer90]. Le défaut de cette méthode est d'interdire la compilation d'une classe de programmes : ceux dont l'automate non minimal est trop gros pour être construit, bien qu'un automate minimal équivalent de taille raisonnable existe.

Pour chaque état de l'automate de contrôle, le compilateur génère un code séquentiel destiné à calculer les sorties mais aussi à choisir l'état successeur. Pour structurer ce code, il ne dispose que du choix déterministe (c'est-à-dire de l'instruction conditionnelle). L'influence de cette opération (*séquentialisation*) est très importante sur la taille du code généré : un mauvais choix de l'ordre des tests et des instructions à générer peut provoquer une nouvelle explosion de la taille du code, et ce dans chaque état. Le compilateur Lustre-V2 utilise un algorithme très simple, de manière à consacrer la puissance de calcul au parcours des états de l'automate. Ce point de vue est sans doute raisonnable, mais la mauvaise qualité du code séquentiel est parfois trop évidente, notamment en ce qui concerne la duplication inutile de code dans les deux branches d'un test.

Enfin, pour limiter l'explosion de l'automate de contrôle, le compilateur dispose éventuellement d'indications fournies par le programmeur. Ce dernier peut en effet écrire des *assertions*, c'est-à-dire des propriétés toujours vérifiées par l'environnement (par exemple l'exclusion ou l'implication de certaines entrées). Correctement utilisée, une assertion peut permettre d'éliminer des comportements impossibles, et donc de simplifier l'automate de contrôle. Mais la prise en compte d'une assertion peut tout aussi bien créer de nouveaux états : paradoxalement, le code produit en tenant de cette indication "simplificatrice" est alors plus gros que si on l'avait ignorée. Le traitement des assertions est trop complexe pour être présenté dans cette introduction, nous retiendrons simplement que le compilateur Lustre-V2 ne sait pas éviter l'éventuelle complication du code due aux assertions. De plus, il ne sait traiter qu'une classe d'assertions dites *causales*.

Objectifs du travail

Les problèmes liés à l'analyse statique et à l'expansion ont été résolus dans le compilateur Lustre-V2. De plus, un outil performant consacré à cette phase existe déjà : il s'agit de la partie haute du compilateur POLLUX[2] développé par F. Rocheteau ([Roc92]). Notre projet porte donc sur la réalisation d'un générateur de code séquentiel (LUSTRE-V3) à partir d'un programme expansé statiquement correct.

Le principe de la génération d'automate n'est pas remis en cause, puisqu'il permet d'obtenir un code très rapide. C'est essentiellement sur la taille du code généré que se porte notre étude, et en premier lieu sur le problème de l'explosion du nombre d'états :

- Dans le cas des programmes intrinsèquement complexes, on étudie de nouvelles heuristiques pour le choix des noyaux de contrôle. Une autre solution pour limiter l'explosion consisterait à faire de la compilation séparée. En effet, après l'expansion d'un nœud interne A dans le code d'un nœud principal B on obtient un automate qui est le produit des automates de A et B. Si par contre A est compilé séparément, et appelé dans l'automate de B comme une simple procédure, le système global n'est que la "somme" des automates de A et B. La compilation séparée est une solution très intéressante, mais nous avons montré [Ray88] qu'il s'agissait essentiellement d'un problème d'analyse statique.

- Pour éviter l'explosion parasite, la solution envisagée est de définir et d'implémenter un algorithme qui produise directement un automate minimal [BFH⁺90, HRR91]. Il s'agit d'une méthode de génération *dirigée par la demande*, c'est-à-dire uniquement par le besoin de calculer les sorties du programme. L'implémentation d'une telle méthode se heurte à la complexité des algorithmes à mettre en œuvre : il faut par exemple être capable de décider si une proposition est une tautologie, ou construire une composition de fonctions booléennes. Pour faire face à ce problème, nous allons utiliser une représentation à base de *graphes binaires de décision* [Ake78, Bry86]. Ce type de représentation est déjà utilisé avec succès dans le domaine de la preuve formelle de circuits [Bil87, BM88].

Si la taille du code produit est directement liée au nombre d'états, il ne faut pas négliger l'influence de la séquentialisation. Cette phase consiste essentiellement à construire pour chaque état une structure de contrôle "branchue" à base de tests de contrôle. Or, si p tests doivent être ouverts, cette structure de contrôle est éventuellement un arbre binaire complet de profondeur p. Pour éviter cette nouvelle explosion de la taille du code, il faut par exemple essayer d'ouvrir les tests le plus tard possible, et évidemment de les refermer au plus tôt. En se basant sur des hypothèses très simples, nous avons dégagé la forme générale d'un algorithme produisant le code "optimal". Cet algorithme est évidemment paramétré par un certain nombre de choix combinatoires. Pour que le temps de compilation soit raisonnable, il faut donc étudier des heuristiques destinées à en réduire la complexité.

Grâce à une représentation efficace à base de graphes de décision, on peut espérer améliorer considérablement le traitement des assertions, et éviter notamment que celles-ci ne compliquent le code plutôt que de le simplifier. Ce type de représentation doit aussi nous permettre de traiter le cas des assertions non-causales. En effet, les travaux liés à la vérification formelle de programmes LUSTRE [Rat92] ont montré que ce problème pouvait être résolu par des manipulations symboliques.

2. ce compilateur produit du code pour un circuit programmable (PAM)

Organisation du document

La première partie de cet ouvrage présente le langage Lustre et les principes de l'analyse statique. Notre générateur de code ne travaille pas directement sur un programme Lustre, mais sur un fichier au format EC. Ce format permet de représenter des nœuds Lustre expansés et statiquement corrects ; les programmes EC sont produits automatiquement à partir de Lustre par l'option -*expand* de l'outil Pollux.

La partie suivante est consacrée à la normalisation du programme expansé. Il s'agit d'une phase intermédiaire destinée à simplifier la génération de code, en éliminant notamment la notion d'horloge. La correction de cette transformation est basée sur une sémantique formelle du programme source. Cette partie s'achève sur la présentation de quelques exemples qui permettent de se familiariser avec les principes de la génération de code.

La dernière partie est entièrement consacrée au générateur de code. Après avoir présenté le sous-ensemble du code portable Lustre-Esterel utilisé par le générateur, nous définissons les conventions de "bon interfaçage" entre le programme produit et son environnement (ces conventions concernent l'implémentation des entrées/sorties et des objets importés). Le compilateur et le document adoptent la dualité classique entre contrôle et données : le contrôle est le domaine des expressions booléennes du programme, les données celui des expressions arithmétiques et des fonctions externes. Les graphes binaires de décision sont tellement importants dans le générateur de code que nous avons inclus leur présentation dans le document. Notre contribution dans ce domaine consiste essentiellement en une généralisation de cette représentation pour des fonctions à arguments booléens et à valeurs dans un ensemble fini quelconque. Ce type de représentation s'avère en effet très efficace pour implémenter et manipuler des actions gardées. Enfin, le problème des assertions nous a conduits à étudier de manière générale la *simplification des fonctions booléennes*, à laquelle nous consacrons un chapitre.

Nous concluons ce document par une présentation concrète du générateur de code et de ses options, et par quelques perspectives de travail.

Première partie

Le langage LUSTRE

Chapitre 1

Définition du langage Lustre

1.1 Expressions et équations

En Lustre, une expression de type τ dénote une suite infinie de valeurs du type τ.

1.1.1 Types

Les types prédéfinis sont les type booléen (`bool`), entier (`int`) et réel (`real`). Les constantes de ces trois types s'écrivent de manière classique : `true` et `false` pour les booléens, comme dans les langages C ou Pascal pour les constantes arithmétiques. Les opérateurs arithmétiques et logiques classiques sont eux aussi prédéfinis. Tout autre type nécessaire à une application doit être déclaré comme externe, avec des conventions que nous étudierons plus tard.

1.1.2 Constantes

Si c est une constante, c dénote en Lustre la suite infinie de c ; par exemple, la constante entière `1` dénote la suite : $(1, 1, 1, \cdots)$.

1.1.3 Opérateurs sur les valeurs

Un opérateur classique `OP` de $\tau_1 \times \tau_2 \times \cdots \times \tau_n$ dans τ, opère point à point sur les suites infinies de ces types. Par exemple :

$$\begin{array}{rrcl} & \textit{si} \;\; X & = & (x_1, x_2, \cdots, x_i, \cdots) \\ & \textit{et} \;\; Y & = & (y_1, y_2, \cdots, y_i, \cdots) \\ \textit{alors} & X\texttt{+}Y & = & (x_1 + y_1, x_2 + y_2, \cdots, x_i + y_i, \cdots) \end{array}$$

Les opérateurs arithmétiques et logiques classiques sont prédéfinis. La définition du langage Esterel [BCG87] a suggéré l'introduction d'un opérateur booléen n-aire dit *d'exclusion* : intuitivement, l'expression `#`$(x_1, \cdots, x_n)$ est vraie si au plus un de ses arguments est vrai, fausse sinon. Cet opérateur permet d'exprimer notamment des *assertions* sur les entrées du programme (§1.3).

1.1.4 Opérateurs sur les suites

Le langage possède deux opérateurs originaux qui permettent de manipuler les suites :

- L'opérateur `pre` permet de décaler une suite dans le temps ; si $X = (x_1, x_2, \cdots, x_i, \cdots)$ alors :

  $$\texttt{pre}\ X \quad = \quad (\texttt{nil}, x_1, x_2, \cdots, x_i, \cdots)$$

  `nil` dénote une valeur indéfinie. Elle exprime ici le fait que la valeur d'une suite avant l'instant initial n'a pas de sens.

- L'opérateur `->` permet d'initialiser une suite ; si $X = (x_1, x_2, \cdots, x_i, \cdots)$ et $Y = (y_1, y_2, \cdots, y_i, \cdots)$ sont deux suites de même type, alors :

  $$X\ \texttt{->}\ Y = (x_1, y_2, \cdots, y_i, \cdots)$$

1.1.5 Equations

Chaque variable "*id*" du programme est définie par une unique équation de la forme :
id = *expression*
Les expressions sont construites avec des constantes, des opérateurs sur les valeurs et les suites, mais aussi des variables et des entrées. La sémantique intuitive d'une équation est donnée par les principes suivants :

Principe de substitution : il y a synonymie complète entre l'expression et la variable. En particulier, dans toute expression, on peut remplacer la variable par l'expression qui la définit, et réciproquement. Il en découle naturellement que l'ordre des équations n'a aucune incidence sur la sémantique d'un programme.

Principe de définition : le comportement de la variable est complètement déterminé par l'expression associée et le comportement des variables qui apparaissent dans cette expression. En particulier, aucune information ne doit être déduite de la manière dont est utilisée la variable dans les autres définitions.

On peut écrire des équations récurrentes comme "`N = 0 ->pre N + 1`". Cette équation définit de proche en proche `N` comme la suite des entiers naturels :

$$\begin{array}{rl} \texttt{pre N} = & (\texttt{nil},\ 0,\ 1,\ 2,\ \cdots) \\ \texttt{N} = & (0,\ \ 1,\ 2,\ 3,\ \cdots) \end{array}$$

A tout instant, la valeur d'une suite peut dépendre de valeurs calculées dans le passé. Par contre des définitions "instantanément" récursives, comme : `Y = Y * Y - 1` sont refusées. En effet, même si ces équations ont des solutions, on ne peut pas toujours les résoudre en un temps borné. Pour être correct, un programme doit donc être *sans blocage*, c'est-à-dire que la valeur d'une suite ne doit pas dépendre instantanément d'elle-même.

1.1.6 Horloges et flots

Il n'y a pour l'instant qu'une notion de temps dans un programme LUSTRE : celle induite par la suite des valeurs. Pour certaines applications, il est cependant intéressant de faire évoluer des sous-systèmes à des rythmes différents.

Prenons par exemple le cas d'un programme qui aurait pour but d'étudier une course. L'unité de temps qui peut permettre de départager les concurrents, et d'établir d'éventuels records, est généralement le centième de seconde. Certains calculs, comme la vitesse moyenne, n'ont de sens que sur une "horloge" moins rapide, comme par exemple la seconde ou la minute. Enfin, certains calculs peuvent être effectués sur la base du nombre de tours de circuit effectués.

1.1.7 Opérateur d'échantillonnage

Les séquences booléennes vont nous permettre de définir des "horloges" plus lentes que l'horloge globale grâce à l'opérateur d'échantillonnage `when`. Si `E` est une expression et `C` une expression booléenne, `E when C` dénote la suite de valeurs extraite de `E` quand `C` est vraie. L'expression `E when C` n'a pas "la même notion de temps" que `E` et `C` : quand l'horloge `C` est fausse, la suite `E when C` n'existe pas (figure 1.1).

$$
\begin{array}{rlcccccl}
\texttt{E} & = (& e_1 & e_2 & e_3 & e_4 & e_5 & \cdots) \\
\texttt{C} & = (& \texttt{true} & \texttt{false} & \texttt{true} & \texttt{true} & \texttt{false} & \cdots) \\
\texttt{X = E when C} & = (& x_1 = e_1 & & x_2 = e_3 & x_3 = e_4 & & \cdots)
\end{array}
$$

FIG. 1.1 – *L'opérateur d'échantillonnage*

Soit `H` une autre expression booléenne, `H when C` dénote une nouvelle suite qui a la même notion de temps que `X = E when C`. On peut donc échantillonner `X` sur cette nouvelle notion de temps : `Y = (E when C) when (H when C)`.

La possibilité d'obtenir des horloges de plus en plus lentes nous amène à définir la notion de *flot* :

> En LUSTRE, toute expression E de type τ dénote un flot de valeurs, c'est-à-dire un couple composé :
>
> – d'une suite de valeurs du type τ,
>
> – d'une horloge, qui est soit l'horloge de base dénotée par l'expression `true`, soit un flot quelconque à valeurs booléennes.

Par définition les constantes sont toutes sur l'horloge de base, et les opérateurs sur les valeurs s'appliquent à des flots de même horloge. Par exemple, l'expression `E + (E when C)` n'est pas correcte.

1.1.8 Opérateur de projection

Pour opérer entre des flots qui n'ont pas la même horloge, il faut les projeter sur une horloge commune grâce à l'opérateur `current`. Si `X` dénote un flot sur l'horloge `C`, `current X` dénote un flot sur l'horloge `H` de `C` ; à chaque instant de l'horloge `H`, si `C` est vrai, `current X` porte la même valeur que `X`, sinon il porte la valeur prise par `X` au dernier instant où `C` était vraie. La figure 1.2 met en évidence l'inégalité : `E` $\neq$ `current(E when C)`.

`E`	= (	e_1	e_2	e_3	e_4	e_5	$\cdots$)
`C`	= (	`true`	`false`	`true`	`true`	`false`	$\cdots$)
X = E when C	= (	e_1		e_3	e_4		$\cdots$)
Y = current X	= (	e_1	e_1	e_3	e_4	e_4	$\cdots$)

FIG. 1.2 – *L'opérateur de projection*

1.2 Nœuds et réseaux

Tout programme peut être vu comme un réseau d'opérateurs connectés par des fils. Par exemple, l'équation "`N = 0 ->pre N + 1`" décrit le réseau de la figure 1.3. Tous les opérateurs de base sont des nœuds prédéfinis, mais le programmeur peut définir et réutiliser ses propres nœuds.

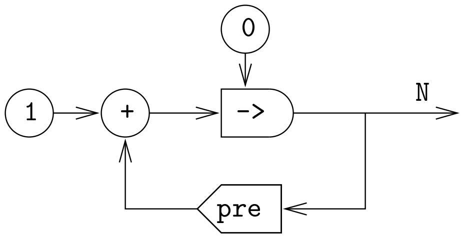


FIG. 1.3 – *Un réseau d'opérateurs*

1.2.1 Définition d'un nœud

Prenons l'exemple d'un compteur généralisé :

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

Ce nœud prend en entrée deux flots entiers et un flot booléen, et retourne le flot entier `n`. Les paramètres d'entrée sont supposés être sur la même horloge. Ce paramètre "sous-entendu" n'est autre que l'horloge de base, c'est-à-dire l'horloge la plus fine connue à l'intérieur du nœud. On peut cependant définir des entrées sur une horloge plus lente, à condition que cette horloge soit elle-même une entrée :

```
node N( x, y : int; h : bool; (z : int when h)) returns ( ...);
```

1.2.2 Instanciation d'un nœud

L'instanciation d'un nœud se fait de manière fonctionnelle : il s'agit donc d'une expression particulière. Par exemple, l'expression `COMPTEUR(0,1,false)` dénote la suite des naturels, `COMPTEUR(0,2,false)` celle des nombres pairs. Le nœud peut être utilisé dans une définition récurrente, pour définir par exemple un compteur modulo 3 :

```
mod3 = COMPTEUR(0,1,pre(mod3)=2);
```

Nous avons vu que l'horloge d'un nœud est celle de ses paramètres. On peut par exemple filtrer les entrées d'un nœud pour qu'il évolue à un rythme plus lent :

```
COMPTEUR((0,1,false) when (mod3 = 0))
```

Par contre, si on se contente de filtrer les sorties, le nœud continue à évoluer sur l'horloge de base :

```
COMPTEUR(0,1,false) when (mod3 = 0)
```

et on obtient un résultat différent (figure 1.4).

`mod3`	= (	0	1	2	0	1	2	0	1	···)
`COMPTEUR((0,1,false) when (mod3 = 0))`	= (	0			1			2		···)
`COMPTEUR(0,1,false) when (mod3 = 0)`	= (	0			3			6		···)

FIG. 1.4 – *Echantillonnage des entrées ou des sorties*

1.2.3 Tuples

Pour pouvoir instancier des nœuds ayant plusieurs sorties, LUSTRE permet d'utiliser des tuples. Par exemple, un nœud réalisant la division entière dont l'entête est :

```
node DIVIDE(x, y : int) returns (q, r : int);
```

peut être instancié de la manière suivante :

```
(quotient, reste) = DIVIDE(X, Y);
```

Si toutes les composantes du tuple sont sur la même horloge, on peut considérer que ce tuple est en fait un flot dont le domaine de valeurs est le produit cartésien des domaines de ses opérandes. On peut alors combiner des expressions de type "tuple" par des opérateurs polymorphes :

```
(min,max) = if x<=y then (x,y) else (y,x);
```

1.3 Les assertions

Comme dans le langage ESTEREL [BCG87], le programmeur peut écrire en LUSTRE des assertions, c'est-à-dire des propriétés vérifiées par l'environnement du programme. La notion d'assertion est cependant plus générale en LUSTRE qu'en ESTEREL. Une assertion est de la forme :

`assert` *expression*

où *expression* est une expression booléenne quelconque. Nous étudions dans cet ouvrage comment utiliser de telles assertions pour optimiser le code produit par le compilateur.

Chapitre 2

Vérifications statiques

La compilation d'un programme LUSTRE est divisée de manière tout à fait classique en une phase de vérifications statiques et une phase de génération de code. Il existe actuellement deux outils consacrés à la première phase. Le premier fait partie intégrante du compilateur LUSTREV2 [Pla88]. Le deuxième est développé par F. Rocheteau dans le cadre du projet POLLUX [Roc92]. Ces deux outils fournissent des programmes statiquement corrects dans un format intermédiaire appelé EC (de l'anglais *expanded code*). L'outil que nous avons développé est uniquement consacré à la deuxième phase, c'est-à-dire à la génération de code séquentiel à partir d'un programme syntaxiquement correct.

Nous présentons brièvement dans ce chapitre les principes de la vérification statique. Certaines vérifications sont tout à fait classiques, comme celle des types. D'autres sont liées à l'aspect "flot de donnée" du langage (inter-blocages, horloges). Enfin, il est parfois impossible de produire du code séquentiel de façon modulaire. C'est pourquoi les appels de nœuds internes sont en fait "expansés" dans le code.

2.1 Vérifications statiques

2.1.1 Inter-blocages

Comme nous l'avons vu dans le paragraphe 1.1.5, les définitions "instantanément" récurrentes sont refusées. La notion d'inter-blocage est bien entendu globale. On ne résout pas le blocage en remplaçant l'équation `X = X*X -1` par le système :

```
X = Y-1;
Y = X*X;
```

Le critère retenu est purement syntaxique, on refuse par exemple de "faux" blocages comme :

```
X = if C then Y else Z;
Y = if C then Z else X;
```

2.1.2 Vérification des types

La vérification des types est assez complexe en LUSTRE, notamment à cause des opérateurs polymorphes et des tuples. Dans le compilateur LUSTREV2, J. Plaice [Pla88] se base sur l'inférence des types dans le langage ML. Au niveau de notre générateur de code, la cohérence des types est garantie par la donnée d'une fonction *type* qui associe son type à chaque expression du programme EC. Ce type est soit un type simple (prédéfini ou externe), soit un vecteur de types simples (dans le cas des tuples).

2.1.3 Vérification des horloges

Ce paragraphe décrit brièvement le traitement des horloges en LUSTRE, qui constitue l'un des aspects originaux de la compilation du langage. Ce traitement consiste à associer une horloge à toute expression du programme, en vérifiant que tout opérateur est appliqué à des opérandes d'horloges convenables, c'est-à-dire que :

- tout opérateur de base, à plus d'un argument, est appliqué à des opérandes de même horloge ;
- tout opérande d'un opérateur `current` est sur une horloge différente de l'horloge de base du nœud où il apparaît ;
- les horloges des paramètres effectifs d'un nœud satisfont les contraintes spécifiées dans la déclaration d'interface du nœud.

Il nous faut d'abord préciser ce que nous entendons par "même horloge". Idéalement, deux expressions sont sur la même horloge si leurs horloges sont définies par le même flot booléen. L'égalité de flots booléens étant évidemment indécidable, on se contente d'une notion plus restrictive : deux expressions booléennes `B1` et `B2` définissent la même horloge, si et seulement si elles peuvent être rendues syntaxiquement identiques par application du principe de substitution (§1.1.5). Ainsi, dans l'exemple suivant :

```
x = a when (y>z);
y = b+c;
u = d when (b+c>z);
v = e when (z<y);
```

`x` et `u` sont sur la même horloge (dénotée par l'expression `b+c>z`), considérée différente de celle de `v`.

Les règles de calcul des horloges appliquées par le compilateur sont décrites formellement dans [CPHP87, Pla88, HL91]. Le calcul est effectué en respectant le principe de définition, c'est-à-dire que l'horloge de toute variable doit être déduite de sa définition, sans tenir compte de la façon dont la variable est utilisée. Par exemple, le programme suivant, où `M` et `N` sont des nœuds retournant un résultat sur la même horloge que leur paramètre d'entrée :

```
X = M(Y); Y = N(X); Z = X+Y+1;
```

est erroné du point de vue des horloges[1]. En effet, s'il est possible de déduire de la définition de `Z` que `X` et `Y` devraient être sur la même horloge que `1` (c'est-à-dire l'horloge de base), cette information ne provient pas de la définition de `X` et `Y`. Cette définition ne fournit que l'égalité des horloges de `X` et `Y`, ce qui est insuffisant.

2.2 Expansion des nœuds

Le générateur de code produit un programme purement séquentiel. Or il est bien connu que la production de code séquentiel à partir d'un programme parallèle ne peut, en général, se faire de façon modulaire. En d'autres termes, on ne peut séquentialiser le code d'un sous-programme parallèle, indépendamment de son contexte d'utilisation. En Lustre, un exemple très simple illustre cette impossibilité. Considérons le nœud suivant :

```
node double_copie(a, b: int) returns (x, y: int);
let x = a ; y = b ; tel.
```

Il est bien évident qu'il y a deux codes séquentiels possibles, pour un cycle d'exécution de ce nœud : soit "`x:=a ; y:=b`", soit "`y:=b; x:=a`".

Le problème est que le choix entre ces deux formes n'est pas indifférent et peut dépendre de la façon dont le nœud est appelé. Par exemple, pour l'appel suivant :

```
(x,y) = double_copie(a,x);
```

qui correspond à la figure 2.1, seule la première forme convient.

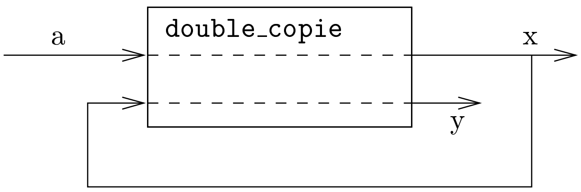


Fig. 2.1 – *Un appel bouclé*

Avant de générer le code, le compilateur réalise donc l'expansion des nœuds internes dans le nœud principal : tout appel de nœud est remplacé par son corps après renommage des paramètres et des horloges. C'est à partir de ce programme "plat" qu'on va pouvoir générer du code séquentiel.

Il est cependant possible, comme nous l'avons montré dans [Ray88], de compiler séparément un nœud LUSTRE. On peut en effet restructurer le nœud en un ensemble de nœuds compilables séparément, appelés par un nœud principal résumant leurs contraintes de séquencement. Seul ce nœud principal doit alors être expansé dans le programme principal. Les instances des nœuds compilés séparément sont alors traitées à peu près comme des appels de fonctions externes[2].

1. Notons que, si la sortie de l'un des nœuds N ou M ne dépend que du passé strict de son entrée, ce programme ne contient pas de blocage.

2. En fait, un nœud externe n'est pas une fonction quelconque. Il possède sa propre mémoire qui doit évoluer chaque fois que l'horloge de l'appel est vraie, et pas seulement quand les sorties du nœud sont nécessaires au programme principal.

2.3 Le format EC

La génération de code séquentiel n'est pas le seul sujet d'étude autour du langage LUSTRE :

– C. Ratel développe l'outil LESAR pour la vérification de propriété sur un programme LUSTRE [Rat92, RHR91].

– F. Rocheteau développe le compilateur POLLUX qui produit du code pour un circuit programmable [Roc92, RH91].

La phase d'analyse statique et d'expansion, nécessaire à toutes ces applications, est déjà implémentée dans POLLUX. Grâce à l'option *-expand*, le compilateur POLLUX permet aux autres applications de récupérer le résultat de son analyse statique : cette option produit à partir de tout programme LUSTRE statiquement correct un fichier dans le format EC (de l'anglais *expanded code*).

Un fichier EC est composé d'un ensemble de déclarations d'objets externes (types, constantes et fonctions), et de la déclaration d'un unique nœud (les éventuels nœuds internes ont été expansés). Un nœud EC est statiquement correct, c'est-à-dire qu'il ne comporte pas d'interblocages et que chaque expression est correcte du point de vue des types et des horloges. Puisque chaque expression est correctement typée, il est très simple de déduire le type d'une expression du type de ses opérandes : tout se passe comme si on disposait d'une fonction *type* qui associe son type à chaque expression. De plus, le format EC permet de synthétiser très simplement les informations issues du calcul d'horloge :

– toutes les horloges complexes (différentes de l'horloge de base) sont repérées par un identificateur (pour celà, de nouvelles variables locales ont éventuellement été introduites) ; dans les expressions EC, les occurrences de l'opérateur `when` sont donc de la forme "*exp* `when` *ident*".

– chaque identificateur (entrée, sortie ou variable interne) est déclaré avec son horloge (l'horloge de base est notée `true`) :

$$\textit{ident-decl} \ ::= \ \texttt{(}\textit{ident} : \textit{type}\texttt{)} \ \texttt{when} \ \textit{clock}$$

$$\begin{array}{rcl} \textit{clock} & ::= & \texttt{true} \\ & | & \textit{ident} \end{array}$$

En connaissant les horloges de chaque identificateur, et en sachant que le programme est correct (tout opérateur s'applique à des opérandes de même horloge) l'horloge de chaque expression peut être retrouvée très simplement. Notons *ident.clock* l'horloge associée à *ident* dans sa déclaration, la fonction suivante (*ck*) associe à une expression l'identificateur de son horloge :

$$\begin{array}{rcl} ck(ident) & = & ident.clock \\ ck(exp\ \texttt{when}\ ident) & = & ident \\ ck(\texttt{current}\ exp) & = & ck(ck(exp)) \\ ck(op(exp_1, \cdots, exp_n)) & = & ck(exp_1) \quad \textit{pour tout autre opérateur}\ \mathrm{op} \end{array}$$

On admettra par la suite que la donnée d'un programme EC équivaut à la donnée d'un nœud LUSTRE expansé et sans inter-blocages, et de deux fonctions *ck* et *type* qui associent respectivement à une expression son horloge et son type.

Deuxième partie

Le programme normalisé

Chapitre 3

Sémantique opérationnelle

Nous présentons dans ce chapitre une sémantique opérationnelle du programme expansé. Les appels de nœuds ont donc disparu du code, et toutes les vérifications statiques ont été effectuées :

– Le programme est sans blocage ; on est donc sûr que la valeur des sorties à chaque instant ne dépend que de la valeur des entrées et d'un nombre borné de valeurs calculées dans le passé.

– Toutes les expressions du programme sont correctement typées ; on suppose donc connue une fonction *type* qui associe à chaque expression le type correspondant.

– Le programme est correct du point de vue des horloges ; on dispose donc d'une fonction *ck* qui associe à chaque expression une expression booléenne dénotant son horloge ; cette expression est soit la constante `true`, soit un identificateur (§2.3).

La sémantique que nous présentons est très proche de celles données par C. Buors [Buo86] et J. Plaice [Pla88]. Nous insistons surtout sur les notions de flots et d'horloges *apparents*.

3.1 Horloges apparentes et flots apparents

Nous avons présenté la sémantique intuitive des expressions LUSTRE en terme de flots (§1.1), c'est-à-dire de couples composés d'une suite de valeurs et d'une horloge. L'horloge peut être l'horloge de base dénotée par l'expression `true`, ou un flot booléen quelconque, possédant sa propre horloge.

Prenons l'exemple de trois expressions dénotant respectivement les flots f_1, f_2 et f_3. Dans ces expressions, `x` dénote un flot quelconque sur l'horloge de base, les expressions `c` et `d` des flots booléens sur l'horloge de base :

```
e1 = (x when c) when (d when c);
e2 = (x when d) when (c when d);
e3 = x when (c and d);
```

Il est clair que ces expressions dénotent des flots différents, puisque leurs horloges sont toutes différentes. Pourtant, si on se place à un instant quelconque de l'horloge de base :

- si les flots $f_{\mathtt{c}}$ et $f_{\mathtt{d}}$ portent la valeur “vrai”, les trois flots f_1, f_2 et f_3 sont définis et portent la même valeur : la valeur courante du flot $f_{\mathtt{x}}$.
- si un des deux flots $f_{\mathtt{c}}$ ou $f_{\mathtt{d}}$ porte la valeur “faux”, alors l'horloge $f_{\mathtt{(c\ and\ d)}}$ est fausse, et les horloges $f_{\mathtt{(d\ when\ c)}}$ et $f_{\mathtt{(c\ when\ d)}}$ sont soit fausses, soit non définies ; dans tous les cas, aucun des flux f_1, f_2 et f_3 n'est défini.

En fait, bien que les horloges dénotées par “`c and d`”, “`c when d`” et “`d when c`” soient différentes, elles permettent de repérer les mêmes instants de l'horloge de base. Cette constatation nous amène à définir la notion *d'horloge apparente* :

Définition : *L'horloge apparente d'un flot quelconque f est un flot booléen sur l'horloge de base qui, à chaque instant, porte la valeur* `true` *si et seulement si f est défini.*

Dans notre exemple, f_1, f_2 et f_3 ont donc la même horloge apparente, qui n'est autre que le flot $f_{\mathtt{(c\ and\ d)}}$. De plus, quand ces trois flots sont définis, ils portent la même valeur : on dit qu'ils ont le même *flot apparent*. Le flot apparent peut être défini comme un flot sur l'horloge apparente. On préfère cependant le définir comme un flot sur l'horloge de base en introduisant une valeur spéciale, $\perp$:

Définition : *Le flot apparent d'un flot quelconque f est un flot sur l'horloge de base qui, à chaque instant, porte la valeur $\perp$ si f n'est pas défini, porte la même que f sinon.*

3.2 Histoires et mémoires

Le but de la sémantique opérationnelle est de décrire un programme LUSTRE comme une “machine” qui construit le flot apparent des sorties à partir du flot apparent des entrées. Une exécution du programme est caractérisée par le prédicat :

$\nu_i \vdash prg : \nu_o$

Les objets ν_i et ν_o peuvent être définis comme des flots de type tuple : à chaque instant, la valeur du flot ν_i est un tuple composé des valeurs courantes des flots apparents des entrées ; de même, la valeur courante du flot ν_o est composée des valeurs courantes des flots apparents des sorties. On préfère cependant définir ν_i et ν_o comme des *histoires*, c'est-à-dire des séquences infinies de *mémoires* :

$\nu ::= \sigma\nu$

Une mémoire est un objet dynamique tout à fait classique, qui permet d'associer une valeur à un identificateur. A tout instant, une mémoire peut être interprétée comme une fonction partielle de l'ensemble des identificateurs dans celui des constantes (y compris la constante $\perp$). Les opérations de base sur les mémoires sont :

Référence : le prédicat $\sigma(id) = k$ signifie que la valeur k a été associée à l'identificateur id dans la mémoire σ.

Affectation : $\sigma[k/id]$ désigne une mémoire σ' définie par :

$$\begin{array}{l} \sigma'(id) = k \quad et \\ (\sigma(id') = k') \Rightarrow (\sigma'(id') = k') \quad \forall id' \neq id \end{array}$$

Filtrage : $\sigma|ids$, désigne une mémoire σ' définie par :

$$(\sigma'(id) = k) \quad \Leftrightarrow \quad (id \in ids) \wedge (\sigma(id) = k)$$

3.3 Fonctionnement cyclique

3.3.1 Evaluation des sorties

La construction de l'histoire des sorties à partir de celle des entrées est décrite de manière cyclique. Un cycle de calcul correspond à la construction d'une mémoire de sortie en fonction d'une mémoire d'entrée :

$\sigma_i \vdash prg : \sigma_o$

Ce prédicat est cependant insuffisant. En effet, à chaque instant, la valeur des sorties dépend non seulement des entrées courantes, mais aussi d'informations héritées du passé :

– Pour évaluer l'expression e_1 `->` e_2, on a besoin de savoir si l'horloge apparente de e_1 et e_2 est vraie pour la première fois dans l'instant courant.

– Pour évaluer la valeur de `pre` e ou `current` e, on a éventuellement besoin de connaître la dernière valeur significative (i.e. différente de $\perp$) prise par le flot apparent de e.

3.3.2 Réécriture du programme

Une façon élégante de mémoriser les informations nécessaires consiste à dire que le programme se réécrit en fin de cycle en un programme contenant les informations suffisantes pour évaluer la valeur des sorties au cycle suivant.

Opérateur `->` : La réécriture de l'opérateur `->` est particulièrement évidente. En effet, sous certaines contraintes liées à son horloge apparente, l'expression e_1 `->` e_2 se réécrit comme e_2.

Opérateurs de mémoire : Pour pouvoir réécrire les opérateurs `pre` et `current`, on leur associe un argument supplémentaire : $\mathtt{pre}(k, e)$, $\mathtt{current}(k, e)$. Cet argument, qui est toujours une constante, est la dernière valeur significative prise par le flot apparent de e. La correspondance avec les opérateurs unaires de LUSTRE est définie par :

$$\begin{array}{rcl} \mathtt{pre}(exp) & = & \mathtt{pre(nil,}exp) \\ \mathtt{current}(exp) & = & \mathtt{current(nil,}exp) \end{array}$$

3.3.3 Un cycle complet de calcul

Au cours d'un cycle, le programme fournit la mémoire des sorties et se réécrit pour le cycle suivant. Ce fonctionnement est caractérisé par le prédicat :

$\sigma_i \vdash prg \stackrel{\sigma_o}{\rightarrow} prg'$

Le fonctionnement cyclique s'en déduit par la règle d'inférence :

$$\frac{\sigma_i \vdash prg \stackrel{\sigma_o}{\rightarrow} prg' \quad \nu_i \vdash prg' : \nu_o}{\sigma_i \nu_i \vdash prg : \sigma_o \nu_o}$$

3.4 Syntaxe abstraite

Avant d'étudier en détail les règles sémantiques, nous définissons une syntaxe abstraite qui ne retient que les informations strictement nécessaires.

$$\begin{array}{lcl}
prg & ::= & ins\ outs\ eqs \\
 & & \\
eqs & ::= & eq \quad | \quad eq\ eqs \\
 & & \\
eq & ::= & id\ \texttt{=}\ exp \\
 & & \\
exp & ::= & k \\
 & | & id \\
 & | & dataop(exp, \cdots, exp) \\
 & | & exp\ \texttt{when}\ exp \\
 & | & exp\ \texttt{->}\ exp \\
 & | & \texttt{pre(}k\texttt{,}exp\texttt{)} \\
 & | & \texttt{current(}k\texttt{,}exp\texttt{)}
\end{array}$$

Pour les programmes, on n'a retenu que la liste des identificateurs d'entrée et de sortie (*ins* et *outs*), et la liste des équations du programme (*eqs*). Dans les expressions, *dataop* désigne indifféremment un opérateur sur les valeurs ou un appel de fonction externe.

3.5 Un cycle de calcul

Le cycle de calcul est réalisé en deux temps, d'abord l'évaluation des équations du programme, puis la réécriture. L'évaluation des équations est définie par le prédicat :

$\sigma_i \vdash eqs : \sigma$

qui signifie que le système d'équations "enrichit" la mémoire d'entrée σ_i pour construire la mémoire courante σ. Dans cette mémoire, on ne fait aucune distinction entre variable locale et sortie. Pour obtenir la mémoire des sorties, il faut la filtrer grâce à l'indication *outs* du programme.

La réécriture des équations est réalisée en fonction de la mémoire courante :

$\sigma \vdash eqs \rightarrow eqs'$

La règle qui définit un cycle de calcul est donc la suivante :

$$\frac{\sigma_i \vdash eqs : \sigma \qquad \sigma \vdash eqs \rightarrow eqs'}{\sigma_i \vdash ins\ outs\ eqs \overset{\sigma|outs}{\rightarrow} ins\ outs\ eqs'}$$

3.6 Evaluation des équations

L'évaluation d'un système d'équations consiste en une séquence d'affectations qui modifie, de proche en proche, la mémoire courante.

$$\frac{\sigma \vdash eq : \sigma' \qquad \sigma' \vdash eqs : \sigma''}{\sigma \vdash eq\ eqs : \sigma''}$$

L'évaluation de la première équation peut cependant être bloquée si elle nécessite une information qui n'est pas encore dans la mémoire courante. Dans ce cas, l'évaluation est différée grâce à la règle :

$$\frac{\sigma \vdash eqs : \sigma' \qquad \sigma' \vdash eq : \sigma''}{\sigma \vdash eq\ eqs : \sigma''}$$

3.7 Evaluation de l'horloge apparente

L'évaluation d'une équation id `=` exp dans la mémoire courante σ est caractérisée par le prédicat :

$\sigma \vdash id = exp : \sigma[v/id]$

La valeur v est celle portée par le flot apparent de exp dans le cycle courant. Si l'horloge apparente de exp est fausse, cette valeur est $\perp$. On introduit donc un prédicat intermédiaire, $\sigma \overset{xck}{\vdash} exp : b$, qui signifie que dans la mémoire courante σ, l'horloge apparente du flot dénoté par exp est évaluée à la constante booléenne b. On rappelle que la fonction ck associe à toute expression LUSTRE une expression booléenne dénotant son horloge :

- L'horloge apparente d'un flot sur l'horloge de base est toujours vraie :

$$\frac{ck(exp) = \texttt{true}}{\sigma \overset{xck}{\vdash} exp : \texttt{true}}$$

- Sinon, si l'horloge apparente de $ck(exp)$ est fausse, celle de exp l'est aussi :

$$\frac{\sigma \overset{xck}{\vdash} ck(exp) : \texttt{false}}{\sigma \overset{xck}{\vdash} exp : \texttt{false}}$$

- Sinon, la valeur de l'horloge apparente est obtenue en évaluant l'expression $ck(exp)$:

$$\frac{\sigma \overset{xck}{\vdash} ck(exp) : \texttt{true} \qquad \sigma \vdash ck(exp) : b}{\sigma \overset{xck}{\vdash} exp : b}$$

3.8 Evaluation d'une équation

Grâce au prédicat $\overset{xck}{\vdash}$, l'évaluation d'une équation s'écrit très simplement :

– Si dans la mémoire courante, l'horloge apparente est fausse, on associe à id la valeur $\perp$:

$$\frac{\sigma \overset{xck}{\vdash} exp : \texttt{false}}{\sigma \vdash id = exp : \sigma[\perp / id]}$$

– Sinon, on évalue exp, et on associe le résultat à id :

$$\frac{\sigma \overset{xck}{\vdash} exp : \texttt{true} \quad \sigma \vdash exp : k}{\sigma \vdash id = exp : \sigma[k/id]}$$

3.9 Evaluation des expressions

– Une constante rend toujours sa valeur :

$$\sigma \vdash k : k$$

– Un identificateur ne peut être utilisé que si une valeur lui est associée dans la mémoire courante, d'où l'importance de la deuxième règle pour l'évaluation des systèmes d'équations :

$$\frac{\sigma(id) = k}{\sigma \vdash id : k}$$

– La valeur d'un opérateur simple est obtenue en combinant les valeurs des opérandes ; $DATAOP$ désigne l'opérateur sémantique associé à $dataop$:

$$\frac{\sigma \vdash exp_1 : k_1 \ \cdots \ \sigma \vdash exp_n : k_n}{\sigma \vdash dataop(exp_1, \cdots, exp_n) : DATAOP(k_1, \cdots, k_n)}$$

– L'opérateur `when` est ignoré :

$$\frac{\sigma \vdash exp : k}{\sigma \vdash exp \ \texttt{when} \ exp' : k}$$

– Le résultat d'un `->` est celui de l'opérande gauche :

$$\frac{\sigma \vdash exp : k}{\sigma \vdash exp \ \texttt{->} \ exp' : k}$$

– La valeur d'un `pre` est la dernière valeur significative :

$$\sigma \vdash \texttt{pre}(k, exp) : k$$

– Le résultat du `current` est, suivant la valeur de l'horloge apparente, la valeur courante ou la dernière valeur significative :

$$\frac{\sigma \overset{xck}{\vdash} \mathit{exp} : \texttt{false}}{\sigma \vdash \texttt{current(}k\texttt{,}\mathit{exp}\texttt{)} : k}$$

$$\frac{\sigma \overset{xck}{\vdash} \mathit{exp} : \texttt{true} \quad \sigma \vdash \mathit{exp} : k'}{\sigma \vdash \texttt{current(}k\texttt{,}\mathit{exp}\texttt{)} : k'}$$

3.10 Réécriture des équations

Le système d'équations est réécrit en séquence ; seules les expressions sont affectées par cette réécriture :

$$\frac{\sigma \vdash \mathit{exp} \to \mathit{exp}' \quad \sigma \vdash \mathit{eqs} \to \mathit{eqs}'}{\sigma \vdash \mathit{id}\texttt{=}\mathit{exp}\ \mathit{eqs} \to \mathit{id}\texttt{=}\mathit{exp}'\ \mathit{eqs}'}$$

– Les constantes et les identificateurs ne sont pas modifiés :

$$\sigma \vdash k \to k \quad \sigma \vdash \mathit{id} \to \mathit{id}$$

– La règle suivante s'applique à tous les opérateurs simples, mais aussi à l'opérateur `when` :

$$\frac{\sigma \vdash \mathit{exp}_1 \to \mathit{exp}'_1 \ \cdots \ \sigma \vdash \mathit{exp}_n \to \mathit{exp}'_n}{\sigma \vdash \mathit{dataop}(\mathit{exp}_1, \cdots, \mathit{exp}_n) \to \mathit{dataop}(\mathit{exp}'_1, \cdots, \mathit{exp}'_n)}$$

– La constante associée à un *memop* (`pre` ou `current`) n'évolue que si l'horloge apparente de l'argument est vraie :

$$\frac{\sigma \overset{xck}{\vdash} \mathit{exp} : \texttt{false} \quad \sigma \vdash \mathit{exp} \to \mathit{exp}'}{\sigma \vdash \mathit{memop}(k, \mathit{exp}) \to \mathit{memop}(k, \mathit{exp}')}$$

$$\frac{\sigma \overset{xck}{\vdash} \mathit{exp} : \texttt{true} \quad \sigma \vdash \mathit{exp} : k' \quad \sigma \vdash \mathit{exp} \to \mathit{exp}'}{\sigma \vdash \mathit{memop}(k, \mathit{exp}) \to \mathit{memop}(k', \mathit{exp}')}$$

– Si dans le cycle courant, l'horloge apparente d'un `->` est fausse, cette expression continuera au cycle suivant à être évaluée comme son opérande gauche ; sinon, elle sera dorénavant évaluée comme son opérande droit :

$$\frac{\sigma \overset{xck}{\vdash} (\mathit{exp}_1 \ \texttt{->}\ \mathit{exp}_2) : \texttt{false} \quad \sigma \vdash \mathit{exp}_1 \to \mathit{exp}'_1 \quad \sigma \vdash \mathit{exp}_2 \to \mathit{exp}'_2}{\sigma \vdash (\mathit{exp}_1 \ \texttt{->}\ \mathit{exp}_2) \to (\mathit{exp}'_1 \ \texttt{->}\ \mathit{exp}'_2)}$$

$$\frac{\sigma \overset{xck}{\vdash} (\mathit{exp} \ \texttt{->}\ \mathit{exp}') : \texttt{true} \quad \sigma \vdash \mathit{exp}_2 \to \mathit{exp}'_2}{\sigma \vdash (\mathit{exp}_1 \ \texttt{->}\ \mathit{exp}_2) \to \mathit{exp}'_2}$$

Chapitre 4

Normalisation du programme source

La compilation débute par une transformation syntaxique dont le but essentiel est de substituer aux opérateurs temporels (`->`, `pre`, `when` et `current`) des constructions syntaxiques plus classiques. Cette transformation permet notamment d'uniformiser la notion de mémoire avec seulement deux opérateurs (`init` et `last`). L'intérêt et la correction de cette transformation sont fondés sur la sémantique opérationnelle, grâce à laquelle on établit l'équivalence entre le programme normalisé et le programme source.

4.1 La fonction *xck*

La plupart des règles de la sémantique opérationnelle sont complexes car elles font intervenir le prédicat $\overset{xck}{\vdash}$. Pour simplifier ces règles, on introduit une fonction *xck* qui associe à toute expression *exp* une expression booléenne telle que :

$$\sigma \vdash xck(exp) : b \quad ssi \quad \sigma \overset{xck}{\vdash} exp : b$$

Cette fonction est directement déduite des règles définissant $\overset{xck}{\vdash}$:

$$\begin{array}{ll} xck(exp) = \texttt{true} & si\ ck(exp) = \texttt{true} \\ xck(exp) = \texttt{if}\ xck(ck(exp))\ \texttt{then}\ ck(exp)\ \texttt{else false} & sinon \end{array}$$

4.2 La fonction *xflow*

Grâce à *xck*, on définit la fonction *xflow* :

$$xflow(exp) \quad = \quad \texttt{if}\ xck(exp)\ \texttt{then}\ exp\ \texttt{else bottom}$$

L'expression `bottom` est simplement une forme textuelle de la constante $\perp$:

$$\forall\sigma \quad \sigma \vdash \texttt{bottom} : \perp$$

L'expression *xflow*(*exp*) n'est pas une expression Lustre correcte, puisqu'elle ne vérifie pas les contraintes d'horloges. Mais du point de vue de l'évaluation, on peut établir l'équivalence suivante, où v désigne indifféremment une constante significative k ou la valeur $\perp$:

$$\sigma \vdash (id\texttt{=}exp) : \sigma[v/id] \quad ssi \quad \sigma \vdash xflow(exp) : v$$

Il en découle que l'on peut obtenir un programme sémantiquement équivalent du point de vue de l'évaluation, en remplaçant dans le programme initial chaque équation "id=exp" par "id=$xflow(exp)$". Avec cette transformation, le prédicat $\overset{xck}{\vdash}$ devient inutile, et la règle d'évaluation peut être simplifiée en conséquence :

$$\frac{\sigma \vdash exp : v}{\sigma \vdash id = exp : \sigma[v/id]}$$

Cette transformation n'est cependant pas correcte du point de vue de la réécriture. En effet, pour la réécriture, le prédicat $\overset{xck}{\vdash}$, et donc la fonction ck, sont nécessaires. Or, la fonction ck n'a pas de sens pour un programme qui ne respecte pas les contraintes d'horloges. Il faut donc définir une transformation globale, qui permette la réécriture.

4.3 La fonction *ximpl*

Les définitions de xck et $xflow$ font maintenant intervenir une fonction $ximpl$. La définition de $xflow$ est légèrement compliquée pour éviter la construction d'expressions conditionnelles triviales du type "`if true then` $ximpl(exp)$ `else bottom`" :

$$\begin{array}{rcll} xck(exp) & = & \texttt{true} & si\ ck(exp) = \texttt{true} \\ xck(exp) & = & \texttt{if } xck(ck(exp)) \texttt{ then } ximpl(ck(exp)) \texttt{ else false} & sinon \\ \\ xflow(exp) & = & ximpl(exp) & si\ ck(exp) = \texttt{true} \\ xflow(exp) & = & \texttt{if } xck(exp) \texttt{ then } ximpl(exp) \texttt{ else bottom} & sinon \end{array}$$

L'expression $ximpl(exp)$, appelée *implémentation* de exp, doit vérifier les propriétés suivantes :

- Son évaluation et sa réécriture ne doivent faire aucune référence à la fonction ck.
- Son évaluation doit donner le même résultat que celle de exp :

  $$\forall\sigma\ (\sigma \vdash exp : k) \Rightarrow (\sigma \vdash ximpl(exp) : k)$$

- Après un nombre quelconque de cycles, les réécritures respectives de exp et de $ximpl(exp)$ doivent encore donner le même résultat lors de l'évaluation. Pour que cette propriété soit vérifiée, il suffit que :

  $$\forall\sigma\ (\sigma \vdash exp \to exp') \Rightarrow (\sigma \vdash ximpl(exp) \to ximpl(exp'))$$

4.3.1 Implémentation des opérateurs simples

La définition de $ximpl$ pour les opérateurs simples ne pose pas de problème. L'opérateur `when`, qui est systématiquement réécrit bien qu'il n'intervienne pas dans l'évaluation, est tout simplement éliminé :

$$\begin{array}{l} ximpl(k) = k \\ ximpl(id) = id \\ ximpl(dataop(exp_1, \cdots, exp_n)) = dataop(ximpl(exp_1), \cdots, ximpl(exp_n)) \\ ximpl(exp \texttt{ when } exp') = ximpl(exp) \end{array}$$

4.3.2 Implémentation des opérateurs de mémoire

Pour définir l'implémentation de l'opérateur `pre`, on introduit un nouvel opérateur noté `last`. Comme le `pre` et le `current`, il a deux opérandes dont le premier est une constante différente de $\perp$. Son deuxième argument est une expression qui peut être évaluée à $\perp$. Intuitivement, la constante associée à un `last` est la dernière valeur significative (différente de $\perp$) prise dans le passé par son deuxième argument. La sémantique opérationnelle du `last` est donnée par :

Evaluation : $\sigma \vdash \texttt{last(}k\texttt{,}exp\texttt{)} : k$

Réécriture : $\dfrac{\sigma \vdash exp : \perp \qquad \sigma \vdash exp \rightarrow exp'}{\sigma \vdash \texttt{last(}k\texttt{,}exp\texttt{)} \rightarrow \texttt{last(}k\texttt{,}exp'\texttt{)}}$

$$\frac{\sigma \vdash exp : k' \qquad k' \neq \perp \qquad \sigma \vdash exp \rightarrow exp'}{\sigma \vdash \texttt{last(}k\texttt{,}exp\texttt{)} \rightarrow \texttt{last(}k'\texttt{,}exp'\texttt{)}}$$

L'implémentation du `pre` est alors définie par :

$$ximpl(\texttt{pre(}k\texttt{,}exp\texttt{)}) \;=\; \texttt{last(}k\texttt{,}xflow(exp)\texttt{)}$$

Cet opérateur va nous permettre d'uniformiser la notion de mémoire :

$$ximpl(\texttt{current(}k\texttt{,}exp\texttt{)}) \;=\; \texttt{if } xck(exp) \texttt{ then } ximpl(exp) \texttt{ else last(}k\texttt{,}xflow(exp)\texttt{)}$$

4.3.3 Implémentation de l'initialisation

Pour définir l'implémentation de l'opérateur `->`, on introduit un opérateur binaire noté `init`. Le premier argument est une constante booléenne, le deuxième une expression booléenne. Sa sémantique opérationnelle est définie par :

Evaluation : $\sigma \vdash \texttt{init(}b\texttt{,}exp\texttt{)} : b$

Réécriture : $\dfrac{\sigma \vdash exp \rightarrow exp'}{\sigma \vdash \texttt{init(false,}exp\texttt{)} \rightarrow \texttt{init(false,}exp'\texttt{)}}$

$$\frac{\sigma \vdash exp : \texttt{false} \qquad \sigma \vdash exp \rightarrow exp'}{\sigma \vdash \texttt{init(true,}exp\texttt{)} \rightarrow \texttt{init(true,}exp'\texttt{)}}$$

$$\frac{\sigma \vdash exp : \texttt{true} \qquad \sigma \vdash exp \rightarrow exp'}{\sigma \vdash \texttt{init(true,}exp\texttt{)} \rightarrow \texttt{init(false,}exp'\texttt{)}}$$

L'implémentation de l'initialisation est alors définie par :

$$ximpl(exp \texttt{ -> } exp') \;=\; \texttt{if init(true,}xck(exp)\texttt{) then } ximpl(exp) \texttt{ else } ximpl(exp')$$

L'opérateur `init`, comme l'opérateur `last`, permet de propager une valeur d'un cycle sur l'autre. Il s'agit donc d'un nouvel opérateur “à mémoire”. En fait, on pourrait se contenter du seul opérateur `last` :

$$ximpl(exp \texttt{ -> } exp') \;=\; \texttt{if last(true,not } xck(exp)\texttt{) then } ximpl(exp) \texttt{ else } ximpl(exp')$$

L'intérêt de ce nouvel opérateur n'apparaît clairement que dans la génération de code. Il est en effet amené à jouer un rôle important dans la structure de contrôle du code produit (chapitre 9).

4.4 Normalisation du programme

La normalisation du programme source est réalisée par la fonction *norm* :

$$\begin{aligned} norm(ins\ outs\ eqs) &= ins\ outs\ norm(eqs) \\ norm(eq\ eqs) &= norm(eq)\ norm(eqs) \\ norm(id\ \texttt{=}\ exp) &= id\ \texttt{=}\ xflow(exp) \end{aligned}$$

L'équivalence sémantique entre le programme normalisé et le programme initial s'exprime de la manière suivante :

$$\forall \nu_i, prg, \nu_o \quad (\nu_i \vdash prg : \nu_o) \;\Leftrightarrow\; (\nu_i \vdash norm(prg) : \nu_o)$$

4.5 Exemple

Nous illustrons la normalisation sur l'exemple suivant :

```
node lapp(onoff : bool) returns (external : int);
var
  internal : int;
  lapping : bool;
let
  external = current(internal when not lapping);
  lapping = false -> if onoff then not pre lapping else pre lapping;
  internal = 0 -> pre internal + 1;
tel.
```

Pour simplifier, le premier argument des opérateurs `init` et `last` n'est pas représenté :

`init` *exp* $\Leftrightarrow$ `init(true,`*exp*`)`

`last` *exp* $\Leftrightarrow$ `last(nil,`*exp*`)`

L'entête du programme, que nous n'avons pas pris en compte dans la définition de *norm*, est recopié tel quel.

```
node lapp(onoff : bool) returns (external : int);
var
  internal : int;
  lapping : bool;
let
  external = if (not lapping) then internal
             else last(if not lapping then internal else bottom);
  lapping = if init(true) then false
            else (if onoff then not init lapping else last lapping);
  internal = if init(true) then 0 else last internal + 1;
tel.
```

Chapitre 5

Représentation interne du programme

5.1 Représentations graphiques

5.1.1 Arbres et graphes acycliques

La représentation graphique la plus évidente d'un programme est son arbre abstrait. Il s'agit d'un arbre reproduisant fidèlement la structure du programme par rapport à la syntaxe du langage. Dans la plupart des compilateurs, on préfère une représentation plus compacte, appelée *graphes orientés acycliques* (ou DAG de l'anglais *directed acyclic graph*), qui permet d'identifier les sous-expressions communes [ASU86]. L'arbre abstrait et le DAG de l'expression :

```
x = if init(true) then (a or b) else (a or b) and not last x
```

sont représentés sur la figure 5.1.

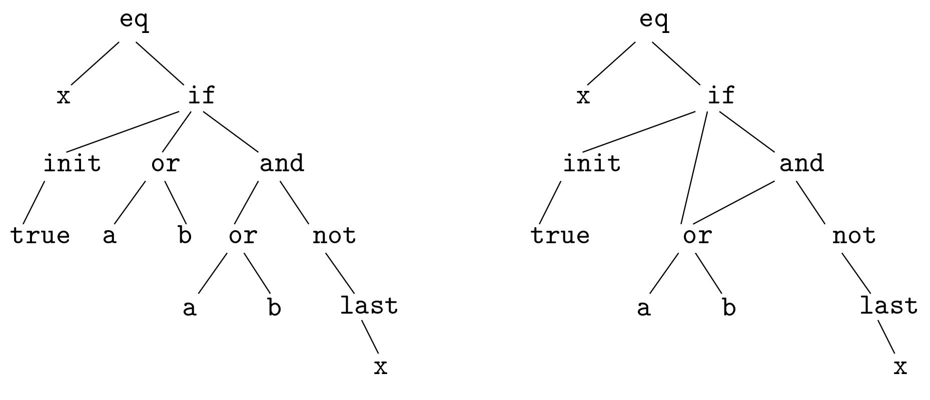


FIG. 5.1 – *Arbre abstrait et DAG*

Dans l'arbre abstrait, on a pris soin de noter le symbole de définition `eq`, pour éviter toute confusion avec l'opérateur de comparaison.

5.1.2 Réseau d'opérateurs

En LUSTRE, le principe de substitution nous suggère d'aller plus loin dans le partage des expressions. En effet, l'équation `x = e` signifie que toute référence à l'identificateur `x` peut être remplacée par l'expression `e`. Dans notre exemple, la référence à l'identificateur `x` est donc remplacée par une référence au graphe associé. On obtient ainsi un graphe orienté éventuellement cyclique, appelé *réseau d'opérateurs* (figure 5.2).

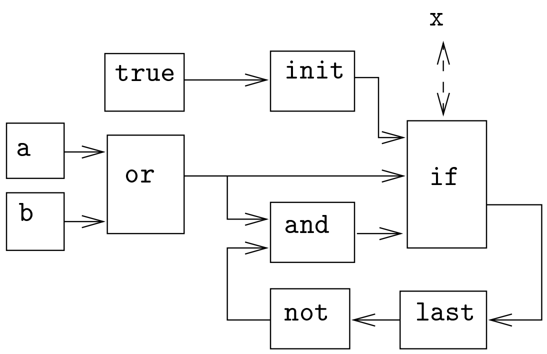


FIG. 5.2 – *Un réseau d'opérateurs*

5.1.3 Finesse du partage

Le programme est donc représenté comme un réseau d'opérateurs. Les opérateurs "zéro-aires" du réseau sont les constantes et les entrées du programme. A partir de ces "feuilles" on construit de proche en proche les autres opérateurs, en vérifiant qu'ils n'existent pas déjà. Cette construction s'apparente donc à celle d'un DAG. La construction de DAG est un problème tout à fait classique en compilation [ASU86]. Dans notre cas, le problème est plus complexe car le réseau d'opérateurs peut justement être cyclique. Pour que le partage des opérateurs soit optimal, il faudrait par exemple traiter le cas suivant :

```
X = if init(true) then 0 else last X + 1;
Y = if init(true) then 0 else last Y + 1;
```

Dans cet exemple, `X` et `Y` pourraient être associés au même réseau d'opérateurs (figure 5.3). Mais, si on veut prendre en compte de tels cas (assez rares), la construction du réseau devient très complexe.

Nous avons préféré implémenter une construction plus simple, de manière à réserver la puissance de calcul du compilateur à la génération de code proprement dite.

5.2 Construction du réseau

5.2.1 Construction des boîtes

Les nœuds du réseau sont appelés des *boîtes*. Une boîte est caractérisée par sa nature, son arité et une liste de pointeurs sur ses boîtes opérandes. Les boîtes d'arité zéro (*feuilles*) peuvent

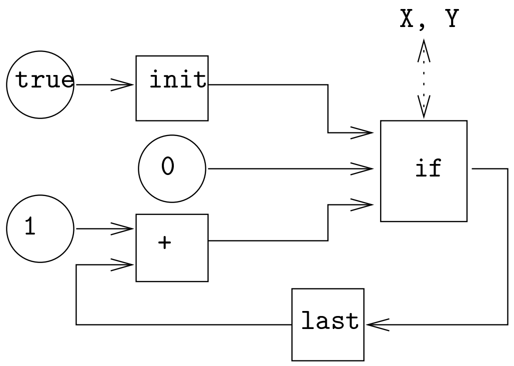


FIG. 5.3 –

être des entrées (nature `input`) auquel cas on leur associe l'identificateur correspondant, ou des constantes (nature `const`) auquel cas on leur associe la valeur correspondante :

box ::= *nature arity operands*
| `input` *id*
| `const` *k*

La construction d'une feuille est réalisée par la procédure *get-leaf*. La construction d'une boîte est réalisée par une procédure *get-box*(*nature*,*operands*), qui vérifie, avant de la construire, si une boîte syntaxiquement équivalente n'existe pas déjà. Cette équivalence prend en compte la commutativité des opérateurs prédéfinis. Par exemple, "$\textit{get-box}(\texttt{and}, b_2, b_1)$" donne le même résultat que "$\textit{get-box}(\texttt{and}, b_1, b_2)$".

Une boîte "muette" peut être construite par un appel de la procédure "*new-box*()", à condition de mettre à jour ultérieurement cette boîte avec un appel de la procédure "$\textit{update}(b_1, b_2)$". Cette procédure recopie dans la boîte b_1 les attributs (nature, arité, opérandes) de la boîte b_2.

5.2.2 Les attributs des identificateurs

Pour mettre en œuvre la construction, on associe à chaque identificateur les attributs suivants :

id ::= *def in-process linked box*

Si *id* est une sortie ou une variable locale, son champ *def* pointe sur l'expression associée dans le programme normalisé :

$$(id.def = exp) \quad \Leftrightarrow \quad \exists\ (id \texttt{ = } exp) \in eqs$$

Le booléen *in-process* est vrai si la boîte associée à *id* est en cours de construction. Le booléen *linked* est vrai si une boîte a déjà été associée à *id*, auquel cas un pointeur sur cette boîte se trouve dans le champ *box*.

5.2.3 La procédure *build-net*

La construction du réseau est réalisée par la procédure *build-net*. Elle commence par construire une boîte pour chaque entrée, puis pour chaque sortie :

build-net(*prg*) {

```
  pour tout id ∈ ins {
          id.box := get-leaf(input id)
          id.linked := true
  }
  pour tout id ∈ outs {
          si (¬ id.linked) put-in-net(id)
  }
}
```

5.2.4 La procédure *put-in-net*

La procédure *put-in-net* commence par marquer le champ *in-process*, de manière a détecter une éventuelle définition récurrente de *id*. Si au retour de la procédure *build-box*, une boîte est attachée à l'identificateur, c'est qu'il s'agit effectivement d'une définition récurrente : la boîte associée à *id* est en fait une boîte muette (voir définition de *build-box*) qu'il faut mettre à jour avec la procédure *update* :

```
put-in-net(id) {
  id.in-process := true
  b := build-box(id.def)
  si (id.linked) alors update(id.box,b)
  sinon {
      id.box := b
      id.linked := true
  }
}
```

5.2.5 La procédure *build-box*

Le seul cas complexe de la procédure *build-box* est celui des identificateurs. En effet, si l'identificateur est en cours de traitement, on est obligé de construire une boîte muette pour stopper la descente récursive. Cette boîte sera mise à jour lors du retour à l'appel de *put-in-net* correspondant :

```
build-box(exp) {
  choix {
      si (exp = k) alors retourner get-leaf(const k)
      si (exp= op(exp1, ···, expn)) alors
          retourner get-box(op,build-net(exp1),···,build-net(expn))
      si (exp = id) alors {
          si (id.linked) alors retourner id.box
          sinon si (id.in-process) alors {
                  id.linked := true
                  id.box := new-box()
                  retourner id.box
          }
```

sinon {
put-in-net(*id*)
retourner id.box
}
}
}
}

5.3 Exemple

La construction est donc insuffisante pour reconnaître l'équivalence des boîtes de `X` et `Y` dans l'exemple 5.1.3. Les boîtes de `init(true)`, `0` et `1` sont partagées, mais les opérateurs qui se trouvent sur le cycle sont dupliqués (figure 5.4).

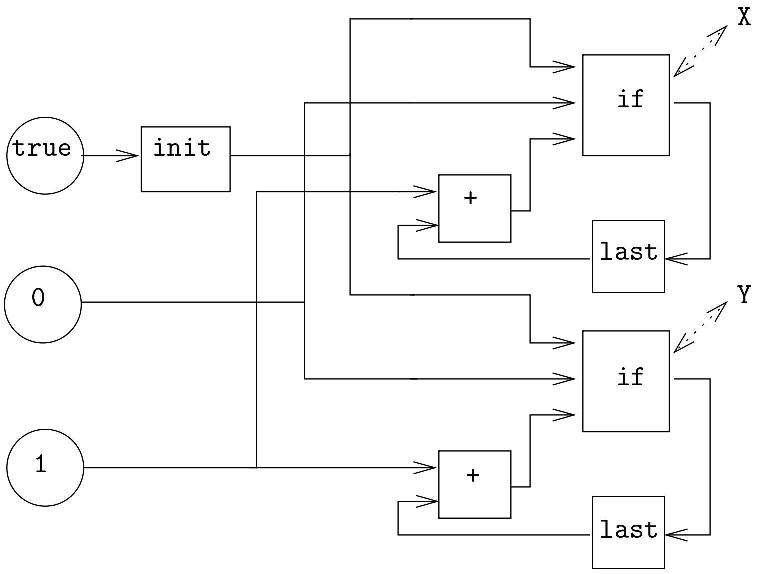


Fig. 5.4 – *Duplication d'un cycle*

Ce cas est cependant extrêmement rare. On obtient généralement un partage "optimal" des sous-expressions communes, même dans le cas de définitions récurrentes. Si par exemple les variables `X` et `Y` sont définies par les équations :

```
X = if init(true) then 0 else last X + 1;
Y = if init(true) then 0 else last X + 1;
```

on obtient le réseau minimal de la figure 5.3.

Chapitre 6

Exemples de compilation

Le but de ce chapitre est de présenter de manière intuitive les principes de la compilation de LUSTRE. La normalisation a déjà permis de mettre en évidence l'aspect essentiellement statique du programme : la réécriture n'affecte pas la structure du programme, mais seulement les valeurs associées aux opérateurs de mémoire. A partir du programme normalisé, on peut construire assez simplement un programme séquentiel. Pour cela, il faut ordonner les équations et associer aux opérateurs de mémoire, non plus une constante mais une variable (au sens case mémoire). On obtient ainsi un programme séquentiel qui fonctionne en *boucle simple*.

Des optimisations peuvent être envisagées pour améliorer ce programme : on peut notamment éliminer de nombreuses variables inutiles. Mais l'inefficacité du code est surtout due à l'absence de structure de contrôle. On peut y remédier en introduisant des instructions conditionnelles, mais surtout en générant un automate de contrôle.

6.1 Fonctionnement en boucle simple

6.1.1 Evaluation

Un des problèmes mis en évidence par la sémantique opérationnelle est l'ordre des équations. Pour que l'évaluation puisse être faite en séquence, il faut ordonner les équations de manière à ce que la définition d'une variable précède son utilisation. Le programme considéré étant sans blocage, il est toujours possible de déterminer statiquement un tel ordre. Dans l'exemple du nœud `lapp` (§4.5), on doit évaluer `external` après `lapping` et `internal`.

Tous les opérateurs classiques s'évaluent de la même manière que des opérateurs impératifs. Les références aux opérateurs `init` et `last` peuvent être remplacées par des références à des variables. Dans notre exemple, on est amené à introduire quatre nouvelles variables (`M0, M1, M2, M3`), qui correspondent respectivement aux expressions :

```
init(true)
last lapping
last internal
last(if not lapping then internal else bottom)
```

La première doit être initialisée à `true`, les autres n'étant, par définition, pas initialisées.

En pseudo-langage impératif, la phase d'évaluation peut se traduire par la séquence d'actions :

```
lapping := if M0 then false else (if onoff then not M1 else M1);
internal := if M0 then 0 else M2 + 1;
external := if (not lapping) then internal else M3;
```

6.1.2 Mémorisation

La phase de réécriture est implémentée par une phase de mémorisation dont le but est de stocker dans les variables supplémentaires (`M0, M1, M2, M3`) les valeurs qui seront nécessaires au cycle suivant.

D'après la sémantique opérationnelle, l'expression `init(` b `, true )` se réécrit quelle que soit la valeur b en `init( false , true )`. La mémorisation de `M0` est donc réalisée par l'affectation :

```
M0 := false
```

La mémorisation de `M1` et `M2` est triviale :

```
M1 := lapping ;
M2 := internal
```

Celle de `M3` est plus complexe. La constante `bottom` signifie en effet que la valeur associée au `last` ne doit pas évoluer. On a donc :

```
M3 := if not lapping then internal else M3
```

6.1.3 Un programme séquentiel simple

Un cycle de calcul est réalisé par l'appel de la procédure `lapp`. Le programme généré est donc esclave de l'environnement : c'est la séquence des appels qui définit l'horloge de base. Le contenu des variables `M0, M1, M2, M3` doit être conservé d'un appel sur l'autre. Ces variables doivent donc être définies en dehors de la procédure. Par contre, les variables locales `lapping` et `internal` pourraient être définies dans la procédure elle-même. Cette distinction entre mémoire rémanente et mémoire d'évaluation n'est cependant pas faite sur l'exemple. Nous avons pris comme convention que le programme et son environnement communiquent via la mémoire globale.

```
var
   boolean onoff;
   integer external;
   boolean lapping;
   integer internal;
   boolean M0 := true;
   boolean M1;
   integer M2, M3;

procedure lapp;
begin
   lapping := if M0 then false else (if onoff then not M1 else M1);
   internal := if M0 then 0 else M2 + 1;
```

```
   external = if (not lapping) then internal else M3;
   M0 := false;
   M1 := lapping;
   M2 := internal;
   M3 := if (not lapping) then internal else M3;
end
```

6.2 Optimisations

6.2.1 Les variables inutiles

La démarche que nous avons suivie pour le traitement des opérateurs de mémoire est systématique : une variable supplémentaire est introduite pour chaque opérateur de mémoire. En fait, ces variables supplémentaires sont souvent inutiles.

Prenons le cas de `M1` : à l'entrée de la procédure, `M1` et `lapping` contiennent la même valeur. L'évaluation de `lapping` peut donc être remplacée par :

```
   lapping := if M0 then false else (if onoff then not lapping else lapping);
```

Or, `M1` n'était utilisé que dans cette évaluation ; on peut donc éliminer cette variable.

Pour `M2` qui n'intervient que dans l'évaluation de `internal`, on peut suivre un raisonnement similaire. Enfin, `M3` évolue exactement de la même manière que la sortie `external`. Le programme, débarrassé de ces variables inutiles, devient :

```
var
   boolean onoff;
   integer external;
   boolean lapping;
   integer internal;
   boolean M0 := true;
procedure lapp;
begin
   lapping := if M0 then false else (if onoff then not lapping else lapping);
   internal := if M0 then 0 else internal + 1;
   external := if (not lapping) then internal else external;
   M0 := false;
end
```

L'analyse des dépendances peut donc nous permettre d'optimiser le nombre de variables.

6.2.2 Tests de contrôle

La variable `M0` est testée deux fois : lors de l'évaluation de `lapping` et de `internal`. On peut optimiser le code en intégrant ces deux affectations sous un test de contrôle. Bien que `external` ne dépende pas directement de `M0`, on a intérêt à intégrer son calcul au test de contrôle. En effet, dans le cas "`M0` = `true`", on peut établir statiquement que "`not lapping` = `true`", et donc que "`external` = `internal`". Enfin l'évolution de `M0` peut être remplacée par "`M0 := if M0 then false else M0`".

On peut donc aussi intégrer son calcul au test de contrôle, avec la convention que l'affectation triviale “`M0 := M0`” ne doit pas être générée :

```
if M0 then
   lapping := false;
   internal := 0;
   external := internal;
   M0 := false
else
   lapping := if onoff then not lapping else lapping;
   internal := internal + 1;
   external := if (not lapping) then internal else external
```

Même sans factorisation, les tests de contrôle permettent d'éviter des affectations triviales. L'affectation :

```
external := if (not lapping) then internal else external;
```

peut être remplacée par :

```
if (not lapping) then external := internal
```

Même chose pour `lapping` :

```
if onoff then lapping := not lapping
```

6.2.3 Génération d'un automate

Même avec la factorisation, le code reste inefficace : la variable `M0` est testée à chaque cycle. Or, on sait que cette variable ne sert qu'à distinguer dynamiquement le premier cycle de tous les autres. Cette distinction peut être faite statiquement en structurant le programme comme un automate. Les deux états de cet automate correspondent aux formes que peut prendre le programme suivant la valeur de `M0`.

```
var
   boolean onoff;
   integer external;
   boolean lapping;
   integer internal;

procedure lapp;
select state
S0 :
   lapping := false;
   internal := 0;
   external = internal;
   state := S1;
S1 :
   if onoff then lapping := not lapping;
   internal := internal + 1;
   if (not lapping) then external := internal;
```

```
   state := S1;
end
```

La mémoire `M0` a été traitée comme une variable d'état. C'est-à-dire qu'on a distingué statiquement les états du programme où cette variable vaut `true` de ceux où elle vaut `false`. Toutes les variables dont le domaine de valeurs est fini pourraient être traitées de la même manière : si une variable ne peut prendre qu'un nombre fini de valeurs $(v_1, v_2, \cdots, v_k)$, on peut distinguer statiquement les k états du programme où cette variable vaut respectivement $v_1, \ldots, v_k$. De plus, il faut que tous les opérateurs sur le domaine considéré soient connus par le compilateur, de manière à effectuer dès la compilation les calculs qui portent sur des constantes.

En fait, le seul domaine fini que "connaît" le compilateur est celui des booléens. Toutes les variables d'état seront donc choisies parmi les mémoires booléennes du programme. Dans notre exemple, la mémoire `lapping` peut être traitée comme une variable d'état. On obtient alors un automate à trois états : l'état `S0` est toujours l'état initial, les états `S1` et `S2` correspondent aux états non initiaux où `last lapping` vaut respectivement `true` et `false`.

```
procedure lapp;
select state
S0 :
   internal := 0;
   external := internal;
   state := S2;
S1 :
   internal := internal + 1;
   if onoff then
      external := internal;
      state := S2;
   else
      state := S1;
S2 :
   internal := internal + 1;
   if onoff then
      state := S1;
   else
      external := internal
      state := S2;
end
```

Ce nouvel automate est plus rapide, mais il est aussi plus "gros". En fait, la taille d'un automate résultant du choix de n variables d'état peut être de 2^n états et $2^n \times 2^n$ transitions. La taille du code généré risque donc d'être exponentiellement plus grande que celle du programme source.

La recherche de "bonnes" variables d'état, qui permettent d'optimiser la vitesse d'exécution avec des automates de taille "raisonnable" est donc un problème crucial de la compilation.

Troisième partie

Génération de code

Chapitre 7

Le format OC

Le compilateur produit un programme impératif exprimé dans un langage appelé OC (pour *object code*). Le langage OC a été défini pour être un format commun aux compilateurs LUSTRE et ESTEREL [PS87]. Le langage OC est avant tout un format intermédiaire. Il peut être traduit automatiquement en des langages impératifs classiques comme C ou ADA, par des outils développés dans l'équipe ESTEREL (occ, ocada). Le minimiseur d'automate ALDEBARAN a été interfacé avec OC ; on dispose donc d'un outil de minimisation des programme OC (ocmin).

Destiné a être produit et lu par des outils informatiques, un programme OC est difficilement lisible pour l'utilisateur. De plus certaines constructions du langage sont plus particulières aux problèmes posés par le langage ESTEREL. Notre but dans cette section n'est donc pas de présenter complètement le langage OC, mais uniquement le sous-langage utilisé pour la traduction des programmes LUSTRE.

7.1 Structure générale d'un programme OC

Les différents objets auxquels fait référence le programme sont regroupés par classes, comme par exemple les types, les variables ou les actions. Une classe est en fait une table qui permet d'associer à chaque objet un index qui lui tient lieu d'identificateur dans le programme. Dans ce chapitre, nous séparons de manière abstraite les tables du programme qui concernent la déclaration d'objets statiques (l'entête), de celles qui concernent la définition du code séquentiel (l'automate).

7.2 L'entête

La partie déclaration est décomposée en plusieurs tables : types, constantes, fonctions, procédures, signaux et variables. Les objets prédéfinis sont repérés par un index, précédé du caractère `$`. L'index des objets prédéfinis fait référence à une table spéciale [PS87].

7.2.1 Les types externes

Les types booléen, entier et réel sont prédéfinis en OC (notés respectivement `_boolean`, `_integer` et `_float`). Tout autre type nécessaire à une application doit être déclaré comme un type externe. Pour chaque type externe, le format OC impose la déclaration de quatre primitives de base :

<type-decl> ::= *name assign-proc-index eq-fun-index ne-fun-index cond-fun-index*

La procédure repérée par l'index *assign-proc-index* est supposée réaliser l'affectation d'une valeur de type *name* à une variable de type *name*. Les fonctions booléennes *eq-fun-index* et *ne-fun-index* permettent de tester l'égalité et la différence de deux valeurs de type *name*. Enfin, la fonction *cond-fun-index* réalise le choix déterministe entre deux valeurs de type *name* (i.e. le *si . . . alors . . . sinon . . .*).

7.2.2 Les constantes externes

La déclaration d'une constante est tout à fait classique : elle associe à un identificateur un index de type (prédéfini ou externe).

<const-decl> ::= *name type-index*

7.2.3 Les fonctions externes

Dans une expression OC, les opérations, même prédéfinies, sont toujours en notation préfixée. Par exemple, l'expression `x + 1` est notée `_plus__integer(x,1)`, où `_plus__integer` est la fonction prédéfinie réalisant l'addition entière.

Toutes les opérations arithmétiques et logiques sont des fonctions prédéfinies en OC [PS87]. Les autres fonctions doivent être importées, et déclarées de la manière suivante :

<func-decl> ::= *name* (*type-index*, $\cdots$, *type-index*) : *type-index*

Nous avons vu (§7.2.1), que la déclaration d'un type externe implique la déclaration de trois fonctions. Pour un type externe d'index k déclaré avec le nom `TYPE_IDENT`, les déclarations de ces trois fonctions doivent être de la forme (`$0` est l'index prédéfini du type booléen) :

`_eq_TYPE_IDENT(`k`,`k`) : $0`

`_ne_TYPE_IDENT(`k`,`k`) : $0`

`_cond_TYPE_IDENT($0,`k`,`k`) :` k

7.2.4 Les procédures

Les seules procédures prédéfinies sont les affectations booléennes, entières et réelles. Une déclaration de procédure externe est de la forme :

<proc-decl> ::= *name* (*type-index*, $\cdots$, *type-index*)(*type-index*, $\cdots$, *type-index*)

La première liste définit les types des arguments passés par référence, la deuxième, ceux des arguments passés par valeur.

Pour chaque type externe d'index k (§7.2.1), la procédure d'affectation est déclarée de la manière suivante :

`_assign_TYPE_IDENT(`k`)(`k`)`

7.2.5 Les signaux

Dans le format OC, les entrées et les sorties sont des *signaux*, c'est-à-dire des entités dont la caractéristique principale est d'être absentes ou présentes. Un signal peut accessoirement "porter" une valeur. Dans ce cas, on peut accéder à cette valeur dans une mémoire associée au signal. Les différentes formes de signaux sont les suivantes :

<signal>	::=	`input :`*name present-action-index* `pure :`
	|	`input :`*name present-action-index* `single :` *var-index*
	|	`output :`*name out-action-index* `pure :`
	|	`output :`*name out-action-index* `single :` *var-index*

Pour les entrées, "*present-action*" est le test qui permet de savoir si le signal est présent. Pour les sorties, "*out-action*" permet de signifier à l'environnement que la sortie correspondante est présente. Enfin, `single:` signifie que le signal est valué, et que la valeur correspondante est disponible dans la variable *var-index* quand le signal est présent.

7.2.6 Les variables

Ce sont des variables classiques (au sens impératif). Elles ne sont pas nommées : c'est leur rang dans la table qui leur tient lieu d'identificateur. Une valeur initiale est éventuellement associée à la variable, sous la forme d'une expression ne contenant que des constantes (cf. définition des expressions) :

<var-decl>	::=	*type-index*
	|	*type-index* `value :` *<initial-value>*

7.3 L'automate

Cette partie regroupe les tables qui concernent la définition du code séquentiel. Dans la syntaxe concrète du langage, cette partie est divisée en :

une table d'actions où sont rangées toutes les opérations impératives de base effectuées par le programme.

une table d'états où sont rangées les séquences d'actions associées aux états du programme.

Cette méthode permet d'associer à chaque action de base un index unique, pouvant être utilisé plusieurs fois dans le code séquentiel proprement dit. Le code résultant est donc particulièrement compact, mais peu lisible. Nous présentons une forme abstraite de cette partie opérative, où les actions sont expansées dans le code séquentiel.

7.3.1 Les expressions

Il s'agit d'expressions algébriques écrites dans un style fonctionnel préfixé :

<expression> ::= `#`*atome*
| `@`*const-index*
| *var-index*
| *fonction-index*(*<expression>*, · · · , *<expression>*)

Les atomes sont des constantes arithmétiques entières ou réelles (`1, 3.14, 2E-5` ...) précédées du symbole `#`. Une constante est repérée par son index précédé du symbole `@`. Deux constantes sont prédéfinies : les booléens `_true` et `_false` (respectivement `@$1` et `@$0`), les autres sont définies dans la table des constantes. Les instances de variable sont tout à fait classiques. Enfin, tous les opérateurs arithmétiques et logiques classiques sont implémentés par des fonctions prédéfinies.

7.3.2 Les actions

Nous présentons une syntaxe abstraite des actions OC, qui permet de différencier les simples actions séquentielles, de celles qui provoquent un branchement dans le code. D'où les notions d'"action" et de "test". Dans la syntaxe concrète de OC, la notion d'affectation n'est pas explicite : il s'agit d'un cas particulier d'appel de procédure dont les arguments sont une variable et une expression de même type. Dans notre compilateur, l'affectation joue un rôle tellement important que nous préférons la distinguer des autres appels de procédure. Enfin, dans notre syntaxe abstraite, le changement d'état de l'automate n'est pas considéré comme une action, mais comme une composante des transitions.

<action> ::= *var-index* := *<expression>*
| *proc-index*(*var-index*, · · · , *var-index*)(*<expression>*, · · · , *<expression>*)
| `output` :*signal-index*

<test-action> ::= `present` :*signal-index*
| `if` : *<expression>*

7.3.3 Les transitions

Nous donnons une syntaxe abstraite du code séquentiel qui différencie les parties éventuellement vides ne contenant que des actions de base ("instructions"), des transitions proprement dites qui se terminent toujours par un changement d'état.

<instruction> ::= ε
| *<action><instruction>*
| *<test-action>* (*<instruction>*)(*<instruction>*)

<transition> ::= `goto` :*state-index*
| *<instruction><transition>*
| *<test-action>* (*<transition>*)(*<transition>*)

7.3.4 L'automate

Cette partie est une séquence de définitions d'états de la forme :

<etat> ::= `state` :*state-index* *<transition>*

Par convention, l'état d'index 0 est l'état initial du programme.

7.4 Exemple

Nous donnons dans ce paragraphe le programme correspondant à l'automate à trois états de la section 6.2.3. Les déclarations en algorithmique classique sont données en commentaires (précédés de `%`). Toutes les tables inutiles ont été omises ; ce programme OC n'est donc pas "tout à fait" correct :

```
signals: 2
   0 output:external 1 single:0
   1 input:onoff 2 single:1
variables: 3
   0 $1       % external : integer
   1 $0       % onoff : boolean
   2 $1       % internal : integer
actions: 7
   0 goto:
   1 output: 0
   2 present: 1
   3 call:$1 (0) (2)         % external := internal
   4 call:$1 (2) (#0)        % internal := 0
   5 call:$1 (2) ($13(2,#1)) % internal := internal + 1
   6 if: 1                   % if: onoff
automaton:
state: 0
    4 3 1 0 2
state: 1
    5 6 ( 3 1 0 2)( 1 0 1)
state: 2
    5 6 ( 1 0 1)( 3 1 0 2)
end:
```

Chapitre 8

Génération de l'interface

Ce chapitre présente la génération de l'interface du programme objet, c'est-à-dire la partie où sont déclarés les objets externes, ainsi que les entrées et les sorties du programme. La plupart des étapes de cette traduction sont triviales, cependant certaines alternatives sont possibles, qui déterminent les conventions de "bon interfaçage" entre le programme produit et son environnement. C'est le cas par exemple des fonctions externes, et des entrées/sorties booléennes.

8.1 Les types et les constantes

Les déclarations de types et de constantes externes du programme source sont traduites de manière triviale dans les tables correspondantes du programme OC. Les primitives associées à chaque type sont elles-aussi déclarées, avec les conventions définies par le format OC (chapitre 7).

8.2 Les fonctions

En Lustre, tous les programmes importés sont déclarés et utilisés comme des fonctions pouvant rendre éventuellement plusieurs résultats. On peut par exemple déclarer :

```
function LOG_NEPERIEN(x : real) returns (y : real);
function DIVISION_ENT(x : int; y : int) returns (q : int; r : int);
```

Or, la notion de fonction rendant plusieurs résultats n'existe pas dans le langage OC. Il convient donc de définir de manière précise la façon dont le compilateur va traiter une définition de fonction externe.

- Une fonction ne rendant qu'un résultat ne pose pas de problème. Par exemple, la déclaration de `LOG_NEPERIEN` est traduite par :

  ```
  LOG_NEPERIEN(_float) : _float
  ```

  Et, de manière tout à fait naturelle, les appels de cette fonction sont des expressions dans le code objet, apparaissant en partie droite d'affectation.

– Pour ce qui est des fonctions à plusieurs résultats, le problème est plus complexe. La solution retenue dans le compilateur Lustre-V2 [PH87], consiste à se ramener au cas à une sortie, en structurant les résultats complexes. La notion de structure n'existant pas dans le langage OC, il faut donc définir le type du résultat de manière abstraite. Par exemple, pour `DIVISION_ENT`, un type `DIVISION_ENT_result` est déclaré. Les primitives standard (comparaisons, choix et affectations) doivent être déclarées, ainsi que les fonctions permettant d'accéder à chaque composante de ce type :

```
DIVISION_ENT_result_q(DIVISION_ENT_result) : _integer
DIVISION_ENT_result_r(DIVISION_ENT_result) : _integer
DIVISION_ENT(_integer,_integer) : DIVISION_ENT_result
```

Cette méthode nous paraît lourde et peu efficace. La solution retenue est de traiter les fonctions à plusieurs résultats comme des procédures externes :

```
DIVISION_ENT(_integer,_integer)(_integer,_integer)
```

8.3 Les entrées et les sorties

8.3.1 Signaux valués

Une séquence de valeurs sur un domaine $\tau \cup \{\perp\}$ peut être vue comme l'histoire d'un signal valué de type τ : chaque occurrence de la valeur $\perp$ signifie que le signal est absent dans l'instant correspondant, l'occurrence d'une valeur $v \in \tau$ signifie qu'il est présent avec la valeur v. Il est donc toujours possible d'implémenter les entrées et les sorties du programme avec des signaux valués. L'émission des sorties est nécessaire : pour l'environnement, qui n'est pas supposé connaître le programme Lustre et sa hiérarchie d'horloges, il s'agit en effet d'une propriété dynamique. Par contre, la présence d'une entrée est, pour le compilateur, une propriété tout à fait statique : le test correspondant ne sera donc jamais utilisé dans le code produit.

8.3.2 Signaux purs

L'implémentation des entrées/sorties booléennes peut être réalisée par des signaux valués, mais aussi par des signaux purs : en effet, une suite de valeurs sur $\{\texttt{true}, \texttt{false}, \perp\}$ peut être vue comme l'histoire d'un signal pur, présent à chaque occurrence de la valeur `true` dans la séquence.

Il est très intéressant de générer des programmes OC avec signaux purs. En effet, ce type de programme peut être traité automatiquement par des outils existant :

Visualisation des automates : La structure générale de l'automate est peu visible dans un programme OC. En particulier, la structure "branchue" des transitions empèche de comprendre facilement sous quelles conditions on passe d'un état à un autre. L'outil Autograph [Roy90] permet de visualiser et manipuler un automate, en ne retenant que les informations liées à la présence et à l'émission des signaux : une transition est étiquetée par une conjonction de présences (qui joue le rôle de condition) et par la séquence d'émissions qui en résulte. Si une entrée booléenne est implémentée par un signal valué, sa présence sur une transition ne signifie pas grand chose (le signal peut être présent avec la valeur *vrai* ou la valeur *faux*). Si par contre elle est implémentée par un signal pur, sa présence devient significative : elle est présente avec la valeur *vrai*. Un raisonnement dual peut être suivi pour les sorties booléennes : la présence du signal pur correspondant signifie que la sortie est *vrai*. Pour obtenir le plus d'informations possible des automates visualisés par AUTOGRAPH, il est donc nécessaire d'avoir des signaux purs.

Génération d'interfaces graphiques : Un programme OC est destiné à être traduit en une procédure C ou ADA ; chaque appel de cette procédure exécute une transition de l'automate OC. Pour simuler un comportement cyclique, il faut donc écrire un programme principal qui joue le rôle de "moteur" : indéfiniment, il saisit les entrées, appelle la procédure et traite ses sorties. La taille d'un tel moteur, surtout si on veut le doter d'un interface graphique agréable, est souvent bien plus importante que celle de la composante réactive elle-même. L'outil SAHARA [Ghe91] a été développé pour obtenir facilement des programmes principaux : l'utilisateur décrit un "tableau de bord" graphique dont les composants (boutons, lampes, cadrans ...) sont reliés aux signaux du programme OC. Cette description est alors compilée pour obtenir un programme principal qui appelle la procédure réactive (obtenue en compilant le programme OC) en fonction du tableau de bord. Dans la description du tableau de bord, le type d'un composant est lié à celui du signal correspondant. En particulier, les boutons-poussoir et les lampes ne peuvent être associés qu'à des signaux purs.

Du point de vue de la compilation, les signaux purs impliquent des contraintes spécifiques :

– une entrée pure ne peut intervenir que dans un test de présence

– une sortie pure se comporte comme une sortie valuée ne pouvant prendre que les valeurs `true` (auquel cas on génère son émission), `false` ou $\perp$ (aucune action n'est alors générée)

L'implémentation des entrées/sorties booléennes en signaux purs influence donc de manière profonde la structure et l'interface du programme objet. Nous avons choisi d'en faire une option de compilation.

Chapitre 9

Identification du contrôle

La structure de contrôle d'un programme OC est un automate d'états fini. Pour construire un tel automate, le compilateur s'appuie sur le comportement temporel des expressions booléennes du programme.

9.1 Exemple

Nous allons présenter la notion d'automate de contrôle sur un exemple très simple. Il s'agit d'un programme dont l'unique sortie booléenne `S` est définie en fonction de l'entrée booléenne `E` par l'équation suivante :

```
S = if init(true) then E else E and not last(S);
```

Du point de vue de la sémantique opérationnelle, l'état initial du programme, que nous appellerons q_0, correspond à une version de ce programme où la constante `true` est attachée à l'opérateur `init`, et la constante `nil` à l'opérateur `last` :

```
S = if init(true,true) then E else E and not last(nil,S);
```

On peut obtenir une équation équivalente du point de vue de l'évaluation en remplaçant les opérateurs de mémoire par la constante associée :

```
S = if true then E else E and not nil;
```

Après simplification, on en déduit que dans l'état initial q_0, la sortie porte la même valeur que l'entrée :

q_0 : `S = E`

D'après la sémantique opérationnelle, l'expression "`init(true,true)`" se réécrit en "`init(false,true)`". Par contre, la réécriture du `last` dépend du résultat de l'évaluation de `S`, c'est-à-dire de la valeur de `E`. On peut donc établir de manière statique que le programme ne peut se réécrire que de deux façons, correspondant aux états que nous appellerons respectivement q_1 et q_2 :

q_1 : `S = if init(false,true) then E else E and not last(true,S)`

q_2 : `S = if init(false,true) then E else E and not last(false,S)`

L'évaluation de `S` dans l'état q_1 est donnée par l'équation simplifiée : `S = false`. On en déduit que le programme, dans cet état, se réécrit indépendamment de la valeur de l'entrée sous la forme q_2. Par contre, dans l'état q_2, l'équation simplifiée est : `S = E`. Comme pour l'état q_0, les réécritures possibles sont, suivant la valeur de l'entrée, les états q_1 ou q_2.

On a donc obtenu, de manière statique, un automate fini (figure 9.1) appelé automate de contrôle du programme.

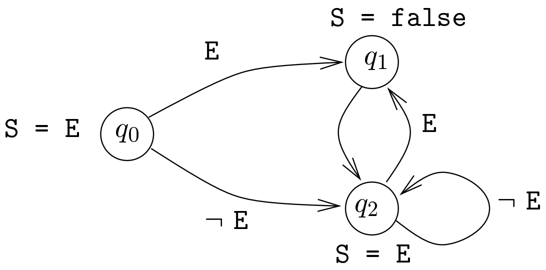


FIG. 9.1 – *Un automate de contrôle*

A chaque état de l'automate correspond une forme simplifiée du programme qui peut servir de base à la génération de code séquentiel. En implémentant les entrées/sorties par des signaux valués, on peut générer l'automate OC suivant (pour simplifier, `S` désigne indifféremment la variable ou le signal associés à la sortie) :

```
state: 0
   S:=E; emit: S; if: E (goto: 1)(goto: 2)
state: 1
   S:=false; emit: S; goto: 2
state: 2
   S:=E; emit: S; if: E (goto: 1)(goto: 2)
```

9.2 Les expressions booléennes

Dans le programme normalisé, les expressions booléennes se décomposent en :

Constante `bottom` : elle intervient dans la définition des sorties et des opérateurs de mémoire.

Mémoires booléennes : les expressions booléennes en `init` et `last` sont appelées *variables d'état*. Dans notre exemple, les variables d'état sont `init(true)` et `last(S)`.

Opérateurs évaluables : il s'agit de tous les opérateurs que le compilateur peut évaluer statiquement : les constantes booléennes, bien sûr, mais aussi tous les opérateurs strictement booléens (`not, or, and, if, #, =, <>`).

Expressions non évaluables : Ce sont toutes les expressions dont l'évaluation ne peut être que dynamique. Il s'agit des entrées booléennes, des appels de fonction externe à résultat booléen, et des comparaisons entre non booléens (`=, <>, >, <`, etc..). Ces expressions sont appelées les *conditions* du programme. Dans notre exemple, la seule condition est l'entrée `E`.

9.2.1 Les éléments de contrôle

Les variables d'état et les conditions sont appelées *éléments de contrôle* du programme. L'identification du contrôle commence par un parcours du réseau d'opérateurs destiné à marquer les boîtes qui correspondent aux éléments de contrôle. On identifie ainsi n variables d'état, notées $svar_i$ avec $i = 1..n$, et m conditions, notées $cond_j$, avec $j = 1..m$.

Dans notre exemple, on identifie deux variables d'état, $svar_1$ et $svar_2$ attachées respectivement aux expressions `init true` et `last S`. L'unique condition, $cond_1$, est attachée à l'entrée `E`.

9.2.2 Les expressions de contrôle

Après le marquage des éléments de contrôle, chaque expression booléenne du réseau peut être exprimée dans la syntaxe abstraite suivante :

$$
\begin{array}{rcl}
\textit{control-exp} & ::= & \texttt{bottom} \\
 & | & \texttt{true} \\
 & | & \texttt{false} \\
 & | & svar_i \\
 & | & cond_i \\
 & | & \texttt{not}\ \textit{control-exp} \\
 & | & \textit{control-exp}\ \texttt{or}\ \textit{control-exp} \\
 & | & \textit{control-exp}\ \texttt{and}\ \textit{control-exp} \\
 & | & \textit{control-exp}\ \texttt{=}\ \textit{control-exp} \\
 & | & \textit{control-exp}\ \texttt{<>}\ \textit{control-exp} \\
 & | & \texttt{if}\ \textit{control-exp}\ \texttt{then}\ \textit{control-exp}\ \texttt{else}\ \textit{control-exp} \\
 & | & \texttt{\#(}\textit{control-exp}, \cdots, \textit{control-exp}\texttt{)}
\end{array}
$$

Dans la suite, pour toute expression booléenne du réseau (exp), on désigne par $ctrl(exp)$ l'interprétation de exp selon la syntaxe des expressions de contrôle.

Une expression de contrôle qui ne contient aucune référence à la constante `bottom` dénote une fonction booléenne dont les arguments sont les éléments de contrôle. Dans notre exemple, l'expression attachée à `S` correspond à la fonction booléenne à trois arguments :

$$\lambda svar_1 svar_2 cond_1.\texttt{if}\ svar_1\ \texttt{then}\ cond_1\ \texttt{else}\ (cond_1 \texttt{and}\ \texttt{not} svar_2)$$

9.3 Evolution des variables d'état

Un état est une mémoire spéciale, notée q, qui permet d'associer à chaque variable d'état sa valeur courante. Dans la suite, on représente souvent un état sous la forme d'un n-uplet booléen :

$$q = (b_1, \cdots, b_n) \quad \Leftrightarrow \quad \forall i \in 1, \cdots, n \quad q(svar_i) = b_i$$

L'état initial q_0 est une mémoire prédéfinie qui associe aux variables d'état du type "`init`*exp*" la valeur `true`, et à celles du type "`last`*exp*" la valeur `nil`.

Nous nous intéressons maintenant à l'évolution des variables d'état, c'est-à-dire à la question : "si le cycle courant est caractérisé par l'état q, quelle sera la valeur associée à $svar_i$ dans le cycle suivant ?"

Cette valeur dépend bien sûr de l'état courant q, mais aussi de la valeur des conditions. Dans la suite, on note c la valeur courante des conditions. Comme pour q, c est indifféremment utilisé comme un m-uplet booléen ou une mémoire :

$c = (b_1, \cdots, b_m) \quad \Leftrightarrow \quad \forall j \in 1, \cdots, m \quad c(cond_j) = b_j$

9.3.1 Les mémoires `init`

Prenons l'exemple simple de la mémoire `init true`. On sait d'après la sémantique opérationnelle que l'expression `init(` b `, true )` se réécrit quel que soit b en `init( false , true )`. On en déduit que l'évolution de la variable d'état correspondante, $svar_i$, est donnée par la fonction booléenne :

$\delta_i \quad = \quad \lambda qc.\texttt{false}$

Dans le cas général, l'évolution d'une mémoire `init(` b `,` exp `)` dépend de sa valeur courante b, et de l'évaluation de exp. D'après la sémantique opérationnelle, on peut établir que si la valeur de la mémoire est `true`, et que exp est évaluée à v, sa valeur au cycle suivant est la négation logique de v. Par contre, si la valeur courante est déjà `false`, elle reste `false` au cycle suivant.

Soit $svar_i$ la variable d'état correspondante, b n'est autre que la valeur associée à cette variable dans l'état courant. De plus, on sait par construction que exp est une expression booléenne qui ne contient pas d'opérateurs `bottom`. Elle peut donc être interprétée comme une fonction qui ne dépend que de l'état courant et des conditions. L'évolution de la variable d'état est donnée par la fonction :

$\delta_i \quad = \quad \lambda qc.\texttt{if } svar_i \texttt{ then not } ctrl(exp) \texttt{ else false}$

9.3.2 Les mémoires `last`

On sait que, par construction (§4.3.2), les expressions en `last` ne peuvent avoir qu'une des deux formes suivantes :

`last` exp
`last if` exp' `then` exp `else bottom`

où les expressions exp et exp', construites respectivement par les fonctions xck et $ximpl$, ne peuvent pas être évaluées à $\bot$. L'évolution d'un `last` de la première forme est très simple, soit $svar_i$ la variable d'état associée :

$\delta_i \quad = \quad \lambda qc.ctrl(exp)$

L'évolution d'un `last` de la deuxième forme est implicitement récursive. En effet, si son opérande est évalué à $\bot$ (c'est-à-dire si son horloge apparente exp' est évaluée à `false`), sa propre valeur reste inchangée :

$\delta_i \quad = \quad \lambda qc.\texttt{if } ctrl(exp') \texttt{ then } ctrl(exp) \texttt{ else } svar_i$

9.3.3 Les fonctions de transition

L'évolution de chaque variable d'état peut donc être définie par une fonction booléenne dont les arguments sont les éléments de contrôle. Ces fonctions sont appelées les *fonctions de transition* et notées δ_i. On peut les construire facilement à partir des du réseau sous la forme d'expressions de contrôle (*control-exp*).

9.4 L'automate de contrôle

Le contrôle d'un programme à n variables d'état et m conditions est donc défini par la donnée de n fonctions booléennes à $n+m$ arguments $\delta_1, \delta_2, \ldots, \delta_n$. L'automate de contrôle de ce programme est défini comme le quadruplet $(Q, C, \rightarrow, q_0)$ où :

- $Q = B^n$ est l'ensemble des états, i.e. l'ensemble des vecteurs de n booléens.
- $C = B^m$ est l'ensemble des conditions, i.e. l'ensemble des vecteurs de m booléens.
- $\rightarrow \in Q \times C \times Q$ est la relation de transition définie par :

  $$q \xrightarrow{c} q' \text{ ssi } \forall i = 1..n \; q'_i = \delta_i(q, c)$$

  Cette relation définit une fonction de $Q \times C$ dans Q, ce qui reflète le déterminisme du langage.
- $q_0 \in Q$ est l'état initial, i.e. la mémoire booléenne initiale.

Ce système de transition n'est pas directement exploitable : le nombre de transitions (potentiellement 2^{n+m}) est en effet prohibitif. Il convient donc de "compacter" les transitions en définissant un nouveau système où les étiquettes ne sont plus des vecteurs de B^m, mais des fonctions booléennes sur B^m. Le nouvel automate est un quadruplet $(Q, T, \mapsto, q_0)$ où :

- $T = B^{B^m}$ est l'ensemble des fonctions booléennes à m arguments.
- $\mapsto \in Q \times T \times Q$ est la nouvelle relation de transition définie par :

  $$q \overset{t}{\mapsto} q' \text{ ssi } \forall c \in B^m (t(c) = 1) \Leftrightarrow (q \xrightarrow{c} q')$$

Enfin, on ne s'intéresse en général qu'à un sous-ensemble des états possibles : $Q_{acc} \subseteq Q$ est l'ensemble des états accessibles depuis l'état initial. Q_{acc} est défini comme le plus petit sous-ensemble de Q vérifiant :

$$q_0 \in Q_{acc}$$
$$(q \in Q_{acc}) \wedge (\exists c / q \xrightarrow{c} q') \Rightarrow q' \in Q_{acc}$$

9.5 La représentation des fonctions booléennes

Les fonctions d'évolution des variables d'état ne sont pour l'instant connues que sous la forme d'expressions algébriques (les expressions de contrôle). Or ce type de représentation se prête mal aux manipulations symboliques nécessaires à la génération des automates de contrôle. Nous allons donc utiliser une représentation interne basée sur les graphes de décision binaires (*binary decision diagrams* ou *bdds* [Ake78]). Ce type de représentation fait actuellement autorité dans le domaine de l'évaluation symbolique, et une abondante littérature lui est consacrée [Mor82, Bry86, Bil87]. Nous nous limitons dans cette section à la présentation de quelques caractéristiques importantes des *bdds*, ainsi que des problèmes particuliers à leur utilisation dans le compilateur LUSTRE.

9.5.1 Les graphes de décision binaires

La remarque qui est à la base des *bdds*, est que toute fonction booléenne à n arguments peut se décomposer, suivant la valeur de son premier argument, en deux fonctions à $n-1$ arguments (décomposition de Shannon [Sha38]) :

$$\begin{array}{rrcl} & f(x_1, x_2, \cdots, x_n) & = & \textit{si} \;\; x_1 \;\; \textit{alors} \;\; f_1(x_2, \cdots, x_n) \;\; \textit{sinon} \;\; f_0(x_2, \cdots, x_n) \\ \textit{avec} & f_1(x_1, \cdots, x_{n-1}) & = & f(1, x_1, \cdots, x_{n-1}) \\ \textit{et} & f_0(x_1, \cdots, x_{n-1}) & = & f(0, x_1, \cdots, x_{n-1}) \end{array}$$

Cette décomposition peut être représentée par un arbre binaire complet avec, par exemple, la convention que les fonctions correspondant à la valeur 1 de l'argument sont à gauche. L'arbre binaire de la fonction $(x \vee y) \wedge z$, décomposée selon l'ordre x, y, z est représenté sur la figure 9.2. L'ordre des arguments étant fixé, cet arbre est évidement unique ; il constitue donc une représentation canonique de la fonction.

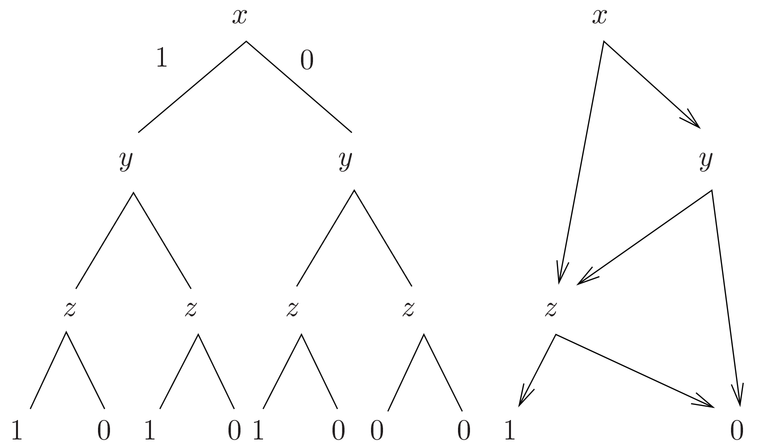


FIG. 9.2 – *Arbre et graphe binaires de la fonction* $(x \vee y) \wedge z$

Dans l'arbre binaire complet, certains nœuds sont inutiles : c'est le cas quand les fils droit et gauche sont identiques. L'élimination de tels nœuds est réalisée en appliquant la règle de simplification suivante : $\begin{array}{c} x \\ / \;\; \backslash \\ \alpha \quad\; \alpha \end{array} \to \alpha$. Après un nombre fini de simplifications, on obtient un arbre dans lequel aucun nœud n'est inutile, appelé *arbre de Shannon* de la fonction. Pour un ordre fixé, cet arbre est encore une représentation canonique de la fonction. Le *bdd* de la fonction est une représentation graphique de cet arbre, dans laquelle les sous-graphes identiques sont partagés (figure 9.2).

Il existe de nombreux résultats théoriques sur les *bdds*, le plus important étant évidemment la canonicité de cette représentation : deux fonctions f et g portant sur les mêmes arguments $x_1, \cdots, x_n$ sont égales si et seulement si leurs *bdds* selon le même ordre des arguments sont identiques.

Dans le compilateur, les *bdds* sont gérés de manière abstraite :

– *true-bdd* dénote la fonction toujours vraie

– *false-bdd* dénote la fonction toujours fausse

– *get-bdd*(i, bdd_1, bdd_0) construit si c'est nécessaire un nœud correspondant au i$^{\text{ème}}$ argument, dont les fils gauche et droit sont respectivement bdd_1 et bdd_0

– toutes les opérations sur les fonctions (égalité, ou, et, négation ...) sont disponibles ; nous les noterons avec les symboles mathématiques usuels $(=, \vee, \wedge, \neg \cdots)$

9.5.2 L'ordre des arguments

La seule contrainte que l'on impose à l'ordre des arguments, est de placer en tête les variables d'état, puis les conditions. De cette manière les *bdds* peuvent être interprétés :

– comme des fonctions de $Q \times C \rightarrow B$, c'est-à-dire des fonctions booléennes dont les arguments sont les éléments de contrôle (variables d'état et conditions)

– comme des fonctions de $Q \rightarrow (C \rightarrow B)$, c'est-à-dire des fonctions qui à tout état associent une fonction booléenne dont les arguments sont les conditions.

Les fonctions de $C \rightarrow B$ sont appelées fonctions instantanées. Elles jouent un rôle très important dans la génération de code, car c'est à partir de celles-ci que l'on génère le code des transitions.

9.5.3 Traduction des expressions algébriques en *bdds*

Cette traduction est réalisée par la fonction *exp-to-bdd*. La définition de cette fonction, calquée sur la syntaxe, est triviale. Par exemple :

$$\textit{exp-to-bdd}(e_1 \texttt{ or } e_2) \quad = \quad \textit{exp-to-bdd}(e_1) \vee \textit{exp-to-bdd}(e_2)$$

A chaque élément de contrôle correspond un indice d'argument dans les *bdds*, qui dépend pour les conditions du nombre de variables d'état (n) :

$$\begin{aligned} \textit{exp-to-bdd}(svar_i) &= \textit{get-bdd}(i, \textit{true-bdd}, \textit{false-bdd}) \\ \textit{exp-to-bdd}(cond_i) &= \textit{get-bdd}(i+n, \textit{true-bdd}, \textit{false-bdd}) \end{aligned}$$

9.5.4 La valeur *nil*

La valeur *nil* est tout à fait acceptable pour une variable d'état : en particulier, dans l'état initial du programme, toutes les variables d'état de la forme `last`(e) valent *nil*. L'ensemble des booléens auquel nous avons fait référence est donc un ensemble à trois valeurs : $B = \{0, 1, nil\}$. Il nous faut donc préciser la sémantique que le compilateur donne aux opérations booléennes sur la valeur *nil*.

La valeur *nil* est absorbante pour tous les calculs booléens, sauf :

– pour le “et” logique avec 0 $(nil \wedge 0 = 0)$

– pour le “ou” logique avec la valeur 1 $(nil \vee 1 = 1)$.

La structure des *bdds* nous oblige à introduire un autre cas où la valeur *nil* n'est pas absorbante, non déductible des deux précédents :

$$si\ \ nil\ \ alors\ \ x\ \ sinon\ \ x\ \ =\ \ x$$

9.6 Le choix du contrôle

Le traitement associé aux éléments de contrôle est donc clairement défini : il s'agit d'évaluer de manière exhaustive le comportement temporel de la mémoire booléenne du programme. Ce type de traitement peut entraîner une véritable explosion de la taille du code généré. En effet, la taille d'un automate modélisant le comportement de n mémoires booléennes est potentiellement de 2^n états et 2^{2n} transitions (une transition pour chaque couple d'états). Pour un programme donné, l'automate “extrême” qui modélise le comportement de toutes les mémoires booléennes est sans doute la structure de contrôle la plus rapide, mais sa taille peut être prohibitive. Il est donc judicieux d'éliminer du contrôle les “mauvais éléments”, c'est-à-dire les mémoires dont la prise en compte multiplie inutilement le nombre d'états pour un gain de vitesse d'exécution dérisoire. De telles mémoires seront traitées non pas comme des variables d'état, mais comme des mémoires quelconques (entières ou réelles).

9.6.1 Cohérence du choix

L'ensemble des variables d'état peut donc être un sous-ensemble quelconque des mémoires booléennes du programme. Pour être exploitable, cet ensemble doit vérifier une propriété de cohérence : si une mémoire *memop*(*exp*) est choisie comme variable d'état, alors toutes les mémoires et les expressions non évaluables qui apparaissent dans *exp* doivent être respectivement des variables d'état et des conditions. Cette sorte de “fermeture” dans le choix des éléments de contrôle permet de définir l'évolution de chaque variable d'état comme une fonction de transition qui ne dépend que des éléments de contrôle.

Une autre contrainte dans le choix du contrôle est liée à l'implémentation des entrées/sorties booléennes par des signaux purs. En effet, une entrée pure ne peut apparaître que comme argument d'une conditionnelle dans le programme OC. L'expression correspondante doit donc être choisie comme condition. Une sortie pure se comporte comme une sortie valuée ne pouvant prendre que les valeurs *true*, *false* ou *bottom*. L'évaluation d'une telle sortie, au même titre que celle d'une variable d'état, doit donc s'exprimer comme une fonction booléenne ne dépendant que des éléments de contrôle.

9.6.2 Heuristiques

Cinq options pour le choix du contrôle sont disponibles dans le compilateur. Trois d'entre elles existaient déjà dans le compilateur LUSTREV2 : il s'agit des deux solutions extrêmes (“tout” ou “rien”), et d'une solution intermédiaire basée sur une heuristique simple (“conditionnelle”). Nous avons complété cet éventail avec deux autres heuristiques donnant des solutions intermédiaires. Nous présentons ces choix de manière intuitive, sans entrer dans les détails de mise en œuvre. Les contraintes liées à la fermeture et aux entrées/sorties pures sont sous-entendues quelle que soit l'heuristique choisie :

Rien : Toutes les expressions booléennes sont considérées comme des données, cette option conduit à générer un automate à un seul état que nous appelons programme en boucle simple (§6.1). La programmation en boucle simple est le seul moyen de garantir que la taille du programme produit soit du même ordre que celle du programme source.

Heuristique "états initiaux" : Cette heuristique est basée sur la remarque pragmatique que l'opérateur `->` de LUSTRE (et donc `init` dans le programme normalisé) est toujours utilisé comme un opérateur de contrôle. Pour un programme comportant n opérateurs `init` (i.e. n horloges), la taille de l'automate correspondant est potentiellement de 2^n états. Mais en fait le nombre des horloges est toujours très limité, et celles-ci sont souvent interdépendantes. Cette heuristique, même pour un programme avec plusieurs horloges, conduit généralement à un automate de taille raisonnable.

Heuristique "expressions conditionnelles" : Cette heuristique est basée sur l'hypothèse que l'opérateur `if` symbolise l'emploi d'une expression booléenne comme élément de contrôle. C'est une remarque très naturelle, mais en fait elle conduit presque toujours à choisir toutes les expressions booléennes.

Heuristique "expressions récurrentes" : Cette heuristique est basée sur la remarque que les mémoires les plus "explosives" sont celles qui ne dépendent que d'expressions non évaluables. En effet, la prise en compte d'une telle mémoire va systématiquement multiplier par deux le nombre d'états, puisque son évolution ne dépend pas du passé. Pour éviter de telles mémoires, cette heuristique choisit en priorité les expressions récurrentes, c'est-à-dire celles qui contiennent une mémoire dont l'évolution dépend de son propre passé (les expressions "`init`" sont naturellement incluses par cette heuristique). Puis cet ensemble de base est complété par les mémoires dont l'évolution est très dépendante des éléments de contrôle déjà choisis.

Tout : Le comportement temporel de toutes les expressions booléennes est évalué de manière exhaustive en fonction des expressions non-évaluables. L'automate produit a évidemment un grand intérêt théorique, puisqu'il synthétise le plus grand nombre d'informations sur les exécutions possibles du programme : il sert notamment de base à la vérification de propriétés [Glo89, Rat92].

Chapitre 10

Identification des données

Dans le chapitre précédent, nous avons vu que le comportement d'une mémoire booléenne pouvait être évalué de manière exhaustive dès la compilation. Ce n'est évidemment pas le cas, par exemple, d'une mémoire entière dont le domaine de valeurs est théoriquement infini. Une telle mémoire va donc nécessiter l'emploi d'une variable dans le code objet. Pour éviter toute confusion avec les variables du programme source, les variables du code OC sont appelées des *registres*. Tous les opérateurs `last` de type entier, réel, externe, voire booléen s'ils n'ont pas été retenus comme variables d'état, nécessitent un registre. D'autres registres sont indispensables pour implémenter les sorties, ou les appels de procédures du programme.

Le but de ce chapitre est d'identifier les registres nécessaires, et d'étudier leur évolution d'un cycle sur l'autre. Cette évolution va se traduire dans le code objet par des actions impératives, généralement de simples affectations. Pour synthétiser les informations relatives aux registres et à leur évolution, nous introduisons la notion de *calcul*.

En première approximation, un calcul est associé à un registre particulier, et synthétise les informations nécessaires à son affectation. Mais cette notion est insuffisante dans le cas des résultats d'appels de procédures externes. En effet, un appel de procédure est une action indivisible qui affecte plusieurs registres. Un calcul est donc caractérisé par son aspect atomique dans le code généré : il est associé à un ou plusieurs registres et traduit le fait que l'évolution de ces registres est réalisée par une action indivisible dans le code OC.

L'ensemble des calculs doit être ordonné pour permettre la séquentialisation du programme. La recherche de cet ordre est basée sur les dépendances entre les calculs ; elle peut provoquer la définition de registres intermédiaires : les *tampons*.

Les actions qu'on doit générer pour chaque calcul dépendent des éléments de contrôle. On définit une représentation canonique des actions, les *actions gardées*, qui met en évidence ces dépendances. La construction des actions gardées constitue la phase ultime du traitement des données.

10.1 Les calculs

Un calcul est une structure interne de la forme :

calcul ::= *mode decl-def*

où *decl-def* est l'expression du réseau associée au calcul, et *mode* un champ variant définissant le type d'action à générer. Les registres où sont stockés le ou les résultats du calcul sont associés au champ *mode*.

10.1.1 Les sorties

Pour chaque sortie implémentée par un signal valué, on introduit un calcul de mode :

mode ::= OUTSIG *var-index signal-index*

où *var-index* est le registre où doit être stockée la valeur de la sortie, et *signal-index* le signal qu'il faut éventuellement émettre. Le calcul des sorties est un cas particulier où l'on doit générer deux actions (affectation et émission). Mais cette séquence garde tout de même un caractère atomique : une sortie peut toujours être émise immédiatement après son affectation.

Dans le cas d'une sortie booléenne implémentée par un signal pur, l'indication *var-index* est inutile, mais *signal-index* demeure :

mode ::= PURE-OUTSIG *signal-index*

10.1.2 Les mémoires en `last`

A tout opérande d'un `last` qui n'est pas déjà une variable d'état ou une sortie, est associé un registre destiné à conserver le résultat d'un cycle sur l'autre. L'action correspondante est une simple affectation :

mode ::= AFFECT *var-index*

En particulier, le mode OUTSIG, pourrait être décomposé en deux calculs : un AFFECT qui met à jour le registre correspondant, et un PURE-OUTSIG qui émet éventuellement le signal. Si l'opérateur `last` porte sur une expression de tuple, il faut autant de registres que l'arité du tuple. Pour ne pas avoir à introduire un mode de calcul spécifique, on suppose que toute expression de la forme $\texttt{last}E$ où E est un tuple, est remplacée par l'expression :

$$(\texttt{last proj}_1(E), \cdots, \texttt{last proj}_k(E))$$

De cette manière, tous les opérateurs de mémoire du programme ne portent que sur des expressions de type simple, et ne nécessitent donc qu'un seul registre.

10.1.3 Les mémoires en `init`

Les opérateurs `init` qui n'ont pas été retenus comme variables d'état nécessitent un registre (*var-index*) dont le traitement est différent de celui des AFFECT : la valeur initiale du registre est `true`, et son évolution est implicitement récursive. On introduit donc un calcul de mode INIT-MEM :

mode ::= INIT-MEM *var-index*

10.1.4 Les appels de procédures

Les appels de fonctions produisant plusieurs résultats vont être implémentés par des appels de procédures externes. Ils nécessitent donc autant de registres qu'il produisent de résultats :

mode ::= PROC-CALL *nb-result tab-result*

Le tableau *tab-result* contient les *nb-result* indices des registres associés à ce calcul. Le problème est de savoir à quelle expression du réseau on doit associer ces registres. Si on les associe aux opérateurs `call`, on risque de multiplier inutilement le nombre de registres, par exemple :

```
(x,y) = if c then F(a,b) else G(d,e);
```

deux registres sont associés à l'appel de `F`, deux autres à celui de `G`. Si `x` et `y` sont déjà des calculs, on a encore deux registres inutiles. L'idée est donc d'associer le calcul à l'opérateur `if` : deux registres seulement sont nécessaires, associés à `x` et `y`.

Cette méthode introduit une nouvelle contrainte dans le choix des éléments de contrôle. En effet, un appel de procédure n'est plus relatif à une procédure particulière : dans notre exemple, suivant la valeur de l'expression `c`, l'une des procédures `F` ou `G` va être appelée. Ce choix ne peut être implémenté dans le code OC que par une instruction conditionnelle :

```
if c then
        F(&x,&y)(a,b)
else
        G(&x,&y)(d,e)
```

Toute expression booléenne qui apparaît dans la définition d'un calcul de mode PROC-CALL doit donc être choisie pour faire partie du contrôle. Comme pour la définition d'une variable d'état, une telle expression va s'exprimer comme une fonction booléenne des éléments de contrôle.

10.1.5 Les conditions

L'évaluation d'une condition peut être faite dans le test lui-même, ou précéder celui-ci. Dans le dernier cas, il faut introduire un registre intermédiaire destiné à conserver le résultat de l'évaluation. Cette solution est plus coûteuse en nombre de registres, mais simplifie beaucoup le traitement des conditions : l'évaluation d'une condition est alors un simple calcul de mode AFFECT. La phase de séquentialisation, dont le rôle est (entre autres) d'ouvrir et de fermer les tests, est aussi considérablement simplifiée : tous les tests sont de la forme "`if:`*var-index*" et une fois que la valeur de la condition est stockée, le test correspondant peut être ouvert et fermé à volonté.

10.2 L'ordre des calculs

Une fois que l'ensemble des calculs a été fixé, il convient d'étudier les relations qui les relient pour déduire une séquence de calcul utilisant au mieux les résultats déjà connus.

10.2.1 Les dépendances

A chaque calcul est associé un champ *deps*, qui représente l'ensemble des calculs dont il dépend instantanément. Ce critère de dépendance est purement syntaxique :

> *Un calcul C associé à l'expression E dépend instantanément de C' associé à E', si E' est une sous-expression de E, apparaissant en dehors d'un opérateur de mémoire : on a alors $C' \in C.deps$.*

Le fait que le programme soit sans blocage nous garantit que la fermeture transitive de cette relation est un ordre partiel. Pour pouvoir réutiliser au cours d'un même cycle un résultat déjà calculé, il est clair que la séquence des actions doit respecter cet ordre.

10.2.2 Les pré-dépendances

On définit pour chaque calcul un champs *predeps*, pour représenter l'ensemble de tous les calculs qui dépendent de la valeur précédente de celui-ci. Il s'agit là aussi d'un critère syntaxique :

> *Un calcul C associé à l'expression E pré-dépends de C' associé à E', si* $\texttt{last}E'$ *est une sous-expression de E ; on a alors $C \in C'$.predeps.*

Un deuxième cas de pré-dépendance doit être défini pour le cas particulier des mémoires de mode INIT-MEM :

> *Un calcul C associé à l'expression E pré-dépends de C' associé à* $\texttt{init}E'$*, si* $\texttt{init}E'$ *est une sous-expression de E.*

Comme pour les dépendances instantanées, nous nous intéressons à la fermeture transitive de cette relation, que nous appelons *relation de pré-dépendance.* Cette relation intervient dans la séquentialisation car, quand un registre sert à conserver une valeur d'un cycle sur l'autre, il ne peut être mis à jour que si tous les calculs qui dépendent de sa valeur précédente ont été effectués. Soit par exemple un calcul X dont l'évolution dépend de $\texttt{last}(Y)$; si la mise à jour de Y est effectuée avant celle de X, le registre correspondant est écrasé et l'ancienne valeur de Y est perdue. On va donc chercher un séquence des calculs qui respecte les pré-dépendances, c'est-à-dire telle qu'aucun calcul ne succède à un calcul dont il pré-dépend. Une telle séquence n'existe pas toujours, car la pré-dépendance, contrairement à la dépendance instantanée, n'est pas une relation d'ordre. Il suffit d'imaginer deux calculs qui pré-dépendent réciproquement l'un de l'autre :

```
x = F(last y);
y = G(last x);
```

Dans ce cas, `x` pré-dépend de `y` et devrait donc être calculé avant `y`, de même que `y` pré-dépend de `x` et devrait donc être calculé avant ce dernier.

10.2.3 Les tampons

Nous avons vu que la pré-dépendance, contrairement à la dépendance instantanée, n'était pas forcément un ordre. Mais l'union de ces deux relations peut aussi créer des problèmes. C'est le cas d'un calcul qui dépend à la fois des résultats courant et précédent d'un même calcul :

```
x = F( last y , y);
```

Ces problèmes sont très simplement résolus en introduisant des registres supplémentaires appelés *tampons.* Par exemple, dans le cas présenté dans la section précédente où deux calculs `x` et `y` pré-dépendent l'un de l'autre, on peut introduire un registre `t` associé à l'expression `last y`. Une séquence possible des mises à jour est alors :

```
y := G(x);
```

```
x := F(t);
t := y;
```

Même chose pour le deuxième exemple, où le calcul x dépend à la fois des valeurs courante et précédente de y :

```
y := ..... ;
x := F(t,y);
t := y;
```

On introduit donc un certain nombre de calculs intermédiaires, caractérisés par le mode :

mode ::= TAMP *var-index calcul*

L'action associée est une simple recopie du registre associé à *calcul*, dans le registre *var-index* associé au tampon.

La recherche d'un nombre minimal de tampons est un problème NP-complet. On utilise donc un algorithme très simple qui se contente de "casser" le premier cycle rencontré, jusqu'à l'obtention d'un ordre. Cet algorithme, déjà implémenté dans le compilateur LUSTRE-V2, donne en général un très petit nombre de tampons pour une complexité raisonnable.

10.3 Evaluation d'un calcul de mode AFFECT

10.3.1 Syntaxe abstraite des définitions

A chaque cycle, l'action à générer pour un calcul de mode AFFECT est une simple affectation. La partie gauche de cette affectation est bien entendu le registre associé au calcul, et la partie droite une expression OC déduite de sa définition déclarative. Une fois que les éléments de contrôle et les calculs ont été choisis, cette définition peut s'exprimer dans une syntaxe simplifiée :

decl-def	::=	*const*	*(constante)*
	|	$svar_i$	*(référence à la valeur d'une variable d'état)*
	|	$cond_i$	*(référence à la valeur d'une condition)*
	|	$calc_i$	*(référence au résultat courant d'un calcul)*
	|	`last`$calc_i$	*(référence au résultat passé d'un calcul)*
	|	*dataop* (*decl-def*, $\cdots$, *decl-def*)	*(opérateur classique)*

Le terme *dataop* regroupe tous les opérateurs classiques (arithmétiques, logiques ...), ainsi que les appels de fonction. Ces opérateurs peuvent se traduire trivialement en OC par des appels de fonctions prédéfinies ou externes. Le cas des références aux résultats de procédures est traité ultérieurement.

Les références à des résultats présents ou passés de calculs vont bien entendu se traduire en OC par des références aux registres correspondants.

Une référence à la constante particulière `bottom` signifie que le registre correspondant ne doit pas évoluer ; elle est donc interprétée comme une référence à ce même registre, avec la convention qu'aucune affectation triviale du type "*var-index* `:=` *var-index*" ne doit être effectivement générée.

Illustrons ces quelques principes sur un exemple très simple tiré d'un programme normalisé :

```
cpt = if c then
```

```
          if init(c) then 0
          else if (x and not last x) then 0 else last cpt + 1
     else bottom
```

Dans le programme, deux variables d'état ont été identifiées (`init(c)` et `last x`), ainsi que deux conditions (`c` et `x`). Ces éléments de contrôle sont notés respectivement $svar_1$, $svar_2$, $cond_1$ et $cond_2$. L'utilisation de `cpt` en argument d'un opérateur de mémoire nous amène à lui associer un calcul de mode AFFECT, noté $calc_1$. Avec ces différentes notations, l'expression associée au calcul devient :

```
if cond1 then
         if svar1 then 0
         else if (cond2 and not svar2) then 0 else last calc1 + 1
else calc1
```

10.3.2 Les expressions gardées

L'affectation doit être intégrée à la structure de contrôle de manière à éliminer les références aux variables d'état et aux conditions. Pour un couple particulier $(q, c) \in Q \times C$, chaque référence peut en effet être remplacée par sa valeur (`true` ou `false`). Après simplification des opérateurs booléens évaluables on obtient une expression ne contenant plus que des *dataop* et des références à des registres. Une telle expression est qualifiée d'*impérative*, en ce sens qu'elle est directement traduisible en OC. L'ensemble des expressions impératives est noté I. Dans notre exemple, pour :

$$(svar_1, svar_2, cond_1, cond_2) \quad = \quad (\texttt{false}, \texttt{true}, \texttt{true}, \texttt{true})$$

et après simplification des opérations booléennes, on obtient l'expression impérative :

$$calc_1.var\text{-}index \texttt{ + 1}$$

L'expression associée à un calcul dénote donc une fonction, qui à un couple de $Q \times C$ fait correspondre une expression impérative de I. Cette fonction va être représentée sous une forme normale appelée *expression gardée*. Il s'agit d'une représentation directement inspirée par les *bdd* : une structure binaire dont les nœuds sont étiquetés par les éléments de contrôle, mais où les feuilles peuvent être des expressions impératives quelconques. L'ordre des arguments est bien entendu le même que pour des *bdd* classiques, ces derniers pouvant être vus comme des expressions gardées particulières dont les seules feuilles possibles sont les constantes booléennes.

$$\begin{array}{lcll} imp\text{-}exp & ::= & const & \textit{(constante)} \\ & | & var\text{-}index & \textit{(référence au contenu d'un registre)} \\ & | & dataop\ (imp\text{-}exp, \cdots, imp\text{-}exp) & \textit{(opérateur classique)} \end{array}$$

$$\begin{array}{lcll} gard\text{-}exp & ::= & imp\text{-}exp & \textit{(feuille)} \\ & | & ctrl_i(gard\text{-}exp)(gard\text{-}exp) & \textit{(nœud binaire “si alors sinon”)} \end{array}$$

Comme pour les *bdd* booléens, on est amenés à s'interroger sur la canonicité de cette représentation. Dans le cas des expressions gardées, cette canonicité ne peut être que relative puisqu'il n'existe pas de représentation canonique des feuilles (les expressions impératives). On est donc amené à définir une relation d'équivalence sur $(Q \times C) \to I$ basée sur l'équivalence syntaxique des expressions impératives :

Deux fonctions f et g de $(Q \times C) \to I$ sont équivalentes si pour un même couple $(q, c) \in Q \times C$ elles donnent deux expressions impératives syntaxiquement équiva-

lentes.

L'équivalence syntaxique des feuilles est garantie par la représentation sous forme de réseau d'opérateurs. Du point de vue de la génération de code, ce critère est tout à fait naturel, puisque le code généré n'est qu'une traduction en OC des expressions impératives.

L'expression gardée correspondant à notre exemple est représentée sur la figure 10.1 sous la forme d'un arbre de Shannon. Dans le compilateur, les sous-arbres communs sont partagés et on manipule des *bdd*. Le registre associé au calcul est noté r ($r = calc_1.var\text{-}index$).

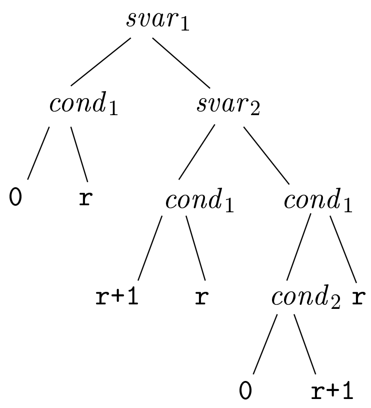


FIG. 10.1 – *L'expression gardée associée à* `cpt`

10.3.3 Construction des expressions gardées

Dans la suite, nous conservons le terme *bdd* pour désigner des expressions gardées quelconques, les *bdd* strictement booléens ne constituant qu'un cas particulier.

La création d'un nœud est réalisée par la fonction *get-bdd* définie dans le chapitre précédent. La composition de plusieurs *bdd* est réalisée par une fonction $apply(dataop, bdd_1, \ldots , bdd_n)$, qui parcourt en parallèle les arguments jusqu'aux feuilles, avant de combiner celles-ci par l'opérateur *dataop*. La règle générale de combinaison des feuilles est triviale :

$$apply(dataop, imp\text{-}exp_1, \ldots , imp\text{-}exp_n) \quad = \quad dataop\ (imp\text{-}exp_1, \cdots, imp\text{-}exp_n)$$

A cette règle générale, on ajoute des règles de simplification pour les opérateurs booléens évaluables (pour les opérateurs commutatifs, on se contente de donner les règles relatives au premier argument) :

$$\begin{aligned}
apply(\texttt{not}, \texttt{true}) &= \texttt{false}\\
apply(\texttt{not}, \texttt{false}) &= \texttt{true}\\
apply(\texttt{or}, \texttt{true}, \alpha) &= \texttt{true}\\
apply(\texttt{or}, \texttt{false}, \alpha) &= \alpha\\
apply(\texttt{and}, \texttt{true}, \alpha) &= \alpha\\
apply(\texttt{and}, \texttt{false}, \alpha) &= \texttt{false}\\
apply(\texttt{=}, \alpha, \alpha) &= \texttt{true}\\
apply(\texttt{=}, \texttt{true}, \texttt{false}) &= \texttt{false}
\end{aligned}$$

$$\begin{aligned}
apply(\texttt{<>}, \alpha, \alpha) &= \texttt{false}\\
apply(\texttt{<>}, \texttt{true}, \texttt{false}) &= \texttt{true}\\
apply(\texttt{if}, \texttt{true}, \alpha, \beta) &= \alpha\\
apply(\texttt{if}, \texttt{false}, \alpha, \beta) &= \beta\\
apply(\texttt{if}, \alpha, \beta, \beta) &= \beta
\end{aligned}$$

Les règles de simplification de l'opérateur d'exclusion sont définies récursivement en introduisant un opérateur `nor` à arité variable non nulle. Comme pour les autres opérateurs commutatifs, on se contente de donner les règles relatives au premier argument :

$$\begin{aligned}
apply(\texttt{\#}, \alpha) &= \texttt{true}\\
apply(\texttt{\#}, \texttt{false}, \alpha_1, \ldots, \alpha_n) &= apply(\texttt{\#}, \alpha_1, \ldots, \alpha_n)\\
apply(\texttt{\#}, \texttt{true}, \alpha_1, \ldots, \alpha_n) &= apply(\texttt{nor}, \alpha_1, \ldots, \alpha_n)\\
apply(\texttt{nor}, \alpha) &= apply(\texttt{not}, \alpha)\\
apply(\texttt{nor}, \texttt{false}, \alpha_1, \ldots, \alpha_n) &= apply(\texttt{nor}, \alpha_1, \ldots, \alpha_n)\\
apply(\texttt{nor}, \texttt{true}, \alpha_1, \ldots, \alpha_n) &= \texttt{false}
\end{aligned}$$

Grâce à cette fonction *apply*, la fonction *exp-to-bdd* qui traduit une expression algébrique en une expression gardée (éventuellement booléenne), s'exprime de manière très simple. On rappelle que par convention les n variables d'état sont les arguments $1, \cdots, n$ des *bdd*, et que les m conditions sont les arguments $n+1, \cdots, n+m$:

$$\begin{aligned}
\textit{exp-to-bdd}(\textit{const}) &= \textit{const}\\
\textit{exp-to-bdd}(\textit{svar}_i) &= \textit{get-bdd}(i, \textit{true-bdd}, \textit{false-bdd})\\
\textit{exp-to-bdd}(\textit{cond}_i) &= \textit{get-bdd}(i+n, \textit{true-bdd}, \textit{false-bdd})\\
\textit{exp-to-bdd}(\textit{calc}_i) &= \textit{calc}_i.\textit{var-index}\\
\textit{exp-to-bdd}(\texttt{last}\,\textit{calc}_i) &= \textit{calc}_i.\textit{var-index}\\
\textit{exp-to-bdd}(\textit{dataop}\ (e_1, \cdots, e_n)) &= apply(\textit{dataop}, \textit{exp-to-bdd}(e_1), \ldots, \textit{exp-to-bdd}(e_n))
\end{aligned}$$

10.3.4 Les actions gardées

Le code à générer en fonction du contrôle est complètement défini par l'expression gardée et le registre associés au calcul. Par souci de synthèse, ces informations vont être réunies pour définir l'action gardée associée au calcul. L'action gardée est une structure binaire calquée sur l'expression gardée, mais dans laquelle les feuilles sont remplacées par l'action OC à générer. Dans le cas d'un calcul de mode AFFECT, associé au registre "`r`" et à l'expression gardée α, la construction de l'action gardée consiste à remplacer dans α chaque feuille *imp-exp* par l'affectation "`r :=` ***imp-exp***". L'affectation triviale "`r := r`" ne devant jamais être générée, on la remplace dans l'action gardée par une action fictive `nop`. La figure 10.2 représente l'action gardée associée au calcul de `cpt`.

10.3.5 Conclusion

La démarche que nous venons de suivre dans le cas des calculs AFFECT peut se résumer en trois étapes :

– recherche d'une expression impérative dénotant l'évolution du calcul à chaque cycle,

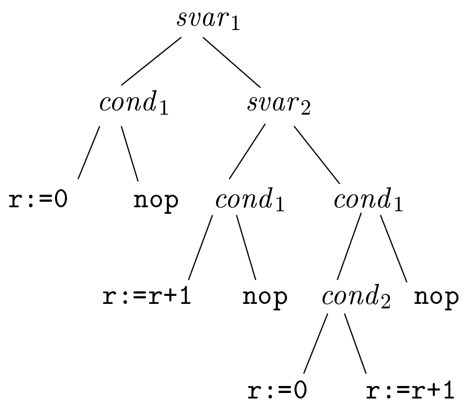


FIG. 10.2 – *L'action gardée associée à* `cpt`

– traduction de cette expression algébrique en une expression gardée,

– transformation de l'expression gardée en action gardée.

Une démarche similaire va être suivie pour les autres types de calculs, des différences pouvant intervenir notamment dans la détermination de l'expression impérative, ou dans la transformation de l'expression gardée en action gardée.

10.4 Evaluation d'un calcul de mode INIT-MEM

A un calcul de mode INIT-MEM sont associés un registre *var-index* et une expression du réseau de la forme "`init(E)`". Ce mode de calcul a été introduit pour indiquer que le registre correspondant devait être initialisé à `true`, par opposition aux registres de mode AFFECT qui ne sont pas initialisés. Par contre, pour tout ce qui concerne les références aux INIT-MEM et leur évolution, on peut se ramener au cas d'une mémoire classique. Cela correspond à remplacer dans le réseau toute expression "`init(E)`" par une expression "`last(X)`", où l'expression `X` est définie récursivement par (§9.3.1) :

```
X = if last(X) then not E else bottom
```

Le calcul de mode INIT-MEM est, dans le nouveau programme, associé à l'expression `X`. On peut alors construire à partir de `X` une expression gardée puis une action gardée comme on l'aurait fait pour un calcul de mode AFFECT.

10.5 Evaluation d'un calcul de mode PURE-OUTSIG

Lors du choix du contrôle, l'expression associée à un calcul de mode PURE-OUTSIG a été traitée de la même manière qu'une variable d'état. Il en résulte que dans notre syntaxe abstraite, cette expression se décompose uniquement en constantes, opérateurs évaluables et références aux éléments de contrôle. L'expression gardée correspondante est donc un *bdd* strictement booléen dont les feuilles sont les constantes `true`, `false` et `bottom`. L'occurrence de la constante `true` signifie que le signal correspondant doit être émis, celle des constantes `false` ou `bottom` qu'aucune action ne doit être générée.

L'action gardée associée à un calcul *calc* de mode PURE-OUTSIG est donc obtenue à partir de l'expression gardée en remplaçant chaque feuille `true` par l'action :

`output :` *calc.signal-index*,

et chaque feuille `false` ou `bottom` par l'action fictive "`nop`".

10.6 Evaluation d'un calcul de mode OUTSIG

Les calculs de mode OUTSIG sont particuliers, car ils nécessitent deux actions : une affectation et une émission. Comme pour un calcul de mode AFFECT, ces actions vont être déduites de l'expression gardée associée. La seule différence réside dans l'interprétation de la constante `bottom`. En effet, une occurrence de cette constante signifie non seulement que le registre ne doit pas être affecté, mais aussi que le signal ne doit pas être émis. On ne peut donc pas assimiler les occurrences de la constante `bottom` à des auto-références comme dans le cas AFFECT.

Pour construire l'action gardée d'un calcul *calc* de mode OUTSIG à partir de son expression gardée, on distingue trois cas :

- Une feuille `bottom` est remplacée par l'action "`nop`" : le registre n'est pas modifié, et le signal n'est pas émis.

- Une référence au registre correspondant est remplacée par l'action :

  `output :` *calc.signal-index*

  En effet, bien que le registre ne soit pas modifié, le signal doit tout de même être émis ;

- Toute autre expression impérative *imp-exp* est remplacée par la séquence :

  calc.var-index`:=` *imp-exp*`; output :` *calc.signal-index.*

10.7 Evaluation d'un calcul de mode PROC-CALL

10.7.1 Syntaxe abstraite des expressions de tuple

A un calcul de mode PROC-CALL sont associés un certain nombre de registres, et une expression de type tuple qui définit l'évolution de ces registres. La syntaxe abstraite des expressions de tuple est la suivante :

tup-exp	::=	`if` *control-exp* `then` *tup-exp* `else` *tup-exp*	
	|	`(`*decl-def*, ···, *decl-def*`)`	*("tuplage")*
	|	*func*(*decl-def*, ···, *decl-def*)	*(appel de fonction)*

L'opérateur `if` porte sur une expression booléenne qui est supposée avoir été choisie pour faire partie du contrôle (§10.1.4). Elle se décompose donc en éléments de contrôle et opérateurs évaluables. Dans une opération de "tuplage", les différentes composantes du tuple sont définies par une expression qui leur est propre : les registres correspondants n'ont pas besoin d'être calculés ensemble. Par contre, dans le cas d'un appel de fonction à plusieurs résultats, les registres sont affectés ensemble, lors de l'appel de la procédure correspondante.

Un calcul PROC-CALL n'est donc pas relatif à l'appel d'une procédure particulière. Il signifie simplement que, en fonction du contrôle, les registres associés devront être calculés ensemble.

10.7.2 Les références à un calcul de mode PROC-CALL

La syntaxe des *decl-def* (§10.3.1) doit être complétée pour introduire les références à la k-ième composante d'un appel de procédure externe : $\texttt{proj}_k\,calc_i$.

La définition de la fonction *exp-to-bdd* doit être complétée pour prendre en compte ce cas. Une nouvelle fonction, *proj-to-bdd*, est introduite pour extraire de la définition d'un tuple les parties qui ne comportent pas d'appel de procédure :

$$\textit{exp-to-bdd}(\texttt{proj}_k\,calc_i) = \textit{proj-to-bdd}(k, calc_i, calc_i.\textit{decl-def})$$

$$\textit{proj-to-bdd}(k, calc_i, \texttt{(}E_1, \cdots, E_n\texttt{)}) = \textit{exp-to-bdd}(E_k)$$
$$\textit{proj-to-bdd}(k, calc_i, \textit{func}(\textit{decl-def}, \cdots, \textit{decl-def})) = calc_i.\textit{tab-result}[k]$$
$$\textit{proj-to-bdd}(k, calc_i, \texttt{if}\ C\ \texttt{then}\ T_1\ \texttt{else}\ T_2) =$$
$$\quad \textit{apply}(\texttt{if}, \textit{exp-to-bdd}(C), \textit{proj-to-bdd}(k, calc_i, T_1), \textit{proj-to-bdd}(k, calc_i, T_2))$$

10.7.3 L'expression gardée d'un calcul de mode PROC-CALL

Le rôle de la fonction *proj-to-bdd* est de limiter les références à une composante de tuple aux seuls cas ou un appel de procédure est nécessaire. De manière duale, quand on construit l'expression gardée associée à un PROC-CALL, on ne doit retenir que les cas où un appel est nécessaire :

$$\textit{exp-to-bdd}(\texttt{(}E_1, \cdots, E_n\texttt{)}) = \texttt{bottom}$$
$$\textit{exp-to-bdd}(\textit{func}(E_1, \cdots, E_n)) = \textit{apply}(\textit{func}, \textit{exp-to-bdd}(E_1), \cdots, \textit{exp-to-bdd}(E_n))$$
$$\textit{exp-to-bdd}(\texttt{if}\ C\ \texttt{then}\ T_1\ \texttt{else}\ T_2) =$$
$$\quad \textit{apply}(\texttt{if}, \textit{exp-to-bdd}(C), \textit{exp-to-bdd}(T_1), \textit{exp-to-bdd}(T_2))$$

La construction de l'action gardée correspondante est définie par deux règles :

- la constante `bottom` est remplacée par l'action `nop` ; cela signifie que pour la valeur correspondante des éléments de contrôle, la valeur des registres correspondants est donnée par un tuple, et qu'il n'y a donc pas lieu de les calculer ensemble ;

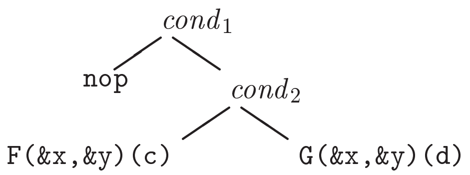
- un appel de fonction "$\textit{func}(\textit{imp-exp}_1, \cdots, \textit{imp-exp}_n)$" est remplacé par l'appel de procédure équivalent : "$\textit{func}(\&calc_i.\textit{tab-result})(\textit{imp-exp}_1, \cdots, \textit{imp-exp}_n)$".

10.7.4 Exemple

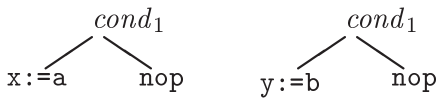
Prenons l'exemple d'un calcul W de mode PROC-CALL, auquel sont associés les registres `x` et `y` et l'expression suivante (`a,b,c,d` sont des constantes, `F,G` des procédures externes) :

`if` $cond_1$ `then` (`a`, `b`) `else if` $cond_2$ `then` `F`(`c`) `else` `G`(`d`)

Si $cond_1$ est vraie, aucun appel de procédure n'est nécessaire : l'action correspondante est donc `nop`. Sinon, suivant la valeur de $cond_2$, on doit appeler une des procédures `F` ou `G` avec les paramètres correspondants :



Soient deux calculs de mode AFFECT, X et Y, auxquels sont attachés respectivement les registres `x` et `y` et les expressions $\texttt{proj}_1 W$ et $\texttt{proj}_2 W$. Pour évaluer X quand $cond_1$ est vraie, il suffit d'extraire la première composante du tuple `(a,b)`. L'action correspondante est donc `x := a`. Dans tous les autres cas, un appel de procédure est nécessaire. La valeur de X se trouve donc dans le premier registre associé à W, qui n'est autre que `x`, l'affectation `x := x` ne devant pas être générée, l'action correspondante est `nop`. Les actions gardées de X et Y sont les suivantes :



Après factorisation, le code correspondant à la séquence W, X, Y est le suivant (`c1` et `c2` sont les registres associés aux deux conditions) :

```
if c1 then { x:= a; y := b }
else if c2 then F(&x,&y)(c)
else G(&x,&y)(d)
```

Chapitre 11

Génération de l'automate

Cette phase a pour but de construire explicitement l'automate de contrôle. Cette entreprise se heurte au problème intrinsèque de l'explosion du nombre des états. En effet, nous avons vu que l'automate de contrôle modélisant le comportement de n variables d'état a, dans le pire des cas, 2^n états et 2^{2n} transitions. Ce que l'on peut traduire en quelque sorte par "la taille du code généré peut être exponentiellement plus grande que celle du programme source", ce qui est un résultat assez gênant pour un compilateur qui se veut efficace.

L'explosion du nombre des états peut avoir deux origines :

- Le programme à compiler est intrinsèquement complexe, c'est-à-dire qu'il n'existe pas d'automate de taille raisonnable modélisant son comportement. Au niveau de l'analyse statique, on aurrait peut être pu limiter la complexité du programme en évitant l'expansion des nœuds internes : cette solution consiste à faire de la compilation séparée [Ray88]. Au niveau du générateur de code, tout ce que l'on peut faire avec un programme trop complexe, c'est jouer sur le choix des variables d'état (§9.6).

- L'automate de contrôle (§9.4) est de taille prohibitive, mais il existe un automate équivalent de taille raisonnable. Une équivalence sur les automates est appelée une *bisimulation* [Par81]. Dans une classe d'équivalence, on s'intéresse particulièrement à l'automate minimal, c'est-à-dire à l'automate équivalent ayant le nombre minimal d'états. L'explosion peut donc être due non pas à la complexité intrinsèque du programme, mais à un algorithme de génération qui ne produit pas un automate minimal.

Dans ce chapitre, nous nous intéressons particulièrement à ce deuxième problème, en présentant les deux algorithmes de génération d'automate implémentés dans le compilateur. L'algorithme *dirigé par les données*, déjà à la base du compilateur LUSTREV2, produit en général des automates non minimaux. Nous avons donc développé un nouvel algorithme, *dirigé par la demande*, qui produit à coup sûr des automates minimaux. Le problème d'explosion, s'il intervient au cours d'une génération dirigée par la demande, ne peut alors être attribué qu'à un "mauvais" choix des variables d'état.

La génération de code séquentiel, indépendante de la manière dont on parcourt les états, est traitée à part, dans le chapitre consacré à la *séquentialisation*.

On rappelle qu'à ce stade de la compilation, on a identifié n variables d'état, m conditions et k calculs ; l'ensemble Q est celui des états, C celui des valeurs des conditions, et A celui des actions OC. L'ensemble des booléens est noté B. Le programme est représenté par :

– les n fonctions de transitions : $\delta_i \in Q \rightarrow C \rightarrow B$

– les k actions gardées : $\lambda_j \in Q \rightarrow C \rightarrow A$

11.1 Génération dirigée par les données

L'automate construit par cet algorithme est très proche de l'automate de contrôle (§9.4) ; la différence essentielle est qu'on ne retient que les états accessibles depuis l'état initial. Chaque état $q \in Q$ correspond à une valeur du vecteur des variables d'état. Il est identifié de manière unique par un index, auquel font référence les actions `goto` des transitions (*state-index*).

11.1.1 Les actions instantanées

Le traitement des données dans un état consiste à associer à chacun des k calculs une action instantanée, c'est-à-dire une action gardée uniquement par des conditions :

$$\forall j \in 1..k \quad \begin{array}{rcl} \Lambda_{qj} : C & \rightarrow & A \\ c & \mapsto & \lambda_j(q,c) \end{array}$$

11.1.2 Le calcul du NEXT-STATE

La transition d'un état q, notée Δ_q, résulte de la combinaison des $\delta_i(q)$:

$$\begin{array}{rcl} \Delta_q : C & \rightarrow & Q \\ c & \mapsto & (\delta_1(q,c), \cdots, \delta_n(q,c)) \end{array}$$

Par souci d'uniformité, l'ensemble des calculs est complété avec une donnée fictive de mode NEXT-STATE (et d'index $g = k+1$), dont le calcul correspond au changement d'état. L'action instantanée modélisant le changement d'état depuis un état q est définie par la fonction :

$$\begin{array}{rcl} \Lambda_{qg} : C & \rightarrow & A \\ c & \mapsto & \texttt{goto:}\,\textit{state-index}(\Delta_q(c)) \end{array}$$

L'introduction de ce calcul fictif va considérablement simplifier le travail de la séquentialisation, mais il faut spécifier le fait que le calcul du NEXT-STATE doit être effectué en dernier. La manière la plus naturelle est de dire que ce calcul dépend de tous les autres : $\textit{deps}(g) = \{1, \cdots, k\}$.

11.1.3 Le parcours des états

L'algorithme ne traite que l'ensemble des états accessibles à partir de l'état initial. L'état initial (q_0) est celui dans lequel les variables d'état de la forme `init`(e) valent 1, et les autres *nil*. L'ensemble des états accessibles connus est noté ACCES. Le traitement d'un état accessible est réalisé par la procédure *gen-state* qui renvoie l'ensemble des successeurs de son argument :

$$\begin{array}{rcl} \textit{gen-state} : Q & \rightarrow & 2^Q \\ q & \mapsto & \{q' \in Q \;/\; \exists c \in C \;\; \Delta_q(c) = q'\} \end{array}$$

Chaque fois qu'un état est traité, il est mis dans l'ensemble TRAITE. Le traitement d'un état fait éventuellement apparaître de nouveaux états accessibles, qu'il faut à leur tour traiter. L'algorithme s'arrête quand tous les états accessibles sont traités :

data-driven{
  ACCES = { q_0 };
  TRAITE = $\emptyset$;
  tant que $\exists$ q $\in$ (ACCES \ TRAITE) {
      ACCES $\cup =$ *gen-state*(q);
      TRAITE $\cup =$ { q };
  }
}

11.1.4 Exemple

```
node Exemple(i: bool) returns (n : int);
var x,y,z : bool;
let
   n = if init(true) then 0 else if last x then 0 else last n + 1;
   x = if init(true) then false else not last x and z;
   y = if init(true) then i
       else if last x then (last y and i) else (last z or i);
   z = if init(true) then true
       else if last x then last z else ((last y and last z) or i);
tel.
```

Dans ce programme, on identifie quatre variables d'état, correspondant aux expressions `init(true), last x, last y, last z,` et une condition (l'entrée `i`). Pour plus de lisibilité, ces éléments de contrôle sont repérés par des identificateurs (respectivement *init*, *lastx*, *lasty*, *lastz* et *i*) plutôt que par des indices. La fonction d'évolution de *init* est par définition la fonction toujours fausse :

$$\delta_{init}(init, lastx, lasty, lastz, i) = 0$$

Celles des autres variables d'état sont exactement celles dénotées par les expressions définissant les variables internes (`x, y` et `z`). Dans le compilateur, ces fonctions sont représentées par des graphes de décision binaires. La figure 11.1 représente une forme simplifiée de ces graphes, où les sous-graphes communs ne sont pas partagés (arbre de Shannon), et où les feuilles sont les fonctions instantanées.

La seule donnée du programme est la sortie entière `n`. Son calcul correspond à une fonction $\lambda_{\mathtt{n}} \in Q \rightarrow C \rightarrow A$ représentée sur la figure 11.2.

Détaillons les différentes étapes de la génération :

- L'état initial, q_0, est caractérisé par les valeurs des variables d'état : $(1, nil, nil, nil)$. Dans cet état, l'action associée au calcul de la sortie est $\lambda_{\mathtt{n}}(q_0) =$ `(n := 0)`. L'évolution des variables d'état est donnée par :

$$\delta_{init}(q_0, i) \;=\; 0$$

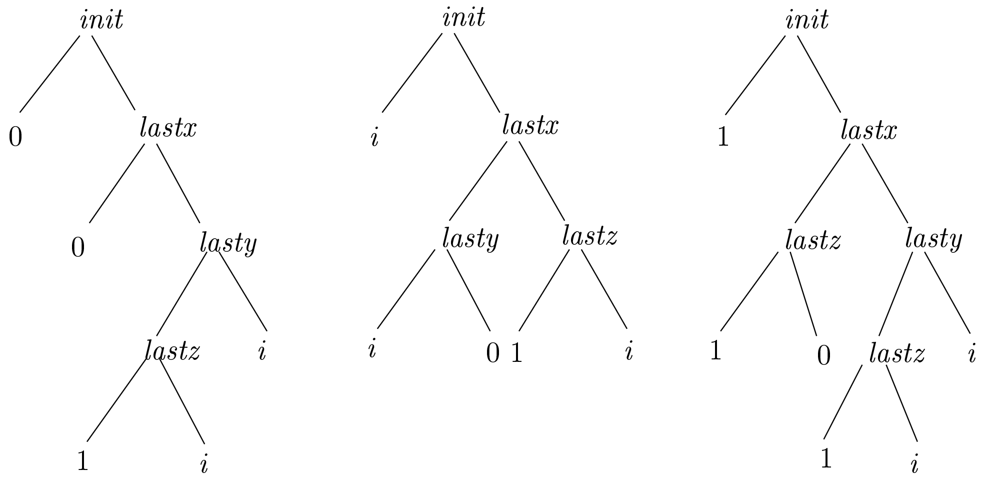


FIG. 11.1 – *Les fonctions* δ_{lastx}, δ_{lasty} *et* δ_{lastz}

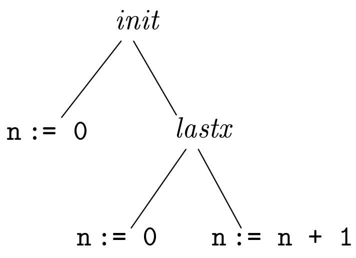


FIG. 11.2 – *La fonction* $\lambda_{\mathtt{n}}$

$$\begin{array}{rcl} \delta_{lastx}(q_0, i) & = & 0 \\ \delta_{lasty}(q_0, i) & = & i \\ \delta_{lastz}(q_0, i) & = & 1 \end{array}$$

La combinaison de ces fonctions fait apparaître deux nouveaux états accessibles :

$$\begin{array}{lrcl} & \Delta_{q_0}(i) & = & si \;\; i \;\; alors \;\; q_1 \;\; sinon \;\; q_2 \\ avec & q_1 & = & (0,0,1,1) \\ et & q_2 & = & (0,0,0,1) \end{array}$$

– Dans l'état $q_1 = (0,0,1,1)$, l'action associée au calcul de la sortie est `n := n + 1`. La transition est construite en combinant les fonctions de transition instantanées :

$$\begin{array}{rcl} \delta_{init}(q_1, i) & = & 0 \\ \delta_{lastx}(q_1, i) & = & 1 \\ \delta_{lasty}(q_1, i) & = & 1 \\ \delta_{lastz}(q_1, i) & = & 1 \end{array}$$

La transition est donc indépendante de l'entrée, et fait apparaître un nouvel état :

$$\begin{array}{lrcl} & \Delta_{q_1}(i) & = & q_3 \\ avec & q_3 & = & (0,1,1,1) \end{array}$$

– Dans $q_2 = (0,0,0,1)$, l'action est `n := n + 1`, et la transition dépend de la valeur de l'entrée. Un nouvel état q_4 est atteint :

$$\begin{array}{lrcl} & \Delta_{q_2}(i) & = & si \;\; i \;\; alors \;\; q_3 \;\; sinon \;\; q_4 \\ avec & q_4 & = & (0,0,1,0) \end{array}$$

– Dans l'état $q_3 = (0,1,1,1)$, l'action est `n := 0`, le calcul de la transition ne fait apparaître que des états déjà atteints :

$$\Delta_{q_3}(i) \;\; = \;\; si \;\; i \;\; alors \;\; q_1 \;\; sinon \;\; q_2$$

– Dans l'état $q_4 = (0,0,1,0)$, l'action est `n := n + 1`, un nouvel état est atteint :

$$\begin{array}{lrcl} & \Delta_{q_4}(i) & = & si \;\; i \;\; alors \;\; q_3 \;\; sinon \;\; q_5 \\ avec & q_5 & = & (0,0,0,0) \end{array}$$

– Enfin, le traitement de $q_5 = (0,0,0,0)$, dans lequel l'action est `n := n + 1`, ne fait apparaître que des états déjà traités :

$$\Delta_{q_5}(i) \;\; = \;\; si \;\; i \;\; alors \;\; q_3 \;\; sinon \;\; q_5$$

L'automate complet est représenté sur la figure 11.3.

Dans cet exemple, on obtient un automate de Moore, c'est-à-dire que les actions sont associées aux états. Cela vient du fait que la sortie est indépendante de la valeur des conditions. Il est clair que si les calculs à effectuer dépendent des conditions, on obtient un automate de Mealy, c'est-à-dire un automate où le calcul des sorties est associé aux transitions.

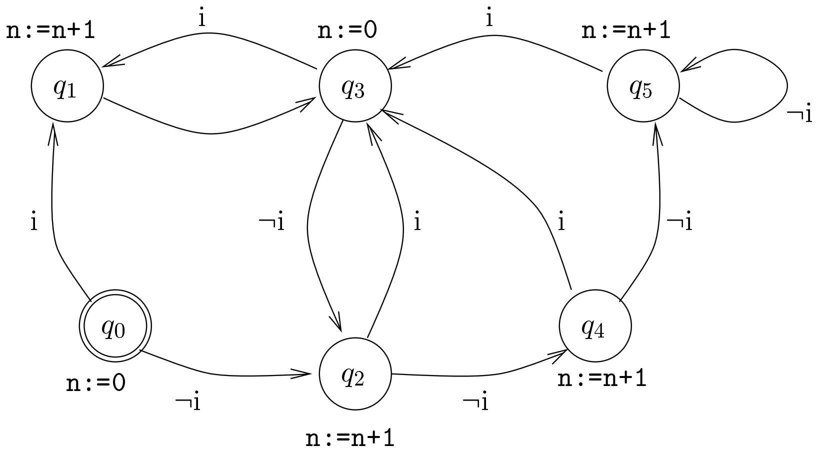


FIG. 11.3 – *L'automate de contrôle du programme* `Exemple`

11.2 Minimalité de l'automate

La taille du code exécutable est directement liée à la taille de l'automate de contrôle, que l'on mesure naturellement en nombre d'états et de transitions. Autrement dit, tout état inutile dans l'automate de contrôle fait croître inutilement la taille du code généré. Nous étudions dans cette section ce qu'est un état inutile, ainsi que les moyens de s'en débarrasser.

11.2.1 Bisimulation et minimisation

Dans l'automate de la figure 11.3, on peut considérer que les états q_0 et q_3 sont équivalents :

- ils exécutent la même action (`n := 0`),
- leurs successeurs pour une même valeur de l'entrée sont les mêmes.

L'équivalence des états q_4 et q_5 s'exprime de la même manière. Par contre, celle de q_2 avec ces deux derniers est un peu plus complexe : il faut les supposer *a priori* équivalents pour pouvoir identifier leurs transitions par ¬i. Enfin, l'état q_1 n'est équivalent à aucun autre :

- il ne calcule pas la même chose que les états q_0 et q_3,
- il calcule la même chose que les états q_2, q_4 et q_5, mais contrairement à ces derniers, la seule action possible après q_1 est `n := 0`.

Une telle équivalence est appelée une bisimulation [Par81]. Une bisimulation est relative à une équivalence initiale. Dans notre cas, nous considérons une équivalence initiale $\overset{\lambda}{\sim}$ basée sur l'égalité de tous les calculs :

$$q \overset{\lambda}{\sim} q' \quad ssi \quad \forall\, j = 1..k\ (\lambda_j(q) = \lambda_j(q'))$$

Une bisimulation est en quelque sorte une extension naturelle de cette équivalence qui prend en compte les transitions de l'automate. Pour que deux états se bisimulent il faut non seulement qu'ils exécutent les mêmes actions, mais qu'après un nombre quelconque de transitions, leurs successeurs respectifs exécutent aussi les mêmes actions :

$$\forall c_1, \cdots, c_m \in C^m \ si\ (q \xrightarrow{c_1} q_1 \cdots q_{m-1} \xrightarrow{c_m} q_m)\ et\ (q' \xrightarrow{c_1} q'_1 \cdots q'_{m-1} \xrightarrow{c_m} q'_m)\ alors\ q_m \overset{\lambda}{\sim} q'_m$$

Ces propriétés peuvent se résumer en définissant les bisimulations de $\overset{\lambda}{\sim}$ comme les solutions de l'équation de point fixe :

$$q \overset{\lambda}{\approx} q \quad ssi \quad (q \overset{\lambda}{\sim} q')\, et\ (\forall c \in C\ \Delta_q(c) \overset{\lambda}{\approx} \Delta_{q'}(c))$$

Cette équation ne définit pas une unique équivalence. En particulier l'égalité sur les états de l'automate est une bisimulation, mais elle ne permet pas de réduire l'automate. On va donc s'intéresser à la bisimulation la plus grossière, c'est-à-dire celle qui distingue le moins possible de classes d'états, tout en vérifiant l'équation de point fixe. Le problème de la minimisation d'un automate selon une bisimulation est tout à fait classique [PT87], et des outils performants existent (ALDEBARAN [Fer88]). Un outil spécifique à la minimisation des automates OC (*ocmin*) a été développé à partir d'ALDEBARAN.

11.2.2 Limitation de la méthode

Pour atteindre un code de taille "minimale", la méthode employée avec le compilateur LUSTREV2 consiste donc à générer avec un algorithme dirigé par les données un premier automate, qui est ensuite minimisé par *ocmin*. Or cette méthode se heurte à l'explosion du nombre d'états : l'automate non-minimal est souvent trop gros pour être construit, alors que l'automate minimal équivalent est de taille raisonnable. Nous sommes donc amenés à envisager de nouvelles méthodes, qui, en mêlant les notions de construction et de minimisation, permettent d'atteindre directement l'automate minimal.

11.3 Génération dirigée par la demande

L'algorithme dirigé par les données distingue des états qui ne diffèrent que sur la valeur de variables d'état ou de calculs qui n'influenceront jamais la valeur des sorties. D'où l'idée d'un algorithme dirigé par la demande, c'est-à-dire uniquement par la nécessité de calculer les sorties. Cet algorithme est une adaptation au cas de LUSTRE des principes de la *génération de modèles minimaux* [BFH+90].

11.3.1 Les super-états

Comme dans les algorithmes de minimisation, l'élément de base n'est plus un état en tant que valeur du vecteur de variables d'état ($q \in Q$), mais un ensemble d'états. D'où la notion de *super-états*, c'est-à-dire de classes d'états de Q que l'on considère momentanément comme équivalents. Une classe d'états peut être vue comme une propriété des variables d'état, c'est-à-dire une fonction de $Q \to B$. Cette capacité à assimiler un ensemble à sa fonction caractéristique va bien sûr nous permettre d'utiliser dans l'algorithme toutes les possibilités de la représentation canonique à base de *bdd*.

11.3.2 Les préconditions

Dans l'algorithme dirigé par les données, la construction des transitions consiste à déterminer la valeur future de chaque variable d'état. Dans notre cas, le problème est plus complexe puisqu'il va falloir déterminer la valeur future d'une *propriété sur les variables d'état*. Ce type de calcul est rendu possible en introduisant la notion de *précondition*. La précondition d'une classe d'états caractérisée par la propriété P, est l'ensemble de tous les couples de $Q \times C$ qui conduisent dans un état de P :

$$Precond(P) \quad = \quad \{(q,c) \in Q \times C \ / \ \Delta_q(c) \in P\}$$

Pour calculer les préconditions, considérons tout d'abord le cas de la classe P_i des états dans lesquels la $i^{\text{ème}}$ variable d'état vaut 1. La précondition de P_i est exactement l'ensemble des couples (q,c) qui conduisent dans un état où la $i^{\text{ème}}$ variable d'état vaut 1, c'est-à-dire les couples pour lesquels la fonction δ_i est évaluée à 1. En confondant l'ensemble $Precond(P_i)$ et sa fonction caractéristique, on peut donc écrire :

$$Precond(P_i) \quad = \quad \delta_i$$

L'ensemble $Precond(P \cup P')$ est celui des couples (q,c) qui conduisent dans P ou dans P', donc :

$$Precond(P \cup P') \quad = \quad Precond(P) \cup Precond(P')$$

Un raisonnement similaire peut être suivi pour les autres opérations ensemblistes (intersection et complémentation), à condition de se limiter à des automates déterministes (ce qui est notre cas). Prenons le contre-exemple d'un automate non-déterministe (figure 11.4) pour lequel :

$$Precond(P \cap P') = \emptyset \not\ni (q,c)$$

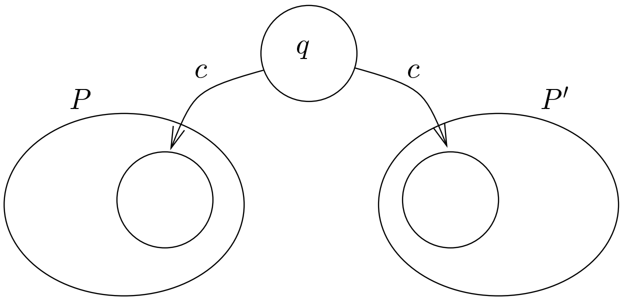


FIG. 11.4 – *Un automate non-déterministe*

Pour les automates déterministes, la fonction *Precond* est un homomorphisme de 2^Q dans $2^{Q \times C}$. En assimilant les ensembles et leurs fonctions caractéristiques, le calcul des préconditions se ramène donc à une composition de fonctions booléennes :

$$Precond(P) \quad = \quad P(\delta_1, \cdots, \delta_n)$$

Nous sommes donc ramenés une fois de plus à un problème de manipulation booléenne, dont l'efficacité est garantie par la représentation à base de *bdd*.

11.3.3 Le parcours des super-états

La génération de l'automate minimal débute en considérant que tous les états sont équivalents : cela correspond à un unique super-état caractérisé par la propriété “toujours vraie”. Au cours de la génération, on peut être amené à diviser une classe :

– si elle contient des états qui diffèrent sur le calcul de la sortie (partition initiale)

– si elle contient des états qui diffèrent sur le calcul de leurs successeurs (aspect bisimulation).

Le déroulement de l'algorithme est donc rythmé par une suite de scissions distinguant des classes d'états qui ne peuvent plus être considérés comme équivalents. Parmi ces classes, seules doivent être considérées celles qui sont accessibles depuis l'état initial.

11.3.4 Exemple

Nous reprenons ici le programme `Exemple`, déjà traité avec l'algorithme dirigé par les données (page 87). Le but de cet exemple est de présenter de manière intuitive le déroulement de la génération d'automate. En particulier, nous ne faisons pas référence à la représentation canonique des fonctions booléennes et des actions gardées, mais utilisons une notation polynômiale plus lisible. Par exemple, l'action gardée λ_n (figure 11.2 page 88) est donnée par :

$\lambda_n = si\ (init \vee lastx)\ alors$ `(n := 0)` $sinon$ `(n := n + 1)`

La construction de l'automate débute avec une classe P caractérisée par la propriété *true*, dans laquelle l'action associée à la sortie est encore inconnue, et dont les successeurs sont évidemment dans P. Ce "pseudo-automate" initial est représenté sur la figure 11.5.

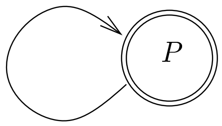


FIG. 11.5 –

La première étape de l'algorithme consiste à regarder si tout les états de P sont équivalents selon la partition initiale $\stackrel{\lambda}{\sim}$. La réponse est non, puisque P contient des états vérifiant $init \vee lastx$ pour lesquels le calcul de la sortie est `n := 0`, et des états vérifiant $\neg init \wedge \neg lastx$ pour lesquels le calcul est `n := n + 1`. La classe P est donc scindée en deux sous-classes :

– $P_1 = init \vee lastx$

– $P_0 = \neg init \wedge \neg lastx$

La seule classe accessible pour l'instant est P_1, puisqu'elle contient l'état initial. La valeur de la sortie y est connue, il faut maintenant évaluer ses transitions.

Initialement, tout ce qu'on sait des transitions, c'est qu'elle conduisent invariablement dans les états de P. La scission de cette dernière nous oblige à distinguer les transitions qui conduisent dans P_1 de celles qui conduisent dans P_0. On sait que P_1 est constituée de l'état initial et de tous les états dans lesquels la variable d'état *lastx* est vraie. Par définition, aucune transition ne conduit à l'état initial. La précondition de P_1 est donc réduite aux couples (q, c) qui conduisent dans un état vérifiant *lastx*, c'est-à-dire les couples pour lesquels $\delta_{lastx}(q, c) = 1$. La fonction

δ_{lastx} (figure 11.1 page 88), interprétée comme une fonction de $Q \to (C \to B)$, peut s'écrire sous la forme :

$$\begin{aligned} \delta_{lastx} = \quad & si\ (init \vee lastx)\ alors\ 0 \\ & sinon\ si\ (lasty \wedge lastz)\ alors\ 1 \\ & sinon\ i \end{aligned}$$

Dans tous les états de P_1, et indépendamment de l'entrée, la fonction δ_{lastx} est donc évaluée à 0. Il en résulte que tous les états P_1 conduisent invariablement dans un état de P_0 (figure 11.6).

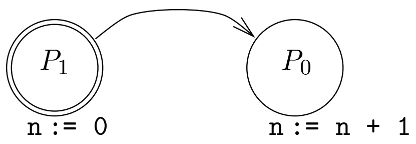


FIG. 11.6 –

On est maintenant certain que des états de la classe P_0 sont accessibles depuis l'état initial. Cette classe doit donc être examinée. L'action associée au calcul de la sortie est connue (`n := n + 1`), il reste à évaluer les transitions. Comme pour P_1, les transitions vont dépendre de la valeur de δ_{lastx}, or :

- P_0 contient des états vérifiant $lasty \wedge lastz$, pour lesquels δ_{lastx} vaut invariablement 1,
- P_0 contient des états vérifiant $\neg lasty \vee \neg lastz$, pour lesquels δ_{lastx} vaut i.

La classe P_0 doit donc être scindée en deux sous-classes :

- $P_{01} = P_0 \wedge (lasty \wedge lastz)$ qui est la sous-classe des états de P_0 dont tous les successeurs sont dans P_1,
- $P_{00} = P_0 \wedge \neg(lasty \wedge lastz)$ qui est la sous-classe des états de P_0 qui par i conduisent dans P_1, et par $\neg i$ conduisent dans P_0.

Les connaissances que nous avons de l'automate minimal sont représentées sur la figure 11.7. On voit que les transitions qui conduisent vers les états de P_0 ne sont que partiellement connues, et doivent être complétées pour "choisir" entre les deux sous-classes P_{01} et P_{00}. Pour cela, on ne va pas remettre en cause les traitements déjà effectués pour choisir entre P_1 et P_0 : seul un complément d'information concernant la valeur de $lasty \wedge lastz$ est nécessaire. On est donc amené à déterminer la précondition de cette propriété, c'est-à-dire les couples (q, c) dans lesquels les fonctions δ_{lasty} et δ_{lastz} sont toutes deux évaluées à vrai.

La seule classe clairement accessible est de nouveau P_1, on sait déjà que tous les successeurs sont dans la classe P_0, reste donc à évaluer la précondition $\delta_{lasty} \wedge \delta_{lastz}$ pour distinguer celles qui vont dans P_{01} de celles qui vont dans P_{00}. Cette précondition est une nouvelle fonction des éléments de contrôle, dont une représentation possible est :

$$\begin{aligned} \delta_{lasty} \wedge \delta_{lastz} = \quad & si\ (init \vee lastx)\ alors \\ & \quad (si\ (init \vee (lasty \wedge lastz))\ alors\ i\ sinon\ 0) \\ & sinon \\ & \quad (si\ (lasty \wedge lastz)\ alors\ 1\ sinon\ i) \end{aligned}$$

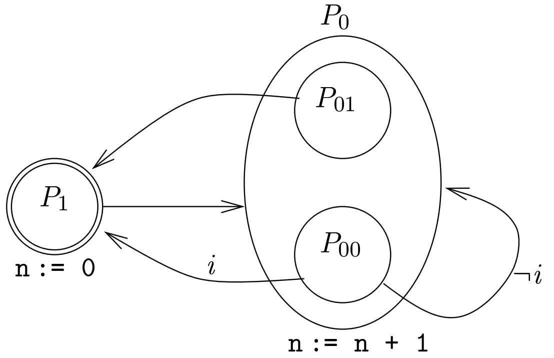


FIG. 11.7 –

Cette représentation a été choisie pour mettre en évidence le rôle de la propriété $P_1 = init \vee lastx$ dans l'évaluation de la précondition. Il apparaît que la classe P_1 contient des états qui vérifient $init \vee (lasty \wedge lastz)$ pour lesquels cette fonction est évaluée à i, et des états vérifiant $\neg(init \vee (lasty \wedge lastz))$ pour lesquels elle est évaluée à 0.

La classe P_1 doit donc être scindée en :

– $P_{11} = P_1 \wedge (init \vee (lasty \wedge lastz))$ qui est la sous-classe des états de P_1 dont tous les successeurs sont, suivant la valeur de i, dans P_{01} ou P_{00},

– $P_{10} = P_1 \wedge \neg(init \vee (lasty \wedge lastz))$ qui est la sous-classe des états de P_1 dont les successeurs sont toujours dans P_{00}.

L'automate de contrôle a donc potentiellement quatre états, et ses transitions sont partiellement connues (figure 11.8). En particulier, les transitions qui conduisent dans P_1 devront être complétées pour répondre à la précondition de $(init \vee (lasty \wedge lastz))$. Aucune transition ne conduisant dans l'état initial, la précondition de cette propriété se réduit à celle de $(lasty \wedge lastz)$. Cette précondition est déjà connue, elle correspond aux couples (q, c) pour lesquels $\delta_{lasty} \wedge \delta_{lastz}$ est évaluée à vraie.

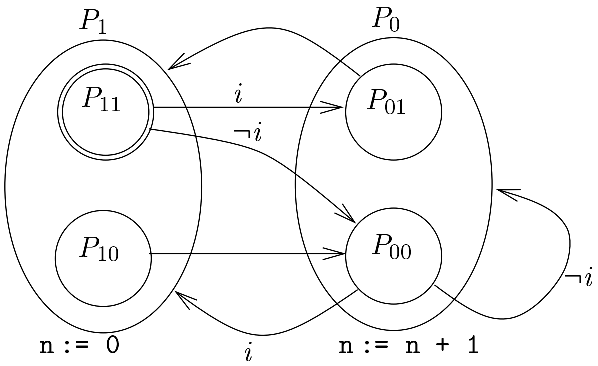


FIG. 11.8 –

La classe de l'état initial, P_{11}, peut être maintenant considérée comme stable car ses transitions sont connues dans l'automate courant. De celle-ci, on peut atteindre les classes P_{01} et P_{00}

qui doivent de nouveau être considérées.

On sait déjà que tous les successeurs de P_{01} sont dans P_1, reste donc à "choisir" entre les deux sous-classes de cette dernière en évaluant $\delta_{lasty} \wedge \delta_{lastz}$. Or, cette classe ne contient qu'un état $(\neg init \wedge \neg lastx \wedge lasty \wedge lastz)$ pour lequel cette fonction est évaluée à 1. Il en résulte que la classe P_{01} conduit invariablement dans la classe P_{11} déjà traitée.

Pour déterminer les transitions de P_{00} vers les deux sous-classes de P_1, on doit évaluer la précondition $\delta_{lasty} \wedge \delta_{lastz}$. Or cette dernière se réduit à i pour tous les états de P_{00}. Comme on sait déjà que les états de P_1 ne peuvent être atteints à partir de P_{00} que si i est vraie, il en résulte que seule la classe P_{11} est accessible. La même précondition doit être évaluée pour déterminer les successeurs de P_{00} dans les deux sous-classes de P_0. Or on sait que ces dernières ne sont atteintes que si i est fausse. Donc la seule sous-classe de P_0 accessible depuis P_{00} est P_{00}.

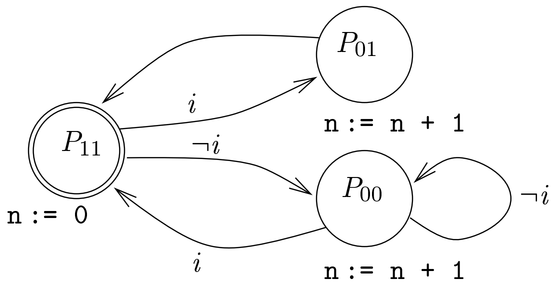


FIG. 11.9 –

Toutes les classes accessibles depuis l'état initial sont donc traitées : l'automate minimal est atteint (figure 11.9).

11.3.5 Evaluation de la sortie

L'algorithme dirigé par la demande est donc rythmé par une suite de scissions qui, de proche en proche, conduisent à l'automate minimal. Comme nous l'avons vu dans l'exemple, certaines de ces scissions sont motivées par la nécessité d'évaluer la sortie du programme. Le but de cette section est de préciser en quoi consiste cette évaluation.

Nous nous limitons pour l'instant au cas d'un programme ne comportant qu'une sortie. L'action gardée associée à la sortie est une fonction $\lambda \in Q \to I$ où I est l'ensemble des actions instantanées $(I = (C \to A))$. L'équivalence initiale dont on doit trouver la plus grande bisimulation est :

$$q \overset{\lambda}{\sim} q' \; ssi \; (\lambda(q) = \lambda(q'))$$

Chaque fois qu'on traite une nouvelle classe accessible P, on est amené à se demander si elle est bien compatible avec $\overset{\lambda}{\sim}$, c'est-à-dire si tous ses états sont équivalents selon $\overset{\lambda}{\sim}$. L'évaluation de la sortie dans une classe P consiste donc à chercher une action instantanée x telle que :

$$\forall \; q \in P \; (\lambda(q) = x)$$

– Si cette action existe, cela signifie que la classe P est bien compatible avec $\overset{\lambda}{\sim}$ et qu'il n'y a donc pas lieu de la remettre en cause pour l'instant. L'action instantanée correspondante doit en outre être conservée car c'est à partir de celle-ci que le code séquentiel va être produit.

– Sinon, la classe d'états doit être remise en cause, de manière à converger vers la partition initiale.

Quand une classe est remise en cause, on peut par exemple la remplacer par le quotient $P/\overset{\lambda}{\sim}$, dont chaque classe est trivialement compatible avec la partition initiale. Mais cette solution va à l'encontre de l'aspect "paresseux" de l'algorithme : elle distingue a priori des états qui ne seront peut-être jamais atteints à partir de l'état initial. Nous lui préférons une méthode qui procède uniquement par division binaire :

Soit une classe P pour laquelle l'équation $\lambda(q) = x$ n'a pas de solution unique. Il existe une action instantanée $v \in (C \to A)$ telle que :

$$\begin{array}{rrcccc} & S_v & = & \{q \in Q/\lambda(q) = v\} & & \\ \textit{avec} & P_1 & = & P \wedge S_v & \neq & \emptyset \\ \textit{et} & P_0 & = & P \wedge \neg S_v & \neq & \emptyset \end{array}$$

Moyennant le choix d'une d'une telle action v, la classe est donc scindée en deux parties non vides, dont au moins une (P_1) est compatible avec la partition initiale. La propriété S_v correspondante est appelée le `pivot` de la scission.

Il est important de souligner que le choix d'une action v comme pivot de la scission n'a aucune influence sur le résultat final : on obtient toujours au bout du compte l'automate minimal. Cependant, un choix judicieux peut permettre de converger plus vite vers cet automate minimal. Il existe par exemple une heuristique simple qui concerne le cas de l'état initial : si la classe à scinder contient l'état initial, on choisira comme pivot de la scission l'action instantanée correspondant à celui-ci. De cette manière, on est certain qu'au moins une des deux sous-classes est à la fois accessible et compatible avec la partition initiale (celle de l'état initial).

11.3.6 L'évaluation des transitions

Comme pour l'algorithme dirigé par les données, le changement d'état va être interprété comme le calcul d'une donnée fictive NEXT-STATE. A ce calcul correspond une action gardée Δ, que chaque classe se doit d'évaluer au même titre que la sortie. Le problème est que cette action gardée est susceptible d'être modifiée au cours du traitement :

– Initialement, tout les états sont supposés équivalents ; la connaissance que nous avons de l'automate minimal se limite à une classe $P = \textit{true}$, et le calcul du NEXT-STATE se réduit donc à :

$$\Delta_0(q, c) = \texttt{goto:}P$$

– Après une première scission, motivée par la nécessité d'évaluer la sortie, la classe initiale P est scindée selon une propriété S en $P_1 = S$ et $P_0 = \neg S$, et l'action gardée associée au NEXT-STATE est modifiée en conséquence :

$$\Delta_1(q, c) = \textit{si Precond}(S) \textit{ alors } \texttt{goto:}P_1 \textit{ sinon } \texttt{goto:}P_0$$

Le nouvel automate comporte donc deux classes, dans lesquelles l'action gardée Δ_1 devra être évaluée.

- L'évaluation du Δ courant, comme celui de la sortie, peut provoquer à son tour de nouvelles scissions, et, à chaque scission, l'action Δ doit être remise à jour. Plaçons-nous à un stade du traitement où le calcul du NEXT-STATE est donné par l'action gardée Δ_i. Un super-état P_α est scindé selon la propriété S en $P_{\alpha 1} = P_\alpha \wedge S$ et $P_{\alpha 0} = P_\alpha \wedge \neg S$. La nouvelle action Δ_{i+1} est construite à partir de Δ_i en remplaçant l'action $\texttt{goto:}P_\alpha$ par une action gardée conduisant, suivant la valeur de $\mathit{Precond}(S)$ dans l'une ou l'autre des sous-classes de P_α :

$$\Delta_{i+1} = \Delta_i[\mathit{si}\ \mathit{Precond}(S)\ \mathit{alors}\ \texttt{goto:}P_{\alpha 1}\mathit{sinon}\ \texttt{goto:}P_{\alpha 0}/\texttt{goto:}P_\alpha]$$

Dans notre exemple, les actions Δ sont successivement :

- initialement :

$$\Delta_0 = \texttt{goto:}P$$

- après la scission de la classe P selon la propriété $\mathit{init} \vee \mathit{lastx}$, (dont δ_{lastx} est la précondition) :

$$\Delta_1 = \mathit{si}\ (\delta_{lastx})\ \mathit{alors}\ \texttt{goto:}P_1\ \mathit{sinon}\ \texttt{goto:}P_0$$

- après la scission de la classe P_0 selon la propriété $\mathit{lasty} \wedge \mathit{lastz}$, (dont la précondition est $\delta_{lasty} \wedge \delta_{lastz}$) :

$$\begin{aligned}\Delta_2 = \ & \mathit{si}\ (\delta_{lastx})\ \mathit{alors}\ \texttt{goto:}P_1 \\ & \mathit{sinon}\ (\\ & \quad \mathit{si}\ (\delta_{lasty} \wedge \delta_{lastz})\ \texttt{goto:}P_{01} \\ & \quad \mathit{sinon}\ \texttt{goto:}P_{00})\end{aligned}$$

- enfin, après la scission de la classe P_1 selon la propriété $\mathit{init} \vee (\mathit{lasty} \wedge \mathit{lastz})$, (dont la précondition est $\delta_{lasty} \wedge \delta_{lastz}$) :

$$\begin{aligned}\Delta_3 = \ & \mathit{si}\ (\delta_{lastx})\ \mathit{alors}\ (\\ & \quad \mathit{si}\ (\delta_{lasty} \wedge \delta_{lastz})\ \texttt{goto:}P_{11} \\ & \quad \mathit{sinon}\ \texttt{goto:}P_{10}) \\ & \mathit{sinon}\ (\\ & \quad \mathit{si}\ (\delta_{lasty} \wedge \delta_{lastz})\ \texttt{goto:}P_{01} \\ & \quad \mathit{sinon}\ \texttt{goto:}P_{00})\end{aligned}$$

11.3.7 Implémentation du calcul des transitions

Pour gérer efficacement les évaluations successives des actions Δ_i, le compilateur doit respecter les règles suivantes :

– Quand la fonction Δ est modifiée après la scission d'une classe P, elle ne doit pas être réévaluée dans tous les super-états, mais uniquement dans ceux qui sont concernés, c'est-à-dire ceux qui ont effectivement des successeurs dans P.

– De plus, quand une classe doit réévaluer la fonction Δ, seule une partie de ses transitions est concernée : celle qui conduit dans la classe qui vient d'être scindée.

Pour respecter la première règle, il suffit de gérer une relation qui associe à chaque super-état ses prédécesseurs actuellement connus.

Pour obéir à la deuxième, la solution retenue consiste à représenter le Δ courant comme un arbre de préconditions qui reflète les scissions déjà effectuées. La figure 11.10 représente l'état de cet arbre pour la fonction Δ_3 de l'exemple. Les nœuds de l'arbre sont étiquetés par la précondition à évaluer. Par convention, la branche gauche correspond à la valeur “vrai” de la précondition, la branche droite à la valeur “faux”. Les transitions partiellement connues sont représentées par des flèches continues. Elles correspondent au “pseudo-automate” représenté sur la figure 11.8.

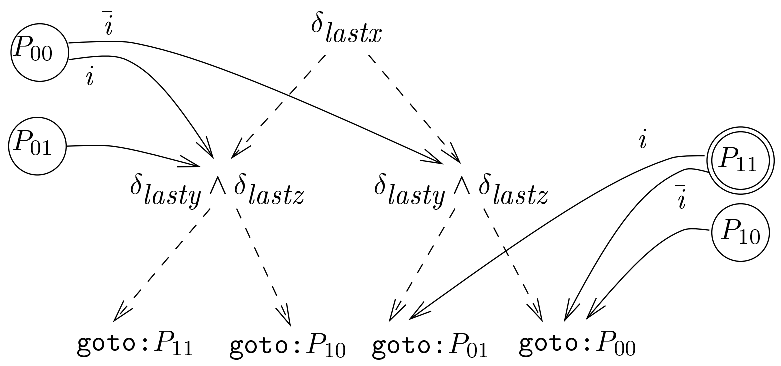


FIG. 11.10 – *L'arbre de la fonction* Δ_3

11.3.8 Programmes comportant plusieurs calculs

Nous nous intéressons maintenant au cas des programmes quelconques, comportant plusieurs calculs. En première approximation, nous avions présenté une partition initiale des états basée sur l'égalité de chaque action instantanée (11.2.1), c'est-à-dire sur l'évaluation de chacun des calculs. L'algorithme dirigé par la demande va nous permettre d'affiner cette partition : deux états ne doivent en effet être distingués que s'ils diffèrent sur l'évaluation d'un calcul qui influence la valeur courante ou future d'une sortie.

Globalement, les calculs se divisent en :

– des sorties, dont l'évaluation est toujours nécessaire,

- des calculs accessoires dont le résultat influence éventuellement la valeur courante des sorties,
- des calculs accessoires dont le résultat influence éventuellement la valeur future des sorties.

Ces différents cas ne sont bien sûr pas exclusifs. Seule l'évaluation des sorties est a priori nécessaire dans une classe ; les calculs accessoires doivent être évalués à la demande. On associe donc à chaque classe un ensemble *to-calc* des calculs nécessaires, initialisé avec l'ensemble des sorties. Au cours de la génération, l'évaluation d'un calcul nécessaire dans un super-état P peut faire apparaître une action instantanée qui dépend :

- du résultat courant d'un calcul accessoire, celui-ci est alors ajouté à l'ensemble *to-calc* de la classe P,
- du résultat précédent d'un calcul accessoire. Celui-ci est alors ajouté à l'ensemble *to-calc* de chacun des prédécesseurs actuellement connus de P, qui devront être réexaminés. De plus, certains prédécesseurs de P ne sont peut-être pas encore connus : on doit donc associer à P un ensemble *to-calc-in-preds* qui doit être inclus aux *to-calc* de chaque nouveau prédécesseur de P.

11.4 Comparaison des résultats

Soit un programme donné, nous notons Γ_{data} l'automate obtenu avec l'algorithme dirigé par les données. La minimisation de cet automate par le programme *ocmin* produit l'automate :

$$\Gamma_{\text{datamin}} = \Gamma_{\text{data}}/\overset{\lambda}{\approx}$$

L'automate Γ_{demand} est obtenu avec l'algorithme dirigé par la demande.

Dans le cas d'un programme ne comportant qu'un seul calcul, on peut montrer trivialement que :

$$\Gamma_{\text{datamin}} = \Gamma_{\text{demand}}$$

Mais ceci n'est pas vrai dans le cas général. En effet, Γ_{demand} peut être vu comme le résultat de la minimisation de Γ_{data} selon une équivalence plus grossière que $\overset{\lambda}{\approx}$.

11.4.1 Le prédicat *demand*

L'équivalence qui est à la base de l'algorithme dirigé par la demande n'est pas basée sur l'égalité de tous les calculs, mais uniquement de ceux qui influencent ou influenceront la valeur des sorties. On est donc ramené à un problème de dépendances entre les calculs ; mais il s'agit ici de dépendances relatives à un état donné :

- Le calcul i dépend du calcul j dans l'état q si l'action instantanée $\lambda_i(q)$ contient une référence instantanée au résultat du calcul j. On note : $i \text{ dep}_q\ j$.
- Le calcul i pré-dépend du calcul j dans l'état q si l'action instantanée $\lambda_i(q)$ contient une référence au résultat précédent du calcul j. On note : $i \text{ predep}_q\ j$.

Comme pour les dépendances globales, il s'agit de dépendances purement syntaxiques. A partir de ces relations, on définit le prédicat *demand*(q,i), qui vaut vrai si l'évaluation du calcul i est nécessaire dans l'état q. On fait intervenir le prédicat *is-out*(i), qui vaut vrai si le calcul i est de mode OUTSIG ou PURE-OUTSIG, ainsi que l'ensemble *succ*(q) des successeurs direct de l'état q :

$$\begin{array}{rcl} \mathit{demand}(q,i) & = & \mathit{is\text{-}out}(i) \\ & \vee & \exists j /\ \mathit{demand}(q,j) \wedge j \ \mathrm{dep}_q\ i \\ & \vee & \exists j\ \exists q' /\ q' \in \mathit{succ}(q) \wedge \mathit{demand}(q',j) \wedge j\ \mathrm{predep}_{q'}\ i \end{array}$$

11.4.2 La bisimulation associée à la demande

Pour chaque calcul i, on peut construire une nouvelle action gardée λ'_i :

$\lambda'_i(q) \quad = \quad \mathit{si}\ \mathit{demand}(q,i)\ \mathit{alors}\ \lambda_i\ \mathit{sinon}\ \mathtt{nop}$

De manière similaire à la définition de $\overset{\lambda}{\sim}$, on définit une relation d'équivalence basée sur ces nouvelles actions gardées :

$q \overset{\lambda'}{\sim} q' \quad \mathit{ssi} \quad \forall\ j = 1 \cdots k\ (\lambda'_j(q) = \lambda'_j(q'))$

On a trivialement :

$q \overset{\lambda}{\sim} q' \quad \Rightarrow \quad q \overset{\lambda'}{\sim} q'$

Dans l'algorithme dirigé par la demande, tout se passe comme si on opérait une minimisation basée sur cette équivalence initiale ; soit $\overset{\lambda'}{\approx}$ la plus grande bisimulation de $\overset{\lambda'}{\sim}$, on a donc :

$\Gamma_{\mathrm{data}}/\overset{\lambda'}{\approx} \quad = \quad \Gamma_{\mathrm{demand}}$

on montre trivialement que :

$\forall q, q' \in Q \quad (q \overset{\lambda}{\approx} q') \Rightarrow (q \overset{\lambda'}{\approx} q')$

L'équivalence $\overset{\lambda'}{\approx}$ est plus grossière que $\overset{\lambda}{\approx}$, elle a donc potentiellement moins de classes. On en déduit que le nombre d'états de Γ_{demand} est inférieur ou égal à celui de l'automate $\Gamma_{\mathrm{datamin}}$.

11.4.3 Expérimentation

La table 11.11 présente les résultats obtenus avec les différentes méthodes de compilation, pour un programme réalisant une montre digitale avec réveil et chronomètre. Le nombre d'instructions "`goto`" n'est qu'une approximation du nombre de transitions. Pour deux automates équivalents, qui ont le même nombre de transitions "théoriques", cette mesure peut varier à cause de différences dans l'ordre d'ouverture des tests.

A propos des temps de compilation, on remarque que la construction de Γ_{demand} est plus rapide que celle de Γ_{data}. Ce résultat semble contraire à l'intuition, quand on connaît la complexité des manipulations symboliques mises en œuvre dans l'algorithme dirigé par la demande. Il s'explique cependant assez bien :

- L'automate produit est beaucoup plus petit. A ce titre, le "gain" en nombre de transitions est plus significatif que le gain en nombre d'états : On montre facilement que si le rapport du nombre d'états entre Γ_{data} et Γ_{demand} est k, celui des transitions est de l'ordre de k^2 (ce qu'on peut vérifier sur l'exemple).

– La construction des transitions dans l'algorithme dirigé par les données est de loin l'opération la plus coûteuse (§11.1.2).

– Dans l'algorithme dirigé par la demande, il est très difficile d'estimer le temps consacré à la construction des transitions. Cependant, on a vu que la structure arborescente des scissions binaires permettait d'optimiser le calcul des préconditions (§11.10).

Pour ce qui est de la taille du code, on observe une différence sensible entre les résultats Γ_{demand} et $\Gamma_{\mathrm{datamin}}$. Cette différence s'explique en partie par le fait que certains calculs accessoires ne sont pas générés par l'algorithme dirigé par la demande, mais surtout par l'influence de la séquentialisation. Cette influence est très nette sur cet exemple, où elle est mise en évidence par la différence entre les nombres d'instructions `goto`.

Programme `MONTRE`	data-driven	data-driven + minimisation	demand-driven
Temps de compilation	8,1 sec. cpu	41,5 sec. cpu	6,8 sec. cpu
Nombre d'états	81	41	41
Nombre de `goto`	1163	474	342
Taille du code exécutable	30,94 Ko	19,15 Ko	16,51 Ko

FIG. 11.11 – *Résultats de compilation du programme* `MONTRE`

Chapitre 12

Séquentialisation

La génération d'automate, qu'elle soit dirigée par les données ou par la demande, produit le même type de résultat. A un état est en effet associée une mémoire qui fait correspondre à chaque calcul une action gardée simplifiée. Une telle action ne dépend que des conditions, et est appelée action instantanée. Dans le cas d'un état produit par l'algorithme dirigé par la demande, certains calculs inutiles à l'évaluation courante et future des sorties n'ont pas été évalués. On contourne ce problème en leur associant l'action `nop`. Dans ce chapitre, nous adoptons une notation textuelle des actions gardées : un nœud $\begin{array}{c} \mathrm{x} \\ / \ \backslash \\ \alpha \quad \beta \end{array}$ est noté "x?(α)(β)".

Le rôle de la séquentialisation est d'intégrer toutes ces actions instantanées dans un programme séquentiel. La structure d'un tel programme est définie dans le paragraphe 7.3.3 page 56 : il s'agit d'une séquence "branchue" se terminant par une action `goto:`. Un des rôles de la séquentialisation est donc d'ouvrir et de fermer des tests de manière à factoriser plusieurs actions sous un même test de contrôle. Pour être correcte, la séquence doit en outre respecter des contraintes liées aux dépendances entre les actions.

12.1 Les tests

Les tests de contrôle du programme vont bien sûr être basés sur les conditions qui apparaissent dans les actions gardées. La règle générale pour une action gardée de la forme "`x?(A)(B)`" consiste à intégrer cette action à un test de contrôle sur la condition "`x`" :

```
if x then
    ...; A; ...
else
    ...; B; ...
```

Certaines variantes peuvent être envisagées. On peut par exemple traduire une action gardée par une expression conditionnelle plutôt que par une instruction conditionnelle. Une autre variante consiste à combiner les conditions avec des opérateurs (`and`, `or` ...).

12.1.1 Expression conditionnelle ou test de contrôle

Une action gardée du type “x?(r:=a)(r:=b)” peut être intégrée à un test sur la condition `x` :

```
if x then { ...; r:=a; ... } else { ...; r:=b; ... }
```

mais peut aussi être traduite comme une action de base en utilisant une expression conditionnelle :

```
r:=if x then a else b;
```

La deuxième solution semble donner un code plus compact. En fait ceci n'est vrai que si le calcul considéré est le seul à être généré sous la condition “x”. En effet, si plusieurs calculs peuvent être effectués sous la condition “x”, on aura évidemment intérêt à privilégier la factorisation.

Pour simplifier, nous fixons donc comme contrainte que les conditions ne peuvent intervenir que dans des instructions conditionnelles. De cette manière, les seules actions qui peuvent être générées sont celles qui apparaissent en feuilles des actions gardées. Seule la distinction entre l'action `nop` et les autres actions est alors nécessaire au cours de la séquentialisation.

12.1.2 Combinaison des conditions

Une action gardée de la forme “x?(y?(A)(B))(B)” peut être intégrée à un test imbriqué sur les conditions “x” et “y” :

```
if x then
    if y then A else B
else B
```

mais on peut aussi construire un test portant sur la valeur de “`x and y`” :

```
if (x and y) then A
else B
```

La deuxième solution est sans doute aussi bonne que l'autre en ce qui concerne la vitesse d'exécution, et le code produit est plus compact. Mais là encore, la factorisation peut remettre en question cet avantage ; c'est par exemple le cas si le calcul peut être inséré dans une séquence qui se factorise bien dans le premier cas, et pas dans le second :

```
if x then
    if y then E; A; F else G; B; H
else I; B; J
```

De plus, l'opération qui consiste à reconnaître qu'un calcul dépend d'une combinaison des conditions, et non pas de chaque condition, est intrinsèquement complexe. Dans notre exemple, il faut tout d'abord remarquer que l'action gardée G peut prendre deux valeurs `A` et `B`. On en déduit alors que le code à générer peut être de la forme :

```
if T(x,y) then A else B
```

où `T` est la fonction caractéristique de l'ensemble :

$$T = \{c \in C / G(c) = \mathtt{A}\}$$

Une forme canonique de cette fonction peut être trivialement déduite de l'action gardée, en remplaçant les occurrences de l'action "`A`" par `true`, et celles de "`B`" par `false` :

(x?(y?(`true`)(`false`))(`false`))

Pour être exploitable dans le code, cette fonction doit alors être traduite en expression algébrique qui soit la plus simple possible ; par exemple : "x and y". Cette dernière opération, qui peut paraître triviale sur notre exemple, est en fait très complexe dans le cas général.

On peut enfin ajouter qu'il n'est pas possible de combiner les entrées pures avec les autres conditions. En effet, les entrées pures ne peuvent intervenir dans le programme que dans un test de contrôle du type "`present :`".

Nous introduisons donc une nouvelle contrainte pour imposer que chaque test porte sur la valeur d'une seule condition. Grâce à cette contrainte, la génération d'un test de contrôle devient très simple. On sait en effet que toute condition est soit une entrée pure, soit un calcul auquel est associé un registre. Tous les tests sont donc de la forme : "`present:` *signal-index*" ou "`if:` *var-index*".

12.2 Ordonnancement des actions

12.2.1 Dépendances et pré-dépendances

La génération du code séquentiel doit répondre à un impératif lié aux dépendances entre les actions. Une action I associée aux registres $r_1, \ldots, r_n$ ne peut être générée que si :

– tous les registres dont elle dépend ont déjà été mis à jour (dépendance instantanée),

– et toutes les actions qui dépendent de la valeur précédente d'au moins un des registres $r_1, \ldots, r_n$ ont été générées (pré-dépendance).

12.2.2 Ordre global ou local

Ces relations découlent naturellement des dépendances entre les calculs. Elles sont cependant plus grossières ; en effet, si un calcul X dépend d'un calcul Y, cela signifie qu'il existe des états dans lesquels l'action associée à X dépend du résultat de l'action associée à Y. Un raisonnement similaire peut être suivi avec les pré-dépendances.

Il en découle que l'ordre des calculs déduit des dépendances et des pré-dépendance est plus contraignant que celui des actions. Un algorithme de séquentialisation basé sur cet ordre global impose dans certains états des contraintes injustifiées ; ces contraintes sont susceptibles d'écarter des possibilités d'ordonnancement intéressantes.

La recherche d'ordres locaux à chaque état a cependant été rejetée pour son coût. La séquentialisation est donc basée sur l'ordre global des calculs. Cet ordre partiel, noté $\triangleleft$, résulte de la combinaison des relations de dépendance et de pré-dépendance. La sémantique intuitive de $x \triangleleft\, y$ est : "x doit être généré avant y". On utilise aussi la relation inverse notée $\triangleright$.

12.3 Factorisation

Les algorithmes de séquentialisation doivent poursuivre deux buts :

- le premier est un critère de correction : il faut en effet assurer que la séquence des actions respecte l'ordre global des calculs,
- l'autre est du domaine de l'optimisation : il concerne la factorisation des actions sous un même test de contrôle.

12.3.1 Influence de l'ordre des calculs

L'ordre des calculs est partiel, on peut donc rechercher des séquences qui regroupent des actions dépendantes des mêmes tests. Prenons l'exemple très simple de trois actions gardées G_1, G_2, G_3 :

$$\begin{aligned} G_1 &= \texttt{y?(A)(B)} \\ G_2 &= \texttt{x?(C)(D)} \\ G_3 &= \texttt{y?(E)(z?(E)(F))} \end{aligned}$$

On suppose en outre que cet ensemble est partiellement ordonné : G_1 et G_2 doivent être générés avant G_3, tandis que G_1 et G_2 sont indépendants. Il en résulte que deux séquences de calcul sont correctes. La première (G_1, G_2, G_3) ne permet pas de factorisation et donne le code suivant :

```
if y then A else B;
if x then C else D;
if y then E else { if z then E else F }
```

Le deuxième (G_2, G_1, G_3) permet la factorisation des deux dernières actions et donne un code meilleur :

```
if x then C else D;
if y then { A ; E } else { B ; if z then E else F }
```

12.3.2 Fermeture des tests

Un autre problème de la séquentialisation consiste à déterminer quand un test doit être fermé. Pour simplifier ce problème, nous avons imposé que la valeur d'une condition devait toujours être stockée dans un registre. De cette manière, un test peut être ouvert et fermé à volonté au cours d'un cycle. Prenons l'exemple des trois actions gardées suivantes, devant impérativement être générées en séquence :

$$\begin{aligned} G_1 &= \texttt{x?(A)(B)} \\ G_2 &= \texttt{C} \\ G_3 &= \texttt{x?(D)(E)} \end{aligned}$$

Bien que l'action G_2 ne dépende pas de la condition `x`, on peut estimer qu'il est tout de même intéressant de l'intégrer au test sur `x` :

```
if x then
   { A ; C ; D }
```

```
else
   { B ; C ; E }
```

Cette solution donne un code plus rapide que celle qui consiste à refermer le test après G_1. Cependant le code associé à G_2 est dupliqué et la taille du code est plus importante. Le problème peut être généralisé en considérant que G_1 et G_3 représentent des ensembles de calculs dépendant de x, et G_2 un ensemble de calculs ne dépendant pas de x, mais devant être générés impérativement entre les calculs de G_1 et de G_3. Le problème revient à déterminer s'il est intéressant de dupliquer le code des calculs de G_2, dans le seul but d'éviter deux tests de la condition x.

12.4 Algorithme général

Il est clair que les problèmes que nous venons d'évoquer (influence de l'ordre, fermeture des tests...) n'ont pas de solution évidente. De nombreuses heuristiques sont envisageables, dont la qualité ne peut être jugée qu'en fonction des espérances de l'utilisateur. Doit-on par exemple privilégier la rapidité d'exécution ou la taille du code ? Le temps de compilation est aussi un facteur important : l'utilisateur est souvent prêt à se contenter d'un code moyennement bon (en taille ou en vitesse) pourvu que le temps de compilation reste raisonnable. Cette remarque est d'autant plus importante que la génération de l'automate est déjà potentiellement exponentielle.

La phase de séquentialisation a donc été conçue de manière très ouverte. Elle est axée sur un algorithme général, basé sur une remarque de "bon sens" : *tant qu'on peut générer des actions sans ouvrir de test, on le fait.* Cet algorithme a l'avantage de cerner de manière précise les problèmes liés à des choix heuristiques. Ces problèmes sont relatifs au choix de la "bonne" condition à ouvrir, et des "bons" calculs à effectuer sous cette condition.

12.4.1 Le paramètre CONDS

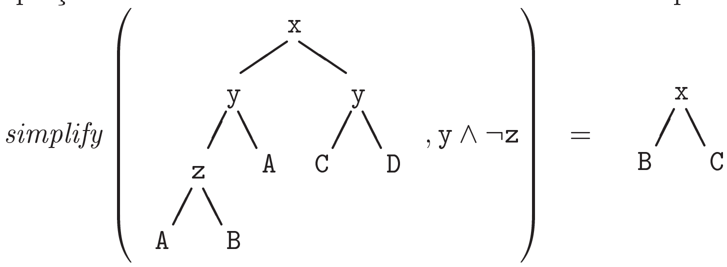
La séquentialisation est réalisée par une fonction récursive *gen-code*. Un des paramètres de cette fonction est un monôme CONDS, portant sur la valeur des conditions, qui indique quelle branche du programme on est en train de générer.

Si le programme final est de la forme :

```
if y then
   ...            % branche 1
   if z then
      ...         % branche 2
   else
      ...         % branche 3
else
   ...            % branche 4
```

on sait par exemple que la branche 1 a été générée avec CONDS= y, et la branche 3 avec CONDS= y $\wedge \neg$ z.

Au cours de la séquentialisation, on est amené à utiliser une fonction *simplify*(G, CONDS) qui réduit l'action gardée G connaissant le monôme CONDS. Cette fonction procède trés simplement en remplaçant dans G les occurences de conditions connues par leurs valeurs. On a par exemple :



Le monôme CONDS intervient aussi dans le prédicat *can-gen*(G,CONDS). Ce prédicat est vrai si on peut générer l'action correspondant à G sans ouvrir de nouveaux tests, c'est-à-dire si G se réduit à une action de base :

$$\textit{can-gen}(G, \text{CONDS}) \quad \Leftrightarrow \quad \textit{simplify}(G, \text{CONDS}) = \texttt{A}$$

12.4.2 Le paramètre TO-GEN

L'autre paramètre de la fonction *gen-code* est le sous-ensemble TO-GEN des calculs à effectuer dans la branche courante. Cet ensemble est partiellement ordonné par la relation $\lhd$ ("doit être généré avant"). On dispose d'une fonction *min*(TO-GEN) qui retourne le sous-ensemble des éléments minimaux de TO-GEN, c'est-à-dire tous les calculs qui peuvent être générés en premier.

12.4.3 Cohèrence des appels récursifs

La fonction *gen-code* est initialement appelée avec le monôme vide et l'ensemble de tous les calculs du programme (y compris le calcul NEXT-STATE, toujours maximum selon $\lhd$).

Au cours d'un appel *gen-code*(CONDS,TO-GEN), on peut être ammené à ouvrir un nouveau test, par exemple sur la condition `x`. On choisit d'abord le sous-ensemble de calcul REC-TO-GEN à générer sous la condition `x`. La fonction *gen-code* est alors appelée récursivement pour générer la branche gauche du test avec le monôme CONDS$\wedge$ `x`, puis la branche droite avec CONDS$\wedge\neg$ `x`.

A tout moment, l'ensemble des calculs du programme est divisé en trois parties :

- les calculs qui ont déjà été traités (DONE),
- les calculs en cours de traitement (TO-GEN),
- les autres, c'est-à-dire les calculs qui n'ont pas encore été générés, mais qui n'ont pas été retenus pour la branche courante.

Pour que globalement la séquence produite respecte l'ordre $\lhd$, il faut évidamment que l'ensemble TO-GEN vérifie la propriété :

$$\forall x \in \text{TO-GEN} \quad (y \lhd x) \Rightarrow (y \in \text{TO-GEN} \vee y \in \text{DONE})$$

Cette propriété est trivialement vérifiée lors de l'appel initial où TO-GEN contient tous les calculs. Pour que cette propriété soit toujours vérifiée, il faut et il suffit que lors d'un appel récursif, le sous-ensemble REC-TO-GEN $\in$ TO-GEN soit choisit de manière à vérifier :

$$\forall x \in \text{REC-TO-GEN}\ \forall y \in \text{TO-GEN} \quad (y \lhd x) \Rightarrow (y \in \text{REC-TO-GEN})$$

12.4.4 Détail de l'algorithme

Dans un premier temps, l'algorithme cherche à générer un maximum d'actions sans ouvrir de test supplémentaire. Cela consiste à rechercher dans *min*(TO-GEN) un calcul dont l'action gardée G vérifie *can-gen*(G,CONDS). L'action de base correspondante est alors engendrée, et le calcul supprimé de l'ensemble TO-GEN. La suppression d'un calcul modifie naturellement l'ensemble *min*(TO-GEN), sur lequel on réitère l'opération.

A la fin de cette première phase, soit l'ensemble TO-GEN est vide, et la génération est finie, soit l'ouverture d'un nouveau test est nécessaire. La condition va bien sûr être choisie parmi celles qui "bloquent" la génération des calculs de *min*(TO-GEN). Ce choix est réalisé par une fonction *choose-cond*(TO-GEN).

Il faut aussi déterminer le sous-ensemble des calculs de TO-GEN qui vont être traités sous le nouveau test. Ce choix est réalisé par la fonction *select*(c,TO-GEN). Le résultat de cette fonction (REC-TO-GEN= *select*(c,TO-GEN)) doit vérifier une propriété relative à l'ordre des calculs :

$$\forall x \in \text{REC-TO-GEN} \quad (y \lhd x) \Rightarrow (y \in \text{REC-TO-GEN} \vee y \notin \text{TO-GEN})$$

On peut en effet montrer par induction que cette contrainte est nécessaire et suffisante pour qu'à tout moment l'ensemble TO-GEN respecte la propriété 12.1.

La fonction *gen-code* est ensuite appelée récursivement pour générer les calculs de REC-TO-GEN dans les branches "vraie" et "fausse" du nouveau test. Après quoi la génération des calculs restants est reprise en séquence.

```
gen-code(TO-GEN,CONDS) {
  tant que ∃ x ∈ min(TO-GEN) tel que can-gen(x,CONDS) {
        genere(x,CONDS);
        TO-GEN := TO-GEN \ {x}
  }
  si TO-GEN≠ ∅ {
        c := choose-cond(TO-GEN);
        REC-TO-GEN := select(c,TO-GEN);
        gen-if(c);
        gen-code(REC-TO-GEN,CONDS∧ c);
        gen-else();
        gen-code(REC-TO-GEN,CONDS∧¬ c);
        gen-fi();
        gen-code(TO-GEN \ REC-TO-GEN,CONDS);
  }
}
```

12.5 Algorithme dérivé

Par son caractère général, l'algorithme que nous venons de présenter n'enlève rien à l'aspect combinatoire du problème de la séquentialisation. Nous avons expérimenté une version de l'algorithme général, où le choix de la condition à ouvrir est basé sur une heuristique simple : choisir la condition qui bloque le plus de calculs de *min*(TO-GEN). Cet algorithme donne un bon résultat en un temps acceptable tant qu'on génère de tout petits automates (moins de dix états). Au delà, le temps consacré à la séquentialisation devient rapidement plus important que celui consacré à la génération de l'automate, et devient prohibitif dès qu'on atteint une centaine d'états (plusieurs dizaines de minutes). Nous avons donc choisi de privilégier le temps de compilation plutôt que la recherche de solutions “optimales”.

12.5.1 Choix d'un ordre total

La complexité de l'algorithme général est due en grande partie au fait que l'ordre des calculs est partiel. En effet, si on impose a priori un ordre total pour tous les états, la fonction *min* donne toujours un singleton $\{x\}$, et l'ensemble des conditions “ouvrables” est réduit à celui des conditions qui bloquent le calcul de x. De plus, il devient inutile de garantir la cohérence de la séquence obtenue (§12.4.3), puisque celle-ci est donnée a priori.

Un ordre total peut être obtenu en effectuant un tri topologique des calculs : l'ensemble à trier contient initialement tous les calculs, puis on retire un à un les élèments minimaux de cet ensemble. La séquence des retrait donne triviallement un ordre total compatible avec $\lhd$. Dans le compilateur, un tel tri est déjà effectué au cours de la recherche des tampons (§ 10.2). On se contente donc d'en garder une “trace” pour la séquentialisation.

12.5.2 Choix de la condition à ouvrir

Deux types de méthodes sont envisageables pour choisir la “bonne” condition à ouvrir : on peut en effet choisir la condition sans se soucier des calculs qui vont être effectués sous celle-ci, ou au contraire tenir compte de ceux-ci dans l'espoir d'optimiser la factorisation. Cette dernière solution consiste en quelque sorte à regrouper les deux fonctions *choose-cond*(TO-GEN) et *select*(c,TO-GEN) en une fonction *choose-and-select*(TO-GEN) dont le rôle est de fournir le meilleur couple possible (c,REC-TO-GEN).

Cette dernière solution est encore une fois trop coûteuse : on ne peut en effet juger de la qualité d'un couple (c,REC-TO-GEN) qu'en anticipant le résultat de la séquentialisation sur l'ensemble REC-TO-GEN. Même si le nombre des conditions ouvrables est réduit, un tel algorithme garde tout de même un caractère combinatoire.

La solution retenue consiste donc à choisir la “première” condition qui bloque la génération de *min*(TO-GEN). Puisque nous avons imposé un ordre total, *min*(TO-GEN) ne contient qu'un élément `x` ; on a $\neg\mathit{can\text{-}gen}(\mathtt{x}, \mathrm{CONDS})$, c'est-à-dire (§12.4.1) :

$$\mathit{simplify}(\mathtt{x}, \mathrm{CONDS}) = \mathtt{c}?(\alpha)(\beta)$$

Le test va donc porter sur la condition `c`, qui apparaît en tête de cette action gardée simplifiée.

12.5.3 Choix de l'ensemble REC-TO-GEN

Après avoir imposé une séquence de calcul et le choix de la condition à ouvrir, la seule imprécision qui demeure est relative à la fonction *select*. Grâce à l'ordre total que nous avons imposé, le choix d'un ensemble REC-TO-GEN ne peut plus être combinatoire. En effet, les calculs susceptibles d'être factorisés sous la condition `c` sont, de proche en proche, les successeurs de

min(TO-GEN).

Trois options sont disponibles dans le compilateur. Les deux premières implémentent les solutions “extrêmes” qui privilégient respectivement la taille du code et la vitesse d’exécution. La troisième option est une solution intermédiaire qui accepte une duplication de code “raisonnable”. Les algorithmes correspondants choisissent dans un ensemble de candidats (initialisé à TO-GEN) les calculs qui vont être effectués sous la condition `c`. Ces algorithmes utilisent la fonction *min* que nous connaissons déjà, mais aussi la fonction duale *max* [1]. Les algorithmes utilisent un prédicat *cond-dep*(`c`,`x`) qui est vrai si la condition `c` apparaît dans l’action gardée associée au calcul `x`. Pour plus de lisibilité, les algorithmes sont présentés dans un style ensembliste et récursif. Ils sont en fait implémentés de manière beaucoup plus efficace dans le compilateur.

Refermer le test au plus tôt : Cette heuristique refuse toute duplication de code, et privilégie donc la taille du programme produit. L’algorithme correspondant est très simple : on accepte de proche en proche les successeurs de min(TO-GEN) qui dépendent de `c`.

$$\begin{aligned} select_0(c, \text{TO-GEN}) = \; & si\ cond\text{-}dep(\mathtt{c}, min(\text{TO-GEN})) \\ & alors\ min(\text{TO-GEN}) \cup select_0(\mathtt{c}, \text{TO-GEN} \ \setminus min(\text{TO-GEN})) \\ & sinon\ \emptyset \end{aligned}$$

Refermer le test le plus tard possible : Cette heuristique implique qu’un test est ouvert une fois et une seule ; elle privilégie donc la rapidité d’exécution. Cet algorithme est en quelque sorte le dual du premier, il procède en effet en éliminant de proche en proche les max(TO-GEN) qui ne dépendent pas de la condition c.

$$\begin{aligned} select_1(c, \text{TO-GEN}) = \; & si\ cond\text{-}dep(\mathtt{c}, max(\text{TO-GEN})) \\ & alors\ \text{TO-GEN} \\ & sinon\ select_1(\mathtt{c}, \text{TO-GEN} \ \setminus max(\text{TO-GEN})) \end{aligned}$$

Solution intermédiaire : Une duplication “raisonnable” est acceptée. Pour être raisonnable, la duplication ne doit concerner que des actions de base. Un calcul qui ne dépend pas de la condition `c` ne peut donc être dupliqué que s’il est calculable dans la branche courante. Pour réaliser cette fonctionnalité, nous définissons une fonction *duplicate* avec un argument supplémentaire (CONDS).

$$\begin{aligned} & duplicate(c, \text{TO-GEN}, \text{CONDS}) = \\ & \quad si\ cond\text{-}dep(\mathtt{c}, min(\text{TO-GEN})) \vee can\text{-}gen(min(\text{TO-GEN}), \text{CONDS}) \\ & \quad alors\ min(\text{TO-GEN}) \cup duplicate(\mathtt{c}, \text{TO-GEN} \ \setminus min(\text{TO-GEN})) \\ & \quad sinon\ \emptyset \end{aligned}$$

Après avoir déterminé ce premier ensemble, il convient d’en retirer de proche en proche les éléments maximaux qui ne dépendent pas de la condition. Cette “correction” peut par exemple être réalisée par la fonction $select_1$.

$$select_2(c, \text{TO-GEN}, \text{CONDS}) = select_1(c, duplicate(c, \text{TO-GEN}, \text{CONDS}))$$

1. le résultat des fonctions *min* et *max* est toujours un singleton dans notre cas

Chapitre 13

Traitement des assertions

Le code généré par le compilateur est basé sur un automate de contrôle qui synthétise le comportement de la mémoire booléenne du programme. Ce comportement est fonction d'un certain nombre de conditions booléennes que le compilateur ne peut pas évaluer de manière statique. Pour limiter le risque d'explosion combinatoire intrinsèque à cette méthode, le programmeur a la possibilité de définir des assertions. Les assertions ont été introduites en LUSTRE en s'inspirant de ce qui existait déjà pour le langage ESTEREL [BCG87]. Il s'agit de signaler au compilateur des propriétés toujours vérifiées par l'environnement. Le compilateur peut alors utiliser ces indications pour élaguer des chemins impossibles, et réduire ainsi le nombre de transitions et d'états de l'automate.

13.1 Assertions instantanées

Un exemple classique d'assertions concerne l'exclusion de certaines entrées booléennes ; l'opérateur n-aire "`#`" a d'ailleurs été introduit dans le langage LUSTRE spécialement pour les assertions, avant d'être accepté dans toutes les expressions booléennes. Prenons l'exemple simple d'un automate qui a deux entrées booléennes `a` et `b`. Nous nous intéressons plus particulièrement aux transitions partant de l'état initial S_0, les autres n'étant pas étiquetées par souci de lisibilité (figure 13.1).

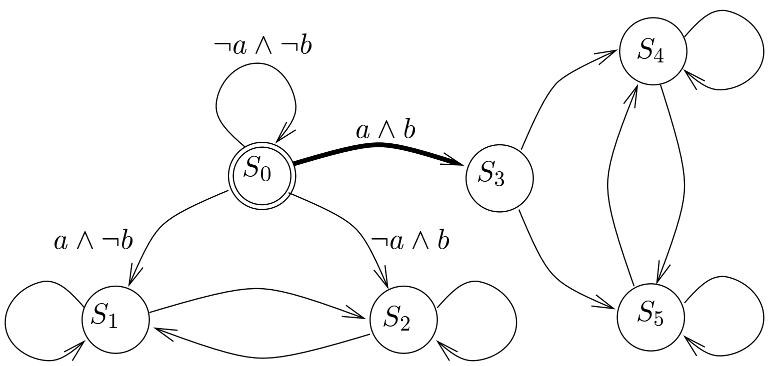


FIG. 13.1 – *Transition violant* `assert #(a,b)`

Le programmeur peut exprimer le fait que les entrées ne peuvent pas être simultanément vraies en écrivant l'assertion : `assert #(a,b)`. Il en résulte que la transition qui conduit de S_0 à S_3 ne pourra jamais être effectuée. Le code séquentiel correspondant à l'état S_0 peut donc être simplifié en conséquence :

```
... ;
if a then goto: S1
else if b then goto: S2
else goto: S0
```

De plus, en supprimant la transition, l'état S_3 ainsi que ses successeurs S_4 et S_5 deviennent inaccessibles. Il est donc inutile pour le compilateur de générer du code pour ces états. En fait ces états n'ont même pas à être traités lors de la génération de l'automate : le parcours des états ne doit en effet considérer que les états certainement accessibles depuis l'état initial, c'est-à-dire accessibles par un chemin qui ne viole jamais l'assertion.

13.2 Assertions temporelles

L'exemple que nous venons de présenter est très simple, puisqu'il exprime une propriété portant sur la valeur courante des entrées. Sur le même modèle, on peut imaginer toute une classe d'assertions "instantanées", qui combinent des entrées et des opérateurs logiques.

Mais on peut vouloir exprimer des propriétés plus complexes, portant sur le comportement temporel des entrées, comme par exemple le fait qu'une entrée booléenne `a` ne peut pas être vraie à deux instants consécutifs. Cette propriété se traduit en LUSTRE par l'assertion :

```
assert (true -> not(a and pre a))
```

La figure 13.2 représente une petite partie d'un automate de contrôle. Si on suppose que la transition $S_0 \xrightarrow{a} S_1$ est la seule qui conduise à S_1, alors la transition $S_1 \xrightarrow{a} S_3$ viole l'assertion et peut donc être supprimée. Le code correspondant à l'état S_1 peut être simplifié : "... ; `goto:` S_2". De plus, si l'état S_3 n'est accessible que par ce chemin "impossible", il ne sera pas traité lors de la génération.

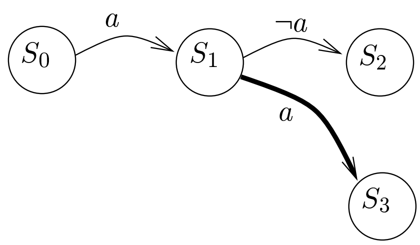


FIG. 13.2 – *Transition violant* `assert (true ->not(a and pre a))`

On peut donc, pour simplifier l'automate, utiliser des assertions portant sur des valeurs passées. Dans l'exemple, nous nous sommes contentés d'utiliser directement le "`pre`" d'une entrée ; mais on pourrait utiliser des "`pre`" plus complexes, dont l'argument est une expression qui dépend elle-même du passé. En fait, une expression booléenne, aussi compliquée soit elle, peut toujours être interprétée comme une propriété de l'environnement : n'importe qu'elle expression booléenne est donc susceptible de simplifier le programme produit.

13.3 Evaluation exhaustive ou paresseuse des assertions

Dans l'exemple précédent, nous avons pu simplifier l'automate grâce à une assertion portant sur une valeur passée. Cette simplification n'a été possible qu'en supposant que la seule transition conduisant dans l'état S_1 était $S_0 \xrightarrow{a} S_1$. Il suffit de compliquer cet exemple en rajoutant un état S_4 et une transition $S_4 \xrightarrow{\neg a} S_1$ pour qu'il en soit autrement (figure 13.3).

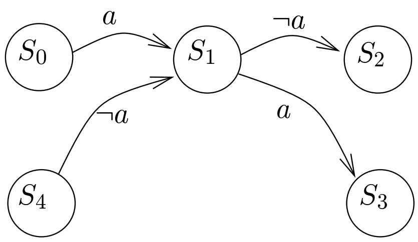


Fig. 13.3 –

Dans ce nouvel exemple, l'état S_1 n'implique plus que `pre a` est vraie. Pour que l'assertion soit évaluée complètement, il faut distinguer deux sous-états de S_1 :

- S_1' auquel on accède par la transition $S_0 \xrightarrow{a} S_1'$
- S_1'' auquel on accède par la transition $S_4 \xrightarrow{\neg a} S_1''$

Ce n'est qu'après avoir fait cette distinction que la transition $S_1' \xrightarrow{a} S_3$ peut être supprimée (figure 13.4).

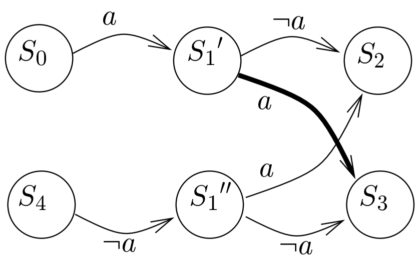


Fig. 13.4 –

Une telle situation peut intervenir quelle que soit la méthode employée pour la génération de l'automate :

dirigée par les données : cela signifie que `pre a` est une mémoire qui influence le calcul de l'assertion mais pas des sorties. Se pose alors le problème de savoir si on lui associe une variable d'état : on risque en effet d'obtenir un automate (figure 13.4) plus complexe que si on avait ignoré l'assertion (figure 13.3).

dirigée par la demande : `pre a` est éventuellement une variable d'état, mais dans l'état S_1, elle n'influe pas sur les valeurs courantes et futures des sorties. Pour obtenir le résultat de la figure 13.4, il faut donc non seulement s'assurer que les mémoires dont dépend l'assertion sont des variables d'état, mais aussi imposer le calcul de l'assertion comme s'il s'agissait d'une sortie.

Dans tous les cas, on peut résumer ce problème en définissant deux types de traitement pour l'assertion :

Evaluation paresseuse : dans cette option, l'automate qu'on aurait obtenu sans tenir compte des assertions est l'automate de référence. Chaque fois qu'on peut simplifier celui-ci, on le fait, mais sans jamais introduire de tests ou d'états supplémentaires.

Evaluation exhaustive : dans cette option, l'assertion est traitée comme une expression booléenne à évaluer complètement. L'automate obtenu, même simplifié, est éventuellement plus complexe que si on avait ignoré l'assertion.

13.4 Assertions contradictoires

Une assertion est contradictoire s'il n'existe aucune histoire des entrées qui la vérifie. On suppose que de telles assertions proviennent d'une erreur du programmeur, et qu'elles doivent donc être détectées.

Dans le cas d'une assertion instantanée, cela signifie que le programmeur a écrit une expression booléenne toujours fausse, comme par exemple "`a and not a`". Pour cet exemple très simple, la mise sous forme canonique suffit à mettre en évidence la contradiction.

Dans le cas des assertions temporelles, il s'agit d'un comportement temporel impossible, comme par exemple "`true ->a and not pre(a)`". Dans ce cas, c'est la construction de l'automate qui met en évidence la contradiction (figure 13.5) : sur cet automate, tous les chemins infinis (i.e. toutes les histoires possibles des entrées) passent par des transitions qui violent l'assertion.

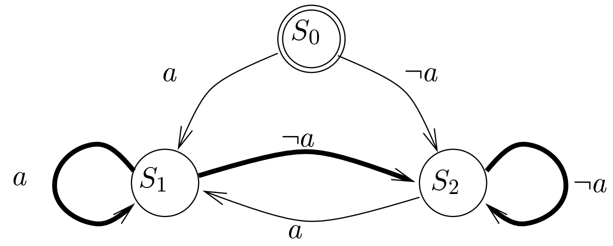


FIG. 13.5 – *L'assertion contradictoire* "`true ->a and not pre a`"

Cet exemple fait apparaître un fait nouveau : l'état S_1 (`pre a = true`), est un *puits*, c'est-à-dire qu'il est impossible de le quitter sans violer l'assertion. On peut montrer trivialement que toute assertion contradictoire provoque des puits. Mais la présence de puits n'est pas forcément synonyme de contradiction, comme nous allons le voir en étudiant la *causalité* des assertions.

13.5 Normalisation des assertions

Nous n'avons pour l'instant parlé que d'assertions sur l'horloge de base. Il faut maintenant définir quel sens on donne à une assertion `assert` *exp*, où *exp* dénote un flot quelconque :

> `assert` *exp* signifie que l'expression *exp* dénote un flot booléen, qui, s'il est présent, porte toujours la valeur `true`.

Cette définition nous amène à définir la notion d'assertion apparente :

Définition : L'assertion apparente d'un flot booléen quelconque f est un flot booléen sur l'horloge de base qui, à tout instant, porte la valeur `false` si f est défini et porte la valeur `false`, et porte la valeur `true` sinon.

Toute assertion `assert` *exp* peut donc être remplacée par une assertion `assert` $xassert(exp)$ si $xassert(exp)$ dénote l'assertion apparente de *exp*. La fonction *xassert* est définie grâce aux fonctions introduites dans le chapitre 4.

$$xassert(exp) \quad = \quad \texttt{if } xck(exp) \texttt{ then } ximpl(exp) \texttt{ else true}$$

La syntaxe abstraite présentée dans le chapitre 3 doit être complétée pour tenir compte des programmes avec assertions :

$$prg \ ::= \ ins\ outs\ eqs \ \mid \ ins\ outs\ assertions\ eqs$$

$$assertions \ ::= \ assertion \ \mid \ assertion\ assertions$$

$$assertion \ ::= \ \texttt{assert}\ exp$$

La normalisation des programmes avec assertions est donnée par :

$$norm(ins\ outs\ assertions\ eqs) = ins\ outs\ \texttt{assert(}norm(assertions)\texttt{)}\ norm(eqs)$$
$$norm(assertion\ assertions) = norm(assertion)\ \texttt{and}\ norm(assertions)$$
$$norm(\texttt{assert}\ exp) = xassert(exp)$$

L'ensemble des assertions, qui représentent un ensemble de propriétés de l'environnement, est donc remplacée par une seule "super-propriété", dénotée par le produit logique des assertions apparentes. Ce passage "ensemble d'assertions/produit d'assertions" n'était pas possible dans le programme source à cause des horloges.

La normalisation des programmes sans assertions est redéfinie pour introduire une assertion triviale :

$$norm(ins\ outs\ eqs) = ins\ outs\ \texttt{assert(true)}\ norm(eqs)$$

De cette manière, tous les programmes normalisés contiennent une unique assertion : `assert`A. Dans la suite, l'expression A sera simplement appelée *l'assertion du programme*.

13.6 L'assertion comme fonction du contrôle

L'assertion d'un programme est destinée à simplifier la structure de contrôle. On doit donc l'interpréter comme une propriété portant sur les éléments de contrôle. Comme toute expression booléenne, l'assertion se décompose en opérateurs évaluables, références au passé booléen et opérateurs non évaluables. L'assertion peut donc être interprétée comme une fonction booléenne, dont les arguments sont ses références au passé et ses sous-expressions booléennes non évaluables.

Dans le cas d'une évaluation exhaustive, tous les arguments de l'assertion sont des éléments de contrôle du programme. Dans le cas d'une évaluation paresseuse, certains ne le sont pas, et posent donc un problème. Prenons un exemple très simple :

```
if c then a else b
```

Dans cette expression, on suppose que `a` et `b` sont des éléments de contrôle, et que `c` n'en est pas un. On peut tout de même déduire une propriété intéressante de cette assertion. En effet, qu'elle que soit la valeur de `c`, l'assertion sera violée si `a` et `b` valent `false`. On peut donc déduire de l'assertion la propriété suivante :

```
a or b
```

13.6.1 Introduction des arguments supplémentaires

La méthode générale consiste à créer un ensemble S d'arguments supplémentaires. L'assertion peut alors être interprétée comme une fonction booléenne :

$$A \in ((Q \times C \times S) \rightarrow B)$$

Contrairement aux éléments de contrôle, il est inutile de distinguer dans S les arguments qui correspondent à des références au passé de ceux qui correspondent à des opérateurs non évaluables. Ces arguments supplémentaires permettent simplement de construire une représentation canonique de l'assertion sous la forme de *bdd*. La forme canonique de l'assertion `if c then a else b`, où `c` est un argument supplémentaire, est représentée sur la figure 13.6. Pour l'ordre des arguments, on impose que les éléments de contrôle apparaissent toujours en tête. Sur notre exemple, on obtient un *bdd* plus gros qu'avec, par exemple, l'ordre `(c, a, b)`. Mais cette méthode simplifie considérablement le traitement suivant, qui consiste à éliminer les arguments supplémentaires.

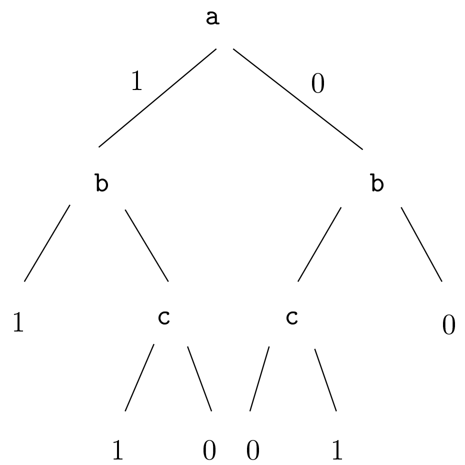


FIG. 13.6 – *Une forme canonique de la fonction "*`if c then a else b`*"*

13.6.2 Elimination des arguments supplémentaires

La forme normale de l'assertion, avec ses arguments supplémentaires, n'est pas directement exploitable pour simplifier le contrôle. Il faut en effet extraire de celle-ci le maximum d'information ne faisant intervenir que les éléments de contrôle. Cette élimination est réalisée par une fonction *simp-assert*, dont la sémantique est donnée par :

simp-assert$(A)(q, c)$ *ssi* $(\exists s \in S / A(q, c, s))$

Sur les *bdd*, cette opération consiste simplement à remplacer les sous-graphes indépendants du contrôle et qui ne sont pas identiquement faux, par la feuille *true-bdd*. Sur notre exemple, on retrouve bien une forme canonique de la fonction `a or b` (figure 13.7).

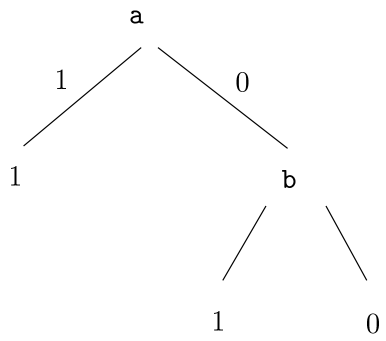


FIG. 13.7 – *La fonction "*$\exists$ `c (if c then a else b)`*"*

13.7 Causalité

13.7.1 Exemple

Dans un exemple précédent, nous avons introduit une assertion pour exprimer que l'entrée `a` ne pouvait pas être vraie à deux instants consécutifs. Nous avons naturellement traduit cette propriété en faisant intervenir la valeur courante de l'entrée. Il est cependant tout à fait possible de l'exprimer en faisant intervenir les deux dernières valeurs `pre(a)` et `pre(pre(a))`. Correctement initialisée, l'assertion devient :

```
assert true ->not(pre(a) and pre(false ->pre(a))).
```

La figure 13.8 représente une petite partie de l'automate correspondant, avec en gras les transitions qui violent l'assertion.

Si on se contente, comme on l'a fait jusqu'à maintenant, de supprimer les transitions "impossibles", on fait de S_3 un *puits*, c'est-à-dire un état sans aucune transition sortante. Du point de vue des exécutions possibles du programme, ce puits ne pose pas de problème : on sait en effet qu'aucune séquence d'entrée ne peut y conduire. On pourrait par exemple le conserver en lui associant une transition non-significative : $S_3 \to S_3$. Cette méthode est cependant peut compatible avec le caractère simplificateur des assertions : la seule méthode raisonnable consiste donc à supprimer cet état.

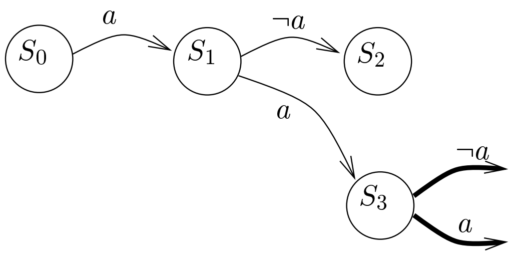


FIG. 13.8 – *Une assertion non-causale*

13.7.2 Définition

Les assertions susceptibles de créer des puits sont dites *non causales*. La causalité d'une assertion $A \in 2^{Q \times C}$ vis-à-vis d'un automate $(Q, C, \rightarrow, q_0)$ est définie formellement par :

$\forall q \in Q \exists c \in C / (q_0 \overset{*}{\rightarrow}_A q) \quad \Rightarrow \quad A(q, c)$

où la relation $\overset{*}{\rightarrow}_A$, est la fermeture transitive de la relation $\rightarrow_A$:

$q \rightarrow_A q' \quad \Leftrightarrow \quad \exists c \in C / (q \overset{c}{\rightarrow} q' \wedge A(q, c))$

Intuitivement, une assertion est causale si tout état accessible depuis l'état initial sans jamais violer l'assertion, peut être quitté par une transition qui ne viole pas l'assertion.

La prise en compte des assertions non causales pose un problème d'efficacité au compilateur. En effet, de telles assertions obligent le compilateur à remettre en cause des transitions et des états déjà générés.

Par défaut, le compilateur suppose que l'assertion fournie par le programmeur est causale. Si un puits est rencontré au cours de la génération, la compilation est interrompue et un message d'erreur est fourni au programmeur.

Pour que le programmeur puisse tout de même écrire, en toute connaissance de cause, des assertions non causales, le compilateur propose une option dans laquelle une assertion quelconque est préalablement transformée en une assertion causale équivalente.

13.8 Construction d'une assertion causale

Le problème des assertions non causales est étudié en détail par C. Ratel dans le cadre de la logique temporelle [Rat92]. Nous nous contentons ici de présenter la méthode permettant de construire une assertion causale équivalente à partir d'une assertion quelconque.

13.8.1 Equivalence sémantique des assertions

Une *exécution* d'un automate $\Gamma = (Q, C, \rightarrow, q_0)$ est une séquence infinie de couples (q_i, c_i) telle que $\forall i \in N \quad q_i \overset{c_i}{\rightarrow} q_{i+1}$. L'ensemble des exécutions de Γ est noté $EX(\Gamma)$. Intuitivement, deux assertions A et A' sont équivalentes vis-à-vis de Γ si toute exécution qui viole l'une, viole l'autre, et réciproquement :

$A \equiv A' \quad ssi$

$\forall (q_i, c_i)_{i \in N} \in EX(\Gamma) \quad (\exists n / \neg A(q_n, c_n)) \Leftrightarrow (\exists m / \neg A'(q_m, c_m))$

13.8.2 Causalité forte

La causalité telle que nous l'avons définie dans la section précédente (§13.7.2) ne peut pas être utilisée. En effet, elle fait intervenir la notion d'accessibilité ; or on voudrait s'assurer de la causalité d'une assertion avant de connaître les états accessibles (i.e. avant la génération de l'automate). On introduit donc une notion de *causalité forte*, qui ne tient pas compte de l'état initial, et donc de l'accessibilité. La causalité forte fait référence à la notion de puits, c'est-à-dire d'états qu'on ne peut pas quitter sans violer l'assertion :

$$puits(A) = \{q \in Q / \forall c \in C \quad \neg A(q,c)\}$$

Intuitivement, A est fortement causale si on ne peut accéder à un puits que par une transition qui viole A. On rappelle que $Precond(E)$ est l'ensemble des transitions (couples (q,c)) qui conduisent dans un état de l'ensemble E (§ 11.3.2 page 92) :

$$A \text{ est fortement causale} \quad ssi \quad A \Rightarrow \neg Precond(puits(A))$$

13.8.3 Relation entre causalité et causalité forte

Une assertion fortement causale est causale si et seulement si l'état initial n'est pas un puits (i.e. $q_0 \notin puits(A)$). En fait, si l'état initial est un puits, l'assertion est trivialement contradictoire.

13.8.4 Construction d'une assertion fortement causale

La fonction :

$$\begin{array}{rcl} F : 2^{Q\times C} & \rightarrow & 2^{Q\times C} \\ A & \mapsto & A \wedge \neg Precond(puits(A)) \end{array}$$

transforme l'assertion A en une assertion sémantiquement équivalente (la démonstration est triviale). De plus, $F(A)$ est inférieure à A, selon l'ordre d'implication $((F(A) \Rightarrow A))$. On en déduit que $F^* = lim_{n\rightarrow\infty} F^n$ existe :

$$\exists k / F(F^k(A)) = F^k(A)$$

L'assertion $F^*(A)$ peut donc être construite en un temps fini. Deux cas sont alors possibles :

- Si l'état initial est un puits de $F^*(A)$, cela signifie que l'assertion initiale A est une contradiction : la compilation est interrompue sur un message d'erreur.
- Sinon, l'assertion $F^*(A)$ est trivialement causale et équivalente à A. La génération de l'automate peut alors être mise en œuvre sous l'assertion $F^*(A)$, et on est sûr de ne jamais rencontrer de puits.

13.9 Prise en compte de l'assertion

Après les manipulations que nous venons de présenter (introduction et élimination des arguments supplémentaires, recherche éventuelle d'une assertion causale), on obtient une fonction booléenne des éléments de contrôle, que nous appelons simplement l'assertion du programme. Cette assertion A va être utilisée à trois niveaux au cours de la génération de code.

13.9.1 Simplification des fonctions de transition et des actions gardées

L'assertion peut être utilisée pour simplifier a priori les fonctions δ_i et λ_j. En effet, plus ces fonctions sont simples, plus leur manipulation symbolique sera efficace. La simplification des *bdd* est réalisée par un opérateur dit de *restriction* et noté $\downarrow$, auquel nous consacrons le chapitre suivant. Chaque fonction δ_i est remplacée avant la génération par une fonction plus simple : $\delta'_i = \delta_i{\downarrow}A$. Même chose pour les actions gardées λ_i qui sont remplacées par : $\lambda'_i = \lambda_i{\downarrow}A$.

13.9.2 Elimination des transitions impossibles

Au cours de la génération d'automate, les fonctions δ_i sont combinées pour construire le Δ de l'état courant, c'est-à-dire l'action gardée correspondant au calcul du NEXT-STATE. Cette action gardée doit à son tour être simplifiée par l'opérateur $\downarrow$ pour éliminer les transitions impossibles : $\Delta' = \Delta{\downarrow}A$.

13.9.3 Elimination des tests inutiles

Au cours de la séquentialisation, l'ouverture d'un nouveau test peut faire apparaître une branche impossible. On peut toujours considérer que l'état en cours de séquentialisation correspond à une propriété des n variables d'état : dans la génération dirigée par la demande, c'est une propriété complexe vérifiée par plusieurs valeurs ; dans la génération dirigée par les données cette propriété n'est vérifiée que par une seule valeur des variables d'état. L'état en cours de traitement est donc caractérisé par une fonction $P \in B^n \rightarrow B$.

De même, le monôme CONDS, qui résume la valeur des tests déjà ouverts (§12.4.1), peut être vu comme une propriété des m conditions : CONDS $\in B^m \rightarrow B$. Dans la branche caractérisée par les fonctions P et CONDS, l'assertion est satisfaisable si et seulement si la fonction :

$$\lambda_{qc}.P(q) \wedge \mathrm{CONDS}(c) \wedge A(q,c)$$

n'est pas identiquement fausse.

Lors de l'ouverture d'un nouveau test $cond_i$, on construit deux fonctions :

$$\begin{aligned} S_i &= \lambda_{qc}.P(q) \wedge \mathrm{CONDS}(c) \wedge cond_i \wedge A(q,c) \\ S_{\neg i} &= \lambda_{qc}.P(q) \wedge \mathrm{CONDS}(c) \wedge \neg cond_i \wedge A(q,c) \end{aligned}$$

Si S_i est identiquement fausse, cela signifie que la branche "vraie" du test ne doit pas être générée ; si $S_{\neg i}$ est identiquement fausse, cela signifie que la branche "fausse" du test ne doit pas être générée. Si les deux fonctions sont identiquement fausses, c'est que l'assertion considérée n'est pas causale.

L'algorithme de séquentialisation (§12.4.4) est modifié en conséquence :

```
gen-code(TO-GEN,CONDS) {
  tant que ∃ x ∈ min(TO-GEN) tel que can-gen(x,CONDS) {
          genere(x,CONDS);
          TO-GEN := TO-GEN \ {x}
  }
  si TO-GEN≠ ∅ {
          c := choose-cond(TO-GEN);
          REC-TO-GEN := select(c,TO-GEN);
```

```
        si (S_c = 0) et (S_¬c = 0)
            erreur(" assertion non causale ")
        sinon si (S_c = 0)
            gen-code(REC-TO-GEN,CONDS∧¬ c);
        sinon si (S_¬c = 0)
            gen-code(REC-TO-GEN,CONDS∧ c);
        sinon {
            gen-if(c);
            gen-code(REC-TO-GEN,CONDS∧ c);
            gen-else();
            gen-code(REC-TO-GEN,CONDS∧¬ c);
            gen-fi();
            gen-code(TO-GEN \ REC-TO-GEN,CONDS);
        }
    }
}
```

Chapitre 14

Simplification des fonctions booléennes

La prise en compte des assertions nous conduit à étudier le problème plus général de la simplification des fonctions booléennes.

Les données du problème sont :

- Une fonction à simplifier : $f \in (B^n \to I) = (I^{B^n})$ où I est un ensemble fini.
- Une hypothèse : $g \in (B^n \to B) = (2^{B^n})$

Dans le cas des fonctions de transition, l'ensemble I est celui des booléens ; dans le cas des actions gardées, I est l'ensemble des actions OC. Intuitivement, notre but est de définir une opération "*f sachant g*" dont le résultat est une fonction $h \in (B^n \to I)$ qui soit :

- équivalente à f quand g est vraie,
- la plus simple possible.

La notion de simplicité est bien entendu relative à une représentation donnée des fonctions. Dans le cas des *bdds*, les critères de comparaison peuvent être basés par exemple sur le nombre de nœuds ou le nombre d'arguments. A cause du partage, il est en fait assez difficile de raisonner sur le nombre de nœuds des *bdds*. On se contente donc de raisonner sur les arbres de Shannon.

Des opérateurs de simplification ont déjà été définis [OC90, Cou91]. Basés sur des heuristiques simples, ils ont l'avantage d'être peu coûteux, mais fournissent parfois des résultats décevants. Le problème de la simplification est en général présenté pour les fonctions strictement booléennes $(B^n \to B)$. Nous l'avons naturellement étendu à des fonctions à valeurs dans un ensemble fini quelconque $(B^n \to I)$.

14.1 Les opérateurs classiques

Nous présentons ici les principes des opérateurs de simplification les plus couramment utilisés.

14.1.1 L'opérateur *constrain*

Le premier est généralement appelé "*constrain*" et noté $\uparrow$([OC90]) :

$\uparrow \in (I^{B^n} \times 2^{B^n}) \to I^{B^n} \cup \{\phi\}$

L'élément "ϕ" est introduit pour représenter le résultat de *constrain* par la fonction identiquement fausse ($\forall f \in I^{B^n} (f \uparrow 0 = \phi)$). Cette valeur n'est pas absolument nécessaire : il suffit de restreindre l'ensemble de définition de l'opérateur. Elle permet cependant une écriture beaucoup plus simple de l'algorithme :

Règles terminales :

$$\begin{array}{rcl} \alpha \uparrow 0 & = & \phi \\ \alpha \uparrow 1 & = & \alpha \\ i \uparrow \beta & = & i \quad si\ (i \in I)\ et\ (\beta \neq 0) \end{array}$$

Descente récursive : il s'agit des règles de parcours en "profondeur d'abord" communes à tous les opérateurs binaires sur les *bdds* :

$$\begin{array}{c} x \\ \alpha \diagup \diagdown \beta \end{array} \uparrow \begin{array}{c} x \\ \gamma \diagup \diagdown \delta \end{array} = combine(x, \alpha{\uparrow}\gamma, \beta{\uparrow}\delta)$$

$$\begin{array}{c} x \\ \alpha \diagup \diagdown \beta \end{array} \uparrow \begin{array}{c} y \\ \gamma \diagup \diagdown \delta \end{array} = combine(x, \alpha \uparrow \begin{array}{c} y \\ \gamma \diagup \diagdown \delta \end{array}, \beta \uparrow \begin{array}{c} y \\ \gamma \diagup \diagdown \delta \end{array}) \quad si\ x < y$$

$$\begin{array}{c} x \\ \alpha \diagup \diagdown \beta \end{array} \uparrow \begin{array}{c} y \\ \gamma \diagup \diagdown \delta \end{array} = combine(y, \begin{array}{c} x \\ \alpha \diagup \diagdown \beta \end{array} \uparrow \gamma, \begin{array}{c} x \\ \alpha \diagup \diagdown \beta \end{array} \uparrow \delta) \quad si\ y < x$$

Combinaison des résultats intermédiaires : il s'agit d'utiliser le résultat partiel ϕ pour faire disparaître un nœud. Dans les autres cas, il s'agit de la combinaison classique, réalisée par la fonction *get-bdd* (§9.5.1 page 9.5.1) :

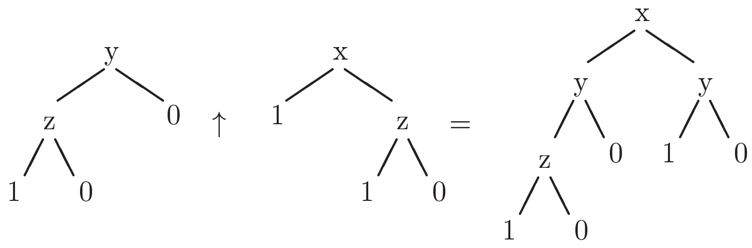
$$\begin{array}{rcl} combine(x, \alpha, \phi) & = & \alpha \\ combine(x, \phi, \beta) & = & \beta \\ combine(x, \alpha, \beta) & = & get\text{-}bdd(x, \alpha, \beta)\ si\ \alpha \neq \phi\ et\ \beta \neq \phi \end{array}$$

L'opérateur $\uparrow$ possède une propriété de *simplification minimale*, concernant le cas des *bdds* strictement booléens. Cette propriété, bien qu'assez évidente, est très importante :

$si\ g \neq\ 0\ et\ g{\Rightarrow}f \quad alors \quad f{\uparrow}g = 1$

$si\ g \neq\ 0\ et\ g{\Rightarrow}\neg f \quad alors \quad f{\uparrow}g = 0$

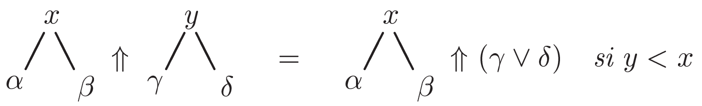
Mis à part ce cas très particulier, il est difficile de caractériser la qualité du résultat. Il est même possible que le résultat soit plus "mauvais" que la fonction initiale. Prenons l'exemple de l'opération $(y \wedge z) \uparrow (x \vee z)$:



Dans cet exemple, le problème vient du fait qu'on introduit dans le résultat un nœud qui n'apparaissait pas dans la fonction initiale. Pour éviter ce problème, un opérateur dérivé du *constrain*, appelé *restrict*, a été proposé [OC90, Cou91].

14.1.2 L'opérateur *restrict*

Les règles qui définissent l'opérateur *restrict* (noté $\Uparrow$) sont les mêmes que celles du *constrain*, sauf la troisième règle de descente récursive. C'est en effet celle-ci qui risque d'introduire dans le résultat un nœud qui n'existait pas dans la fonction à simplifier :



$$\alpha \Uparrow \beta \;=\; \alpha \uparrow \beta \quad sinon$$

Cet opérateur, comme le *constrain*, possède la bonne propriété de simplification minimale. Mais en plus, il garantit une qualité relative du résultat. En effet, l'arbre de Shannon du résultat, mesuré en nombre de nœuds, est toujours plus petit ou égal à l'arbre de Shannon de la fonction initiale.

14.1.3 Limitation de l'opérateur *restrict*

L'opérateur *restrict* apporte donc un "plus", mais il ne permet pas toujours d'obtenir des résultats pourtant intuitivement très simples :

$$(\neg x \vee y) \; sachant \; (x \vee y) \;=\; y$$

Avec l'ordre $x < y$, le résultat est $\neg x \vee y$. Par contre, avec l'ordre $y < x$, on obtient bien la fonction y. Les opérateurs *restrict* et *constrain* ont un défaut commun : la synthèse de bas en haut du résultat privilégie en effet la simplification des nœuds les plus profonds dans l'arbre.

14.2 L'ensemble des solutions

14.2.1 Cas général

Nous appelons $under(f, g)$ l'ensemble des fonctions équivalentes à f quand g est vraie :

$$under(f, g) = \{h \in I^{B^n} / \forall x \in B^n (g(x)) \Rightarrow (f(x) = h(x))\}$$

Pour obtenir des algorithmes plus efficaces que le *restrict*, nous avons étudié une méthode en deux phases :

1. Construction d'une représentation de l'ensemble $under(f, g)$

2. Recherche dans cet ensemble de la fonction "la plus simple"

14.2.2 Cas booléen

Dans le cas où f est une fonction booléenne, on peut donner une interprétation intéressante de l'ensemble $under(f, g)$. En effet, la propriété $h \in under(f, g)$ peut se décomposer en :

$g \Rightarrow (f \Rightarrow h)$ *c'est-à-dire* $(g \wedge f) \Rightarrow h$

$g \Rightarrow (h \Rightarrow f)$ *c'est-à-dire* $h \Rightarrow (\neg g \vee f)$

L'ensemble des solutions est donc l'intervalle des fonctions comprises (selon l'implication) entre $g \wedge f$ et $\neg g \vee f$:

$under(f, g) = \{h \ / \ g \wedge f \Rightarrow h \Rightarrow \neg g \vee f\} = [g \wedge f, \neg g \vee f]$

14.3 Les *phi-bdds*

14.3.1 Définition

Pour construire l'ensemble des solutions, nous utilisons une représentation inspirée des *bdds*, dans laquelle les feuilles appartiennent à $I \cup \{\phi\}$. L'interprétation sémantique d'un tel graphe est définie par la fonction $\mathcal{S}$. L'image d'un *phi-bdd* par la fonction $\mathcal{S}$ est un ensemble de fonctions, c'est-à-dire un élément de $\mathcal{P}(B^n \to I)$.

– L'image d'une feuille i est le singleton composé de la fonction qui vaut toujours "i" :

$$\mathcal{S}(i) \quad = \quad \{\lambda X.i\}$$

– L'image de ϕ est l'ensemble de toutes les fonctions de B^n dans I :

$$\mathcal{S}(\phi) \quad = \quad B^n \to I$$

– Enfin, l'image d'un nœud binaire est définie par :

$$\mathcal{S}\left(\begin{array}{ccc} & x & \\ & / \ \backslash & \\ \alpha & & \beta \end{array} \right) \quad = \quad \{h \in B^n \to I \ / \exists f \in \mathcal{S}(\alpha) \ \ \exists g \in \mathcal{S}(\beta) \ \ h = si\ x\, alors\ f\, sinon\ g\}$$

14.3.2 Construction des *phi-bdds*

Grâce à la fonction *apply* (§10.3.3 page 10.3.3), on construit le *phi-bdd* suivant :

$\alpha = apply(\texttt{if}, g, f, \phi)$

On peut alors montrer par induction que α est une représentation de l'ensemble des solutions :

$$\mathcal{S}(\alpha) = under(f, g)$$

La représentation sous forme de *phi-bdd* étant trivialement canonique, on écrira simplement :

$$under(f, g) = apply(\texttt{if}, g, f, \phi)$$

Tout ensemble H qui vérifie la propriété :

$$\exists f \in (B^n \rightarrow I)\ \exists g \in (B^n \rightarrow B) / H = under(f, g) \tag{14.1}$$

peut donc être représenté par un *phi-bdd*.

La réciproque est vraie ; en effet, soit α un *phi-bdd* :

- on construit une fonction $f \in (B^n \rightarrow I)$, en remplaçant dans α chaque occurrence de la feuille ϕ par un élément particulier $i \in I$,
- on construit une fonction booléenne $g \in (B^n \rightarrow B)$, en remplaçant dans α chaque feuille appartenant à I par la valeur 1, chaque ϕ par la valeur 0,
- on a alors trivialement : $\alpha = apply(\texttt{if}, g, f, \phi) = under(f, g)$.

La classe des ensembles de fonctions représentables par un *phi-bdd* est donc exactement celle des ensembles qui nous intéressent. Dans le cas strictement booléen, cela signifie que la classe des ensembles représentables par un *phi-bdd* est celle des intervalles (au sens de l'implication).

14.3.3 Opérations sur les *phi-bdds*

On peut imaginer de nombreuses opérations sur les *phi-bdds*. La plus simple est l'appartenance d'une fonction à un ensemble représenté par un *phi-bdd*.

Pour ce qui est des opérations ensemblistes, le problème est plus complexe. En effet, pour pouvoir implémenter une opération ensembliste, il faut être sûr que le résultat puisse être représenté par un *phi-bdd*. Par exemple, l'union de deux ensembles représentés par des *phi-bdds* ne peut pas toujours être représentée par un *phi-bdd*. Les fonctions booléennes nous fournissent un contre-exemple très simple :

$$\{\lambda X.1\} \cup \{\lambda X.0\} = \{\lambda X.1, \lambda X.0\}$$

Cette union n'étant pas un intervalle, on ne peut pas la représenter par un *phi-bdd*.

Par contre, on peut montrer que l'intersection de deux ensembles vérifiant la propriété 14.1, si elle est non vide, vérifie elle aussi la propriété 14.1. La condition nécessaire et suffisante pour que l'intersection soit non vide peut s'exprimer de la manière suivante :

$$(under(f_1, g_1) \cap under(f_2, g_2)) \neq \emptyset \quad ssi \quad (\forall x \in B^n (g_1(x) \wedge\ g_2(x)) \Rightarrow (f_1(x) = f_2(x)))$$

Si cette propriété est vérifiée, on peut alors construire une fonction f, ou une fonction f' :

$$\begin{aligned} \forall x \in B^n\ f(x) &= si\ g_1(x)\, alors\ f_1(x)\, sinon\ f_2(x) \\ \forall x \in B^n\ f'(x) &= si\ g_2(x)\, alors\ f_2(x)\, sinon\ f_1(x) \end{aligned}$$

et on a trivialement :

$$under(f_1, g_1) \cap under(f_2, g_2) = under(f, g_1 \vee g_2) = under(f', g_1 \vee g_2)$$

14.3.4 Implémentation de l'opérateur $\cap$

L'opérateur $\cap$, qui peut rendre le résultat $\emptyset$, est défini sur les *phi-bdds* par les règles suivantes (la commutativité de l'opérateur est sous-entendue) :

Règles terminales :

$$
\begin{array}{rcll}
\alpha \cap \phi & = & \alpha & \\
i \cap i & = & i & \forall\, i \in I \\
i \cap i' & = & \emptyset & \forall\, i \neq i' \in I
\end{array}
$$

Descente récursive : Seule la première règle est intéressante. Les deux autres consistent simplement à "équilibrer" la descente récursive :

$$
\begin{array}{rcl}
\begin{array}{c} x \\ \diagup\;\diagdown \\ \alpha \quad \beta \end{array} \cap \begin{array}{c} x \\ \diagup\;\diagdown \\ \gamma \quad \delta \end{array} & = & \emptyset \quad si\ (\alpha \cap \gamma = \emptyset)\, ou\ (\beta \cap \delta = \emptyset) \\
 & = & get\text{-}bdd(x, \alpha \cap \gamma, \beta \cap \delta) \quad sinon
\end{array}
$$

$$
\begin{array}{c} x \\ \diagup\;\diagdown \\ \alpha \quad \beta \end{array} \cap\ i = get\text{-}bdd(x, \alpha \cap\ i, \beta \cap\ i)
$$

$$
\begin{array}{c} x \\ \diagup\;\diagdown \\ \alpha \quad \beta \end{array} \cap \begin{array}{c} y \\ \diagup\;\diagdown \\ \gamma \quad \delta \end{array} = get\text{-}bdd\left(x, \alpha \cap \begin{array}{c} y \\ \diagup\;\diagdown \\ \gamma \quad \delta \end{array}, \beta \cap \begin{array}{c} y \\ \diagup\;\diagdown \\ \gamma \quad \delta \end{array}\right) \quad si\ x < y
$$

14.4 Simplification des *phi-bdds*

14.4.1 Algorithme de bas en haut

Simplifier un *phi-bdd* consiste à remplacer les occurrences de ϕ par une valeur permettant d'éliminer un nœud. L'algorithme de simplification le plus évident procède de bas en haut : le résultat partiel ϕ est éliminé par la fonction *combine* (§14.1.1) :

$$
\begin{array}{rcl}
simp\text{-}bu(i) & = & i \\
simp\text{-}bu(\phi) & = & \phi \\
simp\text{-}bu\left(\begin{array}{c} x \\ \diagup\;\diagdown \\ \alpha \quad \beta \end{array} \right) & = & combine(x, simp\text{-}bu(\alpha), simp\text{-}bu(\beta))
\end{array}
$$

Le résultat obtenu est exactement celui de l'opérateur *constrain*, cet algorithme n'a donc qu'un intérêt théorique :

$$
simp\text{-}bu(under(f, g)) = f \uparrow g
$$

14.4.2 Algorithme de haut en bas

De manière duale, nous avons défini un algorithme qui procède de haut en bas, c'est-à-dire qui privilégie l'élimination de la racine du *phi-bdd* :

$$
\begin{array}{rcl}
\textit{simp-td}(i) & = & i \\
\textit{simp-td}(\phi) & = & \phi \\
\textit{simp-td}\left(\begin{array}{ccc} & x & \\ & / \ \backslash & \\ \alpha & & \beta \end{array}\right) & = & \textit{simp-td}(\alpha \cap \beta) \quad \textit{si}\ (\alpha \cap \beta) \neq \emptyset \\
\textit{simp-td}\left(\begin{array}{ccc} & x & \\ & / \ \backslash & \\ \alpha & & \beta \end{array}\right) & = & \textit{get-bdd}(x, \textit{simp-td}(\alpha), \textit{simp-td}(\beta)) \quad \textit{sinon}
\end{array}
$$

A partir de cet algorithme, on définit un nouvel opérateur de simplification, appelé *new-restrict*, et noté $\downarrow$:

$$f \downarrow g \quad = \quad \textit{simp-td}(\textit{under}(f, g))$$

14.4.3 Exemple

Reprenons l'exemple de $(\neg x \vee y)$ *sachant* $(x \vee y)$:

$$
\textit{under}\left(
\begin{array}{ccc} & x & \\ y & & 1 \\ 1 \quad 0 & & \end{array}
,\
\begin{array}{ccc} & x & \\ 1 & & y \\ & & 1 \quad 0 \end{array}
\right)
=
\begin{array}{ccc} & x & \\ y & & y \\ 1 \quad 0 & & 1 \quad \phi \end{array}
$$

Avec l'algorithme "de bas en haut", on commence par simplifier le sous-arbre droit en remplaçant le ϕ par 1, on obtient alors le résultat du *constrain* :

$$
\begin{array}{ccc} & x & \\ y & & y \\ 1 \quad 0 & & 1 \quad 1 \end{array}
=
\begin{array}{ccc} & x & \\ y & & 1 \\ 1 \quad 0 & & \end{array}
$$

Avec l'algorithme "de haut en bas", on essaie en priorité d'éliminer la racine. Le calcul de l'intersection revient à instancier chaque occurrence de ϕ de manière à identifier les deux sous-arbres. Sur cet exemple, on voit qu'en remplaçant ϕ par 0, les deux sous-arbres deviennent identiques. La racine de l'arbre peut alors être éliminée :

$$
\begin{array}{ccc} & x & \\ y & & y \\ 1 \quad 0 & & 1 \quad 0 \end{array}
=
\begin{array}{c} y \\ / \ \backslash \\ 1 \quad 0 \end{array}
$$

14.4.4 Propriétés de l'opérateur *new-restrict*

L'opérateur $\downarrow$ possède trivialement la propriété de simplification minimale. Pour ce qui est de la dualité *constrain*/*restrict*, cet opérateur est plutôt de type *constrain* que *restrict*. En effet, l'ensemble *under*(f, g) peut contenir des nœuds provenant uniquement de la fonction g. Il est donc impossible de garantir que le résultat soit toujours une forme “simplifiée” de f.

L'intérêt essentiel du *new-restrict* est qu'il privilégie la réduction du support de la fonction, c'est-à-dire l'ensemble des variables qui apparaissent dans le *bdd* de la fonction. En effet, si on arrive à supprimer la racine d'un *bdd*, la variable correspondante disparaît du support de la fonction. Avec un algorithme “de bas en haut”, l'élimination totale d'une variable est beaucoup plus aléatoire. Or, réduire le support d'une fonction est très intéressant quand on doit, comme dans notre compilateur, effectuer de nombreuses compositions de fonctions (calcul des préconditions).

Conclusion

Structure générale du générateur de code

Ce programme est écrit en C++ et fonctionne actuellement sur SUN3 et SUN-SPARC. Il s'agit d'un générateur de code et non d'un compilateur complet : il produit du code séquentiel dans le format OC [PS87] à partir d'un programme LUSTRE expansé, statiquement correct. L'analyse statique et l'expansion des nœuds LUSTRE sont réalisées par la partie avant du compilateur POLLUX[Roc92], qui produit un fichier dans le format EC (§2.3 page 22).

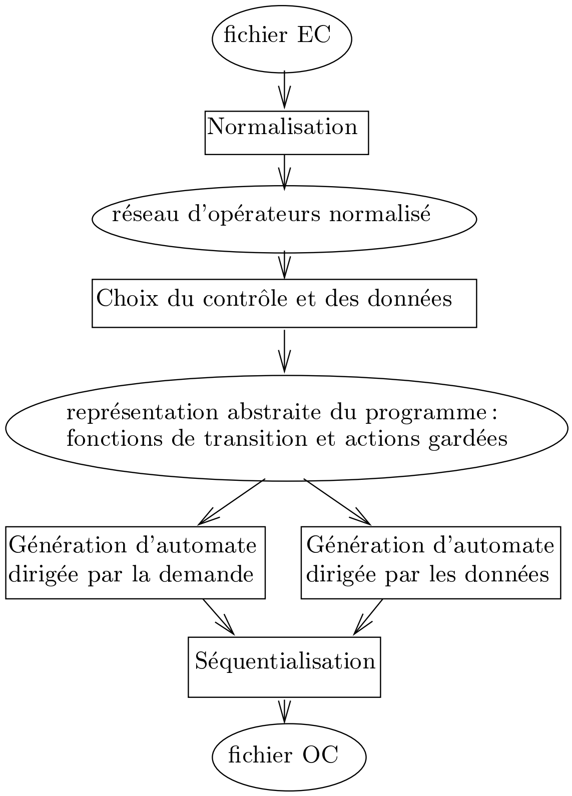


FIG. 14.1 – *Structure du générateur de code*

La figure 14.1 présente la structure de l'outil. Parallèlement aux modules présentés sur cette figure, le générateur comporte deux modules très importants :

– l'un est consacré à la représentation et à la gestion des expressions (réseau d'opérateurs) ;
– l'autre à la représentation et à la manipulation symbolique des fonctions à arguments booléens (graphes binaires de décision).

L'ensemble représente environ 18000 lignes de code C++, et 300 Ko de code exécutable (sur SUN-SPARC).

Utilisation du générateur

La génération de code à partir d'un fichier EC est réalisée par une commande de la forme :

`lv3 [<options>] <ec file>`

On peut cependant générer du code directement à partir d'un fichier LUSTRE grâce à la commande `lus2oc`, qui enchaîne un appel à la partie avant du compilateur POLLUX, et un appel au générateur LUSTRE-V3. Avec cette commande, il faut désigner le nœud qu'on souhaite compiler `<main node>` :

```
lus2oc <lustre file><main node>[<options>]
```

Nous récapitulons ici les différentes options disponibles.

Implémentation des entrées/sorties booléennes

Par défaut, les entrées et sorties booléennes sont implémentées par des signaux valués. Avec l'option `-pure` elles sont implémentées par des signaux purs (§8.3.2 page 60).

Choix du contrôle

Ce choix est paramétré par les options `-0, -1, -2, -3, -4`. La numérotation est historique : il n'y a pas de relation logique entre le numéro de l'option et la finesse du choix effectué. Ces options ne concernent que le choix des variables d'état (§9.6.2 page 70). Le programme choisit au mieux l'ensemble des conditions en tenant compte, bien sûr, des variables d'état, mais aussi d'impératifs liés aux horloges et à la factorisation.

Avec l'option `-0`, toutes les mémoires booléennes sont traitées comme des données. On obtient donc un programme en boucle simple (un seul état). Pour obtenir des automates non triviaux, le programmeur dispose de quatre options, qui déterminent quelles sont les mémoires booléennes retenues comme variables d'état :

– `-1` les `init`, et les `last` apparaissant dans la condition d'un opérateur `if` ;
– `-2` toutes les mémoires booléennes ;
– `-3` les `init`, et les mémoires "récurrentes" ;
– `-4` uniquement les `init`.

Génération de l'automate

Par défaut, celle-ci est réalisée par l'algorithme dirigé par la demande. Le programmeur peut choisir la méthode dirigée par les données avec l'option `-data` (chapitre 11).

Séquentialisation

Les options suivantes correspondent aux variantes de factorisation présentées dans la section 12.5.3 page 111 :

- `-S2` (factorisation faible) : le générateur cherche à éviter la duplication de code en refermant les tests au plus tôt.
- `-S3` (factorisation moyenne) : la duplication d'actions de base est acceptée.
- `-S4` (factorisation forte) : un test ne peut être ouvert qu'une seule fois dans chaque branche du programme ; en contrepartie, la duplication est évidemment très importante.

Une autre variante, que nous n'avons pas présentée dans cet ouvrage, consiste à ne jamais refermer les tests : `-S5`. La duplication inutile étant maximale, le code obtenu est généralement très mauvais. Cette option est cependant utile si on veut visualiser l'automate obtenu avec l'outil Autograph [Roy90].

Traitement des assertions

L'option `-assert` provoque l'évaluation exhaustive des assertions, comme s'il s'agissait de sorties (§13.3 page 115). Il peut parfois arriver que même après simplification, le programme obtenu soit plus gros que si on avait ignoré les assertions. C'est pourquoi l'évaluation est par défaut paresseuse. L'option `-causal` construit une assertion causale à partir des assertions du programme (§13.8 page 120).

Messages d'erreur liés aux assertions

Par défaut, le générateur ne fait aucun traitement particulier pour vérifier la cohérence des assertions ; si un problème lié aux assertions apparaît, la compilation s'arrête sur un message d'erreur :

- `ASSERTIONS STATIQUEMENT CONTRADICTOIRES` : la conjonction des assertions, en temps que fonction booléenne des éléments de contrôle, est identiquement fausse. Cette erreur est détectée lors de la mise sous forme canonique des expressions. Il est clair qu'une telle assertion relève d'une erreur du programmeur.
- `ASSERTIONS NON CAUSALES` : ce message apparaît au cours de la construction de l'automate (qu'elle soit dirigée par la demande ou les données). Il signifie que la prise en compte des assertions a créé un puits dans l'automate, mais pas forcément qu'elles sont contradictoires (§13.7.2 page 120). Le programmeur peut alors essayer de compiler avec l'option `-causal`, qui construit une assertion causale équivalente aux assertions initiales.

- `ASSERTIONS DYNAMIQUEMENT CONTRADICTOIRES` : ce message apparaît au cours de la construction d'une assertion causale (`-causal`) et signifie que les assertions décrivent un comportement temporel impossible des éléments de contrôle. Il s'agit ici d'une véritable erreur de programmation, et non pas, comme pour les assertions non causales, d'un problème de génération de code.

L'évaluation des assertions étant par défaut paresseuse, le générateur peut donc produire du code malgré des assertions contradictoires. Pour être certain de la cohérence des assertions, le programmeur doit utiliser la conjonction d'options : `-assert -causal`.

Perspectives

Choix du contrôle

Le choix d'un "bon" ensemble de variables d'état a une influence primordiale sur la taille du code produit ; en effet, une "mauvaise" variable d'état peut multiplier par deux le nombre d'états, et ceci pour un gain négligeable en temps d'exécution.

Le générateur dispose de plusieurs options pour le choix des variables d'états, basées sur des heuristiques globales (toutes les mémoires récurrentes, toutes les expressions en `init` $\cdots$). On peut imaginer bien d'autres heuristiques plus ou moins complexes, mais il semble peu réaliste d'obtenir un choix automatique toujours satisfaisant.

Il serait donc intéressant de permettre au programmeur de désigner lui-même les variables booléennes qui doivent être traitées comme du contrôle. On pourrait pour cela définir deux types booléens : "booléen-donnée" et "booléen-contrôle". Une telle dualité existe dans les langages synchrones Esterel et Signal [LBBG86, LGLL91], où se côtoient des signaux purs, associés à la notion de contrôle, et des booléens, toujours considérés comme des données.

Compilation séparée

Nous avons vu qu'au cours de l'analyse statique, les nœuds internes sont expansés dans le code du nœud principal. En faisant un parallèle avec les langages classiques, on peut dire que les nœuds Lustre sont utilisés non pas comme des procédures, mais plutôt comme des *macros* dont on doit faire l'inclusion textuelle. Cette méthode est une des causes les plus importantes de l'explosion combinatoire de la taille du code. En effet, après l'expansion d'un nœud B dans le corps d'un nœud principal A, on génère un automate qui est en quelque sorte le produit des automates de A et B. Si par contre on est capable de compiler séparément les deux nœuds, le code résultant n'est que la "somme" des automates de A et B.

La phase d'expansion est nécessaire dans certains cas, comme nous l'avons vu avec l'exemple du nœud `double_copie` (§2.2 page 21). Nous avons cependant montré dans [Ray88] qu'il était toujours possible de limiter l'expansion au strict nécessaire : au cours de l'analyse statique, le nœud à expanser est restructuré en un ensemble minimal de sous-nœuds dont la séquentialisation ne pose pas de problème. Les contraintes de séquencement entre ces différents sous-nœuds sont résumées dans un nœud principal simplifié, qui seul doit être expansé.

La compilation séparée est essentiellement un problème d'analyse statique. Au niveau du générateur de code, il faut cependant prévoir un nouveau type de calcul. En effet, un nœud LUSTRE compilé séparément ne peut pas être traité comme une simple procédure externe. C'est un objet qui comporte une partie opérative (le code séquentiel) partagée par toutes les instances, et une mémoire rémanente (opérateurs `pre` et `current`) propre à chaque instance. De plus, la mémoire interne de chaque instance doit évoluer au rythme de son horloge, et pas seulement quand ses sorties sont nécessaires au nœud principal.

La compilation séparée est en cours de réalisation par C. Dubois, dans le cadre d'un projet CNAM.

Génération d'automate réduit

Dans l'algorithme dirigé par les données, un état de l'automate OC correspond à une valeur du vecteur des variables d'états. La fonction caractéristique est donc un monôme complet, et on peut mettre en œuvre une simple méthode énumérative pour construire l'automate. En revanche, il est fréquent que les classes de l'automate minimal ne soient pas des monômes complets, et qu'on puisse donc l'atteindre avec cette méthode.

Dans l'algorithme dirigé par la demande, un état de l'automate OC correspond à un ensemble quelconque des vecteurs d'états. On est donc amené à manipuler des fonctions booléennes générales. En revanche, on est assuré que la méthode de construction symbolique conduit toujours à l'automate minimal.

Entre ces deux extrêmes, on peut imaginer une solution intermédiaire, imposant que les états soient représentés par des monômes quelconques :

– Moins contraignants que des monômes complets, ils permettent d'obtenir, sinon des automates minimaux, du moins des automates réduits.

– Plus spécifiques que des fonctions quelconques, ils permettent de simplifier considérablement les manipulations symboliques.

Pour mettre en œuvre une telle méthode, la structure générale de l'algorithme dirigé par la demande reste correcte : il suffit d'imposer que les scissions successives (§11.3 page 91) ne portent que sur la valeur d'une variable d'état, et non plus sur une propriété quelconque. Avec cette restriction, de nombreuses optimisations peuvent être envisagées, en utilisant notamment une représentation plus adaptée que les *bdds*.

Le tableau suivant met en évidence la position intermédiaire de ce nouvel algorithme :

Représentation des états :	monômes complets	monômes quelconques	fonctions quelconques
Type d'algorithme :	énumératif	symbolique simplifié	symbolique
Automate obtenu :	quelconque	réduit	minimal

Le code en boucle simple

Dans les algorithmes de séquentialisation, on impose un ordre global des calculs et des tests (§12.5 page 109). Ce point de vue est tout à fait raisonnable quand on doit générer de gros automates : le parcours des états coûte déjà très cher, il faut donc minimiser le temps consacré à la séquentialisation. Avec l'option `-0`, qui produit toujours un seul état, on pourrait consacrer plus de temps à la séquentialisation, et donc améliorer la factorisation [1].

A l'opposé, il serait très intéressant de disposer d'un algorithme qui ne fait aucune factorisation. Le code obtenu est alors une simple séquence d'actions, chaque action correspondant à un seul calcul. Ce type de code est sans doute inefficace, mais sa taille est trivialement linéaire par rapport à celle du programme source.

1. Cette remarque reste valable pour l'option `-1`, qui produit souvent un automate à deux états

Bibliographie

[Ake78] Akers (S. B.). – Binary decision diagrams. *IEEE Transactions on Computers*, vol. C-27, n° 6, juin 1978.

[ASU86] Aho (A.), Sethi (R.) et Ullman (J.). – *Compilers : Principles, Techniques and Tools.* – Addison-Wesley, 1986.

[BCG87] Berry (G.), Couronné (P.) et Gonthier (G.). – Programmation synchrone des systèmes réactifs , le langage ESTEREL. *Technique et Science Informatique*, vol. 4, 1987, pp. 305–316.

[Ber89] Berry (G.). – Real time programming: Special purpose or general purpose languages. *In : IFIP Congress.*

[BFH+90] Bouajjani (A.), Fernandez (J-C.), Halbwachs (N.), Raymond (P.) et Ratel (C.). – Minimal model generation. *In : Workshop on Computer-Aided Verification.*

[Bil87] Billon (J-P.). – *Perfect Normal Forms For Discrete Functions.* – Rapport de Recherchen° n^o 87019, BULL, juin 1987.

[BM88] Billon (J-P.) et Madre (J-C.). – Original concepts of PRIAM, an industrial tool for efficient formal verification of combinational circuits. *In : IFIP WG 10.2 Int Working Conference on the Fusion of Hardware Design and Verification*, éd. par Milne (G.). – North Holland.

[Bry86] Bryant (R. E.). – Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, vol. C-35, n° 8, 1986.

[Buo86] Buors (C.). – *Sémantique opérationelle du langage* LUSTRE. – D.E.A., Université de Grenoble, juin 1986.

[Cou91] Coudert (O.). – *SIAM : une boîte a' outils pour la vérification de systèmes séquentiels.* – Thèse, Ecole Nationale Supe'rieure des Te'le'communications, octobre 1991.

[CPHP87] Caspi (P.), Pilaud (D.), Halbwachs (N.) et Plaice (J. A.). – LUSTRE: a declarative language for programming synchronous systems. *In : 14th ACM Symposium on Principles of Programming Languages.*

[Fer88] Fernandez (J-C.). – *Aldebaran : un système de vérification par réduction de processus communicants.* – Thèse, Université Joseph Fourier, Grenoble, 1988.

[Fer90] Fernandez (J-C.). – An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, vol. 13, n° 2-3, mai 1990.

[Ghe91] Gherardi (G.). – *Le Syste'me* SAHARA. – Rapport de Recherchen° 10/91, CMA Ecole des Mines, juillet 1991.

[Glo89] Glory (A-C.). – *Vérification de propriétés de programmes flots de données synchrones.* – Thèse, Université Joseph Fourier, Grenoble, décembre 1989.

[HCRP91a] Halbwachs (N.), Caspi (P.), Raymond (P.) et Pilaud (D.). – Programmation et vérification des systèmes réactifs à l'aide du langage flot de données synchrone LUSTRE. *Technique et Science Informatique*, vol. 10, n° 2, 1991.

[HCRP91b] Halbwachs (N.), Caspi (P.), Raymond (P.) et Pilaud (D.). – The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE, Special Issue on Synchronous Programming*, vol. A paraître, 1991.

[HL91] Halbwachs (N.) et Lagnier (F.). – *Se'mantique statique du langage* LUSTRE- *Version 3.* – Rapport Techniquen° SPECTRE L15, IMAG, Grenoble, février 1991.

[HP85] Harel (D.) et Pnueli (A.). – On the development of reactive systems. *In : Logic and Models of Concurrent Systems,* NATO *Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems.* – Springer Verlag.

[HRR91] Halbwachs (N.), Raymond (P.) et Ratel (C.). – Generating efficient code from data-flow programs. *In : Third International Symposium on Programming Language Implementation and Logic Programming.*

[Kah74] Kahn (G.). – The semantics of a simple language for parallel programming. *In : IFIP 74.* – North Holland.

[LBBG86] LeGuernic (P.), Benveniste (A.), Bournai (P.) et Gautier (T.). – SIGNAL, a data flow oriented language for signal processing. *IEEE-ASSP*, vol. 34, n° 2, 1986, pp. 362–374.

[LGLL91] LeGuernic (P.), Gautier (T.), LeBorgne (M.) et LeMaire (C.). – Programming real time applications with SIGNAL. *Proceedings of the IEEE, Special Issue on Synchronous Programming*, vol. A paraître, 1991.

[Mor82] Moret (B. M. E.). – Decision trees and diagrams. *ACM Computing Surveys*, vol. 14, n° 4, décembre 1982.

[OC90] O. Coudert (J-C. Madre). – A unified framework for the formal verification of sequential circuits. *In : International conference on computer aided design (ICCAD).*

[Par81] Park (D.). – Concurrency and automata on infinite sequences. *In : 5th GI-Conference on Theorical Computer Science.* – Springer Verlag, 1981. LNCS 104.

[PH87] Plaice (J. A.) et Halbwachs (N.). – LUSTRE-V2 *User's guide and reference manual.* – Rapport Techniquen° SPECTRE L2, IMAG, Grenoble, octobre 1987.

[Pla88] Plaice (J. A.). – *Sémantique et compilation de* LUSTRE*, un langage déclaratif synchrone*. – Thèse, Institut National Polytechnique de Grenoble, 1988.

[PS87] Plaice (J. A.) et Saint (J-B.). – The LUSTRE-ESTEREL portable format. – 1987. Rapport non publié, INRIA, Sophia Antipolis.

[PT87] Paige (R.) et Tarjan (R.). – Three partition refinement algorithms. *SIAM J. Comput., No. 6*, vol. 16, 1987.

[Rat92] Ratel (C.). – LESAR *: un outil pour la vérification d'invariants de programmes* LUSTRE. – Thèse, Université Joseph Fourier, Grenoble, 1992.

[Ray88] Raymond (P.). – *Compilation séparée de programmes* LUSTRE. – D.E.A., IMAG, Grenoble, juin 1988.

[RH91] Rocheteau (F.) et Halbwachs (N.). – Implementing reactive programs on circuits, a hardware implementation of LUSTRE. *In :* REX *Workshop on Real-Time: Theory in Practice, DePlasmolen (Netherlands).*

[RHR91] Ratel (C.), Halbwachs (N.) et Raymond (P.). – Programming and verifying critical systems by means of the synchronous data-flow programming language LUSTRE. *In : ACM-SIGSOFT'91 Conference on Software for Critical Systems,* New Orleans.

[Roc92] Rocheteau (F.). – *Programmation d'un circuit massivement parallèle à l'aide d'un langage déclaratif synchrone*. – Thèse, Institut National Polytechnique de Grenoble, 1992.

[Roy90] Roy (V.). – AUTOGRAPH*, Un outil de visualisation pour les calculs de processus*. – Thèse, Université de Nice, 1990.

[Sha38] Shannon (C. E.). – A symbolic analysis of relay and switching circuits. *Transactions AIEE*, vol. 57, 1938, pp. 305–316.