Sur la répartition de programmes synchrones

Alain Girault

THÈSE

présentée par

Alain GIRAULT

pour obtenir le grade de DOCTEUR

de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

(Arrêté Ministériel du 30 mars 1992)

(Spécialité : Informatique)

Sur la répartition de programmes synchrones.

Date de soutenance : Vendredi 28 janvier 1994

Composition du jury :	Président	G. Mazaré
	Rapporteurs	G. Berry
		P. Le Guernic
	Examinateurs	P. Caspi
		J.L. Bergerand

Thèse préparée dans le cadre d'une convention CIFRE entre l'équipe SPECTRE et la société MERLIN GERIN, sous la direction de P. Caspi et de J.L. Bergerand.

Remerciements

Je tiens à remercier :

Monsieur Guy Mazaré, Directeur de l'ENSIMAG pour m'avoir fait l'honneur de présider le jury de cette thèse.

Messieurs Gérard Berry, maître de recherche à l'Ecole Nationale Supérieure des Mines, et Paul Le Guernic, directeur de recherche à l'INRIA, pour l'intérêt qu'ils ont porté à ce travail et avoir accepté de le juger. Leurs critiques et conseils m'ont beaucoup aidé à améliorer mon manuscrit et je leurs en suis reconnaissant.

Messieurs Paul Caspi, chargé de recherche au CNRS, et Jean-Louis Bergerand, responsable recherche et développement à Merlin Gerin, qui se sont partagés la tâche d'encadrer ce travail et ont su le faire progresser par leurs conseils judicieux.

Messieurs Jean-Claude Fernandez, maître ce conférence à l'Université Joseph Fourier, Claude Girault, professeur à l'Université Paris VI, Nicolas Halbwachs, directeur de recherche au CNRS, et Xavier Nicollin, maître ce conférence à l'Institut National Polytechnique de Grenoble, qui ont également relu mon manuscrit et dont les remarques m'ont été fort précieuses.

Cette thèse a été réalisée, dans le cadre d'une convention CIFRE établie entre le Laboratoire de Génie Informatique de l'IMAG et la société Merlin Gerin. Je remercie donc Chantal Robach, Joseph Sifakis et Jean-Marc Rollet pour m'avoir accueilli au sein de leur unité respective : l'Unité de Génie Matériel et l'équipe Spectre du LGI d'une part, et le service SES à Merlin Gerin d'autre part.

Je remercie également tous les membres de ces équipes avec lesquels j'ai eu grand plaisir à travailler ...

et surtout Isabelle.

Sommaire

Liste des Figures

Introduction

L'approche synchrone

La programmation synchrone a été étudiée dès les années 80 pour faciliter la programmation des *systèmes réactifs*. On peut citer en particulier les travaux de R.Milner sur les calculs de processus synchrones [44].

L'appellation *systèmes réactifs*, introduite par D.Harel et A.Pnueli dans [33], désigne les systèmes informatiques dont le rôle est de réagir continûment à leur environnement physique, celui-ci étant incapable de se synchroniser avec le système, par exemple parce que l'environnement ne peut attendre. Le temps de réponse d'un tel système doit bien évidemment lui permettre de réagir à la vitesse que lui impose son environnement. Cette classe de systèmes s'oppose, d'une part aux systèmes transformationnels (les programmes classiques qui disposent de leurs entrées dès leur initialisation, et qui délivrent leurs résultats lors de leur terminaison), et d'autre part aux systèmes interactifs (qui interagissent continûment avec leur environnement, mais à leur vitesse propre : par exemple les systèmes d'exploitation). La classe des systèmes réactifs englobe la plupart des systèmes industriels dits *temps réel* : systèmes de contrôle-commande, automatismes, systèmes de surveillance de processus, de traitement du signal, mais aussi d'autres systèmes comme les protocoles de communication ou les interfaces homme-machine. Outre leur aspect réactif, ces applications présentent deux autres caractéristiques importantes :

- **Le parallélisme :** un pilote automatique d'avion doit contrôler *en même temps* le roulis et le tangage.
- **La sûreté de fonctionnement :** ces applications sont souvent critiques, et exigent donc une grande sûreté.

En réaction avec les principes du parallélisme informatique, basé sur l'idée d'entrelacement [34], la programmation synchrone s'est fondée sur un principe de simultanéité : toutes les activités parallèles partagent la même échelle de temps, qui est de plus discrète. Toutes les activités peuvent donc être datées sur cette échelle, ce qui présente de nombreux avantages [9] :

- Les raisonnements temporels s'en trouvent facilités.
- L'indéterminisme issu de l'entrelacement disparaît, ce qui facilite la mise au point et la vérification des programmes.

La première idée qui est venue à l'esprit des concepteurs de la démarche a été de projeter cette échelle de temps discrète sur le temps physique. Comme l'échelle est discrète, cela veut dire qu'il ne se passe *rien* entre deux instants consécutifs de cette échelle : tout doit se passer comme si la machine exécutant le programme était infiniment rapide. C'est ce qu'on appelle *l'hypothèse de synchronisme*.

En réalité, une telle machine n'existe pas, mais il suffit pour la simuler que toute entrée puisse être traitée et produise ses sorties avant qu'une nouvelle entrée ne survienne. Pour vérifier que cette condition est bien remplie, il suffit d'avoir des informations sur les fréquences maximales d'entrées, et de pouvoir majorer le temps d'exécution du programme objet correspondant à chaque instant du programme synchrone. Pour ce faire, les langages synchrones se sont restreints volontairement à des constructions qui puissent être compilables en une structure de contrôle sous forme d'automate d'états fini déterministe, dont les transitions soient des programmes séquentiels déterministes et acycliques agissant sur une mémoire finie. Ces transitions, dont le temps d'exécution est mesurable statiquement, correspondent aux "programmes des instants".

Les principaux langages utilisant strictement l'hypothèse de synchronisme sont ESTEREL [9, 10, 12], LUSTRE [18, 29, 30], SIGNAL [38, 39] et ARGOS [41, 35]. On peut citer en outre SAGA [6, 49] qui est un atelier graphique de programmation synchrone fondé sur LUSTRE. SAGA a été initialement développé par MERLIN GERIN et est actuellement industrialisé par VERILOG. Les travaux sur la compilation de ces langages ont conduit à définir un format de codage des automates : c'est le format OC (pour "object code" [50, 47]). Il est le format de sortie des compilateurs des langages ESTEREL, LUSTRE et ARGOS.

La répartition

La démarche que nous venons d'exposer est tout-à-fait cohérente. Cependant, il y a des domaines d'application pour lesquels elle ne suffit pas. Citons-en trois :

- **Les systèmes de traitement de signal complexes :**
  Ces systèmes nécessitent de grosses puissances de calcul. C'est essentiellement le problème auquel s'est attaquée l'équipe SYNDEX de l'INRIA [25] en se proposant de répartir des programmes écrits dans le langage synchrone SIGNAL sur des réseaux de transputers.

- **Les systèmes de contrôle-commande répartis :**
  Ces systèmes sont très souvent répartis géographiquement pour des raisons de localisation de capteurs ou d'actionneurs, de compatibilité de gammes, et de sûreté de fonctionnement. C'est par exemple le cas du système de contrôle-commande de centrale nucléaire CO3N4 développé chez MERLIN GERIN à l'aide de l'atelier SAGA. La solution utilisée est alors de programmer séparément chaque calculateur. Pour être maîtrisable, cette solution exige que l'on étudie soigneusement la décomposition de l'application et peut correspondre à une stratégie du type "diviser pour régner". Cependant, on peut estimer préférable de représenter l'application sous la forme d'un programme synchrone unique, de la mettre au point et de la vérifier sous cette forme, puis de chercher à produire le programme objet de chaque calculateur en garantissant un comportement d'ensemble similaire, en un sens à préciser, à celui qui a été vérifié. Remarquons que, contrairement au cas précédent, le

découpage de l'application en calculateurs n'a pas à être automatique, mais doit obéir à des directives du programmeur.

- **Les tâches de longue durée :**
  Il se peut qu'une même application contienne des composants ayant des dynamiques très diverses, de sorte qu'une machine "infiniment rapide" pour certains composants ne puisse pas être considérée comme telle pour d'autres. Ce cas, qui se produit aussi dans des applications telles que CO3N4, oblige à programmer séparément les divers composants, ce qui n'est pas très grave s'il sont indépendants, mais est plus gênant si ces composants communiquent, car cela rend difficile une vérification d'ensemble. Les solutions que l'on peut proposer sont essentiellement les mêmes que dans le cas précédent, à ceci près que les divers composants pourront aussi bien résider sur le même calculateur, et que l'on parlera donc de répartition en processus plutôt qu'en calculateurs.

C'est aux problèmes soulevés par ces deux derniers cas que nous nous sommes attaqués. Il s'agissait donc :

- de se fixer un environnement d'exécution et des primitives de communication qui permettront aux programmes répartis de s'exécuter et de communiquer,
- de trouver un moyen de produire automatiquement les programmes des processus en fonction d'un programme synchrone unique et de directives de répartition,
- et de chercher à comprendre les similarités existant entre exécutions centralisées et réparties d'un même programme synchrone.

Logiquement, nous aurions dû commencer par le dernier point, puisqu'il pose une question fondamentale, remettant en cause la vision initiale de la programmation synchrone telle que nous venons de l'exposer. Mais, comme c'est souvent le cas, c'est plutôt ici la théorie qui s'est adaptée à la pratique. La solution proposée au chapitre 8 est pourtant simple, et réside dans la dissociation entre le temps logique (celui des instants synchrones) et le temps physique. On aboutit ainsi au concept assez paradoxal d'ordre partiel défini de façon synchrone, c'est-à-dire séquentiellement.

La question des environnements d'exécution a été résolue elle aussi de façon empirique, en s'inspirant de ce qui avait été fait pour les exécutions centralisées. Quant à celle des primitives de communication, elle est intimement liée à la méthode de répartition.

Le cœur de ce travail porte sur la méthode de répartition. L'équipe Ep-Atr de l'INRIA-Rennes qui développe le langage synchrone Signal, et qui s'est posé le même problème, a choisi une méthode consistant à ré-utiliser le compilateur Signal centralisé pour produire les programmes répartis. Le problème s'est alors posé des informations à joindre à chaque fragment du programme à répartir pour que les programmes objet produits coopèrent harmonieusement, par exemple sans interblocage. Ces informations étant, on le verra au chapitre 2, assez nombreuses, le compilateur centralisé a été en fait conçu dès le début comme devant être capable de traiter ces informations. C'est ce qui a été appelé la "méthode d'abstraction de graphes", exposée par O.Maffeïs dans sa thèse [40]. Cette méthode de répartition du code source évite les problèmes d'explosion de la taille du code objet dans le cas d'une grosse application.

Le problème s'est posé différemment à Grenoble. D'une part, le problème de compilation n'a pas été pensé initialement en vue de traiter de telles informations. Et d'autre part, l'optimisation de la compilation centralisée a été poussée plus loin qu'à Rennes (l'algorithme de génération de code "demand-driven" de P.Raymond produisant des automates minimaux [52]), ce qui rendait difficile le contrôle du comportement du compilateur. Il fallait en effet être certain qu'une optimisation sur un fragment ne risquait pas de nuire à la coopération harmonieuse avec les autres fragments. Cela nous a amenés à nous poser le problème de production de programmes répartis en termes d'un postprocesseur du compilateur centralisé : c'est le répartiteur de code OC. Cela, en effet ne présentait pas que des inconvénients :

- Cela permettait de profiter de toutes les améliorations du compilateur centralisé, et il y en a eu de nombreuses [52].
- Cela permettait d'appliquer le répartiteur à tous les autres langages se compilant en OC, entres autres ESTEREL et ARGOS.
- Cela semblait bien cohérent avec la philosophie consistant à mettre au point en centralisé, avec toutes les facilités d'observation que cela suppose, puis ensuite à répartir sans se reposer les questions de mise au point.

Pendant que se déroulait ce travail, de nouveaux résultats sur le même sujet on été obtenus par l'équipe MEIJE de l'INRIA-Sophia-Antipolis qui développe le langage ESTEREL :

- **Sur les tâches asynchrones** : Il s'agit là, contrairement à ce que nous avons appelé tâches de longue durée, de tâches dont la durée est imprévisible. La prise en compte de telles tâches a été obtenue grâce à l'introduction de primitives manipulant des ordres (instantanés au sens du point de vue synchrone) de contrôle de ces tâches [46].
- **Sur les "Communicating Reactive Processes"** : Cela a donné l'idée de faire communiquer entre eux des processus synchrones grâce à ces primitives, chaque processus étant vu par les autres comme une tâche asynchrone [11].
- **Sur la traduction** ESTEREL-LUSTRE : Des préoccupations liées à la conception de circuits ont conduit à la mise en évidence de possibilités de passerelles entre langages synchrones impératifs à la ESTEREL, et déclaratifs à la LUSTRE [8]. Ainsi, un préprocesseur de répartition au niveau LUSTRE aurait pu être utilisé à partir d'ESTEREL ou d'ARGOS.

Enfin, nos travaux ont intéressé C.Jard et B.Caillaud de l'équipe PAMPA de l'IRISA-Rennes. La thèse de ce dernier [14] porte en effet sur la distribution asynchrone d'automates et les programmes réactifs synchrones peuvent tout à fait être décrits dans ce cadre.

Plan de lecture

Les langages synchrones se répartissent en deux catégories, les langages déclaratifs et les langages impératifs. Nous présentons dans le chapitre 1 un représentant de chaque catégorie, LUSTRE et ARGOS. Nous présentons également les techniques de compilation en automate qui leur sont

associées, ainsi que le langage OC, format de codage des automates, sur lequel s'appuie tout notre travail.

Dans le chapitre 2 nous exposons la méthode d'abstraction de graphes dans le cadre du langage LUSTRE, ainsi que le principe de notre méthode de répartition.

L'algorithme de parallélisation avec tout ce qui lui est attaché (expression de la répartition, primitives de communication) est ensuite présenté au chapitre 3. Tout au long, un exemple de programme OC sera étudié en détail afin de bien comprendre les mécanismes mis en jeu. Nous concluons ce chapitre en relevant les défauts de cet algorithmes, sur lesquels nous revenons dans les chapitres ultérieurs.

Puis nous donnons au chapitre 4 la preuve formelle de l'équivalence fonctionnelle entre le comportement du programme centralisé et celui du programme réparti.

Les trois chapitres suivants décrivent des algorithmes palliant des défauts mis en évidence à la fin du chapitre 3, à savoir :

- chapitre 5 : perte de l'équivalence temporelle entre le programme centralisé initial et le programme réparti obtenu,
- chapitre 6 : redondance des messages échangés entre les programmes répartis,
- chapitre 7 : non minimalité des programmes répartis au sens de l'équivalence observationnelle.

Chacune de ces transformations s'applique localement, c'est-à-dire séparément à chaque composantes réparties du programme, mais ne remet pas en cause l'équivalence fonctionnelle entre le comportement du programme centralisé et celui du programme réparti.

Dans le chapitre 8 nous étudions le relâchement du synchronisme qui résulte de la répartition minimale. Nous introduisons pour cela une nouvelle sémantique de LUSTRE qui permet d'expliquer, dans le cadre de ce langage "flots de données", cette perte du synchronisme.

Dans le chapitre 9 nous traitons de l'exécution des programmes répartis et de l'interface entre l'environnement asynchrone et le programme synchrone.

Enfin dans le chapitre 10 nous présentons les réalisations pratiques associées à ce travail, c'est-à-dire le répartiteur de programmes OC, ainsi que la chaîne globale de compilation d'un programme synchrone centralisé vers un système réparti.

Chapitre 1

Les langages synchrones

Nous avons présenté dans l'introduction l'approche synchrone. Cela nous a amenés à introduire le temps discret et l'hypothèse de synchronisme. Nous étudions dans ce chapitre deux exemples de langages illustrant les diverses façons d'appréhender cette approche : tout d'abord nous présentons un langage déclaratif, LUSTRE, puis un langage impératif, ARGOS. Les lecteurs intéressés par une présentation globale des langages synchrones pourront se reporter à [28]. Nous concluons le chapitre en étudiant la compilation en automate des programmes synchrones et le format OC.

1.1 Le langage déclaratif LUSTRE

LUSTRE est un langage à flots de données synchrone inspiré de LUCID [2], des réseaux de Kahn [36], et de divers formalismes utilisés en Automatique, que nous ne détaillerons pas ici. Il appartient à la classe des langages déclaratifs synchrones et est de type fonctionnel. Dans la classe des langages déclaratifs synchrones, on trouve également SAGA qui est l'équivalent graphique de LUSTRE, et SIGNAL qui est relationnel. Nous présentons rapidement l'aspect flot de données du langage, puis sa syntaxe et sa sémantique dans le formalisme fonctionnel introduit par P.Caspi dans [16].

1.1.1 Aspects fondamentaux

LUSTRE est donc un langage déclaratif, c'est-à-dire que les objets manipulés ne sont pas définis par la manière dont on doit les calculer mais par des équations spécifiant leurs propriétés. De plus, c'est un langage flot de données, ce qui implique que les objets manipulés (variables ou expressions) sont des suites éventuellement infinies de valeurs, ce que nous appelons des flots.

Les variables LUSTRE représentent donc des flots. D'une part toute variable est déclarée avec le type de ses valeurs. D'autre part toute variable est ou bien une entrée du programme, ou bien est définie par une équation et une seule. Les équations s'entendent au sens mathématique du terme : l'équation "`x = e;`" signifie que la variable `x` et l'expression `e` ont la même suite de

valeurs.

Ceci donne les deux grands principes de LUSTRE :

- **Principe de substitution :** l'équation "`x = y;`" permet de substituer `x` à `y` et inversement dans tout le programme.
- **Principe de définition :** une variable est complètement définie par sa déclaration et par l'équation où elle apparaît en partie gauche.

Il en résulte que l'ordre des équations est indifférent, de même que l'ajout ou la suppression de variables intermédiaires pour nommer des sous-expressions. En outre, cela suggère naturellement la notion de sous-programme : un ensemble d'équations peut être empaqueté en un nouvel opérateur, appelé *nœud*, et réutilisable dans le programme. Une déclaration de nœud consiste en une spécification d'interface donnant les paramètres d'entrée et de sortie avec leurs types, et un système d'équations définissant les sorties et éventuellement les variables locales en fonctions des entrées et des variables locales.

Nous approfondissons dans les sections suivantes les constructions du langage.

1.1.2 Aperçu de la syntaxe

Le programme suivant définit le nœud `mux` :

```
node mux (m:int) returns (c:bool; y:int);
var x:int;
let
  y = if c then current (x) else pre (y - 1);
  c = true -> (pre (y) = 0);
  x = m when c;
tel;
```

Le nœud `mux` prend en entrée un flot `m` de type entier, et délivre en sortie les flots `c` de type booléen, et `y` de type entier. Pour calculer ses sorties, le nœud `mux` utilise le flot interne `x` de type entier. L'ensemble des équations fournit ensuite la définition de chaque flot local et de sortie, en fonction d'eux-mêmes et des flots d'entrée. Nous expliquons dans la section 1.1.3 ce que calcule le nœud `mux`.

Les expressions utilisées dans les équations sont construites à l'aide de :

- flots constants : `true`, `0`, `1`, ...
- opérateurs usuels étendus aux flots : `-`, `+`, `=` (comparaison), `if then else`, ...
- opérateurs spécifiques du langage : `pre`, `->`, `when`, `current`,
- nœuds définis par l'utilisateur : `mux`.

Il est aussi possible de définir des types externes, ainsi que des fonctions externes importées d'un langage hôte.

1.1.3 Aperçu de la sémantique

Nous nous plaçons dans l'ensemble des suites finies ou infinies d'un domaine de valeurs V :

$$V^\infty = V^* \cup V^\omega$$

Nous adoptons classiquement les notations suivantes :

- "()" représente la suite vide,
- "*nil*" une valeur indéfinie,
- "." l'opérateur de concaténation sur les suites,
- pour tout flot `x`, $tete(\texttt{x})$ représente le premier élément de `x`,
- et pour tout flot `x`, $reste(\texttt{x})$ représente le flot `x` auquel on a enlevé le premier élément.

Nous adoptons de plus la convention typographique selon laquelle `x` désigne toujours un flot, et x une valeur. Nous avons alors les définitions suivantes :

- Les constantes sont les suites infinies de la forme :

  $$\texttt{0} = 0.0$$

  Le tableau suivant donne les suites de valeurs des deux constantes `true` et `0` :

`true`	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	...
`0`	0	0	0	0	0	0	0	0	...

- Les opérateurs usuels s'étendent terme à terme :

  $$\begin{aligned} \texttt{x + y} \quad = \quad & si\ \texttt{x} = ()\ ou\ \texttt{y} = ()\ alors\ () \\ & sinon\ (tete(\texttt{x}) + tete(\texttt{y})).(reste(\texttt{x}) + reste(\texttt{y})) \end{aligned}$$

  Le tableau suivant donne des exemples de suites de valeurs pour les flots `x`, `y` et `x + y` :

`x`	0	3	5	2	12	9	27	10	...
`y`	7	8	35	0	69	1	6	54	...
`x + y`	7	11	40	2	81	10	33	64	...

- L'opérateur `->` sert à définir une valeur initiale :

  $$\begin{aligned} \texttt{x -> y} \quad = \quad & si\ \texttt{x} = ()\ ou\ \texttt{y} = ()\ alors\ () \\ & sinon\ tete(\texttt{x}).second(\texttt{x}, \texttt{y}) \end{aligned}$$

  $$\begin{aligned} second(\texttt{x}, \texttt{y}) \quad = \quad & si\ \texttt{x} = ()\ ou\ \texttt{y} = ()\ alors\ () \\ & sinon\ tete(\texttt{y}).second(reste(\texttt{x}), reste(\texttt{y})) \end{aligned}$$

  Le tableau suivant donne des exemples de suites de valeurs pour les flots `x`, `y` et `x -> y` :

x	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7 ...
y	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7 ...
x -> y	x_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7 ...

- L'opérateur `pre` sert à mémoriser la valeur précédente d'un flot en introduisant un retard :

$$\texttt{pre (x)} = pre(nil, \texttt{x})$$

$$pre(x, \texttt{x}) = si\ \texttt{x} = ()\ alors\ ()\ sinon\ x.pre(tete(\texttt{x}), reste(\texttt{x}))$$

Le tableau suivant donne des exemples de suites de valeurs pour les flots `x` et `pre (x)` :

x	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7 ...
pre (x)	nil	x_0	x_1	x_2	x_3	x_4	x_5	x_6 ...

- L'opérateur `when` est un échantillonneur qui ne transmet la valeur de `x` que lorsque `c` vaut *true* :

$$\texttt{x when c} = si\ \texttt{x} = ()\ ou\ \texttt{c} = ()\ alors\ ()\ sinon\ si\ tete(\texttt{c})\ alors\ tete(\texttt{x}).(reste(\texttt{x})\ \texttt{when}\ reste(\texttt{c}))\ sinon\ reste(\texttt{x})\ \texttt{when}\ reste(\texttt{c})$$

Nous disons alors que le flot `c` est l'*horloge* du flot `x`. Le tableau suivant donne des exemples de suites de valeurs pour les flots `x`, `c` et `x when c` :

x	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7 ...
c	$false$	$true$	$false$	$true$	$false$	$false$	$true$	$true$ ...
x when c		x_1		x_3			x_6	x_7 ...

- L'opérateur `current` sert à projeter un flot après un échantillonnage. Il a une entrée implicite, l'horloge de son argument. Nous définissons dans la section 1.1.4 ce qu'est l'horloge d'un flot.

$$\texttt{current (x)} = current(nil, \texttt{c}, \texttt{x})$$

$$current(x, \texttt{c}, \texttt{x}) = si\ \texttt{c} = ()\ alors\ ()\ sinon\ si\ tete(\texttt{c})\ alors\ si\ \texttt{x} = ()\ alors\ ()\ sinon\ tete(\texttt{x}).current(tete(\texttt{x}), reste(\texttt{c}), reste(\texttt{x}))\ sinon\ x.current(x, reste(\texttt{c}), \texttt{x})$$

Le tableau suivant est un exemple d'échantillonnage et de projection réalisé avec les opérateurs `when` et `current` :

h	$false$	$true$	$false$	$true$	$false$	$false$	$true$	$true$ ...
x	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7 ...
y = x when h		x_1		x_3			x_6	x_7 ...
z = current y	nil	x_1	x_1	x_3	x_3	x_3	x_6	x_7 ...

Etant données toutes ces définitions, le comportement d'un nœud LUSTRE correspond à la plus petite solution de son système d'équations, au sens de l'ordre préfixe sur les flots défini dans [36].

Ainsi le nœud `mux` de la section 1.1.2 décrémente la première valeur de `m` et ré-échantillonne une nouvelle valeur de `m` dès qu'il arrive à 0. Nous pouvons par exemple avoir les flots suivants :

m	3	7	0	8	6	7	1	...
c	*true*	*false*	*false*	*false*	*true*	*false*	*false*	...
x	3				6			...
y	3	2	1	0	6	5	4	...

1.1.4 Aperçu du calcul d'horloge

Naturellement, l'exécution d'un programme LUSTRE ne consiste pas à fournir des flots de sortie en fonction de flots d'entrées, mais correspond à une évaluation paresseuse. Celle-ci consiste à enrichir les flots de sortie au fur et à mesure que des valeurs sont ajoutées en entrée.

Nous souhaitons que cette évaluation paresseuse se fasse en temps de réaction et mémoire bornés (voir l'introduction). Or, pour que cela soit possible, certaines expressions doivent être interdites.

Par exemple, l'expression

```
x + (x when c)
```

ne peut être évaluée en mémoire bornée que si le flot `c` n'a qu'un nombre fini et connu de valeurs *false*, ce qui en pratique est invérifiable. Donc, cette expression doit être interdite.

Le calcul d'horloge de LUSTRE a pour objet d'interdire de telles expressions. Pour cela, une et une seule horloge (flot booléen) doit pouvoir être statiquement attribuée à chaque flot. Intuitivement, tout flot qui n'est jamais échantillonné a pour horloge le flot constant `true`, et tout flot qui est échantillonné a pour horloge le second argument de l'opérateur d'échantillonnage `when`. Cela explique pourquoi il n'est pas nécessaire de fournir l'argument horloge à l'opérateur `current`.

1.1.5 Analyse de causalité

Il peut y avoir en LUSTRE des problèmes de causalité, sous la forme de définitions cycliques. En effet, le principe de définition, que nous avons donné à la section 1.1.1, interdit à une variable de dépendre instantanément d'elle même. Par exemple, il est impossible de donner un sens à la définition `x = x+1`. Une telle définition est considérée comme un interblocage. La détection de ces blocages est effectuée par une simple analyse statique.

1.2 Le formalisme graphique ARGOS

Alors que LUSTRE est un représentant des langages déclaratifs synchrones, ARGOS appartient à la classe des langages impératifs. On trouve également dans cette classe le langage ESTEREL.

La première différence est qu'ARGOS est à la fois graphique et textuel alors qu'ESTEREL est purement textuel. Mais il y a aussi des différences dans les styles de programmation, dues au fait qu'ESTEREL utilise des structures de contrôle complexes alors qu'ARGOS utilise uniquement les automates.

1.2.1 ARGOS et les STATECHARTS

Le plus connu des formalismes graphiques fondés sur une représentation d'automates parallèles et hiérarchisés est les STATECHARTS [32], définis par D.Harel et A.Pnueli. Cependant la sémantique des STATECHARTS n'est pas complètement conforme à la philosophie des langages synchrones dans la mesure où la composition parallèle peut donner naissance à un indéterminisme implicite. C'est pourquoi nous présentons le langage ARGOS, qui adopte la même démarche que les STATECHARTS, mais avec une sémantique complètement formalisée et de plus conforme à l'approche synchrone.

En ARGOS un programme est soit un automate soit le résultat d'un opérateur lui-même appliqué à un ou plusieurs programmes. Chaque automate est dessiné avec des boîtes et des flèches. Chaque opérateur a une syntaxe graphique dont la représentation constitue le squelette du programme.

1.2.2 Automate simple

Un processus simple est directement décrit par un automate. Les états sont nommés, les transitions sont étiquetées et il y a un unique état initial. Les étiquettes des transitions possèdent une partie *entrée* et une partie *sortie*, toutes deux composées d'événements. La partie *entrée* est une conjonction d'événements et de négations d'événements ; par convention, l'événement $\overline{a}$ est la négation de l'événement a. La partie *sortie* est une conjonction d'événements, omise quand elle est vide. Intuitivement, lorsque la partie *entrée* est vérifiée, alors la transition peut être tirée et les événements de la partie *sortie* sont émis.

Si deux transitions partant du même état peuvent être tirées en même temps, le comportement du système est indéterministe. C'est le cas de l'automate de la figure 1.1 si les deux événements a et b surviennent simultanément :

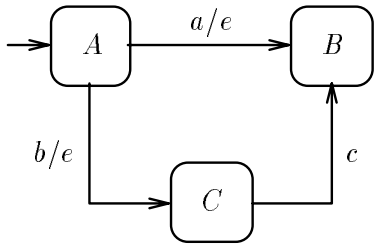


Figure 1.1: Automate ARGOS au comportement indéterministe

Cet indéterminisme explicite n'est pas interdit en ARGOS, mais une option de compilation permet

de vérifier que, dans le programme complet, cet indéterminisme est levé. Pour la figure 1.1, cela peut être vrai si a et b ne sont jamais émis en même temps.

Enfin il existe en ARGOS trois classes d'événements :

- les événements d'entrée : ils ne peuvent pas être émis dans le programme ;
- les événements de sortie : ils ne peuvent apparaître comme entrée sur aucune transition ;
- les événements internes : ils sont déclarés locaux dans un sous-processus.

1.2.3 Mise en parallèle

En ARGOS, la mise en parallèle de deux sous-systèmes se note en les mettant dans une "boîte", et en les séparant par une ligne pointillée :

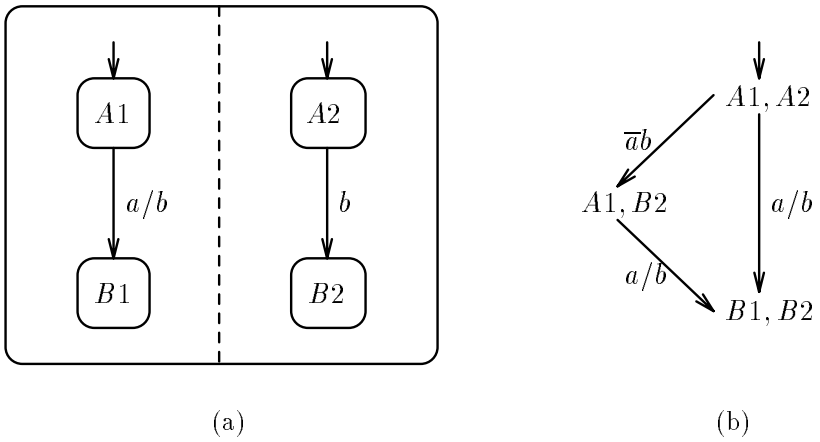


Figure 1.2: Composition parallèle en ARGOS

La figure 1.2 (a) montre la mise en parallèle de deux automates simples qui sont donc les "fils" du processus ainsi représenté. La figure 1.2 (b) donne l'automate équivalent construit selon les règles suivantes :

- L'état initial est le couple formé des états initiaux des deux composants.
- Quand un composant peut réagir à une entrée, il est obligé de le faire. Selon l'entrée, zéro, un ou deux composants peuvent réagir.
- Si les deux composants réagissent en même temps, la sortie est la conjonction des sorties des deux composants.

Remarquons que les composants communiquent entre eux au moyen de l'événement b, et que la communication est instantanée : c'est ce qu'on appelle la "diffusion synchrone". Enfin la composition parallèle est commutative et associative : elle se généralise donc à un nombre quelconque d'arguments.

1.2.4 Déclaration d'événements locaux

Pour rendre un événement d'un processus *local* à ce processus, il suffit de le mettre dans un cartouche rattaché à la boîte du processus. C'est le cas de l'événement b dans la figure 1.3 (a) :

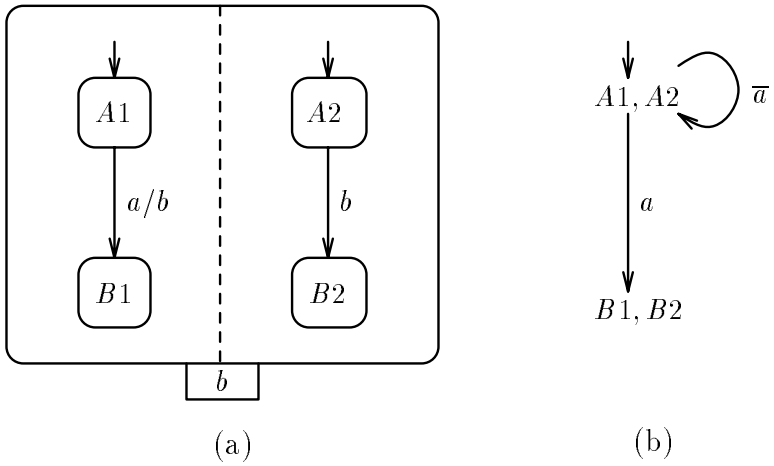


Figure 1.3: Composition parallèle avec un événement local

Cette opération limite la diffusion de ces événements qui sont invisibles à l'extérieur de la boîte et qui ne peuvent parvenir de l'extérieur. La figure 1.3 (b) donne l'automate équivalent : b ayant été rendu local, a est le seul événement d'entrée du processus. Lorsque a survient, il provoque la réaction du premier composant et donc l'émission de b, ce qui provoque alors la réaction du second composant. Ces deux réactions sont synchrones.

1.2.5 Décomposition hiérarchique

La décomposition hiérarchique d'un automate A consiste à considérer certains de ses états comme des processus. Syntaxiquement, on représente le sous-processus à l'intérieur de la boîte de l'état à décomposer :

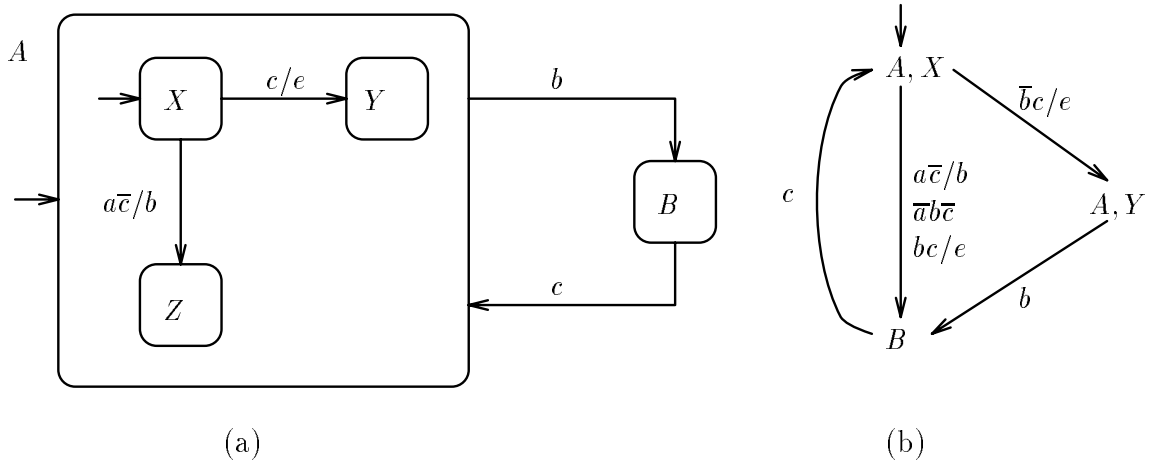


Figure 1.4: Décomposition hiérarchique en ARGOS

Considérons le processus représenté par l'automate de la figure 1.4 (a). En entrant dans l'état A, le sous-processus est activé dans son état initial X. En quittant l'état A, le sous-processus est tué. Les événements émis par un sous-processus, s'ils ne sont pas internes à ce sous-processus, sont visibles par leur "père" : c'est le cas de b. Inversement, tout événement visible par le processus principal est visible de tout sous-processus actif sauf si ce sous-processus a un événement interne de même nom. Ainsi en passant dans l'état Z et en émettant l'événement b, le sous-processus force son "père" à le tuer. La figure 1.4 (b) donne l'automate correspondant au processus de la figure 1.4 (a).

1.2.6 Analyse de causalité

Il peut y avoir en ARGOS des problèmes de causalité. Par exemple certains processus n'ont pas de comportement et d'autres font apparaître un indéterminisme implicite (à différencier de l'indéterminisme explicite mis en évidence à la figure 1.1). Le compilateur effectue donc une détection complète des cycles de causalité. Pour cela il construit l'automate complet du processus et vérifie que, dans chaque état, il existe une unique transition pour chaque monôme des événements d'entrée.

1.2.7 Style de programmation

ARGOS permet aux programmeurs de penser leur application en termes d'automates, ce qui introduit bien évidemment un style de programmation particulier. Ainsi la terminaison d'un processus par suicide (figure 1.4) permet d'implémenter aussi bien la terminaison par exception, la terminaison normale et les interruptions, au prix certes d'une perte de modularité.

1.3 Compilation des langages synchrones

1.3.1 Compilation en automate

Avant la génération de code, les programmes synchrones passent par une phase de vérifications statiques, dont les points les plus significatifs sont : analyse de causalité pour ESTEREL et ARGOS et calcul d'horloges pour LUSTRE, SIGNAL et SAGA. Nous n'entrerons pas dans les détails de ces vérifications, laissant le lecteur se reporter aux articles de référence des langages [10, 41, 30, 38, 6] ou à [28].

C'est pour le langage ESTEREL qu'a été proposée en premier la compilation en automate. Par la suite, cette méthode a été adaptée aux langages LUSTRE et ARGOS.

Le principe consiste à tirer partie du déterminisme du langage. Celui-ci permet en effet de construire à la compilation l'arbre des comportements du programme. Pour un processus ARGOS, il suffit de construire l'automate qui lui est équivalent. Dans le cas d'ESTEREL, l'évaluation des instructions de contrôle permet, à partir d'un état donné du programme et des événements d'entrée, de connaître le prochain état du programme. Enfin dans le cas de LUSTRE, ce sont les opérations sur les booléens (conditionnelles et changements d'horloges) qui représentent le contrôle ; la synthèse de la structure de contrôle consiste donc à simuler à la compilation le comportement de ces booléens.

Nous obtenons donc l'arbre des comportements du programme. Cet arbre est certes infini, mais il peut être replié en un graphe fini, c'est-à-dire un automate d'états fini comportementalement équivalent au programme initial, et couplé à une mémoire bornée contenant les valeurs des variables non booléennes. Le succès de cette méthode de compilation est assuré d'une part par le déterminisme du langage, et d'autre part par les vérifications statiques qui sont effectuées pendant la première phase.

Le synchronisme des langages limite l'explosion du nombre des états : les transitions ne sont engendrées que par des événements externes et toutes les opérations internes sont exécutées simultanément et factorisées. De plus dans le cas d'ESTEREL et de LUSTRE, l'automate produit est généralement minimal, au sens du critère de la bisimulation forte [21]. Dans tous les cas, il est possible d'appliquer un algorithme de minimisation. Enfin, pour des questions évidentes de gain de place mémoire, les actions de l'automate sont factorisées dans une table et identifiées dans le programme par leur index.

Les avantages d'une telle méthode de compilation sont nombreux :

- Le programme équivalent obtenu est *séquentiel*.
- La structure en automate d'états fini permet de valider l'hypothèse de synchronisme.
- On peut appliquer par la suite tous les outils travaillant sur des automates :
  - générateurs de code,
  - outils de minimisation d'automates,
  - interfaces avec des outils de preuve formelle,

- outils de visualisation d'automates,
- générateurs d'interface,
- et répartiteurs de code.

1.3.2 Les formats communs

Les récents travaux sur les formats communs aux langages synchrones ont conduit à proposer trois formats intermédiaires : IC ("imperative code") pour les langages impératifs, GC ("graph code") pour les langages déclaratifs [47], et OC ("object code") pour le codage des automates. C'est ce qu'illustre la figure 1.5 :

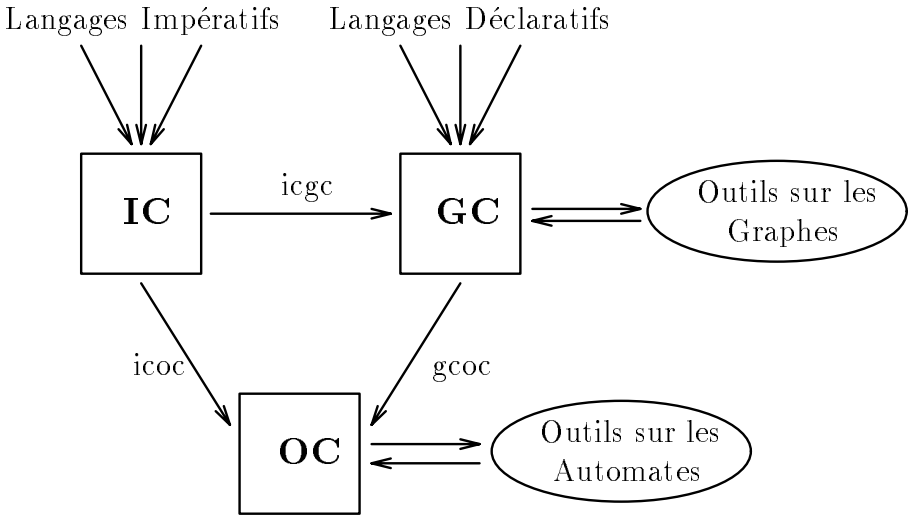


Figure 1.5: Formats communs des langages synchrones

1.3.3 Le format OC

Nous concluons par une présentation sommaire du format OC.

Nous venons de voir qu'un programme OC comprend une liste d'états, avec pour chacun d'entre eux du code sans boucle correspondant à une réaction atomique du programme. Quant aux actions présentes dans le code, elles sont de deux types :

- Actions de contrôle :
  - branchements binaires : `if` (test d'une expression), `dsz` (pour "decrement skip to zero") et `present` (test de présence d'un signal),
  - changements d'état : `goto`.
- Actions séquentielles :

- affectations de variables : `x:=e`,
- émissions de sorties : `output(s)`,
- appels de procédures externes.

A cela il faut ajouter une action *implicite* `go` de synchronisation avec l'environnement. En effet, la notion d'état en OC correspond à celle de réaction atomique du programme : une transition est une réaction à un stimulus de l'environnement. Ceci est matérialisé par l'action `go` présente à chaque début d'état. Celle-ci peut être vue comme une attente de valeurs des signaux d'entrée de la part de l'environnement.

Les entrées sont donc matérialisées par des mémoires mises à jour par l'interface du programme. Dans la version OC2, elles peuvent être ou bien des signaux ou bien des capteurs ("sensor" dans la terminologie OC). Toutefois, la dernière version OC5 a supprimé les capteurs. Les sorties quant à elles sont exclusivement des signaux dont la valeur doit être explicitement émise dans le corps du programme.

En outre, un programme OC se présente sous la forme d'une procédure qui effectue à chaque appel *une* transition de l'automate. Il faut donc y adjoindre une interface qui tourne en boucle perpétuelle et qui est chargée de récolter les entrées du programme et d'activer l'automate.

Enfin pour des raisons de lisibilité, tous les programmes OC que nous donnons dans les chapitres suivants sont présentés sous une forme "expansée", c'est-à-dire que les actions y apparaissent en clair et non plus sous forme d'index référant à la table des actions.

Chapitre 2

Problématique de la répartition

2.1 L'abstraction de graphes et le compilateur LUSTRE

LUSTRE étant un langage parallèle à flots de données, nous pourrions penser, a priori, qu'il est possible de compiler séparément et vers du code séquentiel des paquets d'équations issues d'un programme, en remplaçant les variables partagées par des lectures ou des écritures dans des files "fifo", avec les conventions suivantes :

- la fonction `put(c,v)` met la valeur `v` à la fin de la file `c`,
- et la fonction `get(c)` bloque si la file `c` est vide, et sinon extrait la valeur de tête de la file `c` et rend cette valeur comme résultat.

L'exemple du nœud `double_copie`, dû à G.Gonthier [27] montre qu'il n'en est rien :

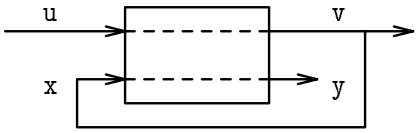


Figure 2.1: Appel bouclé d'un nœud LUSTRE

Les équations LUSTRE correspondant au programme de la figure 2.1 sont les suivantes :

```
y = x;
v = u;
x = v;
```

Nous choisissons de les fractionner en deux paquets de la façon suivante :

```
(1) y = x;                                  (2) x = v;
(1) v = u;
```

où les annotations (1) et (2) désignent le fractionnement en paquets. Ce programme peut donner les programmes séquentiels suivants :

```
prog1:                                      prog2:
 for(;;){                                    for(;;){
   put("y",get("x"));                          put("x",get("v"));
   put("v",get("u"));                        }
 }
```

Il est facile de voir que ce programme réparti est bloquant.

La solution proposée par O.Maffeïs dans [40] consiste, en première approximation, à ajouter à chaque fragment de programme les contraintes d'ordonnancement des actions séquentielles qui résultent des autres fragments. C'est ce qu'on appelle les abstractions des autres fragments.

Par convention, l'abstraction "`v >> x`" signifie que `v` doit être calculé avant `x`. L'exemple précédent donne alors :

```
(1) y = x;                                  (2) x = v;
(1) v = u;
(1) v >> x;
```

Nous obtenons alors par compilation séparée le programme réparti correct :

```
prog1:                                      prog2:
 for(;;){                                    for(;;){
   put("v",get("u"));                          put("x",get("v"));
   put("y",get("x"));                        }
 }
```

Remarquons cependant que, dans l'exemple précédent, le fragment `1` ne crée pas de contrainte dans le fragment `2`. Nous pouvons alors considérer l'exemple suivant :

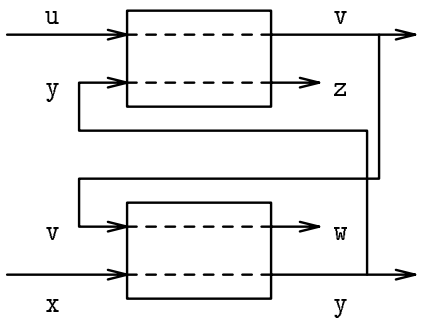


Figure 2.2: Deux appels bouclés entrecroisés d'un nœud LUSTRE

Ses équations LUSTRE sont :

```
(1) z = y;                                  (2) y = x;
(1) v = u;                                  (2) w = v;
```

Nous constatons qu'aucun fragment ne crée de contrainte dans l'autre. Cependant, nous pouvons obtenir par compilation séparée un programme réparti incorrect :

```
prog1:                                      prog2:
 for(;;){                                    for(;;){
   put("z",get("y"));                          put("w",get("v"));
   put("v",get("u"));                          put("y",get("x"));
 }                                           }
```

Le problème est ici que, bien qu'aucun fragment n'induise de contrainte dans l'autre, tout ordonnancement de l'un crée immédiatement une contrainte sur l'autre, et réciproquement. Ces difficultés conduisent à des taxinomies complexes des fragments qui sont ou non compilables séparément, qui sont à rapprocher des relations de compilation séparée de P.Raymond [51].

Une autre difficulté de la compilation séparée tient au traitement des conditionnelles, illustré par l'exemple suivant :

```
(1) z = if c then y else x;                 (2) y = g(v);
(1) v = f(u);
(1) v >> y;
```

Le code naïf :

```
prog1:                                      prog2:
 for(;;){                                    for(;;){
   put("v",f(get("u")));                       put("y",g(get("v")));
   y = get("y");                             }
   if(get("c")) put("z",y);
   else put("z",get("x"));
 }
```

n'est pas optimal par rapport à celui qui ne calcule `y` qu'en cas de besoin :

```
prog1:                                      prog2:
 for(;;){                                    for(;;){
   if(get("c")) {                              put("y",g(get("v")));
     put("v",f(get("u")));                   }
     put("z",get("y"));
     }
   else put("z",get("x"));
 }
```

En revanche, la compilation `prog1` n'est pas correcte si le fragment 2 exécute un calcul séquentiel :

```
(1) z = if c then y else x;                 (2) y = y0 >> g(pre y,v);
(1) v = f(u);
(1) v >> y;
```

Nous constatons que `v >> y` n'est pas une abstraction assez détaillée de `2`, et que la relation d'ordonnancement doit être renseignée par des conditions de calcul. C'est ce qui est proposé pour SIGNAL dans [40], ainsi que dans le format commun GC [47], où ces conditions de calcul sont des horloges. En LUSTRE, cela pose une difficulté supplémentaire du fait que l'ordonnancement peut être exprimé par des horloges *ou bien* par des expressions conditionnelles.

2.2 Principe de la répartition

Nous avons présenté dans l'introduction plusieurs motifs de répartition :

- implémentation physique ou imposée par la localisation des capteurs et des actionneurs,
- tolérance aux fautes,
- et amélioration des performances.

Remarquons que cette répartition est le plus souvent dirigée par le concepteur du système. C'est à ce cas que nous allons nous intéresser. Par conséquent, il faut des directives de répartition qui dépendent évidemment du langage de programmation. Nous pouvons par exemple assigner chaque variable du programme à un et un seul site, un site désignant une machine ou même un simple processus. Par la suite, chaque variable ne sera calculée que sur son site.

Il y a, a priori, trois stratégies possibles pour parvenir à cette répartition :

- Compiler séparément les fragments de programme source de chaque site et les faire communiquer. Ce pourrait être la solution idéale car elle semble être la plus simple. Malheureusement, nous avons vu dans la section 2.1 qu'en toute généralité, la compilation séparée de tels fragments de programmes en programmes séquentiels déterministes n'est pas correcte. P.Raymond propose toutefois dans [51] des critères pour déterminer si un fragment de programme LUSTRE est compilable séparément, c'est-à-dire indépendamment de son contexte d'appel. De même, ESTEREL fournit des critères pour compiler séparément des modules (mode "cascade"). En revanche, la compilation séparée en programmes exécutés non séquentiellement est toujours possible. C'est d'ailleurs la solution adoptée dans le système SYNDEX pour les programmes SIGNAL [25].
- Obtenir par compilation globale un programme séquentiel pour chaque fragment, de telle sorte que ces programmes puissent communiquer harmonieusement. C'est la solution qui a été retenue pour les programmes SIGNAL (voir à ce sujet l'introduction).
- Compiler le programme source en un programme séquentiel OC unique, puis paralléliser ce programme en autant de programmes qu'il y a de sites, de sorte que chaque site n'effectue que les calculs qui lui incombent. Cela permet notamment de mettre au point au préalable le code séquentiel centralisé. Un tel algorithme de parallélisation a été proposé dans [13].

Nous donnons au chapitre 3 une version améliorée de l'algorithme de parallélisation proposé dans [13]. Cet algorithme procède de la façon suivante : l'automate est répliqué sur tous les sites

de la répartition, puis seules sont conservées les actions qui dépendent du site courant ; il faut alors rajouter des actions de communication afin que les programmes s'échangent leurs variables.

C'est une méthode analogue qui est présentée par D.Callahan et K.Kennedy dans [15] : l'application est tout d'abord programmée dans un langage classique (en l'occurrence FORTRAN auquel ont été rajoutées des actions permettant de spécifier une répartition), puis ce programme est réparti brutalement en plusieurs programmes communiquant entre eux par des échanges de messages asynchrones.

Les programmes ainsi obtenus sont certes particulièrement inefficaces, mais dans la mesure où le programme de départ est écrit en FORTRAN, nous pouvons lui appliquer tous les algorithmes de transformation existants (propagation des constantes, élimination des sous-expressions, transformation des boucles ...). L'application de ces algorithmes d'optimisation permet alors d'obtenir des programmes répartis utilisant au mieux la répartition spécifiée.

Chapitre 3

Répartition fonctionnelle

3.1 Introduction

Nous avons motivé dans l'introduction la répartition des systèmes réactifs. Puis nous avons étudié diverses solutions dans le chapitre 2 et décidé de passer d'abord par une phase de compilation en automate (décrite à la section 1.3.1). Nous exposons dans ce chapitre notre algorithme de parallélisation.

Il procède de la façon suivante : les directives de répartition de l'utilisateur conduisent à localiser chaque action sur un unique site ; puis l'automate est répliqué sur tous les sites, et sur un site donné, seules sont conservées les actions localisées sur ledit site ; des communications sont alors ajoutées afin que les programmes des différents sites s'échangent les variables dont ils ont besoin. L'intérêt d'une telle méthode est d'une part de profiter des améliorations du programme centralisé, et d'autre part d'appliquer a posteriori des optimisations sur le programme réparti. Nous étudions ces optimisations aux chapitres 6 et 7.

Nous présentons tout d'abord les directives de répartition, puis les primitives de répartition et les hypothèses qui sont faites sur le réseau. Nous exposons alors les étapes successives de l'algorithme, avec notamment deux stratégies de placement des communications. Enfin nous comparons ces deux stratégies et nous concluons sur les problèmes restant à résoudre.

Tout au long, un exemple de programme OC sera étudié en détail afin de bien comprendre les mécanismes mis en jeu.

3.1.1 Expression de la répartition

Les directives de répartition doivent se traduire en fin de compte par la localisation de chaque action sur un site. Cette localisation doit bien entendu être unique et non ambiguë.

En Lustre, les primitives de répartition vont porter sur les entrées/sorties et les variables locales du nœud principal : l'utilisateur indique pour chacune de ces variables le site sur lequel elles doivent être calculées ; puis par propagation dans le réseau flot de données, toutes les variables

du programme sont affectées à un unique site.

En ESTEREL, chaque objet du programme OC généré par le compilateur est annoté avec sa provenance dans le programme source. Il est par conséquent trivial de localiser les variables, signaux et capteurs du programme.

En ce qui concerne ARGOS, c'est moins facile. L'utilisateur peut certes spécifier pour chaque événement d'entrée, de sortie ou local, son site d'implantation. Mais contrairement à LUSTRE et ESTEREL, le style de programmation impératif empêche de voir comment les variables du programme vont être localisées à partir de ces directives.

Quant à OC, le seul lien de communication entre un programme et son environnement (et par conséquent avec l'utilisateur) est matérialisé par les signaux et les capteurs (il est à ce sujet recommandé de se reporter au manuel de référence OC [50]). Les directives de répartition vont donc porter sur ces signaux et capteurs. En pratique, l'utilisateur doit donner la liste des sites sur lesquels il veut que son programme soit réparti, avec pour chacun d'entre eux la liste des signaux et capteurs qui vont en dépendre. Ces directives de répartition doivent permettre la localisation de façon non ambiguë de toutes les variables (et indirectement de toutes les actions) du programme. Dans le cas contraire, l'utilisateur doit reformuler ses directives de répartition.

3.1.2 Primitives de communication

Il reste à spécifier le moyen de communication utilisé. Les langages temps réel classiques (ADA, OCCAM...) sont basés sur les différentes formes de rendez-vous. On dit souvent que le rendez-vous est un moyen de communication synchrone. Certes, l'échange de message est synchrone, mais l'établissement du rendez-vous est asynchrone [7]. En outre, le rendez-vous impose des temps d'attente inutiles, et il a de plus été montré que les communications asynchrones réduisent le parallélisme et l'efficacité [23]. C'est pourquoi ce moyen fut d'emblée rejeté.

Les files d'attente offrent l'avantage indéniable de permettre le décalage entre l'émission et la réception, ce qui permet de minimiser les temps d'attente et d'être ainsi le moins sensible possible au temps de réponse du réseau. C'est donc le moyen de communication que nous avons retenu.

Il reste encore à choisir le type et le placement des files d'attente. Entre deux sites communicants, nous pouvons avoir :

- Une file par site, ce qui nécessite d'identifier les messages, sauf si le protocole de communication assure que l'ordre des messages est préservé. Dans ce cas il suffit que les émissions soient effectuées sur le site émetteur dans le même ordre que les réceptions sur le site récepteur. C'est en outre la solution retenue dans [15].

- Une file par variable, ce qui est plus souple mais beaucoup plus gourmand au niveau de l'environnement d'exécution (le nombre de variables étant a priori toujours plus grand que le nombre de sites).

Nous avons privilégié l'économie d'environnement et donc choisi la première solution.

Concrètement, quand un site a besoin de la valeur d'une variable appartenant à un autre site, la valeur de celle-ci lui est envoyée par l'intermédiaire d'une file d'attente "fifo". Nous faisons sur le réseau l'hypothèse suivante :

Hypothèse 3.1 (réseau de communication)
Le réseau de communication préserve l'ordre et l'intégrité des messages.

Nous disposons pour ces émissions de deux files d'attente par paire de sites, une dans chaque sens. Les échanges de messages se font au moyen des actions suivantes :

- Pour une émission : `put(destination,valeur)` où `destination` désigne le site vers lequel on émet.
- Pour une réception : `variable:=get(source)` où `source` désigne le site de qui on reçoit.

Il reste a choisir entre des files d'attente bornées et non bornées :

- Des files non bornées imposent que la réception soit bloquante quand la file est vide. Par contre, l'émission n'est jamais bloquante.
- Des files bornées imposent que la réception soit bloquante quand la file est vide, et que de plus l'émission soit bloquante quand la file est pleine. Cette solution est par conséquent plus coûteuse puisqu'à chaque émission il faut tester l'état de la file d'attente.

Là encore, le souci d'économie nous a fait choisir la première solution.

3.1.3 Présentation de l'algorithme

Le but de la répartition est donc d'obtenir, à partir d'un programme OC et de directives de répartition, n programmes OC communiquant au moyen de files d'attente, et dont le comportement global soit fonctionnellement équivalent à celui du programme initial. Le problème consiste donc à placer les `put` et les `get` de telle sorte que les valeurs émises soient affectées aux bonnes variables.

Le code dans chaque état est purement séquentiel. C'est donc par souci de simplicité que l'algorithme de répartition que nous présentons opère au niveau des états.

L'algorithme se décompose en trois étapes :

- réplication et assignation,
- placement des émissions,
- placement des réceptions.

Nous présentons successivement ces trois étapes.

3.1.4 Notations

Pour toute la suite du chapitre, nous définissons les prédicats suivants :

- Une action *dépend* d'un site si et seulement si ce site doit exécuter ladite action.
- Une variable *dépend* d'un site si et seulement si ce site calcule localement ladite variable ; nous disons également que ce site est *propriétaire* de cette variable.
- Un site a *besoin* d'une variable si et seulement si ce site doit exécuter une action utilisant ladite variable, en partie droite d'une affectation ou dans un branchement.

3.2 Présentation d'un exemple

Pour chaque étape, nous étudions l'application des divers algorithmes mis en œuvre au travers d'un exemple de programme OC dont nous ne donnons que le code de l'état 0, mis en forme pour des raisons de lisibilité :

state 0

`if (x) then`
`  y:=x;`
`  output(y);`
`else`
`  x:=true;`
`  y:=x;`
`  output(y);`
`endif`
`goto 1;`

Nous décidons de répartir ce programme sur deux sites, avec les directives de répartition suivantes : `y` appartient au site 0 et `x` au site 1.

3.3 Etape de réplication et d'assignation

Le problème est d'attribuer un site à toutes les variables du programme. Au départ, seuls les signaux, ainsi que les variables qu'ils véhiculent, et les capteurs sont localisés. Par propagation, nous attribuons à chaque variable un site unique de calcul.

L'algorithme d'assignation consiste alors pour chaque action à déterminer la liste des sites qui doivent l'exécuter, c'est-à-dire :

- une action de contrôle (`if`, `dsz`, `present` ou `goto`) : tous les sites,
- une affectation : le site dont *dépend* la variable affectée.

Le contrôle est entièrement répliqué sur chaque site : tous les fragments de programmes ont en effet la même structure de contrôle. Nous obtenons avec notre exemple la localisation suivante :

site	state 0
(0,1)	`if (x) then`
(0)	`y:=x;`
(0)	`output(y);`
(0,1)	`else`
(1)	`x:=true;`
(0)	`y:=x;`
(0)	`output(y);`
(0,1)	`endif`
(0,1)	`goto 1;`

Pour chaque action, nous avons indiqué la liste des sites qui doivent l'exécuter, c'est-à-dire la liste des sites dont *dépend* ladite action.

3.4 Etape de placement des émissions (put)

Nous travaillons successivement sur chaque état de l'automate. Dans chaque état, le programme se présente sous la forme d'un graphe orienté sans circuit dont les nœuds sont des actions et les feuilles des `goto`. Le principe de l'algorithme de placement des émissions est d'associer à chaque site `s` un ensemble $\texttt{Need}_\texttt{s}$ contenant toutes les variables dont le site `s` va certainement avoir *besoin*, si leur valeur n'a pas été envoyée précédemment par leur site *propriétaire* respectif. Le calcul des ensemble $\texttt{Need}_\texttt{s}$ permet alors de placer les `put` de telle sorte qu'un site qui a *besoin* de la valeur d'une variable pour une action donnée la reçoive avant cette action. Nous donnons deux variantes de l'algorithme :

- La stratégie *si besoin* où une variable est envoyée au moment où le site de destination en a effectivement besoin.
- La stratégie *au plus tôt* où une variable est envoyée au moment où elle est calculée.

3.4.1 Stratégie *si besoin*

L'algorithme de placement des émissions consiste, pour chaque site `s`, à placer aux feuilles du graphe un ensemble $\texttt{Need}_\texttt{s}$ initialement vide, puis à propager ces ensembles jusqu'à la racine du graphe (propagation "en arrière") de la façon suivante :

- Quand on atteint une action qui *dépend* du site `s`, si pour cette action le site `s` a *besoin* d'une variable `x` qui *dépend* d'un site autre que `s`, alors on ajoute `x` à l'ensemble $\texttt{Need}_\texttt{s}$ (les branchements ont eux aussi *besoin* de variables).

- Quand on atteint une affectation `x:=exp`, pour tous les sites `s` tels que $\texttt{x} \in \texttt{Need}_\texttt{s}$, on insère l'action `put(s,x)` immédiatement après l'affectation (forcément, la variable `x` ne *dépend* d'aucun de ces sites `s`). Puis on retire `x` de chaque ensemble $\texttt{Need}_\texttt{s}$ concerné.

- Quand on atteint une fermeture de test, on duplique les ensembles $\texttt{Need}_\texttt{s}$, et on continue dans chaque branche `then` et `else`.

- Quand on atteint un test `if`, `dsz` ou `present`, pour chaque site `s` :

  - on construit l'intersection des ensembles $\texttt{Need}_\texttt{s}^\texttt{then}$ et $\texttt{Need}_\texttt{s}^\texttt{else}$ venant des deux branches `then` et `else` ;
  - dans la branche `then` (resp. `else`), on insère une action `put(s,x)` pour chaque variable `x` de l'ensemble $\texttt{Need}_\texttt{s}^\texttt{then} - (\texttt{Need}_\texttt{s}^\texttt{then} \cap \texttt{Need}_\texttt{s}^\texttt{else})$; autrement dit on envoie une variable au moment où le site cible en a besoin et non au moment où elle est calculée ;
  - on continue avec l'intersection $\texttt{Need}_\texttt{s}^\texttt{then} \cap \texttt{Need}_\texttt{s}^\texttt{else}$.

- Quand on atteint la racine du graphe, pour chaque site `s` on insère au début du graphe une action `put(s,x)` pour chaque variable `x` de l'ensemble $\texttt{Need}_\texttt{s}$.

Avec notre exemple, nous obtenons le placement des émissions suivant :

site	state 0	$\texttt{Need}_0$	$\texttt{Need}_1$	
(1)	`put(0,x);`	$\emptyset$	$\emptyset$	③
(0,1)	`if (x) then`	{x}	$\emptyset$	
(1)	`  put(0,x);`	$\emptyset$	$\emptyset$	②
(0)	`  y:=x;`	{x}	$\emptyset$	
(0)	`  output(y);`	$\emptyset$	$\emptyset$	
(0,1)	`else`			
(1)	`  x:=true;`	$\emptyset$	$\emptyset$	
(1)	`  put(0,x);`	$\emptyset$	$\emptyset$	①
(0)	`  y:=x;`	{x}	$\emptyset$	
(0)	`  output(y);`	$\emptyset$	$\emptyset$	
(0,1)	`endif`	$\emptyset$	$\emptyset$	
(0,1)	`goto 1;`	$\emptyset$	$\emptyset$	

L'algorithme a inséré trois `put` :

- le `put(0,x)` numéro ① parce que $\texttt{x} \in \texttt{Need}_0$ et `x` est affectée par le site 1 ;
- le `put(0,x)` numéro ② parce que $\texttt{x} \in \texttt{Need}_0^\texttt{then} - (\texttt{Need}_0^\texttt{then} \cap \texttt{Need}_0^\texttt{else})$;
- le `put(0,x)` numéro ③ parce que $\texttt{x} \in \texttt{Need}_0$ et on atteint la racine du graphe.

3.4.2 Stratégie *au plus tôt*

Le but est ici d'insérer chaque `put` le plus tôt possible, c'est-à-dire juste après le dernier calcul de la variable envoyée. Dans ce cas l'algorithme de placement des `put` est le même que l'algorithme précédent, sauf au moment où on traite un test `if`, `dsz` ou `present` : il faut alors continuer avec l'union des deux ensembles $\texttt{Need}_s$ provenant des deux branches `if` et `else` et non plus avec l'intersection. De plus, il n'y a pas de `put` à insérer après l'action de test :

site	state 0	$\texttt{Need}_0$	$\texttt{Need}_1$	
(1)	`put(0,x);`	$\emptyset$	$\emptyset$	②
(0,1)	`if (x) then`	{x}	$\emptyset$	
(0)	`y:=x;`	{x}	$\emptyset$	
(0)	`output(y);`	$\emptyset$	$\emptyset$	
(0,1)	`else`			
(1)	`x:=true;`	$\emptyset$	$\emptyset$	
(1)	`put(0,x);`	$\emptyset$	$\emptyset$	①
(0)	`y:=x;`	{x}	$\emptyset$	
(0)	`output(y);`	$\emptyset$	$\emptyset$	
(0,1)	`endif`	$\emptyset$	$\emptyset$	
(0,1)	`goto 1;`	$\emptyset$	$\emptyset$	

L'algorithme a cette fois-ci inséré deux `put` :

- le `put(0,x)` numéro ① parce que $\texttt{x} \in \texttt{Need}_0$ et `x` est affectée par le site 1 ;
- le `put(0,x)` numéro ② parce que $\texttt{x} \in \texttt{Need}_0$ et on atteint la racine du graphe.

Nous comparons ces deux stratégies à la section 3.6, après avoir étudié le placement des réceptions.

3.4.3 Réalisation et complexité

Pour effectuer les calculs de complexité algorithmique, nous adoptons les notations suivantes :

- Q représente l'ensemble des états de l'automate, et $|Q|$ représente son nombre d'états.
- nb_{site} représente le nombre de sites de la répartition.
- $nb_{var}(s)$, nb_{var} et moy_{var} représentent respectivement le nombre de variables du site s, le nombre total de variables et le nombre moyen de variables de chaque site.
- $nb_{act}(q)$ et moy_{act} représentent respectivement le nombre d'actions de l'état q et le nombre moyen d'actions dans chaque état.
- Pour toute fonction f, $\mathcal{T}(f)$ et $\mathcal{M}(f)$ représentent ses coûts de calcul en temps et en mémoire.

L'implémentation utilise la représentation des variables OC sous forme d'indice référant à la table des variables. Les ensembles $\texttt{Need}_\texttt{s}$ sont donc mémorisés sous la forme de tableaux de bits, la case n contenant 1 si la variable d'indice n appartient à l'ensemble considéré, et 0 sinon. De la sorte, les opérations d'ajout, de retrait et les tests d'appartenance aux ensembles $\texttt{Need}_\texttt{s}$ sont réalisées en $O(moy_{var})$. Les opérations sur les ensembles, intersection et union, sont réalisées en $O(moy_{var})$: ce sont des opérations logiques sur des booléens. Le graphe des actions étant mémorisé sous la forme de listes chaînées, l'insertion d'une nouvelle action est réalisée en $O(1)$.

Toutes les opérations pouvant être effectuées en $O(moy_{var})$, le coût en temps du placement des `put` est moy_{var} fois le coût du parcours du graphe des actions de chaque état, soit $moy_{var} \times \sum_{q \in Q} nb_{act}(q)$. Donc :

$$\mathcal{T}(\text{placement des } \texttt{put}) = O(|Q| \times moy_{act} \times moy_{var})$$

Quant au coût en mémoire, il est égal à la taille des ensembles $\texttt{Need}_\texttt{s}$, soit $nb_{site} \times \sum_s nb_{var}(s)$. Donc :

$$\mathcal{M}(\text{placement des } \texttt{put}) = O(nb_{site} \times nb_{var})$$

3.5 Etape de placement des réceptions (`get`)

Comme pour les émissions, nous travaillons successivement sur chaque état. Il ne reste plus qu'à placer les `get`, de telle sorte que les actions `x:=get(s)` apparaissent dans le programme du site `t` dans le même ordre que les `put(t,x)` dans le programme du site `s`. L'hypothèse 3.1 page 39 assure alors que les valeurs échangées entre deux sites au moyen d'un `put`/`get` correspondent toujours aux mêmes variables d'un côté et de l'autre.

Le principe de l'algorithme de placement des `get` est de simuler à tout instant l'état des files d'attente. Pour cela nous définissons pour chaque couple de sites (`t`,`s`) une file d'attente $\texttt{Fifo}_{\texttt{t}\triangleright\texttt{s}}$ contenant les variables du site `t` envoyées au site `s`. Ces variables seront insérées dans la file d'attente suivant l'ordre dans lequel elles sont envoyées. L'algorithme consiste alors à placer à la racine du graphe du programme une file $\texttt{Fifo}_{\texttt{t}\triangleright\texttt{s}}$ vide pour chaque couple de sites (`t`,`s`), puis à propager ces files jusqu'aux feuilles du graphe (propagation "en avant") de la façon suivante :

- Quand on atteint une action `put(s,x)` sur le site `t`, on ajoute `x` à la queue de la file $\texttt{Fifo}_{\texttt{t}\triangleright\texttt{s}}$.
- Quand on atteint une action qui *dépend* du site `s`, si pour cette action le site `s` a *besoin* d'une variable `x` qui *dépend* d'un autre site `t`, alors nécessairement $\texttt{x} \in \texttt{Fifo}_{\texttt{t}\triangleright\texttt{s}}$. On extrait alors la tête de la file $\texttt{Fifo}_{\texttt{t}\triangleright\texttt{s}}$ jusqu'à ce que la variable `x` soit extraite, et pour chaque variable `h` extraite, on insère l'action `h:=get(t)` sur le site `s`. Ainsi on assure que les variables sont extraites de la file d'attente exactement dans le même ordre suivant lequel elles y ont été insérées.
- Quand on atteint un test `if`, `dsz` ou `present`, on duplique les files $\texttt{Fifo}_{\texttt{t}\triangleright\texttt{s}}$, et on continue l'algorithme dans les deux branches `then` et `else`.
- Quand on atteint une fermeture de test, pour chaque couple de sites (`t`,`s`) :

- on détermine le plus grand suffixe commun $\texttt{Suff}_{t\triangleright s}$ des deux files $\texttt{Fifo}^{\texttt{then}}_{t\triangleright s}$ et $\texttt{Fifo}^{\texttt{else}}_{t\triangleright s}$ venant des deux branches `then` et `else` ; ce préfixe commun contient les variables, envoyées par le site `t` au site `s`, qui sont situées à la queue des deux files d'attente, et donc qui ont été envoyées le plus récemment ; ici le but est de placer les réceptions le plus tard possible, afin de minimiser le temps d'attente imposé par le réseau ;
- on construit la file $\texttt{Rest}^{\texttt{then}}_{t\triangleright s} = \texttt{Fifo}^{\texttt{then}}_{t\triangleright s} - \texttt{Suff}_{t\triangleright s}$ (resp. `else`) ;
- dans la branche `then` (resp. `else`), on vide la file $\texttt{Rest}^{\texttt{then}}_{t\triangleright s}$ (resp. `else`), et pour chaque variable `h` extraite en tête de la file, on insère l'action `h:=get(t)` sur le site `s` ;
- on continue l'algorithme avec $\texttt{Suff}_{t\triangleright s}$.

- Quand on atteint une feuille, pour chaque couple de sites (`t`,`s`), on vide la file $\texttt{Fifo}_{t\triangleright s}$, et pour chaque variable `h` extraite en tête de la file, on insère l'action `h:=get(t)` sur le site `s`.

3.5.1 Stratégie *si besoin*

Le tableau suivant montre ce que donne cet algorithme avec notre programme d'exemple, dans le cas où les `put` ont été placés avec la stratégie *si besoin* :

site	state 0	$\texttt{Fifo}_{1\triangleright 0}$	$\texttt{Fifo}_{0\triangleright 1}$	
(1)	`put(0,x);`	···x	··	
(0)	`x:=get(1);`	··	··	①
(0,1)	`if (x) then`	··	··	
(1)	`put(0,x);`	···x	··	
(0)	`x:=get(1);`	··	··	②
(0)	`y:=x;`	··	··	
(0)	`output(y);`	··	··	
(0,1)	`else`			
(1)	`x:=true;`	··	··	
(1)	`put(0,x);`	···x	··	
(0)	`x:=get(1);`	··	··	③
(0)	`y:=x;`	··	··	
(0)	`output(y);`	··	··	
(0,1)	`endif`	··	··	
(0,1)	`goto 1;`	··	··	

L'algorithme insère donc trois `get` :

- le `x:=get(1)` numéro ① parce que $x \in \texttt{Fifo}_{1\triangleright 0}$ et le site 0 a *besoin* de `x` pour effectuer le test `if (x)` ;
- le `x:=get(1)` numéro ② parce que $x \in \texttt{Fifo}_{1\triangleright 0}$ et le site 0 a *besoin* de `x` pour effectuer l'affectation `y:=x` ;
- le `x:=get(1)` numéro ③ parce que $x \in \texttt{Fifo}_{1\triangleright 0}$ et le site 0 a *besoin* de `x` pour effectuer l'affectation `y:=x`.

Et donc le programme final est :

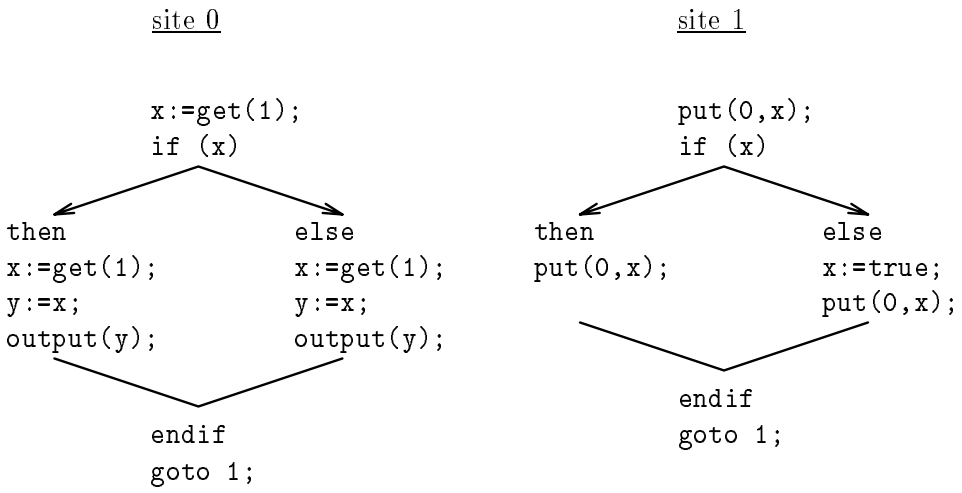


Figure 3.1: Programme OC réparti sur deux sites : stratégie *si besoin*

3.5.2 Stratégie *au plus tôt*

Le tableau suivant montre ce que donne cet algorithme avec notre programme d'exemple, dans le cas où les `put` ont été placés avec la stratégie *au plus tôt* :

site	state 0	$\texttt{Fifo}_{1\triangleright 0}$	$\texttt{Fifo}_{0\triangleright 1}$	
(1)	`put(0,x);`	$\cdots$[x]	$\cdot\cdot$[·]	
(0)	`x:=get(1);`	$\cdot\cdot$[·]	$\cdot\cdot$[·]	①
(0,1)	`if (x) then`	$\cdot\cdot$[·]	$\cdot\cdot$[·]	
(0)	`y:=x;`	$\cdot\cdot$[·]	$\cdot\cdot$[·]	
(0)	`output(y);`	$\cdot\cdot$[·]	$\cdot\cdot$[·]	
(0,1)	`else`			
(1)	`x:=true;`	$\cdot\cdot$[·]	$\cdot\cdot$[·]	
(1)	`put(0,x);`	$\cdots$[x]	$\cdot\cdot$[·]	
(0)	`x:=get(1);`	$\cdot\cdot$[·]	$\cdot\cdot$[·]	②
(0)	`y:=x;`	$\cdot\cdot$[·]	$\cdot\cdot$[·]	
(0)	`output(y);`	$\cdot\cdot$[·]	$\cdot\cdot$[·]	
(0,1)	`endif`	$\cdot\cdot$[·]	$\cdot\cdot$[·]	
(0,1)	`goto 1;`	$\cdot\cdot$[·]	$\cdot\cdot$[·]	

L'algorithme insère donc deux `get` :

- le `x:=get(1)` numéro ① parce que $x \in \texttt{Fifo}_{1\triangleright 0}$ et le site 0 a *besoin* de `x` pour effectuer le test `if (x)` ;
- le `x:=get(1)` numéro ② parce que $x \in \texttt{Fifo}_{1\triangleright 0}$ et le site 0 a *besoin* de `x` pour effectuer l'affectation `y:=x`.

Et donc le programme final est :

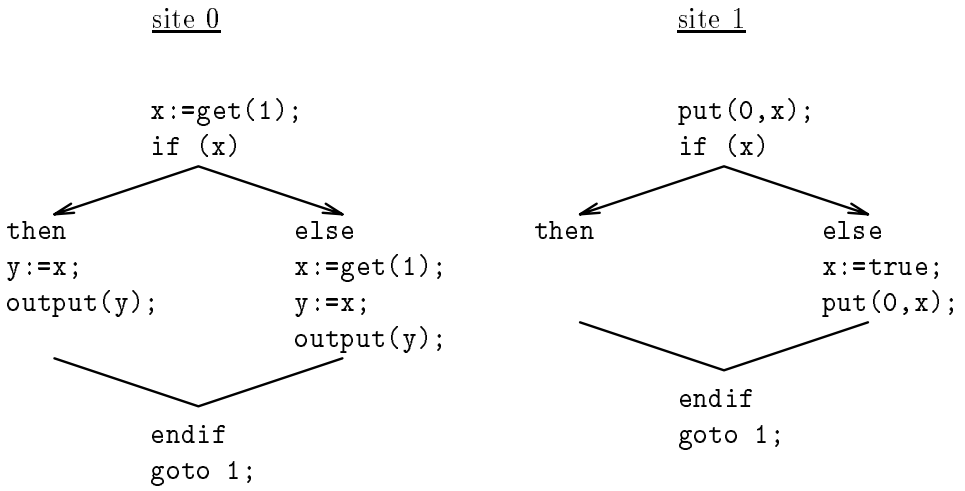


Figure 3.2: Programme OC réparti sur deux sites : stratégie *au plus tôt*

3.5.3 Réalisation et complexité

Nous nous posons à présent le problème de l'implémentation à la compilation des files d'attente $\texttt{Fifo}_{\texttt{t}\triangleright\texttt{s}}$: nous choisissons pour cela une liste chaînée, en gardant un pointeur sur le premier élément et sur le dernier. Ainsi insérer une nouvelle variable ou retirer la tête de file peut se faire en $O(1)$. Reste le calcul du préfixe maximal de deux files d'attente : pour simplifier, nous décidons que la taille de toute file $\texttt{Fifo}_{\texttt{s}\triangleright\texttt{t}}$ est $nb_{var}(s)$, et donc le temps d'un calcul d'un préfixe est $O(nb_{var}(s))$. En conservant les notations introduites à la section 3.4.3, nous obtenons :

$$\mathcal{T}(\text{placement des } \texttt{get}) = O(|Q| \times moy_{act} \times moy_{var})$$

Quant au coût en mémoire, c'est la taille des files $\texttt{Fifo}_{\texttt{s}\triangleright\texttt{t}}$, soit $\sum_s \sum_t nb_{var}(s)$. Donc :

$$\mathcal{M}(\text{placement des } \texttt{get}) = O(nb_{site} \times nb_{var})$$

En définitive, le placement des `put` et des `get` étant deux opérations consécutives, nous avons :

- $\boxed{\mathcal{T}(\text{répartition fonctionnelle}) = O(|Q| \times moy_{act} \times moy_{var})}$
- $\boxed{\mathcal{M}(\text{répartition fonctionnelle}) = O(nb_{site} \times nb_{var})}$

3.6 Comparaison des stratégies *si besoin* et *au plus tôt*

La stratégie *au plus tôt* minimise les délais de communication. En effet, pour une variable donnée, le `put` est placé juste après le calcul de cette variable, soit le plus tôt possible, alors que le `get`

est placé juste avant son utilisation, soit le plus tard possible. Par contre, quand une variable n'est nécessaire que dans une seule des deux branches d'un test, alors il y aura un message inutile dans l'autre branche. Au contraire, la stratégie *si besoin* minimise le nombre de messages mais induit des temps d'attente plus longs, parce que le `get` est bloquant.

Considérons par exemple le programme OC suivant :

```
state 0
y:=10;
if (c) then
     x:=y;
else
     y:=y-1;
endif
goto 1;
```

Nous décidons de répartir ce programme sur deux sites, avec les directives suivantes : `y` appartient au site 0 et `x` et `c` appartiennent au site 1. Après localisation, duplication et placement des `put` et des `get`, nous obtenons :

si besoin		*au plus tôt*	
site	state 0	site	state 0
(1)	`put(0,c);`	(1)	`put(0,c);`
(0)	`y:=10;`	(0)	`y:=10;`
(0)	`c:=get(1);`	(0)	`put(1,y);`
(0,1)	`if (c) then`	(0)	`c:=get(1);`
(0)	`  put(1,y);`	(0,1)	`if (c) then`
(1)	`  y:=get(0);`	(1)	`  y:=get(0);`
(1)	`  x:=y;`	(1)	`  x:=y;`
(0,1)	`else`	(0,1)	`else`
(0)	`  y:=y-1;`	(0)	`  y:=y-1;`
(0,1)	`endif`	(1)	`  y:=get(0);`
(0,1)	`goto 1;`	(0,1)	`endif`
		(0,1)	`goto 1;`

Faisons quelques remarques.

La valeur de la variable `c` est échangée de la même façon quelle que soit la stratégie. Le `put` est fait le plus tôt possible, juste après la mise à jour de `c` qui est faite implicitement au début de l'état. Le `get` est fait le plus tard possible, juste avant l'utilisation de `c`, au moment du test `if (c)`.

Quand à la variable `y`, cela dépend de la stratégie choisie :

- Stratégie *si besoin* : la valeur de la variable `y` n'est échangée que dans la branche `then` où elle est nécessaire ; le `put` est fait le plus tôt possible, juste après le test ; le `get` est fait le plus tard possible, juste avant l'utilisation de `y`.

- Stratégie *au plus tôt* : la valeur de la variable `y` est échangée dans les deux branches, même dans la branche `else` où elle est inutile ; le `put` est fait le plus tôt possible, juste après le calcul de `y` ; les `get` sont fait le plus tard possible, dans la branche `then` juste avant l'utilisation de `y` et dans la branche `else` juste avant la fermeture du test ; enfin le message inutile ne peut pas être supprimé : le `put` est fait avant le test, et donc on doit obligatoirement avoir un `get` dans chaque branche.

Par conséquent, avec la stratégie *au plus tôt* et dans le cas où `c` vaut `false`, la valeur de `y` est différente à la fin de l'état 0 sur le site 0 (où elle vaut 9) et sur le site 1 (où elle vaut 10). Mais cela n'est pas incohérent puisque si dans les états suivants le site 1 a de nouveau besoin de `y`, alors sa dernière valeur lui sera à nouveau envoyée.

Par ailleurs, quand une variable est envoyée avant un test, puis rafraîchie dans une des deux branches alors que sa valeur est requise dans les deux, alors la stratégie *si besoin* génère un message inutile. Il apparaît donc qu'avec cette stratégie, le nombre de messages n'est pas non plus minimal. Toutefois, la différence avec la stratégie *au plus tôt* est que cette fois ci, le message en trop est redondant (c'est-à-dire que la valeur échangée est déjà connue par le site qui la reçoit) et va donc pouvoir être éliminé grâce aux techniques usuelles d'analyse statique. Nous étudions cette optimisation en détail dans le chapitre 6.

Enfin le choix de la stratégie est laissé à l'utilisateur sous la forme d'une option de compilation (voir le chapitre 10).

3.7 Conclusion et problèmes à résoudre

Les langages synchrones permettent de programmer des systèmes réactifs en conservant leur parallélisme naturel au niveau du programme. La méthode que nous venons de présenter permet de produire automatiquement du code séquentiel réparti, à partir d'un programme synchrone. Dans la mesure où le programme est dans un premier temps compilé, analysé et corrigé sur un monoprocesseur, cette méthode permet de produire du code réparti avec la même sûreté que s'il s'agissait de code séquentiel simple. Enfin, les programmes répartis obtenus ainsi ne nécessitent pas de noyau temps réel compliqué pour leur exécution, puisque les communications sont réduites à des protocoles très simples, à savoir des files d'attente.

Même si l'automate du programme initial est minimal, au sens du critère de la bisimulation forte [21], il se peut que les programmes répartis ne le soient plus, dans la mesure où l'on a effacé des actions qui contribuaient éventuellement à distinguer des états de l'automate initial. Il est alors toujours possible de minimiser indépendamment chaque automate réparti. Cependant, on s'aperçoit alors souvent que cette minimisation pourrait être plus poussée si l'on pouvait appliquer un critère d'équivalence observationnelle, dans le cas de branches n'effectuant aucune action visible. Nous étudions au chapitre 7 comment appliquer les techniques de bisimulation à la répartition fonctionnelle.

D'autre part nous avons vu à la section 3.6 que la procédure de placement des émissions pouvait, dans certains cas, générer des messages redondants. Nous voyons au chapitre 6 comment, par analyse statique globale des programmes répartis, on peut minimiser les communications.

De plus, l'algorithme que nous venons de présenter fournit un programme réparti sans interblocage et ayant la même sémantique fonctionnelle que le programme initial. Cependant, rien n'assure que la sémantique temporelle soit préservée. Par exemple, il se peut que certains sites se comportent strictement comme des producteurs de valeurs vis-à-vis des autres sites. Puisque l'émission n'est jamais bloquante, ces sites producteurs peuvent alors s'emballer et provoquer un débordement des files d'attente. Nous étudions au chapitre 5 comment on peut respecter la sémantique temporelle des programmes répartis.

Enfin, un des problèmes importants qui se pose dans le cas de tout algorithme de parallélisation est l'équivalence fonctionnelle entre le comportement du programme centralisé et celui du programme réparti. Nous donnons au chapitre 4 une preuve de cette équivalence fonctionnelle. Pour cela nous utilisons une représentation des automates OC par des systèmes de transitions étiquetées et une formalisation de l'algorithme par des fonctions de transformation, dont nous montrons qu'elles préservent le comportement du programme centralisé.

Chapitre 4

Preuve de l'algorithme de parallélisation

Nous avons présenté au chapitre 3 une façon d'obtenir, à partir d'un programme OC centralisé et de directives de répartition, n programmes OC communicant au moyen de files d'attente. Nous montrons dans ce chapitre que le comportement global des n programmes répartis est fonctionnellement équivalent à celui du programme initial centralisé.

Cette preuve repose d'une part sur une représentation formelle des programmes OC sous forme de système de transitions étiquetées, inspirée du calcul CCS de Milner [43]. D'autre part la synchronisation induite par la définition que nous donnons de la composition parallèle s'apparente à celle des rendez-vous, au sens de la mise en commun de points d'ordres partiels, et ne réalise pas réellement la communication par files d'attente telle que le fait l'algorithme de parallélisation. Mais une modélisation plus réaliste et opérationnelle poserait des problèmes complexes de comportements infinis et d'équité. Représenter formellement les communications par files d'attente imposerait en effet d'introduire les valeurs transmises et de représenter les programmes par des fonctions, comme cela est fait dans [4]. Toutefois, cela n'est pas limitatif car un rendez-vous peut être transformé, d'un côté en une écriture dans une file d'attente, et de l'autre en une lecture dans cette même file d'attente.

La particularité de cette preuve réside dans le modèle théorique de l'algorithme de parallélisation. Il utilise en effet une représentation interne des programmes sous forme de systèmes de transitions étiquetées par des ordres partiels et non plus par des actions. Chacune de ces étiquettes est donc un ordre partiel représentant une action *et* les dépendances qu'elle crée avec les autres actions du programme. Ces dépendances sont directement déduites des dépendances de données existant naturellement dans le programme centralisé initial. A partir de cette représentation interne, répartir un programme OC revient à faire une projection sur chaque site. Il suffit alors de transformer les dépendances restantes en des primitives de communication. En définitive, notre modèle théorique comporte trois étapes : transformation du système de transitions étiquetées par les actions du programme OC en un système de transitions étiquetées par des ordres partiels ; projection de ce système sur chaque site ; remplacement des dépendances en primitives de communication. La preuve de l'algorithme de parallélisation consiste donc à montrer que chacune des trois étapes préserve le comportement du programme centralisé initial.

Nous présentons tout d'abord la représentation formelle des programmes OC, c'est-à-dire les systèmes de transitions étiquetées, puis les bases théoriques des ordres partiels et le modèle théorique de notre algorithme de parallélisation, avant d'en donner la preuve proprement dite.

4.1 Formalisation de l'approche

4.1.1 Système de transitions étiquetées du programme OC

Nous avons présenté le format OC dans la section 1.3.3. En résumé, un programme OC est un automate d'états fini déterministe couplé à une mémoire bornée. Dans chacun de ses états, il y a du code purement séquentiel. Ce code est représenté sous la forme d'un graphe d'actions orienté et sans circuit (DAG pour "directed acyclic graph"), obtenu au moyen des constructeurs suivants :

- des actions correspondant à des affectations de variables internes, des émissions de signaux ou des appels de procédures externes ;
- une action implicite `go` de synchronisation avec l'environnement ;
- des branchements binaires déterministes de la forme $cq + c'q'$, où c et c' sont respectivement l'action d'évaluation à **vrai** et à **faux** d'une expression booléenne ;
- le ";" permettant d'exécuter deux actions en séquence ;
- le `goto` permettant de construire l'automate.

Nous pouvons par conséquent représenter un programme OC par un système de transitions étiquetées fini, déterministe et binaire $S = \{Q,\ q,\ A,\ \rightarrow\}$, tel que :

- Q est l'ensemble des états de S : ce sont les états de tous les DAGS des états de l'automate OC.
- q appartenant à Q est l'état initial : c'est l'état initial du DAG de l'état initial de l'automate OC.
- A est l'alphabet des actions : ce sont toutes les actions de l'automate OC (actions visibles), plus les actions internes (invisibles), c'est-à-dire les changements d'état et fermetures de test. Les actions visibles sont chacune représentées par une étiquette distincte.
- $\rightarrow$ est la fonction de transition définie de $Q \times A$ dans Q.

4.1.2 Représentation algébrique des systèmes de transitions étiquetées

Les définitions suivantes formalisent cette représentation :

Définition 4.1 (*trd*)
Un terme CCS régulier déterministe binaire (au sens où le facteur de branchement est au plus 2), noté *trd*, est défini par :

$$q ::= nil \mid x \mid aq \mid cq + c'q' \mid recx.q$$

où x appartient à un ensemble fini de variables, a, c et c' appartiennent à l'alphabet fini des actions A, et $c \neq c'$.

Nous définissons alors les actions effectuées par les *trd* de la façon suivante :

Définition 4.2 (actions d'un *trd*)

- $aq \xrightarrow{a} q$
- $cq + c'q' \xrightarrow{c} q$
- $cq + c'q' \xrightarrow{c'} q'$
- si $q \xrightarrow{a} q'$ alors $recx.q \xrightarrow{a} q'[recx.q/x]$

Un système de transitions étiquetées déterministe fini binaire est alors défini par :

Définition 4.3 (*sted*)
Un système de transitions étiquetées déterministe fini binaire, noté *sted*, est un *trd* fermé (toutes les variables sont liées par un *rec*) et bien gardé (toutes les variables sont sous la portée d'une action : il n'y a pas de boucle vide telle que $recx.x$).

Un tel système $\{Q,\ q,\ A,\ \rightarrow\}$ sera identifié avec son état initial q.

4.1.3 Comportement d'un *sted*

Soit $\{Q,\ q,\ A,\ \rightarrow\}$ un *sted*. Nous notons $[\![q]\!]$ son comportement, défini formellement par :

Définition 4.4 (comportement d'un *sted*)
Soit $\{Q,\ q,\ A,\ \rightarrow\}$ un *sted*. Son comportement $[\![q]\!]$ est l'ensemble des traces maximales, finies ou infinies, qu'il peut générer : $[\![q]\!] \subseteq A^\infty = A^\star \cup A^\omega$.

Par "trace" nous entendons "mot reconnu" par le système de transitions étiquetées en considérant que tous ses états sont terminaux. Une trace est alors maximale si et seulement si le système de transitions étiquetées ne peut plus reconnaître d'action.

Enfin à toute séquence d'exécution d'un *sted*, nous pouvons associer une unique trace de son comportement définie par :

Définition 4.5 (trace associée à une séquence d'exécution)
Pour toute séquence d'exécution σ du *sted* q de la forme : $\sigma = q \xrightarrow{a_0} q_1 \cdots q_{n-1} \xrightarrow{a_{n-1}} q_n \cdots$ il existe une unique trace $s(\sigma)$ dans $[\![q]\!]$: $s(\sigma) = a_0.a_1 \ldots a_{n-1} \ldots$

4.1.4 Exemple de programme OC

Considérons le programme OC suivant :

	state 0	state 1	
a	`y:=0;`	`if (x=0) then`	c
b	`output(y);`	`  x:=1;`	d
	`goto 1;`	`  y:=1;`	e
		`else`	c'
		`  y:=y*2;`	f
		`endif`	
		`output(y);`	b
		`goto 1;`	

Ce programme comporte 6 actions, `y:=0`, `output(y)`, `if (x=0)`, `x:=1`, `y:=1` et `y:=y*2`, respectivement notées a, b, c, d, e et f. A cela il faut ajouter l'action `go` exécutée implicitement à chaque début d'état. L'action `goto 1` ainsi que la fermeture du test sont des actions invisibles et n'apparaissent donc pas. Par convention, la branche `then` du test est étiquetée c et la branche `else` c'. Le système de transitions étiquetées de ce programme est donc :

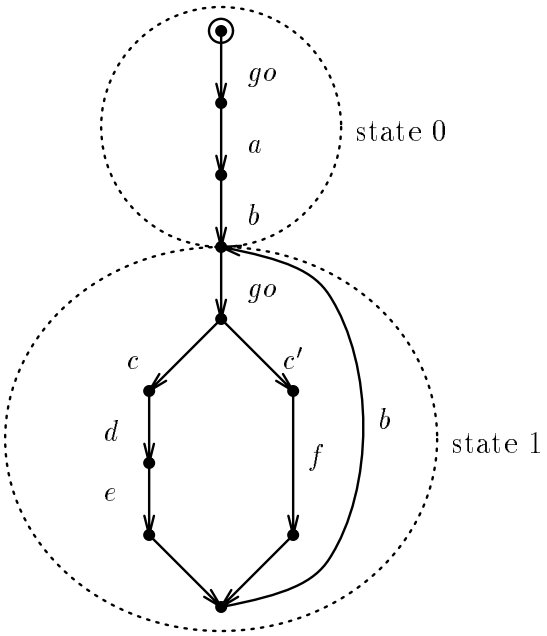


Figure 4.1: Exemple de système de transitions étiquetées

Et sa représentation sous forme de *sted* est :

$$go.a.b.rec\ x.[go.(c.d.e.b.x + c'.f.b.x)]$$

Remarquons que cette représentation algébrique est exponentielle. En effet pour une fermeture de test, le système de transitions étiquetées successeur est dupliqué. Ici c'est le cas de $b.x$.

4.1.5 Relations de dépendance et de commutation entre les actions

Les actions OC sont ordonnées par leurs dépendances de données. Nous pouvons donc établir deux relations sur les actions du système de transitions étiquetées :

- La relation de dépendance $\mathcal{D}$ est symétrique et réflexive.
- Sa relation complémentaire est la relation de commutation $\mathcal{C}$. Elle est symétrique mais pas transitive.

Pour cela, pour toute action a, nous notons respectivement $Var_{in}(a)$ et $Var_{out}(a)$ les ensembles de variables d'entrée et de sortie de l'action a :

Définition 4.6 (dépendance d'actions)
Soient a et b deux actions : $Var_{out}(a) \cap Var_{in}(b) \neq \emptyset \Rightarrow a\mathcal{D}b$.

Reprenons l'exemple de la section 4.1.4. Nous avons :

$$\left.\begin{array}{l} a \text{ est l'action } \texttt{y:=0} \text{ donc } Var_{out}(a) = \{y\} \\ b \text{ est l'action } \texttt{output(y)} \text{ donc } Var_{in}(b) = \{y\} \end{array}\right\} \Rightarrow Var_{out}(a) \cap Var_{in}(b) = \{y\} \Rightarrow a\mathcal{D}b$$

Pour un branchement de la forme $cq + c'q'$, les deux actions c et c' ont forcément les mêmes ensembles Var_{in} et Var_{out}. Par conséquent, elles commutent avec les autres actions de la même manière :

Hypothèse 4.1 (commutation des branchements)
Pour tout branchement c et pour toute action a, $c\mathcal{C}a \iff c'\mathcal{C}a$.

L'action `go` peut être vue comme une affectation aux variables d'entrées de l'automate et elle ne doit donc commuter avec aucune action utilisant ces variables. De plus, nous avons vu dans l'introduction que les langages synchrones sont utilisés dans le cadre de la programmation des systèmes réactifs. D'une part l'action `go` est une requête de réaction venant de l'environnement, et d'autre part les émissions de signaux `output` constituent les réponses du programme à cet environnement. Pour des raisons de réactivité, il est donc impératif que le `go` ne commute avec aucun `output`.

4.1.6 Relation d'équivalence entre les traces d'actions

La relation de commutation $\mathcal{C}$ induit une relation d'équivalence $\sim$ sur A^∞. Intuitivement, deux traces sont équivalentes si et seulement si elles peuvent être rendues identiques en commutant leurs actions. Les notations suivantes permettent de définir formellement l'équivalence de deux traces :

- Pour toute trace s de longueur $|s|$, $\#s$ est l'ensemble $\{1, 2, \ldots, |s|\}$.
- Pour toute trace s et pour tout $n \leq |s|$, $s(n)$ est la $n^{\text{ième}}$ action de la trace s.

Définition 4.7 (équivalence de traces)
Soient s et s' deux traces d'actions. $s \sim s'$ si et seulement si il existe une bijection φ de $\#s$ dans $\#s'$, telle que pour tout n, $s(n) = s'(\varphi(n))$, et pour tous n et n', $n \leq n'$ et $s(n)\mathcal{D}s(n')$ implique $\varphi(n) \leq \varphi(n')$.

4.1.7 Expression de la répartition

Nous avons vu au chapitre 3 comment exprimer la répartition au niveau des programmes sources. Nous supposons ici que l'ensemble des actions A est partitionné en n sous-ensembles A_i, un pour chaque site. Nous écrivons par la suite que l'action a *dépend* du site i si et seulement si $a \in A_i$. Clairement, pour un branchement de la forme $cq + c'q'$, les deux actions c et c' doivent appartenir au même site :

Hypothèse 4.2 (localisation des branchements)
Pour tout c et pour tout i, $c \in A_i \iff c' \in A_i$.

Il faut également décomposer le `go` en n actions `go i`, une pour chaque site. Pour des raisons de réactivité, le `go` ne commute avec aucun `output`. Pour préserver la réactivité, chaque `go i` ne doit donc commuter avec aucun `output` dépendant du site i :

Hypothèse 4.3 (commutation des `go`)
Pour tout site i et pour toute émission de signal a, si $a \in A_i$, alors $a\mathcal{D}go\ i$.

4.2 Enoncé du problème

Répartir un *sted* q sur n sites consiste à trouver un ensemble V d'actions de synchronisation et n systèmes $(q_i)_{i=1,n}$, agissant respectivement sur $A_i \cup V$, et tels que :

$$[\![q]\!]mod \sim \; = \; [\![(q_1[\![q_2]\!] \ldots [\![q_n)/A]\!]mod \sim$$

où $mod \sim$ est le quotient d'un sous-ensemble de A^∞ modulo $\sim$, où $[\![$ et $/$ sont des opérateurs de composition parallèle et de restriction qui seront définis par la suite (respectivement par les définitions 4.25 et 4.22), et où le comportement $[\![q]\!]$ est défini par la définition 4.4.

Le principe est de transformer le *sted* q en un système de transitions fini, étiquetées non plus par des actions mais par des graphes indiquant l'action à effectuer plus ses dépendances (noté *stod*). Un tel graphe sera représenté sous la forme d'un multi-ensemble partiellement ordonné (*pomset* pour "partially ordered multi-set") fini. Avant d'expliquer comment nous effectuons cette transformation, nous donnons des définitions de base sur les *pomset*.

L'opération que nous allons appliquer au *sted* représentant notre programme à répartir consiste à rajouter à chaque action ses dépendances avec les actions passées et futures. Pour cela nous introduisons un ensemble fini A' de dépendances, et nous transformons chaque action de l'ensemble fini des actions A en un *pomset* fini sur $A \cup A'$. Nous étudions dans la section 4.4.1 comment s'effectue cette transformation.

4.3 Systèmes de transitions étiquetées par des ordres

Avant de donner le modèle théorique de notre algorithme de parallélisation, nous présentons des définitions de base sur les ordres partiels et les systèmes de transitions étiquetées par des ordres partiels.

4.3.1 Ordres partiels

Définition 4.8 (ordre partiel)
Un ordre partiel o est un couple (E, R) où :

- E est un ensemble de base,
- R est une relation d'ordre sur E (réflexive, antisymétrique et transitive) : $R \subseteq E \times E$.

Nous utilisons indifféremment les notations xRx' et $(x, x') \in R$. De plus, pour tout ordre o, nous notons respectivement E_o et R_o l'ensemble et la relation d'ordre associés à o.

Nous pouvons définir plusieurs relations d'ordre sur les ordres partiels :

Définition 4.9 (relations d'ordre sur les ordres partiels)
Les ordres $\subseteq$, $\preceq$, P et S sont définis sur les ordres partiels par :

- o est une restriction de o' ($o \subseteq o'$) si et seulement si $E_o \subseteq E_{o'}$ et $R_o = R_{o'} \cap E_o \times E_o$.

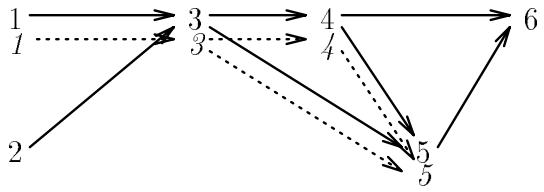


Figure 4.2: Ordre restriction d'un ordre partiel

- o est plus partiel que o' ($o \preceq o'$) si et seulement si $E_o = E_{o'}$ et $R_o \subseteq R_{o'}$.

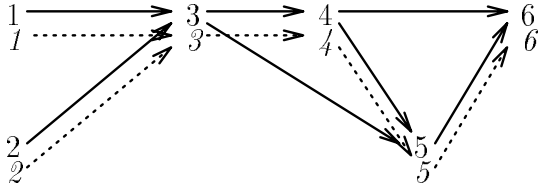


Figure 4.3: Ordre plus partiel qu'un ordre partiel

- o est un préfixe de o' ($o\ P\ o'$) si et seulement si $o \subseteq o'$ et $x \in E_o \ \wedge\ x' R_{o'} x \ \Rightarrow\ x' \in E_o$.

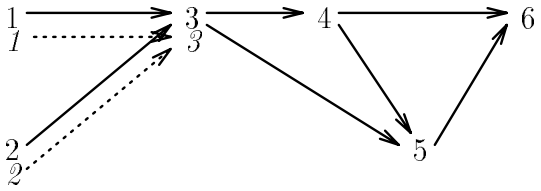


Figure 4.4: Ordre préfixe d'un ordre partiel

- o est un suffixe de o' ($o\ S\ o'$) si et seulement si $o \subseteq o'$ et $x \in E_o \ \wedge\ x R_{o'} x' \ \Rightarrow\ x' \in E_o$.

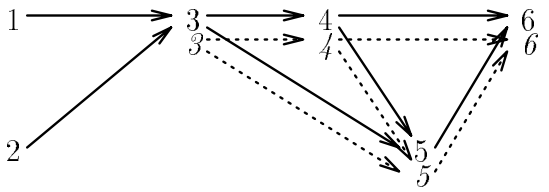


Figure 4.5: Ordre suffixe d'un ordre partiel

Enfin la fermeture transitive de la relation $\mathcal{R}$ est notée classiquement $\mathcal{R}^+$.

4.3.2 Ordres partiels étiquetés

Définition 4.10 (*lpo*)
Un ordre partiel étiqueté (*lpo* pour "labelled partial order") est un triplet $((E, R), A, \lambda)$ tel que :

- (E, R) est un ordre partiel,
- A est un alphabet fini et non vide,
- λ est une fonction d'étiquetage de E dans A, telle que $\lambda(x) = \lambda(y)$ implique ou bien xRy ou bien yRx.

Ce choix de fonction d'étiquetage constitue l'*hypothèse de non auto-concurrence* qui sera utilisée dans la suite :

Hypothèse 4.4 (hypothèse de non auto-concurrence)
Soit o un *lpo* : $\lambda_o(x) = \lambda_o(y) \Rightarrow xR_o y$ ou $yR_o x$.

Par exemple, le *lpo* o tel que $E_o = \{1,2,3,4,5\}$, $R_o = \{(1,2),(2,3),(3,4),(3,5)\}$, $A_o = \{a,b,c\}$ et $\lambda_o = \{(1,a),(2,a),(3,b),(4,c),(5,b)\}$ est :

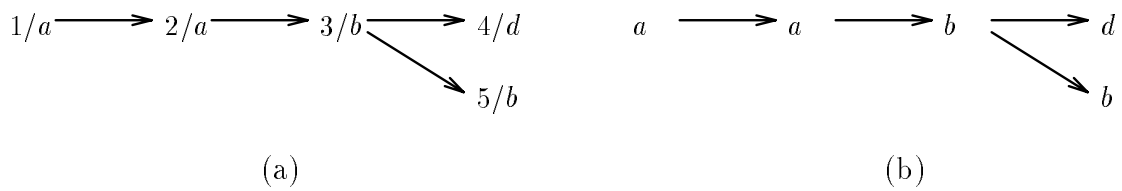


Figure 4.6: Exemple de *lpo*

Dans la suite, nous omettons les numéros dans l'ensemble de base E pour ne garder que les étiquettes (figure 4.6 (b)).

Pour tout *lpo* o, nous notons respectivement E_o, R_o, A_o et λ_o l'ensemble, la relation d'ordre, l'alphabet et la fonction d'étiquetage associés à o. Les ordres définis précédemment sur les ordres partiels (définition 4.9) s'appliquent aussi aux *lpo* :

Définition 4.11 (relations d'ordre sur les *lpo*)
Les ordres $\subseteq$, $\preceq$, P et S sont définis sur les *lpo* par :

- $o \subseteq o'$ si et seulement si $(E_o, R_o) \subseteq (E_{o'}, R_{o'})$ et $A_o \subseteq A_{o'}$ et $\lambda_o = \lambda_{o'} \cap E_o \times E_o$.
- $o \preceq o'$ si et seulement si $(E_o, R_o) \preceq (E_{o'}, R_{o'})$ et $A_o = A_{o'}$ et $\lambda_o = \lambda_{o'}$.
- $o\ P\ o'$ si et seulement si $o \subseteq o'$ et $(E_o, R_o)\ P\ (E_{o'}, R_{o'})$.
- $o\ S\ o'$ si et seulement si $o \subseteq o'$ et $(E_o, R_o)\ S\ (E_{o'}, R_{o'})$.

La définition 4.4 définit le comportement d'un *sted* en termes de traces d'actions générées. Nous définissons de façon analogue le comportement d'un *stod*. Il faut pour cela définir la concaténation de deux *lpo*. Nous choisissons la définition suivante, proposée par B.Caillaud. Elle utilise la notion de *lpo* disjoints, que nous définissons au préalable :

Définition 4.12 (*lpo* disjoints)
Deux *lpo* o et o' sont disjoints si et seulement si $E_o \cap E_{o'} = \emptyset$.

Définition 4.13 (concaténation de deux *lpo*)
La concaténation de deux *lpo* o et o' disjoints est le plus petit (pour l'ordre $\preceq$) *lpo* o'' tel que $E_{o''} = E_o \cup E_{o'}$, $o\ P\ o''$ et $o'\ S\ o''$.

Deux cas peuvent se présenter :

- Si $A_o \cap A_{o'} = \emptyset$, alors la concaténation de o et o' est leur union.
- Si $A_o \cap A_{o'} \neq \emptyset$, alors l'hypothèse 4.4 de non auto-concurrence oblige tout élément maximal de o d'une étiquette donnée à être plus petit que l'élément minimal de o' de même étiquette (figure 4.7).

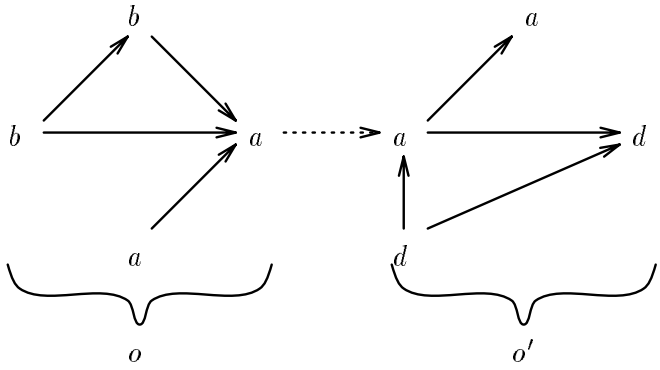


Figure 4.7: Concaténation de deux *lpo*

4.3.3 Systèmes de transitions étiquetés par des *lpo*

Pour les définir, il est nécessaire de définir au préalable les multi-ensembles partiellement ordonnés (*pomset* pour "partially ordered multi-sets") :

Définition 4.14 (équivalence de deux *lpo*)
Soient o et o' deux *lpo* : $o \equiv o'$ si et seulement si il existe un isomorphisme ψ de o vers o' préservant la relation d'ordre R ainsi que la fonction d'étiquetage λ.

Par conséquent, ψ est une bijection de E_o dans $E_{o'}$ telle que :

- $\psi(x) R_{o'} \psi(x')$ si et seulement si $x R_o x'$,
- $\lambda_{o'}(\psi(x)) = \lambda_o(x)$.

Ceci permet de définir les *pomset* :

Définition 4.15 (*pomset*)
Un *pomset* est une classe d'équivalence modulo $\equiv$ de *lpo*.

Les définitions 4.11, 4.12 et 4.13 s'étendent naturellement aux *pomset*. Dorénavant, nous considérons l'ensemble $P(A)$ des *pomset* ainsi que l'ensemble $FP(A)$ des *pomset* finis, sur un alphabet A fini. Nous pouvons donc définir des systèmes de transitions étiquetées par des *pomset* finis :

Définition 4.16 (*stod*)
Un système de transitions déterministe fini sur des ordres (*stod*) est un quadruplet :

$$\{Q,\ q \in Q,\ FP(A),\ \rightarrow\}$$

où Q est un ensemble d'états, q l'état initial, $FP(A)$ l'alphabet des actions et $\rightarrow$ la fonction de transition définie de $Q \times FP(A)$ dans Q.

Un tel système $\{Q,\ q \in Q,\ FP(A),\ \rightarrow\}$ sera identifié avec son état initial q. Nous définissons de plus son comportement $[\![q]\!]$ par :

Définition 4.17 (comportement d'un *stod*)
Soit $\{Q,\ q,\ FP(A),\ \rightarrow\}$ un *stod*. Son comportement $[\![q]\!]$ est l'ensemble des *pomset* maximaux (pour l'ordre préfixe), finis ou infinis, qu'il peut générer : $[\![q]\!] \subseteq P(A)$.

De même que les traces du comportement d'un *sted* sont obtenues par concaténation de ses étiquettes (qui sont des actions), les *pomset* du comportement d'un *stod* sont obtenus par concaténation de ses étiquettes (qui sont elles-mêmes des *pomset*).

Enfin à toute séquence d'exécution d'un *stod*, nous pouvons associer un unique *pomset* de son comportement :

Définition 4.18 (*pomset* associé à une séquence d'exécution)
Pour toute séquence d'exécution σ du *stod* q de la forme : $\sigma = q \stackrel{a_0}{\rightarrow} q_1 \cdots q_{n-1} \stackrel{a_{n-1}}{\rightarrow} q_n \cdots$ il existe un unique *pomset* $o(\sigma)$ dans $[\![q]\!]$: $o(\sigma) = a_0.a_1 \ldots a_{n-1} \ldots$

4.4 Modèle théorique de notre algorithme de parallélisation

Du point de vue fonctionnel, dans un programme réparti sur plusieurs sites, l'émission d'un message constitue une *autorisation à s'exécuter* pour les actions postérieures à la réception de ce message. De façon similaire, la réception d'un message constitue une *demande d'autorisation* de la part des actions antérieures à l'émission de ce message.

Notre idée est de modéliser de telles autorisations par des dépendances entre actions. Pour ordonner les actions et les dépendances, nous transformons le *sted* du programme en un *stod*. Chaque étiquette est un *pomset* liant une action a avec *en amont* des demandes d'autorisation de la part des actions qui ne commutent pas avec a, et *en aval* des autorisations pour les actions qui ne commutent pas avec a.

4.4.1 Transformation d'un *sted* en un *stod*

La fonction de transformation opère sur le *sted* représentant le programme centralisé. Elle le transforme en un *stod* en remplaçant toute action a par un *pomset* fini, donnant les dépendances de a avec les autres actions. Les *pomset* sont par conséquent étiquetés sur l'alphabet des actions et sur l'alphabet des dépendances. Nous choisissons donc comme ensemble d'actions de synchronisations l'ensemble des dépendances sur les actions, c'est-à-dire $V = A \times A$. La fonction f transforme le *sted* q en le *stod* $f(q)$ et est définie par :

Définition 4.19 (fonction f de transformation)

$$\frac{q \stackrel{a}{\rightarrow} q'}{f(q) \stackrel{f(a)}{\rightarrow} f(q')}$$

où $f(a)$ est le *pomset* sur $A \cup V$ tel que $R = \{((a', a), a) : a' \mathcal{D} a\} \cup \{(a, (a, a')) : a \mathcal{D} a'\}$, ce que nous pouvons noter :

$$f(a) = \left(\begin{array}{ccccc} (a', a) & & & & (a, a') \\ \text{où } a' \mathcal{D} a & \rightarrow & a & \rightarrow & \text{où } a \mathcal{D} a' \end{array} \right)$$

Par exemple, si $b\mathcal{D}a$ et $d\mathcal{D}a$, alors, $\mathcal{D}$ étant réflexive et symétrique, nous avons :

$$f(a) = \boxed{\begin{array}{ccccc} (b,a) & & (a,a) & & (d,a) \\ & \searrow & \downarrow & \swarrow & \\ & & a & & \\ & \swarrow & \downarrow & \searrow & \\ (a,b) & & (a,a) & & (a,d) \end{array}}$$

Maintenant, pour tout *pomset* $o \in P(A)$, nous notons $l(o)$ l'ensemble de ses extensions linéaires définies par :

Définition 4.20 (extensions linéaires d'un *pomset*)
Pour tout *lpo* o, $l(o)$ est l'ensemble des traces s telles qu'il existe une bijection φ de E_o vers $\#s$ telle que pour tout $x \in E_o, \lambda_o(x) = s(\varphi(x))$ et pour tous $x, x', x R_o x' \Rightarrow \varphi(x) \leq \varphi(x')$.

Voici un exemple d'ordre avec ses extensions linéaires :

$$o = \left\{ \begin{array}{ccccccc} a & \rightarrow & a & \rightarrow & b & \rightarrow & d \\ & & & & & \searrow & \\ & & & & & & b \end{array} \right\} \qquad l(o) = \{a.a.b.d, a.a.b.b\}$$

Intuitivement, pour un *sted* q, $l([\![f(q)]\!])$ est l'ensemble des traces d'actions et de dépendances d'actions de q, ordonnées selon la relation $\mathcal{D}$.

Les dépendances servent à ordonner correctement les actions : c'est donc le comportement des *stod* restreint au seul alphabet des actions qui nous intéresse. Nous établissons dans la proposition suivante que le comportement de $f(q)$ restreint à A est identique, aux commutations près, au comportement de q :

Proposition 4.1
$C([\![q]\!]) = l([\![f(q)]\!]/A)$

où $C([\![q]\!])$ est l'ensemble des classes d'équivalences des traces de $[\![q]\!]$, définies par :

Définition 4.21 (classe d'équivalence de traces)
Pour toute trace s, nous notons $C(s)$ la classe d'équivalence de s modulo la relation $\sim$ induite par la relation de commutation $\mathcal{C}$.

Avant d'en donner la preuve, nous définissons la restriction d'un *pomset* à un alphabet d'actions, puis nous donnons un lemme préliminaire nécessaire à la preuve.

4.4.2 Opérateur de restriction sur les *pomset*

Pour le passage des *stod* aux *sted* nous avons besoin d'une opération de restriction sur les *pomset* :

Définition 4.22 (restriction)
Pour tout *pomset* $o \in P(A)$ et pour tout sous-ensemble $B \subseteq A$, la restriction de o à l'ensemble B est le *pomset* o/B obtenu en ne gardant que les éléments étiquetés sur B. Donc :

- $E_{o/B} = \lambda_o^{-1}(B)$,
- $R_{o/B} = \{(x,y) | \lambda_o(x) \in B \wedge \lambda_o(y) \in B \wedge x R_o y\}$,
- $A_{o/B} = B$,
- $\lambda_{o/B} = \{(x, \lambda_o(x)) | \lambda_o(x) \in B\}$.

Cette définition s'étend naturellement aux sous-ensembles de $P(A)$.

4.4.3 Résultat intermédiaire sur l'égalité des comportements

Toute trace peut être considérée comme un ordre linéaire, et pour toute trace s nous notons $o(s)$ son ordre associé, défini par :

Définition 4.23 (ordre linéaire associé à une trace)
Pour toute trace s, il existe une bijection φ de $\#s$ vers $E_{o(s)}$ telle que pour tout $n, s(n) = \lambda_{o(s)}(\varphi(n))$ et pour tous $n, n', \varphi(n) R_{o(s)} \varphi(n') \Longleftrightarrow n \leq n'$.

Voici un exemple de trace et son ordre linéaire associé :

$$s = a.b.c.c.d \qquad o(s) = \{a \rightarrow b \rightarrow c \rightarrow c \rightarrow d\}$$

Dans la section 4.2, nous avons exprimé le problème de la répartition en termes d'égalité des comportements du système centralisé et du système réparti. Or pour un *sted* q, $[\![q]\!] mod \sim$ est égal par définition à $\bigcup_{s \in [\![q]\!]} C(s)$. D'autre part, le comportement d'un *stod* est un *pomset*. Le lemme suivant, dû à R.Cori et Y.Métivier dans [20], permet de relier les deux :

Lemme 4.1
Pour toute trace s, si $((E_{o(s)}, R_{o(s)}), A_{o(s)}, \lambda_{o(s)})$ est son *lpo* associé, alors nous avons :

$$C(s) = l(R(s, \mathcal{D})^+)$$

la relation $R(s, \mathcal{D})$ étant définie par :

$$\forall x, x', x \; R(s, \mathcal{D}) \; x' \Longleftrightarrow x \; R_{o(s)} \; x' \text{ et } \lambda_{o(s)}(x) \; \mathcal{D} \; \lambda_{o(s)}(x')$$

Intuitivement, $x \, R(s, \mathcal{D}) \, x'$ si l'action x est antérieure à l'action x' dans la trace s, et si l'action x' ne commute pas avec l'action x. Le lemme 4.1 signifie par conséquent que la classe d'équivalence $C(s)$ de s modulo $\sim$ est l'ensemble des extensions linéaires de la fermeture transitive de l'ordre $R(s, \mathcal{D})$. Autrement dit, il signifie que $C(s)$ est l'ensemble des traces d'actions ordonnées par leur ordre d'apparition dans s et par la relation de dépendance $\mathcal{D}$.

Preuve du sens $\supseteq$:

Soit $s' \in l(R(s, \mathcal{D})^+)$. Il faut montrer que $s' \sim s$, c'est-à-dire, d'après la définition 4.7, qu'il existe une bijection f de $\#s$ dans $\#s'$ telle que :

$$\forall n, \qquad s(n) = s'(f(n)) \tag{4.1}$$

$$\forall n, n', \qquad n \leq n' \wedge s(n) \mathcal{D} s(n') \Rightarrow f(n) \leq f(n') \tag{4.2}$$

Or $(E_{o(s)}, R(s,\mathcal{D})) \preceq (E_{o(s)}, R_{o(s)})$. Donc $s' \in l(R^+_{o(s)})$. Donc, d'après la définition 4.20, il existe une bijection g de $E_{o(s)}$ vers $\#s'$ telle que :

$$\forall x, \quad \lambda_{o(s)}(x) = s(g(x)) \tag{4.3}$$
$$\forall x, x', \quad x R_{o(s)} x' \Rightarrow g(x) \leq g(x') \tag{4.4}$$

De plus, d'après la définition 4.23 de $o(s)$, il existe une bijection h de $\#s$ vers $E_{o(s)}$ telle que :

$$\forall n, \quad s(n) = \lambda_{o(s)}(h(n)) \tag{4.5}$$
$$\forall n, n', \quad h(n) R_{o(s)} h(n') \iff n \leq n' \tag{4.6}$$

Des équations 4.3 et 4.5 nous déduisons que $\forall n,\ s(n) = s'(g(h(n))) = s'((g \circ h)(n))$, c'est-à-dire l'équation 4.1 en posant $f = g \circ h$.

Maintenant, supposons $n \leq n'$ et $s(n) \mathcal{D} s(n')$. Grâce aux équations 4.5 et 4.6, nous déduisons $h(n) R_{o(s)} h(n')$ et $\lambda_{o(s)}(h(n)) \mathcal{D} \lambda_{o(s)}(h(n'))$. Ceci implique, par définition de $R(s,\mathcal{D})$, que $h(n) R(s,\mathcal{D}) h(n')$. Donc $h(n) R(s,\mathcal{D})^+ h(n')$. D'après l'équation 4.4, nous déduisons finalement que $(g \circ h)(n) \leq (g \circ h)(n')$, c'est-à-dire l'équation 4.2 toujours en posant $f = g \circ h$.

Preuve du sens $\subseteq$:

Soit $s' \in C(s)$. Il faut montrer que $s' \in l(R(s,\mathcal{D})^+)$, c'est-à-dire, d'après la définition 4.20, qu'il existe une bijection f de $R(s,\mathcal{D})^+$ vers s' telle que $\forall x, x', x R(s,\mathcal{D})^+ x' \Rightarrow f(x) \leq f(x')$.

Par hypothèse, $s' \in C(s)$: $s \sim s'$ donc d'après la définition 4.7, il existe une bijection g de $\#s$ dans $\#s'$ telle que :

$$\forall n, \quad s(n) = s'(g(n)) \tag{4.7}$$
$$\forall n, n', \quad n \leq n' \wedge s(n) \mathcal{D} s(n') \Rightarrow g(n) \leq g(n') \tag{4.8}$$

De plus, d'après la définition 4.23 de $o(s)$, il existe une bijection h de $\#s$ vers $E_{o(s)}$ telle que :

$$\forall n, \quad s(n) = \lambda_{o(s)}(h(n))$$
$$\forall n, n', \quad h(n) R_{o(s)} h(n') \iff n \leq n'$$

En exprimant ceci à l'aide de h^{-1}, nous obtenons :

$$\forall x, \quad s(h^{-1}(x)) = \lambda_{o(s)}(x) \tag{4.9}$$
$$\forall x, x', \quad x R_{o(s)} x' \iff h^{-1}(x) \leq h^{-1}(x') \tag{4.10}$$

De plus, des équations 4.7 et 4.9, nous déduisons que :

$$\forall x, \lambda_{o(s)}(x) = s(h^{-1}(x)) = s'((g \circ h^{-1})(x))$$

Maintenant, soient x et x' tels que $x R(s,\mathcal{D})^+ x'$. Alors il existe une suite finie $x_0, x_1, \ldots, x_n$ telle que :

$$x = x_0 R(s,\mathcal{D}) x_1 R(s,\mathcal{D}) \ldots x_{n-1} R(s,\mathcal{D}) x_n = x'$$

Donc pour tout i, nous avons :

$$x_i R_{o(s)} x_{i+1} \ \wedge \ \lambda_{o(s)}(x_i) \mathcal{D} \lambda_{o(s)}(x_{i+1})$$

Grâce aux équations 4.9 et 4.10, nous déduisons que pour tout i :

$$h^{-1}(x_i) \leq h^{-1}(x_{i+1}) \ \wedge \ s(h^{-1}(x_i)) \mathcal{D} s(h^{-1}(x_{i+1}))$$

Donc, d'après l'équation 4.8, pour tout i, nous avons :

$$(g \circ h^{-1})(x_i) \leq (g \circ h^{-1})(x_{i+1})$$

Finalement, par transitivité de $\leq$, nous concluons :

$$(g \circ h^{-1})(x) \leq (g \circ h^{-1})(x')$$

Donc en posant $f = g \circ h^{-1}$, nous avons bien une bijection f de $R(s, \mathcal{D})^+$ vers s' vérifiant la définition 4.20, et par conséquent, $s' \in l(R(s, \mathcal{D})^+)$.

4.4.4 Preuve de la proposition 4.1

Soit σ une séquence d'exécution de q de la forme :

$$\sigma = q \stackrel{a_0}{\rightarrow} q_1 \cdots q_{n-1} \stackrel{a_{n-1}}{\rightarrow} q_n \cdots$$

Il lui correspond une unique séquence d'exécution $f(\sigma)$ de $f(q)$ de la forme :

$$f(\sigma) = f(q) \stackrel{f(a_0)}{\rightarrow} f(q_1) \cdots f(q_{n-1}) \stackrel{f(a_{n-1})}{\rightarrow} f(q_n) \cdots$$

Soit $s(\sigma)$ la trace représentant le comportement de σ, définie par la définition 4.5 :

$$s(\sigma) = a_0.a_1 \ldots a_{n-1} \ldots$$

Soit de même $o(f(\sigma))$ le *pomset* représentant le comportement de $f(\sigma)$, défini par la définition 4.18 :

$$o(f(\sigma)) = f(a_0).f(a_1) \ldots f(a_{n-1}) \ldots$$

D'après le lemme 4.1, nous déduisons :

$$C(s(\sigma)) = l(R(o(s(\sigma)), \mathcal{D})^+) \tag{4.11}$$

Montrons à présent que :

$$o(f(\sigma))/A = R(o(s(\sigma)), \mathcal{D})^+ \tag{4.12}$$

A chaque état q_i de $f(\sigma)$, il y a un seul élément étiqueté sur A : appelons-le x_i. Or $x_i R_{o(f(\sigma))} x_j$ si et seulement si il existe une séquence croissante $i = l_0, l_1, \ldots l_{n-1}, l_n = j$ telle que :

$$(\lambda(x_{l_0})) \mathcal{D}(\lambda(x_{l_1})) \ldots \mathcal{D}(\lambda(x_{l_n}))$$

C'est précisément la définition de $R(o(s(\sigma)), \mathcal{D})^+$. Donc, d'après les équations 4.11 et 4.12, nous déduisons que :

$$C(s(\sigma)) = l(o(f(\sigma))/A)$$

Donc finalement, nous concluons :

$$C([\![q]\!]) = l([\![f(q)]\!]/A)$$

4.4.5 Exemple

Dans toute la suite, nous considérons le *sted* $q = a.(c.b.nil + c'.d.nil)$. L'alphabet des actions est $A = \{a, b, c, c', d\}$ et le comportement est $[\![q]\!] = \{acb, ac'd\}$. Supposons que nous ayons les dépendances $a\mathcal{D}c$, $a\mathcal{D}c'$, $c\mathcal{D}d$ et $c'\mathcal{D}d$, plus les dépendances déduites par symétrie et réflexivité. Alors $f(q) = f(a).(f(c).f(b).nil + f(c').f(d).nil)$ avec :

$$f(a) = \begin{array}{|ccccc|}\hline (c,a) & & (a,a) & & (c',a) \\ & \searrow & \downarrow & \swarrow & \\ & & a & & \\ & \swarrow & \downarrow & \searrow & \\ (a,c) & & (a,a) & & (a,c') \\ \hline\end{array} \quad f(b) = \begin{array}{|c|}\hline (b,b) \\ \downarrow \\ b \\ \downarrow \\ (b,b) \\ \hline\end{array} \quad f(d) = \begin{array}{|ccccc|}\hline (c',d) & & (d,d) & & (c,d) \\ & \searrow & \downarrow & \swarrow & \\ & & d & & \\ & \swarrow & \downarrow & \searrow & \\ (d,c') & & (d,d) & & (d,c) \\ \hline\end{array}$$

$$f(c) = \begin{array}{|ccccc|}\hline (a,c) & & (c,c) & & (d,c) \\ & \searrow & \downarrow & \swarrow & \\ & & c & & \\ & \swarrow & \downarrow & \searrow & \\ (c,a) & & (c,c) & & (c,d) \\ \hline\end{array} \quad f(c') = \begin{array}{|ccccc|}\hline (d,c') & & (c',c') & & (a,c') \\ & \searrow & \downarrow & \swarrow & \\ & & c' & & \\ & \swarrow & \downarrow & \searrow & \\ (c',d) & & (c',c') & & (c',a) \\ \hline\end{array}$$

Avec la définition 4.13 de la concaténation des *lpo*, nous trouvons son comportement $[\![f(q)]\!] = \{f(a).f(c).f(b), f(a).f(c').f(d)\}$:

$$[\![f(q)]\!] = \left\{ \begin{array}{|ccccc|}\hline (c,a) & & (a,a) & & (c',a) \\ \downarrow & \searrow & \downarrow & \swarrow & \\ \downarrow & & a & & \\ \downarrow & \swarrow & \downarrow & \searrow & \\ \downarrow (a,c) & & (a,a) & & (a,c') \\ \downarrow \downarrow & & & & \\ \downarrow (a,c) & & (c,c) & & (d,c) \\ \downarrow & \searrow & \downarrow & \swarrow & \\ \downarrow & & c & & \\ \searrow & \swarrow & \downarrow & \searrow & \\ (c,a) & & (c,c) & & (c,d) \\ & & & & \\ & & (b,b) & & \\ & & \downarrow & & \\ & & b & & \\ & & \downarrow & & \\ & & (b,b) & & \\ \hline\end{array} , \begin{array}{|ccccc|}\hline (c,a) & & (a,a) & & (c',a) & \\ & \searrow & \downarrow & \swarrow & & \searrow \\ & & a & & & \downarrow \\ & \swarrow & \downarrow & \searrow & & \downarrow \\ (a,c) & & (a,a) & & (a,c') & \downarrow \\ & & & & \downarrow & \downarrow \\ (d,c') & & (c',c') & & (a,c') & \downarrow \\ & \searrow & \downarrow & \swarrow & & \downarrow \\ & & c' & & & \downarrow \\ & \swarrow & \downarrow & \searrow & & \swarrow \\ (c',d) & & (c',c') & & (c',a) & \\ \downarrow & & & & & \\ (c',d) & & (d,d) & & (c,d) & \\ & \searrow & \downarrow & \swarrow & & \\ & & d & & & \\ & \swarrow & \downarrow & \searrow & & \\ (d,c') & & (d,d) & & (d,c) & \\ \hline\end{array} \right\}$$

Donc en restreignant à A, nous trouvons :

$$[\![f(q)]\!]/A = \left\{ \begin{array}{|c|}\hline a \\ \downarrow \\ c \\ \\ b \\ \hline\end{array} , \begin{array}{|c|}\hline a \\ \downarrow \\ c' \\ \downarrow \\ d \\ \hline\end{array} \right\}$$

Par conséquent, $l([\![f(q)]\!]/A) = \{a.c.b, a.b.c, b.a.c, a.c'.d\}$: nous retrouvons bien le fait que les actions a et c ne commutent pas, de même pour les actions a, c' et d, et que l'action b est concurrente des autres. Pour une exécution séquentielle, elle peut aussi bien être exécutée avant a, après c, ou entre a et c.

4.4.6 Projection et répartition

La répartition un *stod* sur plusieurs sites se fait par simple projection. Nous définissons conjointement la projection et la composition parallèle.

Nous définissons l'opérateur ϕ_i qui projette le *stod* q sur le site i de la façon suivante :

Définition 4.24 (opérateur de projection d'un *stod*)

$$\frac{q \xrightarrow{o} q'}{\phi_i(q) \xrightarrow{\phi_i(o)} \phi_i(q')}$$

La projection de o est fonction de l'étiquette d'origine du *sted*. Intuitivement, si c'est une action n'appartenant pas au vocabulaire du site i, alors il ne faut garder que les dépendances liant une action du vocabulaire du site i. Si par contre l'action appartient au vocabulaire du site i, alors il faut la conserver telle quelle avec ses dépendances. Quant aux branchements, puisque nous voulons que la structure de contrôle du programme centralisé soit conservée sur tous les sites (voir la section 3.3), il faut les projeter quel que soit leur site d'appartenance.

Nous définissons alors le comportement d'un programme réparti de la façon suivante :

Définition 4.25 (comportement réparti)
Soient $(q_i)_{i=1,n}$ n *stod* définis respectivement sur les alphabets $(V_i)_{i=1,n}$. Le comportement de $[\![(q_1[\![\cdots [\![q_n)]\!]$ est l'ensemble des *pomset* o, minimaux pour l'ordre $\preceq$ et tels que :

$$\exists\, \Sigma, \text{ tuple de séquences d'exécution } \sigma_i \text{ de } q_i, \text{ tel que } \forall i, o(\sigma_i) \preceq o/V_i \tag{4.13}$$

$$\nexists\, \Sigma', \text{ plus grand point à point pour l'ordre préfixe que } \Sigma \text{ et vérifiant 4.13} \tag{4.14}$$

Cette définition est cohérente avec l'interprétation des dépendances comme des autorisations et des demandes d'autorisation.

Considérons par exemple les deux *stod* $q_1 = A.nil$ et $q_2 = B.nil$, où A et B sont les *pomset* suivants :

$$A = \left\{ \begin{array}{ccccccccc} a & \to & a & \to & a & & & & \\ & \searrow & & \searrow & & \searrow & & & \\ & & u & \to & u & \to & u & & \end{array} \right\} \qquad B = \left\{ \begin{array}{ccccccccc} u & \to & u & \to & u & & & & \\ & \searrow & & \searrow & & \searrow & & & \\ & & b & \to & b & \to & b & & \end{array} \right\}$$

Leurs vocabulaires respectifs sont : $V_1 = \{a, u\}$ et $V_2 = \{b, u\}$. Nous avons alors :

$$[\![q_1 \,\|\, q_2]\!] = \left\{ \begin{array}{ccccccccc} a & \rightarrow & a & \rightarrow & a & & & & \\ & \searrow & & \searrow & & \searrow & & & \\ & & u & \rightarrow & u & \rightarrow & u & & \\ & & & \searrow & & \searrow & & \searrow & \\ & & & & b & \rightarrow & b & \rightarrow & b \end{array} \right\}$$

L'étiquette u représente dans le *pomset* A une autorisation d'exécution destinée à l'action b, et dans le *pomset* B une demande d'autorisation de la part de l'action a. L'opérateur $\|$ effectue donc une synchronisation entre les deux *stod* par mise en commun des points étiquetés par u. Par contre les actions a et b ne sont pas synchronisées directement : la seule contrainte est que b soit exécutée *après* a. Autrement dit, la synchronisation entre les actions induite par l'opérateur $\|$ correspond à la communication par files d'attente mise en œuvre dans la répartition fonctionnelle.

4.4.7 Opérateur de synchronisation des traces

Définition 4.26 (ensembles associés à une relation)
Nous définissons pour toute relation $\mathcal{X}$ les ensembles suivants :

- $a\mathcal{X} = \{(a, x) : a\mathcal{X}x\}$,
- $\mathcal{X}a = \{(x, a) : x\mathcal{X}a\}$,
- $A\mathcal{X} = \bigcup_{a \in A} a\mathcal{X}$,
- et $\mathcal{X}A = \bigcup_{a \in A} \mathcal{X}a$.

A partir de ces définitions, nous définissons, pour toute action a qui n'est pas un branchement, l'opérateur ϕ_i de projection d'un *pomset* sur le site i par :

Définition 4.27 (projection du *pomset* d'une action)

$$\phi_i(f(a)) = \left\{ \begin{array}{l} \textbf{si } a \in A_i \textbf{ alors } f(a) \\ \textbf{sinon } \tilde{f}(a)/(A_i\mathcal{D} \cup \mathcal{D}A_i) \end{array} \right. \quad \text{avec } \tilde{f}(a) = \left(\begin{array}{ccc} (a', a) & & (a, a') \\ \text{où } a'\mathcal{D}a & a & \text{où } a\mathcal{D}a' \end{array} \right)$$

4.4.8 Opérateur de synchronisation totale des branchements

Pour tout branchement c, l'opérateur ϕ_i de projection d'un *pomset* sur le site i est défini par :

Définition 4.28 (projection du *pomset* d'un branchement)
$\phi_i(f(c)) = f(c)/(A_i\mathcal{D} \cup \mathcal{D}A_i \cup \{c\})$

Donc que le branchement c appartienne au vocabulaire du site i ou non, il est projeté sur le site i. Toutefois, seules les dépendances du *pomset* $f(c)$ liant c avec des actions du site i sont conservées. C'est pour cela que nous parlons de synchronisation totale des branchements.

4.4.9 Exemple

Nous reprenons l'exemple de la section 4.4.5, en supposant deux sites 0 et 1, et la partition suivante des actions : $A_0 = \{a, c, c'\}$ et $A_1 = \{b, d\}$. Nous calculons donc avec la définition 4.26 :

- $A_0\mathcal{D} = \{(a,a),(a,c),(a,c'),(c,c),(c,a),(c,d),(c',c'),(c',a),(c',d)\}$
- $\mathcal{D}A_0 = \{(a,a),(c,a),(c',a),(c,c),(a,c),(d,c),(c',c'),(a,c'),(d,c')\}$
- $A_1\mathcal{D} = \{(b,b),(d,d),(d,c),(d,c')\}$
- $\mathcal{D}A_1 = \{(b,b),(d,d),(c,d),(c',d)\}$

Par projection, nous trouvons donc :

$$\phi_0(f(a)) = \begin{array}{ccc} (c,a) & (a,a) & (c',a) \\ & \searrow \downarrow \swarrow & \\ & a & \\ & \swarrow \downarrow \searrow & \\ (a,c) & (a,a) & (a,c') \end{array} \quad \phi_0(f(b)) = \square \quad \phi_0(f(d)) = \begin{array}{cc} (c',d) & (c,d) \\ (d,c') & (d,c) \end{array}$$

$$\phi_0(f(c)) = \begin{array}{ccc} (a,c) & (c,c) & (d,c) \\ & \searrow \downarrow \swarrow & \\ & c & \\ & \swarrow \downarrow \searrow & \\ (c,a) & (c,c) & (c,d) \end{array} \quad \phi_0(f(c')) = \begin{array}{ccc} (d,c') & (c',c') & (a,c') \\ & \searrow \downarrow \swarrow & \\ & c' & \\ & \swarrow \downarrow \searrow & \\ (c',d) & (c',c') & (c',a) \end{array}$$

Par conséquent, le comportement du *stod* $f(q)$ projeté sur le site 0 est :

$$[\![\phi_0(f(q))]\!] = \left\{ \begin{array}{ccc} (c,a) & (a,a) & (c',a) \\ & \searrow \downarrow \swarrow & \\ & a & \\ & \swarrow \downarrow \searrow & \\ (a,c) & (a,a) & (a,c') \\ \downarrow & & \\ (a,c) & (c,c) & (d,c) \\ & \searrow \downarrow \swarrow & \\ & c & \\ & \swarrow \downarrow \searrow & \\ (c,a) & (c,c) & (c,d) \end{array} , \begin{array}{ccc} (c,a) & (a,a) & (c',a) \\ & \searrow \downarrow \swarrow & \\ & a & \\ & \swarrow \downarrow \searrow & \\ (a,c) & (a,a) & (a,c') \\ & & \downarrow \\ (d,c') & (c',c') & (a,c') \\ & \searrow \downarrow \swarrow & \\ & c' & \\ & \swarrow \downarrow \searrow & \\ (c',d) & (c',c') & (c',a) \\ \downarrow & & \\ (c',d) & & (c,d) \\ & & \\ (d,c') & & (d,c) \end{array} \right\}$$

Pour le site 1, nous trouvons :

$$\phi_1(f(a)) = \boxed{\ } \quad \phi_1(f(b)) = \boxed{\begin{array}{c}(b,b)\\ \downarrow \\ b \\ \downarrow \\ (b,b)\end{array}} \quad \phi_1(f(d)) = \boxed{\begin{array}{ccc}(c',d) & (d,d) & (c,d)\\ & \searrow \ \downarrow \ \swarrow & \\ & d & \\ & \swarrow \ \downarrow \ \searrow & \\ (d,c') & (d,d) & (d,c)\end{array}}$$

$$\phi_1(f(c)) = \boxed{\begin{array}{c}(d,c)\\ \downarrow \\ c \\ \downarrow \\ (c,d)\end{array}} \quad \phi_1(f(c')) = \boxed{\begin{array}{c}(d,c')\\ \downarrow \\ c' \\ \downarrow \\ (c,d')\end{array}}$$

$$[\![\phi_1(f(q))]\!] = \left\{ \boxed{\begin{array}{c}(d,c)\\ \downarrow \\ c \\ \downarrow \\ (c,d)\\ \\ (b,b)\\ \downarrow \\ b \\ \downarrow \\ (b,b)\end{array}}, \boxed{\begin{array}{ccc}(d,c') & & \\ \downarrow & & \\ c' & & \\ \downarrow & & \\ (c',d) & & \\ \downarrow & & \\ (c',d) & (d,d) & (c,d)\\ & \searrow \ \downarrow \ \swarrow & \\ & d & \\ & \swarrow \ \downarrow \ \searrow & \\ (d,c') & (d,d) & (d,c)\end{array}} \right\}$$

4.4.10 Placement des communications

Une dépendance entre deux actions implique forcément une dépendance de données entre des deux actions (définition 4.6). Au niveau des comportements projetés sur les sites $(i)_{1\leq i\leq n}$, il faut donc transformer les dépendances en communications : un (a,b) devient une émissions sur le site de a et une réception sur le site de b. Nous définissons donc deux étiquettes spéciales que nous ajoutons aux alphabets d'actions des sites correspondants :

Définition 4.29 (étiquettes des communications)
$put(a,b)$ correspond à l'action `put(site(b),var(a))` et consiste à envoyer les variables de sortie de l'action a au site dont dépend l'action b.
$get(a,b)$ correspond à l'action `var(b):=get(site(a))` et consiste à recevoir les variables d'entrée de l'action b du site dont dépend l'action a.

Reprenons l'exemple de la section 4.4.5. Nous constatons que les comportements projetés $[\![\phi_0(f(q))]\!]$ et $[\![\phi_1(f(q))]\!]$ (calculés à la section 4.4.9) comportent beaucoup trop de communications. En effet pour une action donnée, toutes les dépendances dans lesquelles elle intervient ne sont pas utiles. C'est en particulier le cas des dépendances qui lient deux actions qui ne seront pas exécutées successivement ou deux actions dépendant du même site. C'est également le cas

des dépendances identiques qui se suivent : une seule des deux doit être conservée. Une fois supprimées ces dépendances inutiles, le comportement final est :

$$[\![\phi_0(f(q))]\!] = \left\{ \boxed{\begin{array}{c} a \\ \downarrow \\ c \end{array}}, \boxed{\begin{array}{c} a \\ \downarrow \\ c' \\ \downarrow \\ (c',d) \end{array}} \right\} \quad [\![\phi_1(f(q))]\!] = \left\{ \boxed{\begin{array}{c} c \\ \\ b \end{array}}, \boxed{\begin{array}{c} c' \\ \downarrow \\ (c',d) \\ \downarrow \\ d \end{array}} \right\}$$

En remplaçant les dépendances par des communications, nous obtenons finalement :

$$[\![\phi_0(f(q))/A]\!] = \left\{ \boxed{\begin{array}{c} a \\ \downarrow \\ c \end{array}}, \boxed{\begin{array}{c} a \\ \downarrow \\ c' \\ \downarrow \\ put(c',d) \end{array}} \right\} \quad [\![\phi_1(f(q))/A]\!] = \left\{ \boxed{\begin{array}{c} c \\ \\ b \end{array}}, \boxed{\begin{array}{c} c' \\ \downarrow \\ get(c',d) \\ \downarrow \\ d \end{array}} \right\}$$

Dans la pratique, le placement des émissions (`put`) et des réceptions (`get`), basé sur les dépendances entre actions, ne place que les communications qui sont nécessaires. C'est ce que nous avons vu au chapitre précédant dans les sections 3.4 et 3.5. Mais par souci de simplicité, nous ne définissons par formellement cette phase de placement *minimal* des communications, ni les stratégies de placement *au plus tôt* et *si besoin*.

4.4.11 Récapitulation

Nous avons vu dans la section 4.1.2 comment un programme OC peut être représenté formellement par un *sted*.

Nous avons vu dans la section 4.2 que répartir un *sted* q sur n sites consiste à trouver un ensemble V d'actions de synchronisation et n systèmes $(q_i)_{i=1,n}$, agissant respectivement sur $A_i \cup V$, et tels que :

$$[\![q]\!] mod \sim \; = \; [\![(q_1 [\![q_2]\!] \ldots [\![q_n)/A]\!] mod \sim$$

Dans la section 4.1.5 nous avons introduit une relation de dépendance $\mathcal{D}$ sur les actions, basée sur les dépendances de données existant entre ces actions au niveau du programme OC. Nous avons également introduit la relation complémentaire $\mathcal{C}$, relation de commutation.

Dans la section 4.1.6 nous avons introduit la relation d'équivalence sur les traces d'actions $\sim$, induite par la relation de commutation $\mathcal{C}$.

Nous avons vu dans la section 4.3.3 comment représenter formellement par un *stod* un programme OC avec les dépendances entre les actions.

Dans la section 4.4.1 nous avons choisi pour V l'ensemble des dépendances sur les actions, c'est-à-dire $V = A \times A$. Nous avons de plus défini une fonction f transformant un *sted* en un *stod*.

Puis dans la section 4.4.6 nous avons défini l'opérateur ϕ_i de projection d'un *stod* sur le site i.

En posant, pour tout site i, $q_i = \phi_i(f(q))$, nous avons bien défini un modèle théorique complet de notre algorithme de parallélisation.

4.5 Preuve de l'algorithme de parallélisation

4.5.1 Comportement d'un *stod* réparti

Pour chaque site i, le *pomset* du comportement est défini sur le vocabulaire $A_i\mathcal{D} \cup \mathcal{D}A_i \cup A_i \cup C$, où C est l'ensemble des branchements.

La proposition suivante établit que le comportement d'un *stod* réparti (définition 4.25) est identique au comportement de ce même *stod* centralisé :

Proposition 4.2
Pour tout *sted* q, $[\![f(q)/A]\!] = [\![(\phi_1(f(q))]\!] \cdots [\![\phi_n(f(q)))/A]\!]$

Preuve du sens $\subseteq$:

Soit σ une séquence d'exécution maximale de $f(q)$, restreinte à A. Nous construisons le tuple $\Sigma = \langle \phi_1(\sigma), \ldots, \phi_n(\sigma) \rangle$. Puisque σ est une séquence maximale, il n'existe pas de tuple Σ' plus grand point à point que Σ.

Montrons alors que le *pomset* $o(\sigma)$ associé à la séquence σ vérifie l'équation 4.13. Cela revient à comparer, pour chaque i, les ordres $o'_i = o(\phi_i(\sigma))$ et $o''_i = o(\sigma)/(A_i\mathcal{D} \cup \mathcal{D}A_i \cup A_i \cup C)$.

Pour une action a n'appartenant pas à A_i, nous avons $\phi_i(f(a)) = \tilde{f}(a)/(A_i\mathcal{D} \cup \mathcal{D}A_i)$. Donc d'après la définition de $\tilde{f}$, la relation d'ordre du *pomset* $\phi_i(f(a))$ est plus partielle que celle de $f(a)/(A_i\mathcal{D} \cup \mathcal{D}A_i)$. Autrement dit, $o'_i \preceq o''_i$.

Preuve du sens $\supseteq$:

Soit Σ un tuple $\langle \sigma_1, \ldots, \sigma_n \rangle$ de séquences d'exécution de $\phi_1(f(q)) \ldots \phi_n(f(q))$. Montrons que le comportement $[\![\Sigma/A]\!]$ est dans $[\![f(q)/A]\!]$. $\forall i$, $\exists \sigma'_i$ séquence d'exécution de $f(q)$ telle que $\sigma_i = \phi_i(\sigma'_i)$. Montrons d'abord que toutes les séquences σ'_i sont identiques.

Elles ont toutes le même état initial : soit q_0 l'état initial de q, alors $\forall i$, $\sigma'_i = f(q_0) \stackrel{f(a_0)}{\rightarrow} f(q_1) \cdots$. Nous procédons par induction jusqu'au premier branchement et nous concluons que toutes les σ'_i ont le même préfixe σ_p. Donc $\forall i$, $\exists \sigma''_i$: $\sigma'_i = \sigma_p.\sigma''_i$ avec $\sigma_p = f(q_0) \stackrel{f(a_0)}{\rightarrow} f(q_1) \cdots f(q_n)$ et $f(q_n) = f(c)f(q') + f(c')f(q'')$ et $\forall i$, $c, c' \in A_i$.

Premier cas : le branchement n'est pas effectué. Mais dans ce cas les séquences σ'_i ne sont plus maximales, ce qui contredit l'hypothèse de l'équation 4.14.

Second cas : le branchement est effectué. Alors $\forall i$, $\sigma''_i = f(q_n) \stackrel{f(c)}{\rightarrow} f(q') \cdots$ ou bien $\sigma''_i = f(q_n) \stackrel{f(c')}{\rightarrow} f(q'') \cdots$ A cause de la synchronisation totale des branchements (section 4.4.8), nécessairement toutes les séquences effectuent le même choix :

- ou bien $\forall i,\ \sigma''_i = f(q_n) \stackrel{f(c)}{\rightarrow} f(q') \cdots$: dans ce cas nous posons $\sigma'_p = \sigma_p.f(c)$,
- ou bien $\forall i,\ \sigma''_i = f(q_n) \stackrel{f(c')}{\rightarrow} f(q'') \cdots$: dans ce cas nous posons $\sigma'_p = \sigma_p.f(c')$.

Donc $\forall i,\ \exists \sigma'''_i\ :\ \sigma'_i = \sigma'_p.\sigma'''_i$ où toutes les séquences σ'''_i ont le même état initial. Nous sommes alors ramenés au point de départ, et par induction nous déduisons que $\exists \sigma'\ :\ \forall i,\ \sigma'_i = \sigma'$. Par conséquent, $\forall i,\ \sigma_i = \phi_i(\sigma')$. Donc $\Sigma = \langle \phi_1(\sigma'), \ldots, \phi_n(\sigma') \rangle$, σ' étant une séquence d'exécution de $f(q)$. Donc $[\![\Sigma/A]\!] \in [\![f(q)/A]\!]$.

4.5.2 Théorème principal

Il ne reste plus qu'à prouver que nous avons bien résolu le problème énoncé à la section 4.2 :

Théorème 4.1
Pour tout *sted* q, $[\![q]\!] mod \sim\ = \ [\![(\phi_1(f(q))[\![\phi_2(f(q))[\![\ldots [\![\phi_n(f(q)))/A]\!] mod \sim$

Preuve :

Nous déduisons des propositions 4.1 et 4.2 que :

$$[\![q]\!] mod \sim\ = \ C([\![q]\!]) \ = \ l([\![f(q)]\!]/A) \ = \ l([\![(\phi_1(f(q))[\![\cdots [\![\phi_n(f(q)))/A]\!])$$

En définitive :

$$[\![q]\!] mod \sim\ = \ [\![(\phi_1(f(q))[\![\phi_2(f(q))[\![\ldots [\![\phi_n(f(q)))/A]\!] mod \sim$$

4.6 Conclusion

Nous avons donc défini une représentation formelle des programmes OC, ainsi qu'un modèle théorique de notre algorithme de parallélisation. Le théorème 4.1 prouve alors que le comportement d'un programme OC centralisé est fonctionnellement équivalent à celui des n programmes OC obtenus par application de cet algorithme.

La preuve utilise une représentation interne des programmes sous forme de systèmes de transitions étiquetées par des ordres partiels. Ces ordres partiels représentent les dépendances existant entre les actions du programme OC, dépendances qui sont directement déduites des dépendances de données entre ces actions.

Cette représentation est un peu lourde, mais elle est nécessaire du fait que la répartition se fait vers un nombre *quelconque* de sites. En effet, on peut, dans le cas de *deux* sites, travailler directement avec les traces d'actions générées. C'est ce qui est fait dans [17]. Toutefois, pour définir le comportement réparti d'un programme, il faut utiliser la notion de mélange équitable de *deux* traces, notion qui ne s'étend pas à *trois* traces.

Une autre solution pour éviter les problèmes d'équité consiste à modéliser les communications par des files d'attente bornées. La réception (`get`) est bloquante quand la file est vide, et l'émission (`put`) est bloquante quand la file est pleine. Mais cette modélisation ne correspond pas à la mise en œuvre de notre algorithme (section 3.1.2).

Chapitre 5

Resynchronisation

Le chapitre 3 présentait une méthode permettant d'obtenir un programme réparti, *fonctionnellement* équivalent au programme centralisé initial. Nous étudions dans ce chapitre les problèmes posés par la préservation de l'équivalence *temporelle* entre les programmes réparti et centralisé. Nous voyons d'abord dans quels cas il peut ne pas y avoir équivalence *temporelle*. Puis nous présentons divers compromis entre l'équivalence *temporelle* et le coût de cette équivalence.

5.1 Sémantique temporelle

Nous avons vu à la section 1.1.4 que le calcul d'horloge de LUSTRE attribue à chaque flot une unique horloge (flot booléen). Ainsi la sémantique temporelle de LUSTRE est : toute variable prend sa $n^{\text{ième}}$ valeur au $n^{\text{ième}}$ instant où son horloge vaut *true*. Ceci découle de l'hypothèse de synchronisme fort qui est posée comme base du langage. Il en va de même pour le langage SAGA puisque celui-ci repose sur la sémantique de LUSTRE. Quant à SIGNAL, on y retrouve la même notion de flot et d'horloge et la sémantique temporelle est la même.

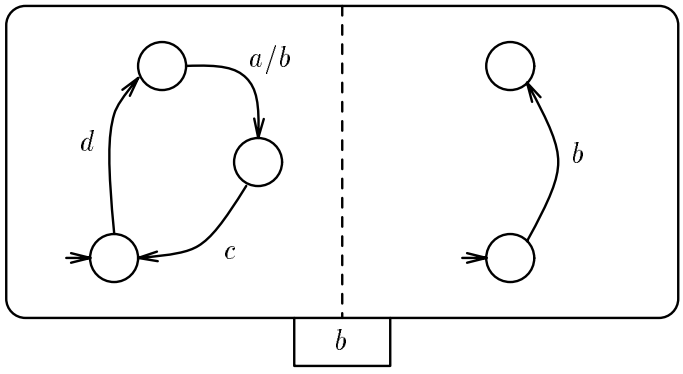


Figure 5.1: Composition parallèle de deux automates ARGOS

En ARGOS l'hypothèse de synchronisme intervient au moment de la composition parallèle de deux automates, en assurant qu'une transition étiquetée par un signal donné est synchrone avec la transition qui émet ce même message : c'est la diffusion synchrone.

Par exemple, dans le cas de la figure 5.1, les deux transitions respectivement étiquetées par a/b et b sont synchrones.

En ESTEREL la sémantique temporelle est également basée sur la diffusion synchrone.

Au niveau du programme OC, cette propriété est assurée par le comportement cyclique du programme. Bien entendu, cela doit rester vrai pour un programme réparti. Pourtant le programme OC réparti sur deux sites dans le chapitre 3 constitue un contre-exemple :

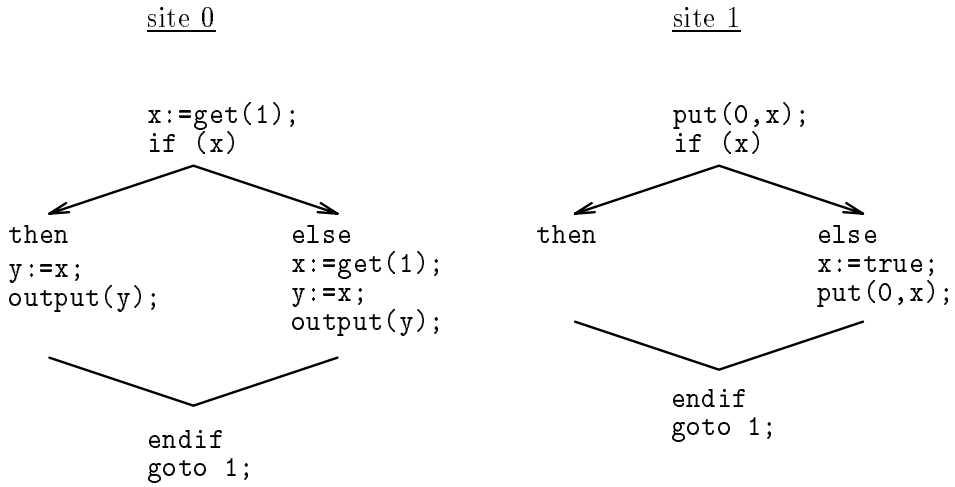


Figure 5.2: Programme OC réparti sur deux sites

La procédure `put` n'étant pas bloquante, le programme du site 1 peut aller aussi vite qu'il le veut, et en particulier beaucoup plus vite que celui du site 0. Par conséquent, les programmes des deux sites n'auront pas la même vitesse et leur comportement conjugué ne sera pas temporellement équivalent à celui du programme centralisé. Qui plus est, la différence entre les vitesses d'exécution des programmes répartis peut entraîner des dépassements de capacité des files d'attente. Nous passons ainsi d'un programme centralisé exécutable en mémoire bornée à un programme réparti qui ne l'est plus.

C'est pour cette raison qu'il est nécessaire de synchroniser les programmes répartis obtenus par la méthode fonctionnelle. A cause de l'absence d'horloge commune entre les sites, la synchronisation ne peut se faire que par des échanges de messages. Nous pouvons envisager deux méthodes de synchronisation différentes, la synchronisation forte et la synchronisation faible, que nous détaillons dans les sections suivantes.

5.2 Synchronisation forte

Nous exigeons que chaque programme ne commence son $n+1^{\text{ième}}$ cycle que quand tous les autres ont achevé leur $n^{\text{ième}}$ cycle. C'est le problème classique de la terminaison répartie. Dans le cas où les liaisons entre sites sont de simples liaisons bipoint, cela nécessite l'envoi, à chaque cycle et par chaque site, de $(nb_{site} - 1)$ messages. Si les liaisons entre sites permettent la diffusion, alors il ne faut plus qu'un envoi supplémentaire à chaque cycle et par chaque site.

Dans le cas d'ESTEREL, le compilateur crée une variable supplémentaire qui dépend de toutes les autres. Ainsi pour la terminaison répartie, il suffit que le site dont *dépend* cette variable envoie un message de synchronisation à tous les autres. Toutefois, pour calculer cette variable, le site *propriétaire* doit recevoir un message de la part de tous les autres sites. En définitive, le coût de la synchronisation forte est le même que dans le cas général, mais une moitié des messages est générée par l'algorithme de placement des émissions fonctionnelles (section 3.4).

Les messages de synchronisation ainsi rajoutés n'ont aucune valeur à transporter : ce sont de simples messages non valués implémentés par les primitives suivantes :

- `put_void(i)` permet d'envoyer un message de synchronisation au site `i`,
- `get_void(i)` permet de recevoir un message de synchronisation de la part du site `i`.

Si le noyau exécutif autorise la diffusion, par exemple au moyen de l'instruction `put` avec une liste de destinataires au lieu d'un seul, alors il suffit de diffuser un message à la fin de chaque cycle. Nous ajoutons donc, selon les cas, à la fin de chaque état et pour chaque site `s`, les instructions suivantes :

sans diffusion	avec diffusion
`put_void(1);`	`put_void(1,...,s-1,s+1,...,n);`
`...`	`get_void(1);`
`put_void(s-1);`	`...`
`put_void(s+1);`	`get_void(s-1);`
`...`	`get_void(s+1);`
`put_void(n);`	`...`
`get_void(1);`	`get_void(`nb_{site}`);`
`...`	
`get_void(s-1);`	
`get_void(s+1);`	
`...`	
`get_void(`nb_{site}`);`	

Bien entendu, les émissions sont effectuées *avant* les réceptions pour éviter les interblocages. L'inconvénient évident de la synchronisation forte est sa lourdeur due au nombre de messages à ajouter. C'est pourquoi on peut choisir d'alléger la contrainte de synchronisme entre les programmes répartis, en optant pour la synchronisation faible détaillée dans la section suivante.

Une autre solution consiste à faire circuler un jeton entre les sites à la fin de chaque cycle. Cela nécessite l'envoi à chaque cycle de $2 \times nb_{site}$ messages puisque le jeton doit effectuer deux tours :

- le premier pour vérifier que tous les programmes ont achevé leur cycle courant,
- et le second pour leur donner l'autorisation de commencer le cycle suivant.

L'inconvénient est que le temps de synchronisation est beaucoup plus long que pour la solution précédente. En effet, une fois que tous les programmes ont fini leur cycle, il faut encore faire circuler le jeton entre tous les sites.

5.3 Synchronisation faible

Nous autorisons ici les cycles n et $n+1$ à se chevaucher, et par conséquent chaque programme ne doit commencer sont $n + 2^{\text{ième}}$ cycle que quand tous les autres ont achevé leur $n^{\text{ième}}$ cycle. Cela nécessite qu'il y ait pour tout couple de sites au moins un échange de message (un dans chaque sens) à chaque cycle. Si ce n'est pas le cas, il faut alors ajouter des échanges de messages supplémentaires. L'avantage est que, contrairement à la synchronisation forte, nous pouvons tirer parti des échanges de messages valués déjà existants.

Remarquons qu'en fait nous disposons de deux options pour synchroniser les programmes répartis : nous pouvons synchroniser entre eux tous les programmes (synchronisation *totale*), où uniquement les programmes qui communiquent déjà (synchronisation *si besoin*). Les figures 5.3 et 5.4 illustrent ces deux options :

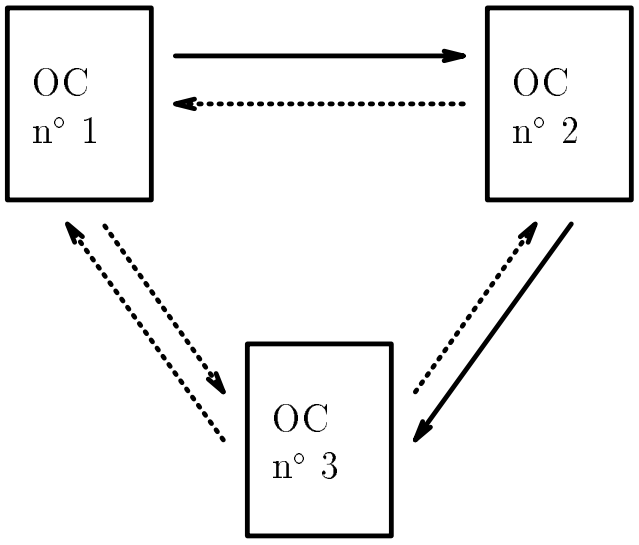


Figure 5.3: Synchronisation faible *totale*

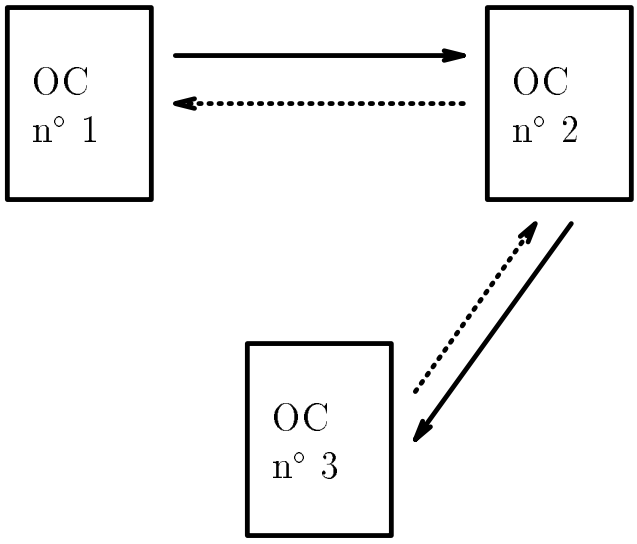


Figure 5.4: Synchronisation faible *si besoin*

Les flèches en noir représentent les communications déjà existantes entre les programmes, alors que les flèches en pointillé représentent les communications qui doivent être ajoutées par l'algorithme de synchronisation.

Nous définissons le graphe de synchronisation comme étant le graphe non orienté dont les sommets sont les sites, et dont les arêtes relient les sites synchronisés ensemble. Dans le cas de la synchronisation faible *totale*, le graphe est complet et il y a au plus un cycle de décalage entre tout couple de sites. Dans le cas de la synchronisation faible *si besoin*, le graphe n'est pas complet et il y a au plus n cycles de décalage entre tout couple de sites, n étant le diamètre du graphe. Si le graphe n'est pas connexe, alors avec la synchronisation faible *si besoin* le nombre de cycles de décalage peut ne pas être borné, et des sites seront complètement désynchronisés. Mais dans ce cas, ces sites sont indépendants et il n'est pas choquant qu'ils soient à ce point désynchronisés.

L'algorithme de synchronisation faible doit être appliqué juste après le placement des `put`. Nous nous contentons d'insérer des émissions de message non valués (`put_void`) ; les `get_void` correspondants seront insérés par l'algorithme de placement des réceptions décrit à la section 3.5. Bien entendu, les `put_void` seront placés le plus tôt possible. Pour chaque état, il faut insérer un `put_void(s)` dans le code de chaque site `t` tel que :

- cas de la synchronisation faible *totale* : le site `t` n'envoie aucun message valué au site `s`,
- cas de la synchronisation faible *si besoin* : le site `t` n'envoie aucun message valué au site `s` et reçoit au moins un message valué provenant du site `s`.

Une solution évidente consiste, pour chaque site `s`, à parcourir le graphe des actions de chaque état en mémorisant les sites auxquels `s` n'a envoyé aucun message, ainsi que les sites qui n'ont envoyé aucun message à `s`. Il ne reste plus alors qu'à insérer au début du graphe des actions les `put_void` nécessaires, en fonction de l'option choisie. Toutefois, il est plus judicieux de n'insérer

les put_void que dans les chemins d'exécution où cela est nécessaire. Ainsi, nous minimisons le nombre de messages de synchronisation.

Comme pour les messages valués, les put_void et les get_void sont placés de telle façon que les put_void sont effectués *avant* les get_void, ce qui assure l'absence d'interblocage. Pour cela, l'algorithme de synchronisation place les put_void le plus tôt possible, et c'est l'algorithme de placement des réceptions (section 3.5) qui place les get_void le plus tard possible, sans faire de distinction entre les messages de synchronisation et les messages valués.

5.3.1 Algorithme de resynchronisation faible *totale*

Dans le cas de la synchronisation faible *totale*, l'algorithme doit mémoriser les sites auxquels s n'a envoyé aucun message. Pour cela nous définissons, pour chaque site s, un ensemble $\texttt{Lout}_\texttt{s}$ contenant initialement tous les sites autres que s. Par la suite, $\texttt{Lout}_\texttt{s}$ contient à tout instant la liste des sites vers lesquels le site s n'a jamais effectué de put.

L'algorithme place un ensemble $\texttt{Lout}_\texttt{s}$ à chaque feuille du graphe et le propage vers la racine (propagation "en arrière") de la façon suivante :

- Quand on atteint une instruction put(t,x) effectuée par le site s, on enlève le site t de l'ensemble $\texttt{Lout}_\texttt{s}$.
- Quand on atteint une fermeture de test, on duplique les ensembles $\texttt{Lout}_\texttt{s}$, et on continue dans les branches then et else.
- Quand on atteint un test if, dsz ou present :
  - on insère dans la branche then (resp. else) une instruction put_void(t) pour chaque site t appartenant à $\texttt{Lout}_\texttt{s}^{\texttt{then}} - \texttt{Lout}_\texttt{s}^{\texttt{else}}$ (resp. $\texttt{Lout}_\texttt{s}^{\texttt{else}} - \texttt{Lout}_\texttt{s}^{\texttt{then}}$),
  - on retire chacun des sites t pour lesquels on vient d'insérer une instruction put_void(t) dans la branche then (resp. else) de l'ensemble $\texttt{Lout}_\texttt{s}^{\texttt{then}}$ (resp. $\texttt{Lout}_\texttt{s}^{\texttt{else}}$),
  - on continue avec l'union des ensembles $\texttt{Lout}_\texttt{s}^{\texttt{then}}$ et $\texttt{Lout}_\texttt{s}^{\texttt{else}}$.
- Quand on atteint la racine du programme, on insère une instruction put_void(t) pour chaque site t restant dans l'ensemble $\texttt{Lout}_\texttt{s}$.

L'ajout de messages de synchronisation donne pour le programme OC du chapitre 3 :

site	state 0	$\texttt{Lout}_0$	$\texttt{Lout}_1$	
(0)	`put_void(1);`	$\emptyset$	$\emptyset$	②
(1)	`put(0,x);`	$\{1\}$	$\emptyset$	
(0,1)	`if (x) then`	$\{1\}$	$\emptyset$	
(1)	`  put_void(0);`	$\{1\}$	$\emptyset$	①
(0)	`  y:=x;`	$\{1\}$	$\{0\}$	
(0)	`  output(y);`	$\{1\}$	$\{0\}$	
(0,1)	`else`			
(1)	`  x:=true;`	$\{1\}$	$\emptyset$	
(1)	`  put(0,x);`	$\{1\}$	$\emptyset$	
(0)	`  y:=x;`	$\{1\}$	$\{0\}$	
(0)	`  output(y);`	$\{1\}$	$\{0\}$	
(0,1)	`endif`	$\{1\}$	$\{0\}$	
(0,1)	`goto 1;`	$\{1\}$	$\{0\}$	

L'algorithme insère donc deux messages de synchronisation :

- le `put_void(0)` numéro ① sur le site 1 parce que $0 \in \texttt{Lout}_1^{\texttt{then}} - \texttt{Lout}_1^{\texttt{else}}$;
- le `put_void(1)` numéro ② sur le site 0 parce que $1 \in \texttt{Lout}_0$.

Il faut après cela placer les `get`, ce qui se fait de la façon indiquée à la section 3.5 :

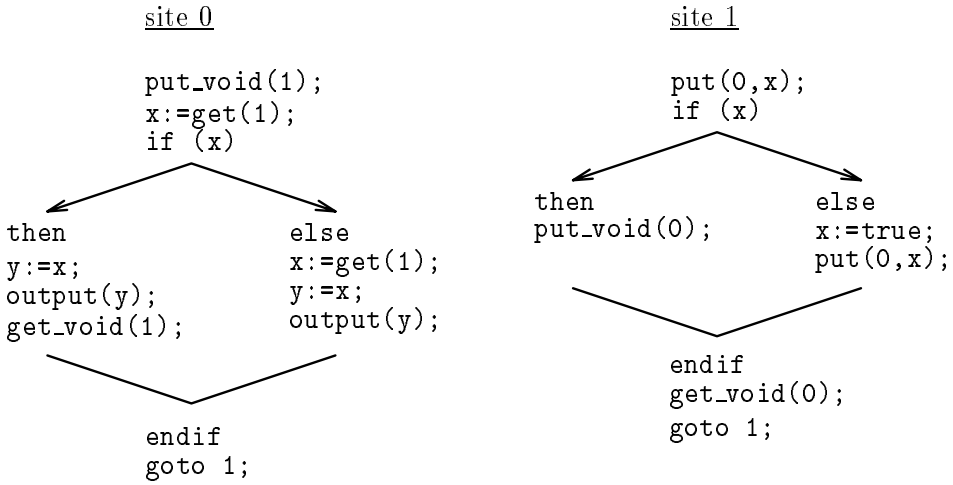


Figure 5.5: Programme OC réparti synchronisé *totalement*

5.3.2 Algorithme de resynchronisation faible *si besoin*

L'algorithme de synchronisation faible *si besoin* est un peu plus lourd puisqu'il faut en plus mémoriser les sites qui n'ont envoyé aucun message à `s`. Nous définissons, pour chaque site `s`, deux ensembles $\texttt{Lin}_\texttt{s}$ et $\texttt{Lout}_\texttt{s}$ contenant initialement tous les sites autres que `s`. Par la suite,

$\texttt{Lout}_s$ contient à tout instant la liste des sites vers lesquels le site `s` n'a jamais effectué de `put`. Par ailleurs, $\texttt{Lin}_s$ contient à tout instant la liste des sites qui n'ont jamais effectué de `put` vers le site `s`.

L'algorithme place deux ensembles $\texttt{Lout}_s$ et $\texttt{Lin}_s$ à chaque feuille du graphe et les propage vers la racine (propagation "en arrière") de la façon suivante :

- Quand on atteint une instruction `put(t,x)` effectuée par le site `s`, on enlève le site `t` de l'ensemble $\texttt{Lout}_s$.
- Quand on atteint une instruction `put(s,x)` effectuée par le site `t`, on enlève le site `t` de l'ensemble $\texttt{Lin}_s$.
- Quand on atteint une fermeture de test, on duplique les ensembles $\texttt{Lin}_s$ et $\texttt{Lout}_s$, et on continue dans les branches `then` et `else`.
- Quand on atteint un test `if`, `dsz` ou `present` :
  - on calcule dans $\texttt{Lthen}_s$ (resp. $\texttt{Lelse}_s$) la différence $\texttt{Lout}_s^{\texttt{then}} - \texttt{Lin}_s^{\texttt{then}}$ (resp. $\texttt{Lout}_s^{\texttt{else}} - \texttt{Lin}_s^{\texttt{else}}$),
  - on insère dans la branche `then` (resp. `else`) une instruction `put_void(t)` pour chaque site `t` appartenant à $\texttt{Lthen}_s - \texttt{Lelse}_s$ (resp. $\texttt{Lelse}_s - \texttt{Lthen}_s$),
  - on retire chacun des sites `t` pour lesquels on vient d'insérer une instruction `put_void(t)` dans la branche `then` (resp. `else`) de l'ensemble $\texttt{Lout}_s^{\texttt{then}}$ (resp. $\texttt{Lout}_s^{\texttt{else}}$),
  - on continue avec l'union des ensembles $\texttt{Lout}_s^{\texttt{then}}$ et $\texttt{Lout}_s^{\texttt{else}}$, et avec l'intersection des ensembles $\texttt{Lin}_s^{\texttt{then}}$ et $\texttt{Lin}_s^{\texttt{else}}$.
- Quand on atteint la racine du programme, on insère une instruction `put_void(t)` pour chaque site `t` restant dans l'ensemble $\texttt{Lout}_s - \texttt{Lin}_s$.

L'ajout de messages de synchronisation donne pour le programme OC du chapitre 3 :

site	state 0	$\texttt{Lout}_0$	$\texttt{Lin}_0$	$\texttt{Lout}_1$	$\texttt{Lin}_1$	
(0)	`put_void(1);`	$\emptyset$	$\emptyset$	{0}	{0}	②
(1)	`put(0,x);`	{1}	$\emptyset$	{0}	{0}	
(0,1)	`if (x) then`	{1}	$\emptyset$	{0}	{0}	
(0)	`  y:=x;`	{1}	{1}	{0}	{0}	
(0)	`  output(y);`	{1}	{1}	{0}	{0}	
(0,1)	`else`					
(0)	`  put_void(1);`	$\emptyset$	$\emptyset$	$\emptyset$	{0}	①
(1)	`  x:=true;`	{1}	$\emptyset$	$\emptyset$	{0}	
(1)	`  put(0,x);`	{1}	$\emptyset$	$\emptyset$	{0}	
(0)	`  y:=x;`	{1}	{1}	{0}	{0}	
(0)	`  output(y);`	{1}	{1}	{0}	{0}	
(0,1)	`endif`	{1}	{1}	{0}	{0}	
(0,1)	`goto 1;`	{1}	{1}	{0}	{0}	

L'algorithme insère donc deux messages de synchronisation :

- le `put_void(1)` numéro ① sur le site 0 parce que $\texttt{Lthen}_0 = \emptyset$, $\texttt{Lelse}_0 = \{1\}$ et donc $1 \in \texttt{Lelse}_0 - \texttt{Lthen}_0$;
- le `put_void(1)` numéro ② sur le site 0 parce que $1 \in \texttt{Lout}_0 - \texttt{Lin}_0$ et on atteint la racine du graphe.

Il faut après cela placer les `get`, ce qui se fait de la façon indiquée à la section 3.5 :

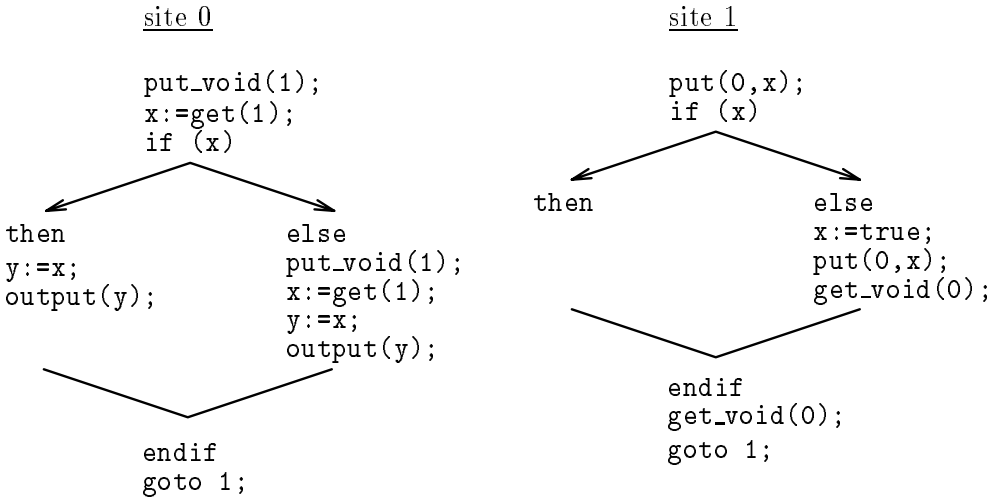


Figure 5.6: Programme OC réparti synchronisé *si besoin*

5.3.3 Calcul de complexité

Il reste à faire le calcul de complexité, qui est analogue à celui effectué pour le placement des `put` (section 3.4.3). Nous conservons les même notations et nous trouvons :

- $\boxed{\mathcal{T}(\text{synchronisation faible } \textit{totale}) = O(|Q| \times moy_{act}| \times nb_{site})}$
- $\boxed{\mathcal{M}(\text{synchronisation faible } \textit{totale}) = O(nb_{site}^2)}$
- $\boxed{\mathcal{T}(\text{synchronisation faible } \textit{si besoin}) = O(|Q| \times moy_{act}| \times nb_{site})}$
- $\boxed{\mathcal{M}(\text{synchronisation faible } \textit{si besoin}) = O(nb_{site}^2)}$

5.4 Conclusion

L'algorithme de parallélisation (chapitre 3) produit un programme réparti fonctionnellement équivalent au programme centralisé initial. Mais il n'assure pas la préservation de la sémantique temporelle, qui est pourtant indispensable pour assurer l'exécution en mémoire bornée. Nous venons de présenter plusieurs algorithmes de resynchronisation permettant de rétablir cette sémantique temporelle et offrant divers compromis entre le coût et l'équivalence temporelle :

type de resynchronisation	équivalence temporelle	coût de la resynchronisation
resynchronisation forte	chaque site ne commence son $n+1^{\text{ième}}$ cycle que lorsque tous les autres ont achevé leur $n^{\text{ième}}$ cycle	il faut envoyer $nb_{site} \times (nb_{site} - 1)$ messages supplémentaires à chaque cycle
resynchronisation faible	chaque site ne commence son $n+2^{\text{ième}}$ cycle que lorsque tous les autres ont achevé leur $n^{\text{ième}}$ cycle	il doit y avoir au moins un échange de message entre chaque site et à chaque cycle

Le choix entre les divers algorithmes est laissé à l'utilisateur sous la forme d'une option de compilation (voir le chapitre 10).

Enfin, dans le cas de la synchronisation faible *si besoin*, il peut y avoir des messages de synchronisation redondants. C'est notamment le cas du programme de la figure 5.6. Comme pour les messages valués redondants, nous voyons au chapitre 6 comment les supprimer.

Chapitre 6

Elimination des messages redondants

L'algorithme de parallélisation du chapitre 3 peut, dans certains cas, générer des messages redondants (voir section 3.6). Les échanges de variables font intervenir le temps de transmission du réseau dans le temps de réaction du programme. Or les domaines d'application de la programmation des systèmes réactifs sont critiques : nucléaire, avionique ... Pour de tels systèmes, le temps de réaction joue un rôle important dans la validation du programme. C'est pourquoi il est nécessaire de détecter et d'éliminer les messages redondants.

Nous exposons dans ce chapitre un algorithme d'élimination des messages redondants. Il concerne aussi bien les messages valués (générés par l'algorithme de parallélisation du chapitre 3) que les messages de synchronisation non valués (générés par l'algorithme de resynchronisation du chapitre 5).

Nous décomposons cet algorithme en deux étapes :

- étape d'analyse statique globale sur la structure de contrôle de l'automate OC (apparentée aux techniques d'analyse statique présentées dans [1]) ;
- étape d'élimination locale sur chaque état de l'automate OC.

Nous motivons cette décomposition par trois raisons :

- Les problèmes posés par ces deux types d'optimisation ne sont pas les mêmes : code purement séquentiel dans les états et boucles dans l'automate.
- La notion d'état est liée à celle de réaction atomique du programme synchrone.
- Il est impératif de ne pas éliminer globalement les messages de synchronisation.

Nous motivons dans un premier temps l'élimination des messages redondants par deux exemples, illustrant respectivement une redondance locale et une redondance globale. Puis nous montrons

comment l'élimination peut être effectuée globalement puis localement. Nous détaillons alors successivement les algorithmes des deux phases de l'élimination, en les illustrant avec les exemples précédemment présentés. Enfin, nous donnons le calcul de complexité total de l'élimination des messages redondants, et nous en discutons les performances, avant de conclure.

6.1 Exemples de redondance des émissions

Il est facile de trouver un exemple de programme réparti comportant un message dont la valeur véhiculée est déjà connue. Il suffit qu'une variable soit initialisée dans un état, puis ne soit jamais modifiée sur le site dont elle *dépend*, tout en étant nécessaire pour un calcul sur un autre site :

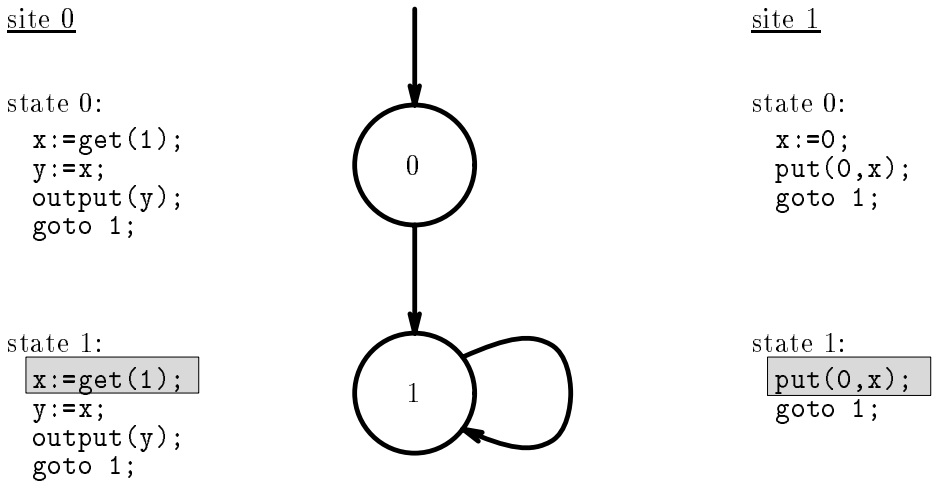


Figure 6.1: Exemple de redondance globale

Ce programme OC réparti a été obtenu avec l'algorithme de parallélisation décrit au chapitre 3. Les directives de répartition choisies font que la variable `y` dépend du site `0`, et que `x` dépend du site `1`. Dans l'état `1`, le message échangé entre les deux automates (`put(0;x)` apparaissant en grisé) est redondant. En effet, la valeur de `x` qui est ainsi transmise n'a pas été recalculée depuis le dernier message. Remarquons que si la variable `x` était une variable d'entrée (c'est-à-dire portée par un signal d'entrée), alors elle serait implicitement mise à jour à chaque changement d'état, et dans ce cas aucun message ne pourrait être supprimé.

Nous pouvons même trouver un exemple de message redondant au niveau d'un état de l'automate. Cela peut se produire avec la stratégie de placement des émissions *si besoin*. Si une variable est calculée et envoyée avant un test, puis rafraîchie dans une seule des deux branches, alors qu'elle est nécessaire pour un calcul (sur un autre site que son "propriétaire") dans les deux branches, alors nous aurons, dans la branche où cette variable n'est pas rafraîchie, un message redondant :

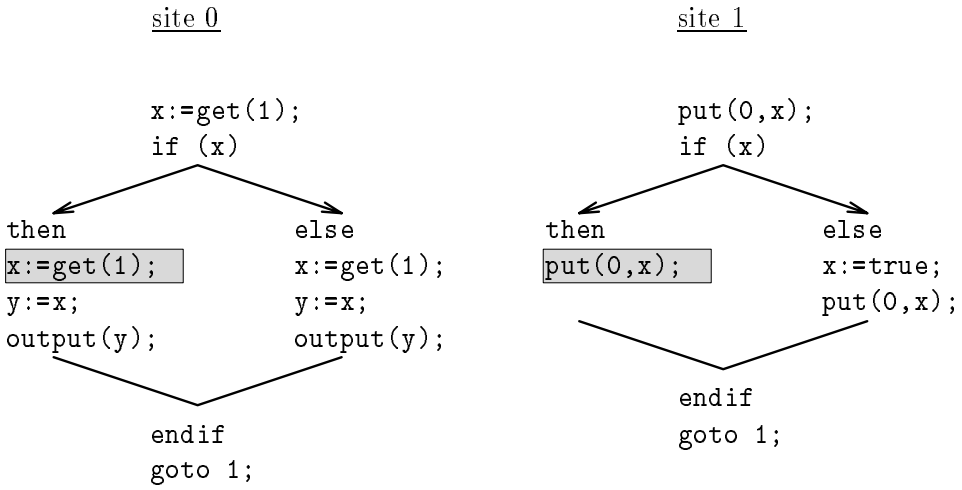


Figure 6.2: Exemple de redondance locale

La localisation des variables `x` et `y` est la même que dans la figure 6.1. Là encore, le message apparaissant en grisé est redondant. Avec la stratégie *au plus tôt*, une telle situation est toutefois impossible : en effet, une variable est envoyée aux sites qui en ont besoin juste après avoir été calculée, et non au moment où elle est utilisée. Dans la figure 6.2, le `put(0;x)` apparaissant en grisé n'existera donc pas (voir la section 3.6 pour plus de détails sur les stratégies de placement des émissions).

Un dernier exemple est illustré par la figure 5.6, page 83. Elle montre un programme OC réparti, auquel on a appliqué l'algorithme de resynchronisation faible *si besoin*, et dont le message de synchronisation échangé entre le site 0 et le site 1 dans la branche `else` est redondant.

6.2 Deux niveaux d'optimisation

Les deux exemples 6.1 et 6.2 nous montrent que l'élimination des messages redondants doit intervenir tant au niveau des états de l'automate, que du programme dans son ensemble. La distinction entre ces deux niveaux est essentielle. D'une part les problèmes posés par ces deux types d'optimisation ne sont pas les mêmes : au niveau d'un état de l'automate le code est purement séquentiel, alors que l'automate lui-même peut comporter des boucles. D'autre part la notion d'état est liée à celle de réaction atomique du programme synchrone : un changement d'état est une réaction à un stimulus de l'environnement, et est accompagné de la mise à jour des variables portées par les signaux d'entrée. Ceci est symbolisé par la présence, au début de chaque état, de l'instruction implicite `go` (voir à ce sujet la section 1.3.3).

Enfin il faut distinguer les messages valués de ceux de synchronisation, à cause des problèmes posés par la synchronisation des programmes répartis (chapitre 5). En effet, enlever un message de synchronisation dans un état de l'automate, sous prétexte qu'il y en a déjà eu un dans l'état précédent, reviendrait à remettre en cause toute la resynchronisation ! Aussi l'élimination des messages de synchronisation n'est permise qu'au niveau d'un état de l'automate. Nous montrons

toutefois comment les messages de synchronisation peuvent être pris en compte par l'étape locale au même titre que les messages valués.

La solution que nous proposons repose sur l'analyse "flots de données" du programme. Elle intervient *avant* la phase de placement des réceptions (`get`) pour éviter de revenir sur cette phase. Par suite, le placement des réceptions (décrit à la section 3.5) se fera sur un programme dont les émissions seront déjà optimisées. Pour chaque site, cette optimisation est divisée en deux étapes :

- **Au niveau de l'automate complet :**
  La première étape de l'algorithme calcule, pour chaque transition, la liste des variables reçues ainsi que la liste des variables affectées au cours de cette transition ; à partir de là nous pouvons calculer, pour chaque état, la liste des variables connues quelle que soit la transition ayant abouti à cet état.
- **Au niveau d'un état de l'automate :**
  La deuxième étape de l'algorithme exploite la liste des variables connues en début de chaque état ; pour chaque action, nous connaissons la liste des variables qu'elle modifie ; nous pouvons donc calculer à chaque instant la liste des variables non-réaffectées depuis leur dernière émission ; par suite nous supprimons un `put` si la variable émise est déjà connue par le site qui la reçoit ; enfin, pour prendre en compte les messages de synchronisation, nous ajoutons à la liste des variables du programme la variable fictive `void`.

Nous présentons successivement les fondements théoriques et les algorithmes correspondant à ces deux étapes.

6.3 Étape d'analyse statique globale

Cette première étape vise à déterminer, pour chaque état, la liste des variables que l'on est sûr de connaître, quel que soit le chemin d'exécution ayant conduit à cet état.

6.3.1 Formalisation

Nous travaillons sur un site donné et nous considérons l'automate projeté sur ce site. A un état p donné, les variables connues sont les variables qui étaient déjà connues dans les états prédécesseurs de p et qui n'ont pas été réaffectées au cours de la transition aboutissant à p, plus les variables reçues au cours de cette même transition et non réaffectées depuis leur émission. Nous introduisons les définitions suivantes :

Définition 6.1 (graphe d'états)

- $\{Q, q_0, \rightarrow\}$ représente l'automate OC ; Q = {états de l'automate}, q_0 est l'état initial et $\rightarrow$ est la fonction de transition ; $|Q|$ représente le nombre d'états de l'automate ; les états sont par convention numérotés de 0 à $|Q| - 1$.
- $succ_n(q_0)$ = {états accessibles depuis l'état initial q_0 en n transitions}.

- $pre(p)$ = {états immédiatement prédécesseurs de l'état p} = $\{q \in Q | q \rightarrow p\}$.

Définition 6.2 (ensembles de variables)

- $E_n(p) = \begin{cases} \{\text{variables connues à l'état } p\} & \text{si } p \in succ_n(q_0), \\ \text{indéfini} & \text{sinon.} \end{cases}$
- $E(p)$ = {variables connues à l'état p}.
- $A(p', p)$ = {variables affectées au cours de la transition $p' \rightarrow p$}.
- $R(p', p)$ = {variables reçues et non réaffectées au cours de la transition $p' \rightarrow p$}.
- $C(p)$ = {variables certainement non connues au début de l'état p}.

Par convention, $\top$ désigne l'ensemble contenant toutes les variables du système. Dans l'exemple de la figure 6.1, nous avons deux variables `x` et `y`, donc $\top = \{\mathtt{x}, \mathtt{y}\}$. Dans toute la suite, le complémentaire d'un ensemble de variables sera pris par rapport à $\top$. Par exemple, $\overline{\emptyset} = \top$.

Le but de la première étape est de calculer pour chaque transition $p' \rightarrow p$ les ensembles $A(p', p)$ et $R(p', p)$, ainsi que pour chaque état p l'ensemble $C(p)$. Or nous avons clairement le lemme suivant :

Lemme 6.1
$\forall p \in Q,\ C(p) = \bigcup_{p' \in pre(p)} [A(p', p) \cap \overline{R(p', p)}]$.

A un état p accessible depuis l'état initial q_0 en n transitions, les variables connues sont les variables qui étaient déjà connues précédemment et qui n'ont pas été réaffectées, plus les variables qui ont été reçues au cours de la dernière transition :

$$\forall p \in succ_{n+1}(q_0),\ E_{n+1}(p) = \bigcap_{\substack{p' \in pre(p) \\ p' \in succ_n(q_0)}} [(E_n(p') - A(p', p)) \cup R(p', p)]$$

Comme nous ne connaissons jamais le chemin d'exécution sur lequel nous nous trouvons, nous devons prendre en compte tous les chemins possibles pour connaître les variables connues à un état p donné :

$$\forall p \in Q,\ E(p) = \bigcap_{\substack{n=0 \\ p \in succ_n(q_0)}}^{\infty} E_n(p)$$

Autrement dit, nous avons le lemme suivant :

Lemme 6.2
Pour tout état p, $E(p)$ est le plus grand point fixe de l'équation :

$$E(p) = \bigcap_{p' \in pre(p)} [(E(p') - A(p', p)) \cup R(p', p)]$$

La forme duale nous permet de supprimer l'opérateur "$-$" de soustraction sur les ensembles, et d'obtenir une forme disjonctive :

$$\begin{aligned}
\forall p \in Q,\ \overline{E(p)} &= \bigcup_{p' \in pre(p)} [\overline{(E(p') - A(p',p))} \ \cap\ \overline{R(p',p)}] \\
&= \bigcup_{p' \in pre(p)} [(\overline{E(p')} \cup A(p',p)) \ \cap\ \overline{R(p',p)}] \\
&= \bigcup_{p' \in pre(p)} [\overline{E(p')} \cap \overline{R(p',p)}] \ \cup \bigcup_{p' \in pre(p)} [A(p',p) \cap \overline{R(p',p)}]
\end{aligned}$$

Le lemme 6.2 prouve que $E(p)$ est un plus grand point fixe. Or la dernière équation fait intervenir $\overline{E(p)}$ et non plus $E(p)$. Donc, grâce au lemme 6.1, nous obtenons finalement le lemme suivant :

Lemme 6.3
Pour tout état p, $\overline{E(p)}$ est le plus petit point fixe de l'équation :

$$\overline{E(p)} = \bigcup_{p' \in pre(p)} [\overline{E(p')} \cap \overline{R(p',p)}] \cup C(p)$$

Autrement dit, nous obtenons un système d'équations linéaires à résoudre sur l'ensemble Q des états de l'automate du programme OC.

6.3.2 Coefficients du système

Remarquons tout d'abord que chaque ligne du système est une équation de point fixe. Ou bien $p \notin pre(p)$ et l'équation du lemme 6.3 s'écrit :

$$\overline{E(p)} = \bigcup_{\substack{p' \in pre(p) \\ p' \neq p}} [\overline{E(p')} \cap \overline{R(p',p)}] \ \cup\ C(p)$$

Ou bien $p \in pre(p)$ et l'équation du lemme 6.3 s'écrit :

$$\overline{E(p)} = [\overline{E(p)} \cap \overline{R(p,p)}] \ \cup \bigcup_{\substack{p' \in pre(p) \\ p' \neq p}} [\overline{E(p')} \cap \overline{R(p',p)}] \ \cup\ C(p)$$

Nous avons établi le lemme 6.3 prouvant que $\overline{E(p)}$ est un plus petit point fixe. Aussi la résolution par itération de cette dernière équation nous donne :

$$\begin{aligned}
\overline{E(p)}^0 &= \emptyset \\
\overline{E(p)}^1 &= \bigcup_{\substack{p' \in pre(p) \\ p' \neq p}} [\overline{E(p')} \cap \overline{R(p',p)}] \ \cup\ C(p) \\
\overline{E(p)}^2 &= [\overline{E(p)}^1 \cap \overline{R(p,p)}] \cup \overline{E(p)}^1 \\
&= \overline{E(p)}^1
\end{aligned}$$

En définitive, nous devons résoudre le système linéaire suivant :

$$\left\{ \begin{array}{l} \vdots \\ \overline{E(p)} = \bigcup\limits_{\substack{p' \in pre(p) \\ p' \neq p}} [\overline{E(p')} \cap \overline{R(p',p)}] \ \cup \ C(p) \\ \vdots \end{array} \right.$$

Pour l'automate de la figure 6.1, qui a deux états, nous avons pour chaque site :

$$\left\{ \begin{array}{l} \overline{E(0)} = \emptyset \\ \overline{E(1)} = [\overline{E(0)} \cap \overline{R(0,1)}] \ \cup \ C(1) \end{array} \right.$$

6.3.3 Algorithme de calcul des coefficients

Le calcul des ensembles A, R et C (c'est-à-dire les coefficients de la matrice et du vecteur du système linéaire) est facile. Il se fait pour chaque site `s` après avoir placé les émissions. Nous définissons deux matrices $A_{\mathtt{s}}$ et $R_{\mathtt{s}}$ de dimension $|Q| \times |Q|$, et un vecteur $C_{\mathtt{s}}$ de dimension $|Q|$. Le graphe orienté représentant l'automate n'est pas complet. Donc les matrices $A_{\mathtt{s}}$ et $R_{\mathtt{s}}$ sont creuses et pour chaque transition $p' \rightarrow p$ inexistante, nous avons les cases suivantes :

$$\left\{ \begin{array}{lcl} A_s[p',p] & := & \emptyset \\ R_s[p',p] & := & \top \end{array} \right.$$

De plus, pour un automate à deux états numérotés 0 et 1, la matrice $A_{\mathtt{s}}$ (respectivement $R_{\mathtt{s}}$) est de la forme :

$$A_{\mathtt{s}} = \left(\begin{array}{cc} A_{\mathtt{s}}[0,0] & A_{\mathtt{s}}[0,1] \\ A_{\mathtt{s}}[1,0] & A_{\mathtt{s}}[1,1] \end{array} \right)$$

A chaque site `s` nous associons un ensemble $\mathtt{Affectees}_{\mathtt{s}}$, contenant toutes les variables qui ont été affectées par un site autre que `s`, et un ensemble $\mathtt{Recues}_{\mathtt{s}}$, contenant toutes les variables reçues par le site `s`. Nous plaçons à la racine du graphe des actions deux ensembles $\mathtt{Affectees}_{\mathtt{s}}$ et $\mathtt{Recues}_{\mathtt{s}}$ vides, pour chaque site `s`. Nous parcourons le graphe d'actions de chaque état p' jusqu'aux feuilles (propagation *en avant*), de la façon suivante :

- Quand on atteint une émission `put(s;x)`, on ajoute `x` à l'ensemble $\mathtt{Recues}_{\mathtt{s}}$.
- Quand on atteint une affectation `x:=exp`, on ajoute `x` à chaque ensemble $\mathtt{Affectees}_{\mathtt{s}}$, pour tout site `s` dont ne *dépend* pas `x`, et on l'enlève de chaque ensemble $\mathtt{Recues}_{\mathtt{s}}$, pour tous ces mêmes sites.
- Quand on atteint un test `if`, `dsz` ou `present`, pour chaque site `s` on duplique les ensembles $\mathtt{Affectees}_{\mathtt{s}}$ et $\mathtt{Recues}_{\mathtt{s}}$, et on continue dans les deux branches du test.
- Quand on atteint une fermeture de test, pour chaque site `s` on fait l'union des deux ensembles $\mathtt{Affectees}_{\mathtt{s}}$ provenant des deux branches, on fait l'intersection des deux ensembles $\mathtt{Recues}_{\mathtt{s}}$ provenant des deux branches, et on continue avec ces nouveaux ensembles.

- Quand on atteint une feuille (un `goto p`), on a fini l'algorithme de calcul, et on remplit les deux matrices :

$$\begin{cases} A_s[p',p] & := & A_s[p',p] \cup \texttt{Affectees}_s \\ R_s[p',p] & := & R_s[p',p] \cap \texttt{Recues}_s \end{cases}$$

  En effet, les résultats calculés dans $\texttt{Affectees}_\texttt{s}$ et $\texttt{Recues}_\texttt{s}$ ne sont que des résultats partiels, puisqu'il peut y avoir plusieurs feuilles dans le même graphe correspondant à la même transition dans l'automate.

Il ne reste plus qu'à calculer les coefficients du vecteur du système, selon le lemme 6.1 :

$$\forall p \in Q,\ C_s[p] := \bigcup_{p' \in Q} (A_s[p',p] \cap \overline{R_s[p',p]})$$

Considérons l'exemple de la figure 6.1. Nous donnons le code complet de l'état 0, avec pour chaque action la liste des sites qui doivent exécuter cette action. Nous montrons de plus comment se propagent les ensembles $\texttt{Affectees}_\texttt{s}$ et $\texttt{Recues}_\texttt{s}$ pour les sites 0 et 1. La dernière ligne contient les résultats :

site	state 0	$\texttt{Affectees}_0$	$\texttt{Affectees}_1$	$\texttt{Recues}_0$	$\texttt{Recues}_1$
(1)	`x:=0;`	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
(1)	`put(0;x);`	{x}	$\emptyset$	$\emptyset$	$\emptyset$
(0)	`y:=x;`	{x}	$\emptyset$	{x}	$\emptyset$
(0)	`output(y);`	{x}	{y}	{x}	$\emptyset$
(0,1)	`goto 1;`	{x}	{y}	{x}	$\emptyset$

Et pour l'état 1, nous obtenons :

site	state 1	$\texttt{Affectees}_0$	$\texttt{Affectees}_1$	$\texttt{Recues}_0$	$\texttt{Recues}_1$
(1)	`put(0;x);`	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
(0)	`y:=x;`	$\emptyset$	$\emptyset$	{x}	$\emptyset$
(0)	`output(y);`	$\emptyset$	{y}	{x}	$\emptyset$
(0,1)	`goto 1;`	$\emptyset$	{y}	{x}	$\emptyset$

En fin de compte, les matrices et les coefficients obtenus sont :

$$R_0 = \begin{pmatrix} \{\texttt{x},\texttt{y}\} & \{\texttt{x}\} \\ \{\texttt{x},\texttt{y}\} & \{\texttt{x}\} \end{pmatrix} \quad R_1 = \begin{pmatrix} \{\texttt{x},\texttt{y}\} & \emptyset \\ \{\texttt{x},\texttt{y}\} & \emptyset \end{pmatrix}$$

$$A_0 = \begin{pmatrix} \emptyset & \{\texttt{x}\} \\ \emptyset & \emptyset \end{pmatrix} \quad A_1 = \begin{pmatrix} \emptyset & \{\texttt{y}\} \\ \emptyset & \{\texttt{y}\} \end{pmatrix}$$

$$C_0 = \begin{pmatrix} \emptyset \\ \emptyset \end{pmatrix} \quad C_1 = \begin{pmatrix} \emptyset \\ \{\texttt{y}\} \end{pmatrix}$$

6.3.4 Substitution du système linéaire

Pour résoudre le système linéaire de la section 6.3.2, nous le triangularisons. Nous effectuons donc les substitutions dans l'ordre $0, \ldots, |Q|-1$. A l'étape i, nous substituons $\overline{E(i)}$ dans les équations $i+1$ à $|Q|-1$, les $i-1$ substitutions précédentes étant déjà effectuées.

$$\begin{cases} \vdots \\ \overline{E(i)} = \bigcup_{i'>i} [\overline{E(i')} \cap \overline{R(i',i)}] \cup C(i) \\ \vdots \\ \overline{E(j)} = \bigcup_{\substack{j'>i \\ j' \neq j}} [\overline{E(j')} \cap \overline{R(j',j)}] \cup C(j) \\ \vdots \end{cases}$$

Substituons $\overline{E(i)}$ dans l'équation numéro j :

$$\begin{aligned} \overline{E(j)} &= \bigcup_{\substack{j'>i \\ j' \neq j}} [\overline{E(j')} \cap \overline{R(j',j)}] \cup [\overline{E(i)} \cap \overline{R(i,j)}] \cup C(j) \\ &= \bigcup_{\substack{j'>i \\ j' \neq j}} [\overline{E(j')} \cap \overline{R(j',j)}] \cup \bigcup_{i'>i} [\overline{E(i')} \cap \overline{R(i',i)} \cap \overline{R(i,j)}] \\ &\quad \cup [C(i) \cap \overline{R(i,j)}] \cup C(j) \\ &= \bigcup_{\substack{j'>i \\ j' \neq j}} [\overline{E(j')} \cap [\overline{R(j',j)} \cup (\overline{R(j',i)} \cap \overline{R(i,j)})]] \\ &\quad \cup [\overline{E(j)} \cap \overline{R(j,i)} \cap \overline{R(i,j)}] \cup [C(i) \cap \overline{R(i,j)}] \cup C(j) \end{aligned}$$

C'est donc à nouveau une équation de point fixe, et comme nous cherchons le plus petit (c'est la même chose qu'à la section 6.3.2), nous pouvons supprimer le terme $[\overline{E(j)} \cap \overline{R(j,i)} \cap \overline{R(i,j)}]$. L'équation j devient alors :

$$\overline{E(j)} = \bigcup_{\substack{j'>i \\ j' \neq j}} [\overline{E(j')} \cap [\overline{R(j',j)} \cup (\overline{R(j',i)} \cap \overline{R(i,j)})]] \cup [C(i) \cap \overline{R(i,j)}] \cup C(j)$$

Par conséquent, la substitution de $\overline{E(i)}$ dans l'équation j revient à effectuer les opérations suivantes sur la matrice et le vecteur du système :

$$\overline{R[j',j]} := \overline{R[j',j]} \cup (\overline{R[j',i]} \cap \overline{R[i,j]}),\ \forall j' > i \text{ et } j' \neq j \quad (6.1)$$

$$C[j] := C[j] \cup (C[i] \cap \overline{R[i,j]}) \quad (6.2)$$

Par substitution dans le système de notre exemple, nous obtenons pour chaque site :

$$\begin{cases} \overline{E(0)} = \emptyset \\ \overline{E(1)} = C(1) \end{cases}$$

6.3.5 Résolution du système linéaire

Une fois toutes les substitutions effectuées, le système se présente ainsi :

$$\begin{cases} \overline{E(0)} & = & \bigcup\limits_{i'>0} [\overline{E(i')} \cap \overline{R(i',0)}] \ \cup \ C(0) \\ \vdots & & \\ \overline{E(i)} & = & \bigcup\limits_{i'>i} [\overline{E(i')} \cap \overline{R(i',i)}] \ \cup \ C(i) \\ \vdots & & \\ \overline{E(|Q|-1)} & = & C(|Q|-1) \end{cases}$$

Pour notre exemple, nous avons pour tout site $s \in \{0,1\}$ le système suivant :

$$\begin{cases} \overline{E_s(0)} = [\overline{E_s(1)} \cap \overline{R_s(1,0)}] \ \cup \ C_s(0) \\ \overline{E_s(1)} = C_s(1) \end{cases}$$

En remplaçant les ensembles $E_{\mathtt{s}}$ et $R_{\mathtt{s}}$ par leurs valeurs respectives, nous retrouvons bien le système de la section 6.3.4 :

$$\text{site } 0 : \begin{cases} \overline{E(0)} & = & \emptyset \\ \overline{E(1)} & = & C_0(1) \end{cases} \qquad \text{site } 1 : \begin{cases} \overline{E(0)} & = & \emptyset \\ \overline{E(1)} & = & C_1(1) \end{cases}$$

Nous calculons donc les ensembles $\overline{E(i)}$ dans l'ordre $|Q|-1, \ldots, 0$, en remplaçant à chaque fois l'ensemble précédemment calculé dans les autres équations. Ainsi, quand on vient de calculer l'ensemble $\overline{E(i)}$, remplacer sa valeur dans l'équation j (avec $j < i$) revient à modifier $C(j)$ de la façon suivante :

$$C[j] := C[j] \cup (C[i] \cap \overline{R[i,j]}) \tag{6.3}$$

A la suite de ces opérations, il n'y a plus qu'à complémenter chaque $C(i)$ pour obtenir dans le vecteur C le résultat escompté.

6.3.6 Résultat pour notre exemple

Pour notre exemple, nous trouvons les ensembles de variables connues suivants :

$$\text{site } 0 : \begin{cases} E(0) & = & \{\mathtt{x},\mathtt{y}\} \\ E(1) & = & \{\mathtt{x},\mathtt{y}\} \end{cases} \qquad \text{site } 1 : \begin{cases} E(0) & = & \{\mathtt{x},\mathtt{y}\} \\ E(1) & = & \{\mathtt{x}\} \end{cases}$$

Dans l'état initial 0, toutes les variables sont connues, ce qui est cohérent avec ce que nous avons dit dans la section 6.3.1. Dans l'état 1, le site 0 est le *propriétaire* de `y` qui est donc naturellement dans $E(1)$. Nous avons en outre $\mathtt{x} \in E(1)$, ce qui signifie que le site 0 connaît également `x`. Quant au site 1, il ne connaît que la variable dont il est le *propriétaire*, à savoir `x`. Ce sont bien les résultats auxquels on pouvait s'attendre à la vue du code du programme.

6.4 Étape d'élimination locale

Nous avons calculé à la section 6.3 les ensembles de variables connues à chaque état de l'automate OC. Il ne reste plus alors qu'à parcourir le graphe d'actions de chaque état, avec ces ensembles de variables connues, pour éliminer tous les messages redondants.

Rappelons qu'au niveau d'un état de l'automate, le code est purement séquentiel et est représenté sous la forme d'un graphe d'actions orienté et sans circuit, avec des actions de deux types :

- les actions de contrôle : branchements binaires (`if`, `dsz` et `present`), ainsi que changements d'état (`goto`),
- les actions séquentielles : affectations (`:=`), émissions de sorties (`output`) et enfin émissions de messages valués (`put`) et non valués (`put_void`).

Pour chaque action, nous connaissons la liste des variables qu'elle modifie. Naturellement, un `put` modifie la variable émise sur le ou les sites destinataires. Il est par conséquent facile de calculer, sur un site donné, et à chaque instant, l'ensemble des variables - autres que celles dont il est le *propriétaire* - qu'il connaît. A partir de là, nous supprimons un `put` si et seulement si la variable émise est déjà connue par le site destinataire.

6.4.1 Algorithme d'élimination locale

A chaque site `s`, nous associons un ensemble $\texttt{Connues}_\texttt{s}$ contenant toutes les variables connues par le site `s`. Pour chaque site `s`, nous plaçons à la racine du graphe d'actions de l'état `p` un ensemble $\texttt{Connues}_\texttt{s}$, initialisé avec l'ensemble $E(\texttt{p})$ du site `s` calculé à l'étape précédente. Nous propageons alors ces ensembles jusqu'aux feuilles du graphe (propagation *en avant*) de la façon suivante :

- Quand on atteint une émission `put(s;x)`, on teste si $\texttt{x} \in \texttt{Connues}_\texttt{s}$; si c'est le cas on efface ce `put(s;x)`, et sinon on ajoute `x` à $\texttt{Connues}_\texttt{s}$.
- Quand on atteint une émission `put_void(s)`, on teste si $\texttt{void} \in \texttt{Connues}_\texttt{s}$; si c'est le cas on efface ce `put_void(s)`, et sinon on ajoute `void` à $\texttt{Connues}_\texttt{s}$.
- Quand on atteint une affectation `x:=exp`, on retire la variable `x` des ensembles $\texttt{Connues}_\texttt{s}$ pour tout site `s` distinct du site *propriétaire* de `x`.
- Quand on atteint un test `if`, `dsz` ou `present`, pour chaque site `s` on duplique l'ensemble $\texttt{Connues}_\texttt{s}$, et on continue l'algorithme dans les deux branches du test.
- Quand on atteint une fermeture de test, pour chaque site `s`, on fait l'intersection des deux ensembles $\texttt{Connues}_\texttt{s}$ provenant des deux branches du test, et on continue avec cette intersection ; en effet, on n'élimine un `put` que si on est certain que la variable est connue dans les *deux* branches du test.

6.4.2 Résultat pour notre exemple

Voici ce que donne l'exécution de l'algorithme de la section 6.4.1 sur l'exemple de la figure 6.1. Pour l'état 0 nous obtenons :

site	state 0	$\texttt{Connues}_0$	$\texttt{Connues}_1$
(1)	`x:=0;`	{x,y}	{x,y}
(1)	`put(0;x);`	{x,y}	{y}
(0)	`y:=x;`	{x,y}	{x,y}
(0)	`output(y);`	{x}	{x,y}
(0,1)	`goto 1;`	{x}	{x,y}

A l'instant où est effectué le `put(0;x)`, la variable `x` n'est pas dans l'ensemble $\texttt{Connues}_1$. Donc ce `put` n'est pas supprimé.

Et pour l'état 1 nous obtenons :

site	state 1	$\texttt{Connues}_0$	$\texttt{Connues}_1$
(1)	`put(0;x);`	{x,y}	{x}
(0)	`y:=x;`	{x,y}	{x}
(0)	`output(y);`	{x}	{x}
(0,1)	`goto 1;`	{x}	{x}

A l'instant où est effectué le `put(0;x)`, la variable `x` est dans l'ensemble $\texttt{Connues}_1$. Donc ce `put` est bel et bien supprimé.

6.5 Complexités et performances des algorithmes

Pour effectuer les calculs de complexité des algorithmes que nous venons de décrire, nous conservons les notations introduites dans les chapitres précédents.

6.5.1 Analyse statique globale

Les coûts en temps et en mémoire du calcul des coefficients du système linéaire (section 6.3.3) sont similaires à ceux du placement des `put` (section 3.4) :

$$\mathcal{T}(\text{calcul des coefficients}) = O(|Q| \times moy_{act})$$

$$\mathcal{M}(\text{calcul des coefficients}) = O(nb_{site} \times nb_{var} \times |Q|^2)$$

Les structures de données existant déjà, le coût en mémoire des substitutions dans le système linéaire (section 6.3.4) est négligeable. Quant au coût en temps, il vaut :

$$\mathcal{T}(\text{substitution}) = nb_{site} \times \sum_{i=0}^{|Q|-1} \sum_{j=i}^{|Q|-1} (\mathcal{T}(\text{opération 6.1}) + \mathcal{T}(\text{opération 6.2}))$$

Or $\mathcal{T}(\text{opération 6.1}) = \sum_{j'=i}^{|Q|-1} O(nb_{var}) = O((|Q| - i) \times nb_{var})$. De plus $\mathcal{T}(\text{opération 6.2}) = O(nb_{var})$. Nous en déduisons le temps de calcul des substitutions :

$$\mathcal{T}(\text{substitution}) = O(nb_{site} \times |Q|^3 \times nb_{var})$$

De même que pour la substitution, le coût en mémoire de la résolution du système linéaire (section 6.3.5) est négligeable. Quant au coût en temps, il vaut :

$$\mathcal{T}(\text{résolution}) = nb_{site} \times \sum_{j=|Q|-1}^{0} \sum_{k=j}^{0} \mathcal{T}(\text{opération 6.3})$$

Comme 6.3 est une simple opération sur des ensembles de variables, $\mathcal{T}(\text{opération 6.3}) = O(nb_{var})$. Donc :

$$\begin{aligned} \mathcal{T}(\text{résolution}) &= nb_{site} \times nb_{var} \times \sum_{j=|Q|-1}^{0} j \\ &= O(nb_{site} \times nb_{var} \times |Q|^2) \end{aligned}$$

Nous pouvons à présent calculer les coûts totaux de la phase d'analyse statique globale :

$$\begin{aligned} \mathcal{T}(\text{analyse globale}) &= \mathcal{T}(\text{calcul des coefficients}) + \mathcal{T}(\text{substitution}) + \mathcal{T}(\text{résolution}) \\ &= O(|Q| \times moy_{act}) + O(nb_{site} \times |Q|^3 \times nb_{var}) + O(nb_{site} \times nb_{var} \times |Q|^2) \\ &= O(nb_{site} \times nb_{var} \times |Q|^3) \\ \mathcal{M}(\text{analyse globale}) &= \mathcal{M}(\text{calcul des coefficients}) + \mathcal{M}(\text{substitution}) + \mathcal{M}(\text{résolution}) \\ &= O(nb_{site} \times nb_{var} \times |Q|^2) + O(1) + O(1) \\ &= O(nb_{site} \times nb_{var} \times |Q|^2) \end{aligned}$$

6.5.2 Elimination locale

Comme nous utilisons les structures de données de la phase d'analyse statique globale, le coût en mémoire de l'élimination locale est négligeable. Quant au coût en temps, c'est le temps de parcours du graphe de chaque état ($O(\sum_{q \in Q} nb_{act}(q))$) fois le coût des opérations sur les ensembles `Connues`. Nous rappelons que moy_{act} désigne le nombre moyen d'actions dans chaque état. Donc le coût en temps de la phase d'élimination locale est :

$$\mathcal{T}(\text{élimination locale}) = O(|Q| \times moy_{act} \times moy_{var})$$

6.5.3 Algorithme d'élimination

Par conséquent, les coûts en temps et en mémoire de l'élimination sont :

$$\begin{aligned} \mathcal{T}(\text{élimination}) &= \mathcal{T}(\text{analyse globale}) + \mathcal{T}(\text{élimination locale}) \\ &= O(nb_{site} \times nb_{var} \times |Q|^3) + O(|Q| \times moy_{act} \times moy_{var}) \\ \mathcal{M}(\text{élimination}) &= \mathcal{M}(\text{analyse globale}) + \mathcal{M}(\text{élimination locale}) \\ &= O(nb_{site} \times nb_{var} \times |Q|^2) + O(1) \end{aligned}$$

En définitive, les coûts théoriques totaux sont :

- $\boxed{\mathcal{T}(\text{élimination}) = O(nb_{site} \times nb_{var} \times |Q|^3 + |Q| \times moy_{act} \times moy_{var})}$
- $\boxed{\mathcal{M}(\text{élimination}) = O(nb_{site} \times nb_{var} \times |Q|^2)}$

Par conséquent, le coût en temps de l'algorithme d'élimination dépend de la structure de contrôle de l'automate OC. Cette structure dépend, pour un même programme synchrone, d'options de compilation choisies par l'utilisateur (voir la thèse de Raymond [52] dans le cas de LUSTRE).

Si l'automate OC a une grosse structure de contrôle, alors moy_{act} sera négligeable devant $|Q|$ et nous aurons : $\mathcal{T}(\text{élimination}) = O(nb_{site} \times nb_{var} \times |Q|^3)$. Si au contraire il a une petite structure de contrôle, alors $|Q|$ sera négligeable devant moy_{act} et nous aurons : $\mathcal{T}(\text{élimination}) = O(|Q| \times moy_{act} \times moy_{var})$.

6.5.4 Performances

Dans le cas du programme de tennis réparti sur deux sites (un pour chaque joueur), `oc2rep` engendre 24 messages (chacun constitué d'un `put` et d'un `get`). L'algorithme d'élimination des messages sans analyse globale des variables connues n'en supprime aucun, alors qu'avec analyse globale il en supprime 5. Nous avons donc, sur cet exemple, un gain de 20%, ce qui est loin d'être négligeable.

6.6 Conclusion

Notre algorithme de parallélisation peut, nous l'avons vu, générer des messages redondants, aussi bien valués que non valués (messages de synchronisation). Or un trop grand nombre de messages ralentit bien évidemment l'exécution d'un programme réparti, et peut ainsi compromettre le succès de la mise en œuvre d'un système réactif.

La solution que nous proposons s'appuie sur une phase préliminaire d'analyse statique globale des programmes OC et permet d'éliminer les messages redondants valués ou non. Cet algorithme intervient juste après le placement des émissions (section 3.4) et le placement des émissions de synchronisation (chapitre 5). Ainsi le placement des réceptions (section 3.5) s'effectue-t-il sur un programme dont les émissions sont optimisées : il ne place donc que le nombre nécessaire de réceptions.

Chapitre 7

Répartition minimale

Notre algorithme de parallélisation réplique entièrement le contrôle (voir section 3.3) sur chaque site. C'est ce que nous avons appelé la synchronisation *totale* des branchements. Le fait qu'un branchement soit exécuté par *tous* les sites alors qu'il ne dépend que d'*un* site implique un échange de messages entre le site propriétaire et chacun des autres sites. Nous étudions dans ce chapitre une méthode permettant d'aboutir à une synchronisation *minimale* des branchements.

Intuitivement, une fois projeté sur un site, un branchement peut avoir ses deux branches qui ne diffèrent plus par les actions qu'elles effectuent. Notre idée est donc de détecter et de réduire un tel branchement, une fois la répartition faite. Afin de réduire le plus possible de branchements, nous représentons les programmes OC sous la forme de systèmes de transitions étiquetées et nous distinguons les actions concrètes (visibles) des actions internes (invisibles). Ainsi en ne considérant que les actions visibles, nous pouvons établir un critère d'équivalence comportementale basé sur les actions observables des systèmes. Afin de déterminer si deux systèmes ont le même comportement observable, nous utilisons des techniques de bisimulation "à la volée".

Nous décomposons donc notre méthode en deux algorithmes :

- algorithme d'équivalence observationnelle entre deux systèmes de transitions étiquetées ;
- algorithme de réduction des tests chargé d'appeler la procédure d'équivalence observationnelle pour chaque test, et de supprimer les tests dont les deux branches sont observationnellement équivalentes.

L'originalité de notre démarche réside dans la définition de ces deux algorithmes par des systèmes de dérivation s'appliquant aux systèmes de transitions étiquetées représentant les programmes OC. Nous justifions ainsi formellement l'équivalence observationnelle en montrant la consistance d'une interprétation pour le système de dérivation. Puis nous déduisons des systèmes de dérivation la mise en œuvre des deux algorithmes dans un langage impératif.

Nous introduisons tout d'abord notre méthode avec des exemples simples. Puis nous présentons successivement la synchronisation minimale des branchements et l'équivalence observationnelle. Nous présentons ensuite la mise en œuvre de ces deux algorithmes. Dans les deux cas nous discutons les choix d'implémentation et les complexités théoriques. Nous résolvons alors le problème

de la redéterminisation d'un programme dont certains branchements ont été supprimés, et de la resynchronisation d'un programme réparti dont les composants n'ont pas tous la même structure de contrôle. Enfin nous récapitulons les étapes de la répartition minimale et nous concluons.

7.1 Principe de la réduction des tests

7.1.1 Motivations

La synchronisation totale des branchements (section 4.4.8) implique que tous les programmes répartis ont la même structure de contrôle, celle du programme initial. Dans le cas d'un programme réparti sur deux sites, il se peut qu'un des fragments ne fasse aucun calcul et se contente de suivre les changements d'état de l'autre fragment, à cause de la synchronisation imposée entre les deux. C'est ce qu'illustre la figure suivante :

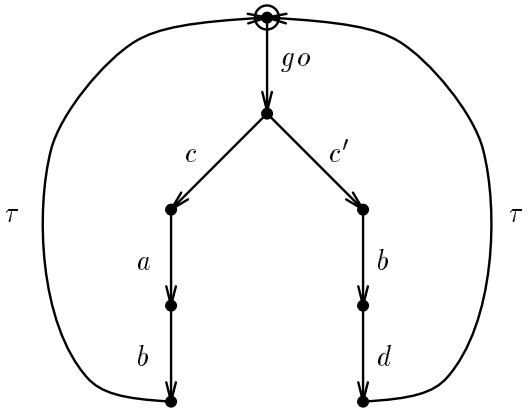


Figure 7.1: Exemple de système de transitions étiquetées

Nous choisissons de localiser les actions a et d sur le site 0, et l'action b sur le site 1, et nous supposons qu'il n'y a aucune relation de dépendance. Le passage des programmes OC aux *sted* (définition 4.3, page 53) nous amène à considérer deux types d'actions :

- les actions visibles : ce sont toutes les actions qui sont locales sur le site du programme considéré,
- les actions invisibles (que nous notons τ) : ce sont les actions qui n'appartiennent pas au site considéré plus les actions internes (changement d'état, fermetures de tests, ...).

La répartition du *sted* de la figure 7.1 sur deux sites donne :

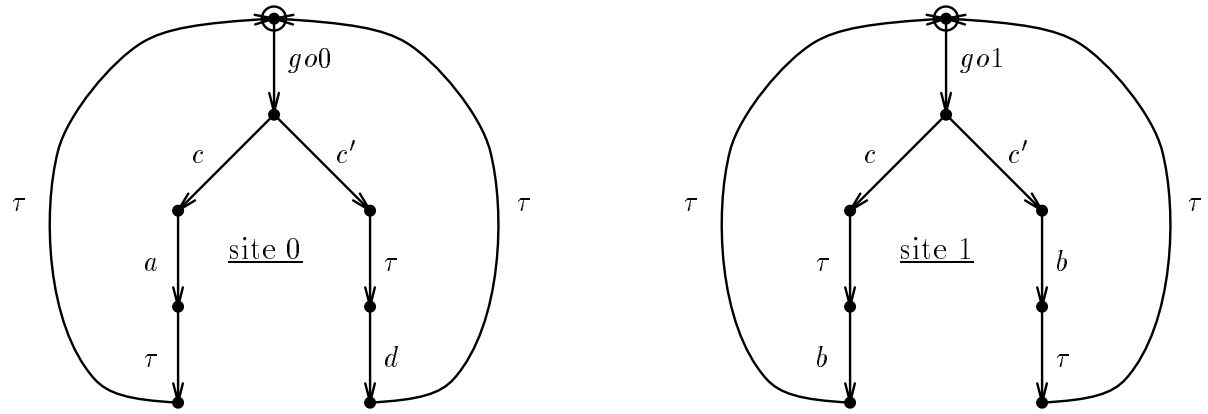


Figure 7.2: Système de la figure 7.1 réparti sur deux sites

Clairement, sur le site 1, le branchement est inutile : le branchement est de la forme $p = cp_1 + c'p_2$ et nous avons les comportements suivants :

- branche gauche du test : $[\![p_1]\!] = \tau.b.\tau.go1.[\![p]\!]$
- branche droite du test : $[\![p_2]\!] = b.\tau.\tau.go1.[\![p]\!]$

Donc si nous faisons abstraction des actions invisibles τ, nous avons $[\![p_1]\!] = [\![p_2]\!]$ et le programme du site 1 a par conséquent le même comportement que le programme suivant :

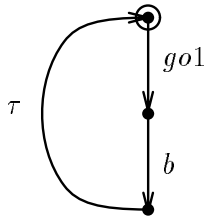


Figure 7.3: Système équivalent au programme de la figure 7.2, site 1

Faire abstraction des τ-actions dans les comportements revient à dire que les systèmes de transitions étiquetées p_1 et p_2 sont équivalents observationnellement, selon le critère d'équivalence observationnelle de CCS [43], ce que nous notons $p_1 \sim_o p_2$.

Nous étudions dans ce chapitre comment simplifier la structure de contrôle des programmes répartis, tout en préservant la sémantique du programme centralisé initial.

7.1.2 Principe

Nous partons du constat qu'une fois projeté sur un site, un branchement peut avoir ses deux branches qui ne diffèrent plus par les actions effectuées. De plus la synchronisation *totale* des branchements (section 4.4.8) impose que tout branchement soit synchronisé sur tous les sites. L'idée est donc de minimiser les programmes répartis *avant* de synchroniser les branchements. Par suite, l'opérateur de synchronisation des branchements ne synchronisera que les branchements non réduits.

L'optimisation que nous proposons consiste à supprimer les branchements dont les comportements des deux successeurs gauche et droit sont équivalents. C'est ce que nous appelons *synchroniser minimalement les branchements.*

La synchronisation minimale des branchements consiste donc, pour un site donné, à supprimer *a priori* tous les branchements. Pour cela nous les remplaçons momentanément par des τ-tests. Puis pour chacun d'entre eux, nous appliquons le critère d'équivalence observationnelle aux deux branches : si les deux branches ne sont pas équivalentes, alors nous remettons ce branchement, sinon nous le supprimons bel et bien. Par rapport à la synchronisation totale des branchements, les programmes répartis comporteront donc moins de tests.

Au départ, pour un système de transitions étiquetées q donné, nous obtenons par répartition fonctionnelle n systèmes $(q_i)_{i=1,n}$ tels que :

$$[\![q]\!] = [\![(q_1 [\![q_2]\!] \ldots [\![q_n)]\!]$$

Le but de la synchronisation minimale est de trouver n systèmes de transitions étiquetées $(q'_i)_{i=1,n}$ tels que :

$$\begin{array}{ccc} q_1 & \sim_o & q'_1 \\ q_2 & \sim_o & q'_2 \\ & \vdots & \\ q_n & \sim_o & q'_n \end{array}$$

Or notre opérateur de composition parallèle a pour effet de mettre en commun les actions identiques des deux côtés (définition 4.25 page 67). Cela correspond donc tout a fait a l'usage conjugué des opérateurs "|" et "\" de CCS, pour lesquels l'équivalence observationnelle est une congruence. Par conséquent nous pouvons remplacer chaque q_i par q'_i :

$$[\![q]\!] = [\![(q'_1 [\![q'_2]\!] \cdots [\![q'_n)]\!]$$

7.1.3 Remarque

L'automate de départ, et donc le système de transitions étiquetées qui le modélise, est minimal au sens du critère de bisimulation forte [21]. Pourtant il se peut qu'à cause de la répartition un branchement ne soit conservé sur aucun des sites. Nous en donnons ici un exemple :

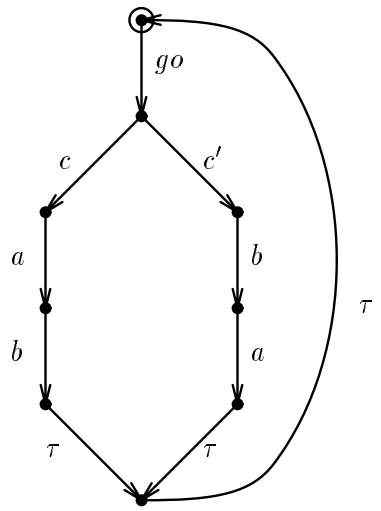


Figure 7.4: Système de transitions étiquetées minimal

En localisant l'action a sur le site 0, et les actions b, c et c' sur le site 1, nous obtenons les deux programmes répartis suivants :

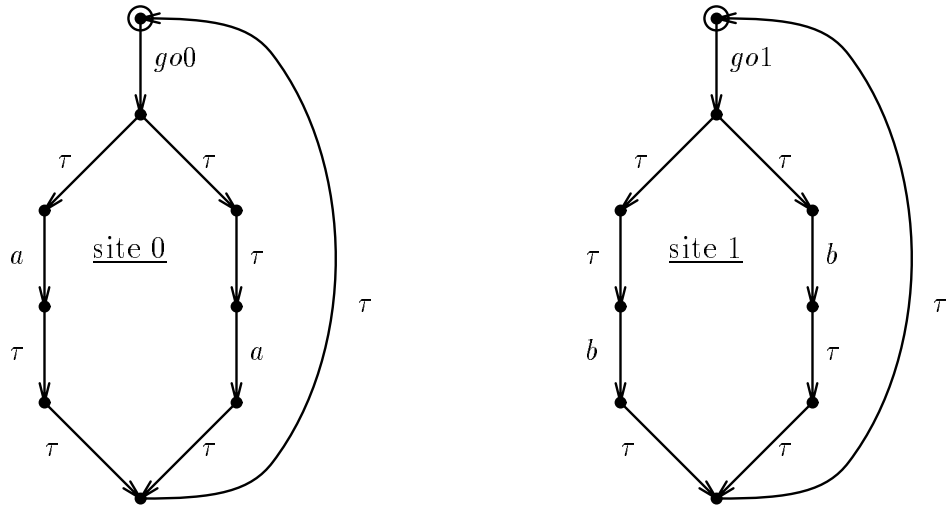


Figure 7.5: Système de la figure 7.4 réparti : $a\mathcal{C}b$

Il y a alors deux possibilités :

- Les actions a et b commutent (la relation de commutation $\mathcal{C}$ a été définie à la section 4.1.5). Dans ce cas l'algorithme de placement des émissions (section 3.4) n'insère aucun `put` dans les branches du test (voir la figure 7.5). Par conséquent sur chaque site les deux branches ont le même comportement et les deux tests peuvent être supprimés.

- Les actions a et b ne commutent pas.
  Dans ce cas l'algorithme de placement des émissions insère un `put`, représenté par l'étiquette $put(a,b)$ après le a de la branche gauche sur le site 1. De même il insère un $put(b,a)$ après le b de la branche droite sur le site 2 (voir la figure 7.6). Par conséquent sur chaque site les deux branches n'ont pas le même comportement et le test doit donc être conservé. Ceci est cohérent puisque, si a et b ne commutent pas, alors la séquence d'actions $a.b$ diffère de la séquence d'actions $b.a$, d'où l'utilité du test.

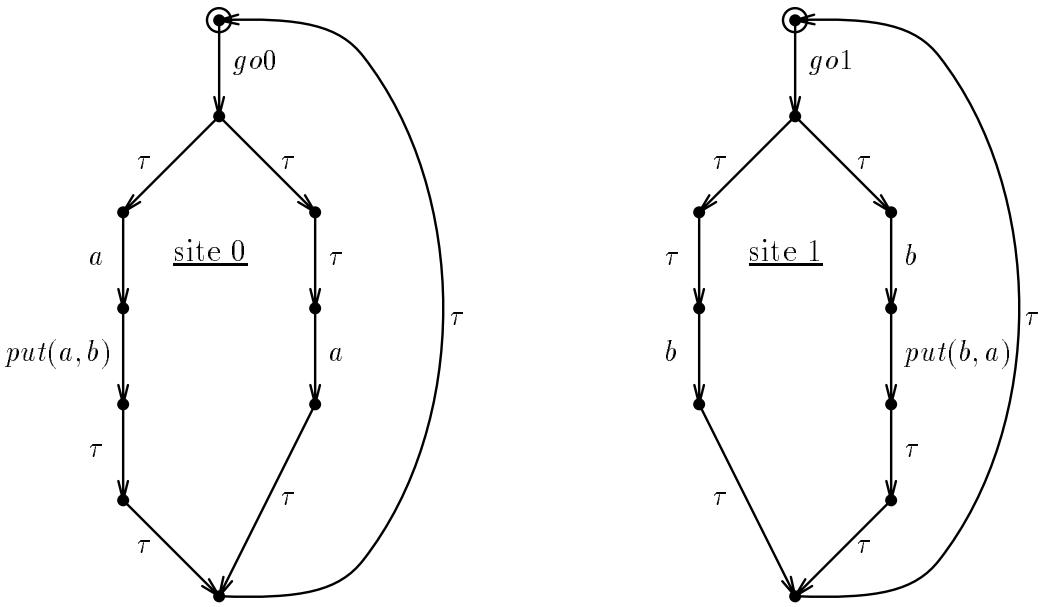


Figure 7.6: Système de la figure 7.4 réparti : $a\mathcal{D}b$

7.1.4 Enchaînement des étapes de la répartition

Dans le premier algorithme (chapitre 4), l'ordre dans lequel nous appliquons les deux opérateurs de synchronisation des traces et des branchements est indifférent. Dans le cas présent, il faut d'abord faire la synchronisation des traces, puis faire la synchronisation minimale des branchements. Considérons en effet l'exemple de la figure 7.6. Si nous appliquions la synchronisation minimale des branchements avant la synchronisation des traces, le *sted* ne comporterait pas les étiquettes $put(a,b)$ et $put(b,a)$, et nous supprimerions le test, alors que nous venons de voir qu'il ne devait pas l'être.

Au niveau de l'implémentation, il faut donc ne placer que les `put` correspondant aux actions autres que les branchements (section 3.4), puis placer les `put` pour les tests qu'on ne peut pas supprimer (chapitre présent). Après cela, nous plaçons les `put` de resynchronisation (chapitre 5), nous éliminons les `put` redondants (chapitre 6), et enfin nous plaçons les `get` (section 3.5). Ainsi, l'algorithme de placement des `get` ne doit être effectué qu'une seule fois.

7.2 Synchronisation minimale des branchements

Nous cherchons à ne synchroniser sur un site donné que les tests qui induisent des comportements différents selon la branche choisie. Comme nous l'avons vu, il faut d'abord supprimer tous les tests.

Définition 7.1 (suppression d'un test)
Pour tout branchement $cq + c'q'$, nous notons $(c, c')\tau q + \tau q'$ le résultat de sa suppression.

Un *sted* où nous avons supprimé des tests est alors un systèmes de transitions étiquetées non déterministe, que nous notons *sten* et définissons formellement de façon analogue aux *sted* :

Définition 7.2 (*trn*)
Un terme CCS régulier binaire non déterministe (*trn*) est défini par :

$$q ::= nil \mid x \mid aq \mid cq + c'q' \mid (c, c')\tau q + \tau q' \mid recx.q$$

Définition 7.3 (*sten*)
Un système de transitions étiquetées non déterministe fini binaire (*sten*) est un *trn* fermé (toutes les variables sont liées par un *rec*) et bien gardé (toutes les variables sont sous la portée d'une action : il n'y a pas de boucle vide telle que $recx.x$).

Nous décrivons la procédure de synchronisation minimale des branchements par un système de dérivation que nous appliquons au *sten* représentant le programmes OC à minimiser.

Les *sten* que nous traitons comportent bien évidemment des boucles. Afin de ne pas traiter plusieurs fois le même état, nous mémorisons dans un ensemble L chaque racine de boucle. Ces racines sont les états de la forme $recx.q$. Les états mémorisés dans L constituent les "hypothèses d'exécution". Par suite nous notons $L : q \rightarrow q'$ si et seulement si q' peut être dérivé de q sous les hypothèses L.

Nous définissons alors l'opérateur g de synchronisation minimale des branchements par :

Définition 7.4 (synchronisation minimale des branchements)
Pour tout système q, $g(q) = q'$ si et seulement si $\emptyset : q \rightarrow q'$.

Nous construisons ainsi des systèmes de transitions étiquetées bien synchronisés, notés *stes* et définis par :

Définition 7.5 (*stes*)
Si $q' = g(q)$, alors q' est un *stes*.

Informellement, les *stes* sont des systèmes où les branchements sont :

- soit de la forme $cq + c'q'$,
- soit de la forme $\tau q + \tau q'$ avec $q \sim_o q'$.

Intuitivement, pour synchroniser minimalement un système de transitions étiquetées, nous devons le parcourir et pour chaque branchement $(c, c')\tau q + \tau q'$ nous devons :

- le transformer en $cq + c'q'$, c'est-à-dire le récrire, si $q \not\sim_o q'$,
- le laisser sous la forme $(c, c')\tau q + \tau q'$, c'est-à-dire le supprimer, si $q \sim_o q'$.

Maintenant le fait de récrire un branchement dans une boucle du système de transition peut modifier le comportement des autres états de cette boucle, et donc remettre en cause des transformations (récritures ou suppressions) précédemment effectuées. Il est donc nécessaire de parcourir plusieurs fois chaque boucle, jusqu'à ce qu'il n'y ait plus aucune récriture à faire, c'est-à-dire jusqu'à ce que le système soit stable.

Le système de dérivation qui décrit l'opérateur g est composé de deux axiomes notés 7.1.A1 et 7.1.A2, et de six règles d'inférence notées 7.1.R1 à 7.1.R6. Ce sont les règles 7.1.R4 à 7.1.R6 qui forcent à parcourir chaque boucle du *sten* jusqu'à ce que celui-ci soit stable :

Système de dérivation 7.1 (synchronisation minimale des branchements)

- axiome 1 : nil se dérive toujours vers nil.
  $L : nil \rightarrow nil$
- axiome 2 : sous les hypothèses $\{q\}$, q dérive vers q.
  $L \cup \{q\} : q \rightarrow q$
- règle 1 : si q dérive vers q', alors aq dérive vers aq' sous les mêmes hypothèses.
  $$\frac{L : q \rightarrow q'}{L : aq \rightarrow aq'}$$
- règle 2 : si q et q' dérivent respectivement vers q_1 et q'_1 sous les mêmes hypothèses, et si q_1 est équivalent observationnellement à q'_1, alors $(c, c')\tau q + \tau q'$ dérive vers $(c, c')\tau q_1 + \tau q'_1$ toujours sous les mêmes hypothèses.
  $$\frac{L : q \rightarrow q_1 \;\; L : q' \rightarrow q'_1 \;\; q_1 \sim_o q'_1}{L : (c, c')\tau q + \tau q' \rightarrow (c, c')\tau q_1 + \tau q'_1}$$
- règle 3 : si q et q' dérivent respectivement vers q_1 et q'_1 sous les mêmes hypothèses, alors $(c, c')\tau q + \tau q'$ dérive vers $cq_1 + c'q'_1$ toujours sous les mêmes hypothèses.
  $$\frac{L : q \rightarrow q_1 \;\; L : q' \rightarrow q'_1}{L : (c, c')\tau q + \tau q' \rightarrow cq_1 + c'q'_1}$$
- règle 4 : nous ajoutons l'hypothèse $recx.q$ à L.
  $$\frac{L \cup \{recx.q\} : q[recx.q/x] \Rightarrow q'[recx.q'/x]}{L : recx.q \rightarrow recx.q'}$$
- règle 5 : nous inférons les hypothèses.
  $$\frac{L \cup \{recx.q\} : q[recx.q/x] \rightarrow q'[recx.q/x] \quad L \cup \{recx.q'\} : q'[recx.q'/x] \Rightarrow q''[recx.q''/x]}{L \cup \{recx.q\} : q[recx.q/x] \Rightarrow q''[recx.q''/x]}$$
- règle 6 : si $q[recx.q/x]$ dérive vers $q[recx.q/x]$, alors le système est stable.
  $$\frac{L \cup \{recx.q\} : q[recx.q/x] \rightarrow q[recx.q/x]}{L \cup \{recx.q\} : q[recx.q/x] \Rightarrow q[recx.q/x]}$$

Ainsi, le branchement $(c, c')\tau q_1 + \tau q_1'$ ne se récrit en $cq_1 + c'q_1'$ sur le site i que si $q_1 \not\sim_o q_1'$ sur le site i. Il ne sera donc pas synchronisé sur le site i dès que $q_1 \sim_o q_1'$. C'est en ce sens que nous parlons de synchronisation *minimale*.

Il nous reste à présent à étudier les points suivants : l'équivalence observationnelle $\sim_o$ de deux systèmes de transitions étiquetées, la mise en œuvre des algorithmes, et la redéterminisation des systèmes de transitions étiquetées synchronisés minimalement. C'est ce que nous voyons dans les sections suivantes.

7.3 Équivalence de comportement observable

Nous nous intéressons aux problèmes posés par l'équivalence comportementale de deux systèmes de transition étiquetés indéterministes. L'équivalence observationnelle est une relation de bisimulation ; nous utilisons les travaux de L.Mounier et J.C.Fernandez que l'on peut trouver dans [45] et [22]. Dans un premier temps, nous présentons les relations de bisimulation en général. Puis nous donnons l'algorithme que nous avons développé dans notre cas précis de réduction des tests, algorithme présenté sous la forme d'un système de dérivation. Nous justifions formellement que ce système met bien en œuvre l'équivalence de comportement de deux systèmes de transitions étiquetées. Enfin nous donnons l'implémentation de l'algorithme d'équivalence observationnelle à la section 7.5.

7.3.1 Relations de bisimulation

Nous proposons tout d'abord une définition intuitive de la sémantique de bisimulation, puis nous en donnons une définition plus formelle.

L'état d'un programme à un instant donné est entièrement déterminé, d'une part par l'ensemble des actions qui peuvent être effectuées à cet instant, et, d'autre part, par les états qui pourront être atteints après exécution de ces actions.

Deux programmes se trouvent alors dans des états identiques si et seulement si leurs comportements possibles à partir de ces états sont les mêmes : à toute évolution de l'un d'eux par une séquence d'actions donnée correspond une évolution de l'autre par la même séquence d'actions, telle qu'à chaque instant les comportements offerts par les états atteints lors de l'exécution de ces séquences soient à leur tour identiques.

La relation de bisimulation formalise cette égalité des comportements en l'exprimant aux niveaux des implémentations : deux systèmes de transitions étiquetées se bisimulent lorsqu'ils représentent des programmes ayant des comportements identiques. Nous définissons donc la bisimulation de façon récursive comme une relation d'équivalence sur l'ensemble des états des deux systèmes de transitions étiquetées :

> Deux états p et q se bisimulent si et seulement si pour tout successeur p' de p par l'action a, il existe un successeur q' de q par la même action a tel que p' et q' se bisimulent, et si pour tout successeur q' de q par l'action a, il existe un successeur p' de p par la même action a tel que p' et q' se bisimulent.

Cette relation est aussi appelée équivalence forte : elle définit "l'égalité" des comportements. Elle ne permet donc pas de prendre en compte les différences d'abstraction entre les programmes. Nous présentons d'autres relations plus faibles, en considérant non plus les actions concrètes du système mais ses actions abstraites, constituées d'un ensemble de séquences d'actions concrètes. La première utilisation de tels langages d'actions est due à D.Austry et G.Boudol dans [3].

Maintenant la bisimulation permet d'établir l'équivalence comportementale au sens des arbres générés. Dans la mesure où les systèmes de transitions que nous obtenons sont déterministes, l'équivalence des arbres est identique à l'équivalence des traces qui nous intéresse.

7.3.2 Bisimulation modulo un critère d'abstraction

Soient $S_i = \{Q_i,\ q_{0i},\ A_i,\ T_i\}_{(i=1,2)}$ deux systèmes de transitions étiquetées, où :

- Q_i est l'ensemble des états de S_i,
- q_{0i} est l'état initial de S_i,
- A_i est l'alphabet des actions de S_i,
- et T_i est la fonction de transition de S_i.

Soit de plus Λ un ensemble de langages disjoints sur $A \cup \{\tau\}$ et soit $\lambda \in \Lambda$. Nous définissons le successeur d'un état par la fonction de transition T et par le langage d'actions λ par :

Définition 7.6 (successeur par un langage d'actions)

$$p \xrightarrow{\lambda}_T q \iff \lambda \ni a_1 a_2 \ldots a_n \wedge \exists q_1, q_2, \ldots, q_{n-1} \in Q \text{ tels que } p \xrightarrow{a_1}_T q_1 \wedge q_1 \xrightarrow{a_2}_T q_2 \wedge \cdots \wedge q_{n-1} \xrightarrow{a_n}_T q$$

De plus, pour une fonction de transition T donnée, nous notons $T_a[p]$ l'ensemble des successeurs de p par l'action a, et par extension $T_\lambda[p]$ l'ensemble des successeurs de p par l'action abstraite λ, ainsi que $T_\Lambda[p]$ l'ensemble des successeurs de p par les actions de Λ :

Définition 7.7 (successeurs par une action abstraite)

- $T_a[p] = \{q \in Q | p \xrightarrow{a}_T q\}$
- $T_\lambda[p] = \{q \in Q | p \xrightarrow{\lambda}_T q\}$
- $T_\Lambda[p] = \{q \in Q | \exists \lambda \in \Lambda : p \xrightarrow{\lambda}_T q\}$

La relation de bisimulation est alors définie comme une famille de relations binaires sur $Q_1 \times Q_2$, paramétrée par Λ :

Définition 7.8 (famille de relations)
Soit l'opérateur $\mathcal{B}_\Lambda : 2^{Q_1 \times Q_2} \to 2^{Q_1 \times Q_2}$ défini par :

$$\mathcal{B}_\Lambda(R) = \{(p_1, p_2) | \forall \lambda \in \Lambda, \quad \forall q_1, (p_1 \xrightarrow{\lambda}_{T_1} q_1 \Rightarrow \exists q_2, (p_2 \xrightarrow{\lambda}_{T_2} q_2 \wedge (q_1, q_2) \in R)) \quad \forall q_2, (p_2 \xrightarrow{\lambda}_{T_2} q_2 \Rightarrow \exists q_1, (p_1 \xrightarrow{\lambda}_{T_1} q_1 \wedge (q_1, q_2) \in R))\}$$

Une relation R sur $Q_1 \times Q_2$ est une bisimulation si et seulement si $R \subseteq \mathcal{B}_\Lambda(R)$. De plus :

Définition 7.9 (équivalence de bisimulation)
L'équivalence de bisimulation modulo Λ est le plus grand point fixe de l'opérateur $\mathcal{B}_\Lambda$.

A partir de cette définition formelle, nous pouvons définir plusieurs équivalences de bisimulation selon divers critères d'abstraction. Par exemple l'équivalence forte correspond au langage suivant :

$$\Lambda_f = \{\{a\} | a \in A \cup \{\tau\}\}$$

Comme nous l'avons vu, elle définit l'égalité des comportements : les actions concrètes coïncident avec les actions abstraites. Pour comparer S_1 et S_2 modulo l'équivalence observationnelle introduite par R.Milner dans [43], il suffit de prendre le langage :

$$\Lambda_o = \{\tau^\star\} \cup \{\tau^\star a \tau^\star | a \in A\}$$

7.3.3 Algorithme de bisimulation

Dans [45], L.Mounier propose un algorithme général de bisimulation s'appliquant à deux systèmes de transitions étiquetées quelconques, $S_i = \{Q_i,\ q_{0i},\ A_i,\ T_i\}_{(i=1,2)}$, identifiés respectivement avec leur état initial q_{01} et q_{02}. Cet algorithme explore le produit synchrone $S_1 \otimes_\Lambda S_2$ (qui dépend du langage d'abstraction Λ), à partir de l'état initial (q_{01}, q_{02}). Dans le cas où S_1 ou S_2 sont déterministes, le critère pour que q_{01} et q_{02} se bisimulent est qu'il n'existe aucun état (p, q) atteignable depuis (q_{01}, q_{02}) et tel que p et q ne se bisimulent pas. Dans le cas général, le critère est similaire quoique plus complexe à cause de l'indéterminisme.

Nous pourrions dans le cas qui nous intéresse utiliser cet algorithme général de bisimulation, mais il faudrait pour cela l'appliquer *à chaque τ-test* du programme, avec à chaque fois un parcours du produit synchrone. C'est pourquoi nous proposons un algorithme plus efficace qui ne nécessite qu'un seul parcours. Cet algorithme repose sur le système d'axiomes et de règles de dérivation 7.2.

Le produit synchrone que nous parcourons comporte bien évidemment des boucles. Afin de ne pas traiter plusieurs fois le même état, nous mémorisons dans l'ensemble H chaque racine de boucle. Ces racines sont les états de la forme $(recx.q_1, q_2)$ ou $(q_1, recx.q_2)$. Les états mémorisés dans H constituent les hypothèses d'exécution. Par suite nous notons $H \xrightarrow{+} (q, q')$ si et seulement si q est bisimilaire à q' sous les hypothèses H.

Maintenant la détection des boucles dans le produit synchrone pose le problème des boucles de τ-transitions dans les systèmes de transitions que nous comparons.

Par exemple, soit le $sten\, q = (c, c')\tau.recx.(\tau.x) + \tau.a.nil$. Nous notons $q_1 = recx.(\tau.x)$ et $q_2 = a.nil$ les deux branches du test. Dans le produit synchrone, le couple (q_1, q_2) dérive par l'action τ vers lui-même et constitue par conséquent une racine de boucle. Mais il ne faudrait pour autant en déduire que $q_1 \sim_o q_2$, c'est-à-dire $\{(q_1, q_2)\} \xrightarrow{+} (q_1, q_2)$, car c'est manifestement faux !

Pour pallier ce problème, nous notons $H \xrightarrow{*} (q, q')$ (au lieu de $\xrightarrow{+}$) si et seulement si l'état (q, q') dérive des hypothèses par des τ-actions. Intuitivement, si (q, q') est dans les hypothèses H, alors

nous en déduisons que $H \xrightarrow{*} (q, q')$, puisque a priori q ou q' est une racine d'une boucle de τ-transitions. Si par la suite q et q' effectuent la même action *visible*, alors nous en déduisons que $H \xrightarrow{+} (q, q')$, puisque nous sommes alors certains que ni q ni q' ne sont la racine d'une boucle de τ-transitions :

Système de dérivation 7.2 (équivalence observationnelle)

- axiome 1 : $H \xrightarrow{+} (p, p)$
- axiome 2 : $H \cup (p, q) \xrightarrow{*} (p, q)$
- règle 1 : $\dfrac{H \xrightarrow{+} (p, q)}{H \xrightarrow{*} (p, q)}$
- règle 2 : $\forall a \neq \tau,\ \dfrac{H \xrightarrow{*} (p, q)}{H \xrightarrow{+} (ap, aq)}$
- règle 3 : $\dfrac{H \xrightarrow{+} (p, q)}{H \xrightarrow{+} (\tau p, q)}$ ainsi que la règle symétrique
- règle 4 : $\dfrac{H \xrightarrow{*} (p, q)}{H \xrightarrow{*} (\tau p, q)}$ ainsi que la règle symétrique
- règle 5 : $\forall c, c' \neq \tau,\ \dfrac{H \xrightarrow{*} (p, q)\ \ H \xrightarrow{*} (p', q')}{H \xrightarrow{+} (cp + c'p', cq + c'q')}$
- règle 6 : $\dfrac{H \xrightarrow{*} (p, p')\ \ H \xrightarrow{+} (p, q)}{H \xrightarrow{+} (\tau p + \tau p', q)}$ ainsi que la règle symétrique
- règle 7 : $\dfrac{H \xrightarrow{*} (p, p')\ \ H \xrightarrow{*} (p, q)}{H \xrightarrow{*} (\tau p + \tau p', q)}$ ainsi que la règle symétrique
- règle 8 : $\dfrac{H \cup (recx.p, q) \xrightarrow{+} (p[recx.p/x], q)}{H \xrightarrow{+} (recx.p, q)}$ ainsi que la règle symétrique

7.3.4 Algorithme d'auto-bisimulation

L'axiome 7.2.A2 tire parti du fait que les deux systèmes de transitions étiquetées que l'on compare ont la même fonction de transition et ne diffèrent que par leur état initial : d'où le terme *auto*-bisimulation.

Nous justifions le système 7.2 par la proposition suivante, dont on trouvera la preuve dans les sections 7.3.5 et 7.3.6 :

Proposition 7.1
$\emptyset \xrightarrow{+} (p, q) \Rightarrow p \sim_o q$.

7.3.5 Preuve de la consistance du système 7.2

Nous donnons une interprétation dont nous prouvons la consistance pour le système 7.2. Puis, dans la section 7.3.6, nous prouvons à l'aide de cette interprétation que ce système calcule bien l'équivalence observationnelle de deux systèmes de transitions étiquetées.

Nous notons pour abréger $\mathcal{F}$ l'opérateur d'équivalence observationnelle :

Définition 7.10 (opérateur $\mathcal{F}$)

$$\begin{aligned}\forall R,\ \mathcal{F}(R) &= \mathcal{B}_{\Lambda_o}(R)\\ &= \{(p,q), \forall \lambda \in \Lambda_o,\ \ \forall p', (p \xrightarrow{\lambda} p' \Rightarrow \exists q', (q \xrightarrow{\lambda} q' \wedge (p',q') \in R))\\ &\qquad\qquad \forall q', (q \xrightarrow{\lambda} q' \Rightarrow \exists p', (p \xrightarrow{\lambda} p' \wedge (p',q') \in R))\}\end{aligned}$$

Intuitivement, si $R \subseteq \mathcal{F}(R)$, alors $\mathcal{F}(R)$ contient les couples de systèmes de transitions étiquetées se bisimulant par toute action abstraite de Λ_o. Nous définissons en outre l'opérateur $\mathcal{G}$ tel que si $R \subseteq \mathcal{G}(R)$, alors $\mathcal{G}(R)$ contient les couples se bisimulant par toute action invisible :

Définition 7.11 (opérateur $\mathcal{G}$)

$$\begin{aligned}\forall R,\ \mathcal{G}(R) = &\ \{(\tau p_1, q_1), (p_1,q_1) \in R\}\\ &\cup \{(\tau p_1 + \tau p_2, q_1), (p_1,q_1) \in R \wedge (p_2,q_1) \in R\}\\ &\cup \{(\tau p_1 + \tau p_2, \tau q_1 + \tau q_2), (p_1,q_1) \in R \wedge (p_2,q_2) \in R\}\end{aligned}$$

Nous définissons à partir de là les opérateurs $\mathcal{G}^*$, $\mathcal{F}^*$ et $\mathcal{F}^+$:

Définition 7.12 (opérateurs $\mathcal{G}^*$, $\mathcal{F}^*$ et $\mathcal{F}^+$)

- $\mathcal{G}^*$ est le plus grand point fixe de l'opérateur $\lambda f.\mathcal{I} \cup \mathcal{G} \circ f$, $\mathcal{I}$ étant l'identité.
- $\mathcal{F}^*$ est le plus grand point fixe de l'opérateur $\lambda f.\mathcal{G}^* \cup \mathcal{F} \circ f$.
- $\mathcal{F}^+ = \mathcal{F} \circ \mathcal{F}^*$.

Nous notons alors R_H le plus grand point fixe de l'opérateur $\lambda r.\mathcal{F}^+(r \cup H)$, et nous définissons les règles d'interprétation suivantes :

Définition 7.13 (règles d'interprétation)

- $I(H \xrightarrow{+} (p,q)) = ((p,q) \in \mathcal{F}^+(R_H \cup H))$
- $I(H \xrightarrow{*} (p,q)) = ((p,q) \in \mathcal{F}^*(R_H \cup H))$

Nous montrons à présent que l'interprétation I est consistante pour le système 7.2 :

Axiome 7.2.A1 :

Il faut montrer que $(p,p) \in \mathcal{F}^+(R_H \cup H)$. Or $\mathcal{F}^+(R_H \cup H) = \mathcal{F}(\mathcal{F}^*(R_H \cup H))$. Comme p et p exécutent forcément les mêmes actions, le résultat découle trivialement des définitions de $\mathcal{F}$ et $\mathcal{F}^*$.

Axiome 7.2.A2 :

Il faut montrer que $(p,q) \in \mathcal{F}^*(R_H \cup H \cup (p,q))$. Par définition, $\mathcal{F}^*(R_H \cup H \cup (p,q)) \supseteq \mathcal{G}^*(R_H \cup H \cup (p,q)) \supseteq R_H \cup H \cup (p,q)$. Donc nous avons bien $(p,q) \in \mathcal{F}^*(R_H \cup H \cup (p,q))$.

Règle 7.2.R1 :

Il faut montrer que $(p,q) \in \mathcal{F}^+(R_H \cup H) \Rightarrow (p,q) \in \mathcal{F}^*(R_H \cup H)$. Or $(p,q) \in \mathcal{F}^+(R_H \cup H)$, ce qui implique $(p,q) \in R_H$ par définition de R_H. Comme $\mathcal{F}^*(R_H \cup H) \supseteq \mathcal{G}^*(R_H \cup H)$ et que $\mathcal{G}^*(R_H \cup H) \supseteq R_H \cup H$, nous obtenons immédiatement $(p,q) \in \mathcal{F}^*(R_H \cup H)$.

Règle 7.2.R2 :

Il faut montrer que $(p,q) \in \mathcal{F}^*(R_H \cup H) \Rightarrow (ap,aq) \in \mathcal{F}^+(R_H \cup H)$. Par définition,

$$\begin{aligned}\mathcal{F}^+(R_H \cup H) = \{(p_1,p_2), \forall \lambda \in \Lambda_o,\ & \forall q_1, (p_1 \xrightarrow{\lambda} q_1 \Rightarrow \exists q_2, (p_2 \xrightarrow{\lambda} q_2 \wedge (q_1,q_2) \in \mathcal{F}^*(R_H \cup H))) \\ & \forall q_2, (p_2 \xrightarrow{\lambda} q_2 \Rightarrow \exists q_1, (p_1 \xrightarrow{\lambda} q_1 \wedge (q_1,q_2) \in \mathcal{F}^*(R_H \cup H)))\}\end{aligned}$$

Or la seule action que peut effectuer ap est a : $ap \xrightarrow{a} p$. De même pour aq : $aq \xrightarrow{a} q$. Comme $(p,q) \in \mathcal{F}^*(R_H \cup H)$ par hypothèse, il est clair que $(ap,aq) \in \mathcal{F}^+(R_H \cup H)$.

Règle 7.2.R3 :

Il faut montrer que $(p,q) \in \mathcal{F}^+(R_H \cup H) \Rightarrow (\tau p,q) \in \mathcal{F}^+(R_H \cup H)$. Par définition de $\mathcal{F}^+(R_H \cup H)$, l'appartenance de (p,q) implique :

$$\forall \lambda \in \Lambda_o, \quad \forall p', \quad p \xrightarrow{\lambda} p' \Rightarrow \exists q', (q \xrightarrow{\lambda} q' \wedge (p',q') \in \mathcal{F}^*(R_H \cup H)) \tag{7.1}$$

$$\forall q', \quad q \xrightarrow{\lambda} q' \Rightarrow \exists p', (p \xrightarrow{\lambda} p' \wedge (p',q') \in \mathcal{F}^*(R_H \cup H)) \tag{7.2}$$

La seule action que peut effectuer τp est τ : $\tau p \xrightarrow{\tau} p$. Donc $\forall \lambda \in \Lambda_o, \forall p', \tau p \xrightarrow{\lambda} p'$ implique $\exists \lambda' \in \Lambda_o, \tau p \xrightarrow{\tau} p \xrightarrow{\lambda'} p'$. D'après l'équation (7.1), $\exists q', q \xrightarrow{\lambda'} q' \wedge (p',q') \in \mathcal{F}^*(R_H \cup H)$. De même, d'après l'équation (7.2), $\exists p', \tau p \xrightarrow{\tau.\lambda} p' \wedge (p',q') \in \mathcal{F}^*(R_H \cup H)$. Donc nous trouvons bien $(\tau p,q) \in \mathcal{F}(\mathcal{F}^*(R_H \cup H)) = \mathcal{F}^+(R_H \cup H)$.

Règle 7.2.R4 :

Il faut montrer que $(p,q) \in \mathcal{F}^*(R_H \cup H) \Rightarrow (\tau p,q) \in \mathcal{F}^*(R_H \cup H)$.

Par définition, $\mathcal{F}^*(R_H \cup H) = \mathcal{G}^*(R_H \cup H) \cup \mathcal{F} \circ \mathcal{F}^*(R_H \cup H)$. Nous pouvons écarter le cas où $(p,q) \in \mathcal{F} \circ \mathcal{F}^*(R_H \cup H)$ car cela implique $H \xrightarrow{+} (p,q)$. Nous en déduisons alors grâce aux règles 7.2.R3 et 7.2.R1 que $H \xrightarrow{*} (p,q)$.

Nous supposons donc $(p,q) \in \mathcal{G}^*(R_H \cup H)$. D'après la définition de $\mathcal{G}$, il est immédiat que $(\tau p,q) \in \mathcal{G}(\mathcal{G}^*(R_H \cup H))$. Or $\mathcal{G}^*(R_H \cup H) \supseteq \mathcal{G}(\mathcal{G}^*(R_H \cup H))$. Nous en déduisons donc que $(\tau p,q) \in \mathcal{F}^*(R_H \cup H)$.

Règle 7.2.R5 :

Il faut montrer que $(p,q) \in \mathcal{F}^*(R_H \cup H) \wedge (p',q') \in \mathcal{F}^*(R_H \cup H) \Rightarrow (cp + c'p', cq + c'q') \in \mathcal{F}^+(R_H \cup H)$.

$cp + c'p'$ ne peut effectuer que deux actions : c ou c'. Il en est de même pour $cq + c'q'$. Ou bien $cp + c'p' \xrightarrow{c} p$. Alors $\exists q, cq + c'q' \xrightarrow{c} q$ avec par hypothèse $(p, q) \in \mathcal{F}^*(R_H \cup H)$, et réciproquement. Ou bien $cp + c'p' \xrightarrow{c'} p'$. Alors $\exists q, cq + c'q' \xrightarrow{c'} q'$ avec par hypothèse $(p', q') \in \mathcal{F}^*(R_H \cup H)$, et réciproquement.

Par ailleurs $\mathcal{F}^+(R_H \cup H) = \mathcal{F}(\mathcal{F}^*(R_H \cup H))$. D'après la définition de $\mathcal{F}$, nous en déduisons $(cp + c'p', cq + c'q') \in \mathcal{F}^+(R_H \cup H)$.

Règle 7.2.R6 :

Il faut montrer que $(p, p') \in \mathcal{F}^*(R_H \cup H) \wedge (p, q) \in \mathcal{F}^+(R_H \cup H) \Rightarrow (\tau p + \tau p', q) \in \mathcal{F}^+(R_H \cup H)$. Contrairement aux autres preuves, celle-ci est contextuelle, c'est-à-dire qu'elle dépend de l'arbre de dérivation. Nous montrons donc tout d'abord que la règle 7.2.R6 n'est utilisée que s'il y a une boucle de τ-transitions sur la branche droite du branchement (resp. gauche pour la règle symétrique).

Construisons l'arbre de dérivation dans lequel s'inscrit cette règle 7.2.R6. D'une part les racines de la branche $H \xrightarrow{*} (p, p')$ sont forcément des axiomes 7.2.A2, et d'autre part il n'y a pas de dérivation par une action visible entre les hypothèses de ces axiomes 7.2.A2 et (p, p'), car sinon nous aurions $H \xrightarrow{+} (p, p')$. Les hypothèses servant aux axiomes 7.2.A2 sont déchargées plus bas dans l'arbre de preuve par des règles 7.2.R8. De même, entre la règle 7.2.R6 et ces règles 7.2.R8, il n'y a pas de dérivation par une action visible car sinon nous aurions à prouver $H \xrightarrow{*} (\tau p + \tau p', q)$ avec la règle 7.2.R7 au lieu de la règle 7.2.R6. Par suite, p' dérive vers $\tau p + \tau p'$ par τ^*.

Par hypothèse, $(p, q) \in \mathcal{F}^+(R_H \cup H)$. Comme $p' \xrightarrow{\tau^*} \tau p + \tau p'$, nous en déduisons par définition de $\mathcal{F}^+$ que $(\tau p + \tau p', q) \in \mathcal{F}^+(R_H \cup H)$.

Règle 7.2.R7 :

Il faut montrer que $(p, p') \in \mathcal{F}^*(R_H \cup H) \wedge (p, q) \in \mathcal{F}^*(R_H \cup H) \Rightarrow (\tau p + \tau p', q) \in \mathcal{F}^*(R_H \cup H)$.

Par définition, $\mathcal{F}^*(R_H \cup H) = \mathcal{G}^*(R_H \cup H) \cup \mathcal{F} \circ \mathcal{F}^*(R_H \cup H)$. Nous pouvons écarter le cas où $(p, q) \in \mathcal{F} \circ \mathcal{F}^*(R_H \cup H)$ car cela rentre dans le cadre de la règle 7.2.R6. De même nous pouvons écarter le cas où $(p, p') \in \mathcal{F} \circ \mathcal{F}^*(R_H \cup H)$ grâce à la règle 7.2.R2.

Nous supposons donc $(p, p') \in \mathcal{G}^*(R_H \cup H)$ ainsi que $(p, q) \in \mathcal{G}^*(R_H \cup H)$. Par définition de $\mathcal{G}$, il est alors immédiat que $(\tau p + \tau p', q) \in \mathcal{G}(\mathcal{G}^*(R_H \cup H))$. Comme $\mathcal{G}^* \supseteq \mathcal{G} \circ \mathcal{G}^*$ et $\mathcal{F}^* \supseteq \mathcal{G}^*$, nous en déduisons $(\tau p + \tau p', q) \in \mathcal{F}^*(R_H \cup H)$.

Règle 7.2.R8 :

Notons K l'hypothèse $(recx.p, q)$. Il faut montrer que $(p[recx.p/x], q) \in \mathcal{F}^+(R_{H \cup K} \cup H \cup K) \Rightarrow (recx.p, q) \in \mathcal{F}^+(R_{H \cup H})$. Il faut donc montrer que $K \subseteq \mathcal{F}^+(R_{H \cup K} \cup H \cup K) \Rightarrow K \in \mathcal{F}^+(R_{H \cup H})$.

Par définition, $R_{H \cup K}$ est le plus grand point fixe de l'opérateur $\lambda f.\mathcal{F}^+(f \cup H \cup K)$. Donc $R_{H \cup K}$ est le plus grand ensemble inclus dans $\mathcal{F}^+(R_{H \cup K} \cup H \cup K)$. Donc nécessairement $K \subseteq R_{H \cup K}$. Donc $R_{H \cup K} \cup H \cup K \subseteq R_{H \cup K} \cup H$ et donc $R_{H \cup K} \subseteq \mathcal{F}^+(R_{H \cup K} \cup H)$. Autrement dit, $R_{H \cup K}$ est un pré-point fixe de $\lambda f.\mathcal{F}^+(f \cup H)$.

Par définition de R_H, nous en déduisons $R_{H \cup K} \subseteq R_H = \mathcal{F}^+(R_H \cup H)$. Par conséquent, $K \in$

$\mathcal{F}^+(R_H \cup H)$.

7.3.6 Preuve de l'algorithme d'auto-bisimulation

Nous venons de prouver que l'algorithme 7.1 calcule les couples de systèmes de transitions étiquetées de $\mathcal{F}^+$:

$$\emptyset \xrightarrow{+} (p,q) \Rightarrow (p,q) \in \mathcal{F}^+(R_\emptyset)$$

Or $R_\emptyset$ est par définition un point fixe de $\mathcal{F}^+$. Nous montrons à présent que les points fixes de $\mathcal{F}$ et les points fixes de $\mathcal{F}^+$ coïncident.

Preuve :

Soit $R = \mathcal{F}(R)$. De la définition de $\mathcal{G}^*$, nous déduisons que $\mathcal{G}^* \circ \mathcal{F} = \mathcal{F}$, ainsi que $\mathcal{G}^* \circ \mathcal{F}^+ = \mathcal{F}^+$. Donc $R = \mathcal{G}^*(\mathcal{F}(R)) = \mathcal{G}^*(R)$. Trivialement, $R = R \cup R = \mathcal{G}^*(R) \cup \mathcal{F}(R) = \mathcal{F}^*(R)$. Nous composons avec $\mathcal{F}$, ce qui nous donne : $\mathcal{F}(R) = \mathcal{F}(\mathcal{F}^*(R)) = \mathcal{F}^+(R)$. Donc finalement, $R = \mathcal{F}(R) = \mathcal{F}^+(R)$.

Réciproquement, soit $R = \mathcal{F}^+(R)$. Donc $\mathcal{G}^*(R) = \mathcal{G}^*(\mathcal{F}^+(R)) = \mathcal{G}^* \circ \mathcal{F}^+(R) = \mathcal{F}^+(R) = R$. Trivialement, $R = R \cup R = \mathcal{G}^*(R) \cup \mathcal{F}^+(R) = \mathcal{F}^*(R)$. Nous composons par $\mathcal{F}$ et nous obtenons finalement : $\mathcal{F}(R) = \mathcal{F}(\mathcal{F}^*(R)) = \mathcal{F}^+(R) = R$.

Ceci nous permet donc de déduire que :

$$\emptyset \xrightarrow{+} (p,q) \Rightarrow (p,q) \in \mathcal{F}(R_\emptyset)$$

Or par définition, le plus grand point fixe de l'opérateur $\mathcal{F}$ est la relation de d'équivalence observationnelle (cf. section 7.3.5). Comme $R_\emptyset$ est un point fixe de $\mathcal{F}$, nous en déduisons la proposition 7.1.

7.3.7 Propagation des hypothèses

Il nous reste à corriger le système 7.1. Par définition de g, opérateur de synchronisation minimale des branchements, $g(q) = q'$ si et seulement si q' est bien synchronisé et se dérive à partir de q. Dans tous les axiomes et règles du système 7.1, nous propageons donc les hypothèses H servant à l'opérateur $\xrightarrow{+}$:

Système de dérivation 7.3 (synchronisation minimale des branchements)

- axiome 1 : $H, L : nil \rightarrow nil$
- axiome 2 : $H, L \cup \{q\} : q \rightarrow q$
- règle 1 : $\dfrac{H, L : q \rightarrow q'}{H, L : aq \rightarrow aq'}$

- règle 2 : elle est chargée d'appeler l'opérateur $\stackrel{+}{\rightarrow}$ avec les hypothèses H :
$$\frac{H, L : q \rightarrow q_1 \;\; H, L : q' \rightarrow q'_1 \;\; H \stackrel{+}{\rightarrow} (q_1, q'_1)}{H, L : (c, c')\tau q + \tau q' \rightarrow (c, c')\tau q_1 + \tau q'_1}$$
- règle 3 : $\dfrac{H, L : q \rightarrow q_1 \;\; H, L : q' \rightarrow q'_1}{H, L : (c, c')\tau q + \tau q' \rightarrow c q_1 + c' q'_1}$
- règle 4 : $\dfrac{H, L \cup \{recx.q\} : q[recx.q/x] \Rightarrow q'[recx.q'/x]}{H, L : recx.q \rightarrow recx.q'}$
- règle 5 :
$$\frac{H, L \cup \{recx.q\} : q[recx.q/x] \rightarrow q'[recx.q/x] \quad H, L \cup \{recx.q'\} : q'[recx.q'/x] \Rightarrow q''[recx.q''/x]}{H, L \cup \{recx.q\} : q[recx.q/x] \Rightarrow q''[recx.q''/x]}$$
- règle 6 : $\dfrac{H, L \cup \{recx.q\} : q[recx.q/x] \rightarrow q[recx.q/x]}{H, L \cup \{recx.q\} : q[recx.q/x] \Rightarrow q[recx.q/x]}$

Enfin, la nouvelle définition de l'opérateur g est :

Définition 7.14 (synchronisation minimale des branchements)
Pour tout système q, $g(q) = q'$ si et seulement si $\emptyset, \emptyset : q \rightarrow q'$.

7.4 Mise en œuvre de la synchronisation minimale

Nous donnons à présent une implémentation algorithmique de l'opérateur g. Nous considérons le système de transitions étiquetées $S = \{Q,\ q_0,\ A,\ T\}$ avec les notations habituelles. Nous rappelons que la synchronisation des traces a déjà été effectuée (cf. section 7.1.4).

Il s'agit d'implémenter le système de dérivation 7.3. C'est un parcours en profondeur du graphe représentant S. Au cours de ce parcours, il faut à chaque τ-test rencontré appeler la fonction *parcours_produit* permettant de déterminer si deux états d'un système de transitions étiquetées indéterministe (deux *sten*) se bisimulent ou pas. Ceci est en effet imposé par la règle 7.3.R2.

7.4.1 Principe

Il faut, pour mettre en œuvre ce parcours sans avoir une complexité en temps de calcul exponentielle, détecter les états du graphe bien synchronisés. Rappelons en effet que la représentation sous forme de *sten* des programmes est exponentielle puisqu'à chaque fermeture de test, l'état successeur est dupliqué (cf. section 4.1.4 page 54). De plus, pour mémoriser les dépendances induites par les boucles du graphe, nous associons à chaque état une liste L non redondante d'hypothèses. C'est la liste des hypothèses du système 7.3.

Pour implémenter cette liste d'hypothèses, il faut marquer les états du graphe avec les informations suivantes :

- "stack" : états du système qui sont dans la séquence d'exécution courante ;

- "root" : états du système qui constituent une racine de boucle.

Quand, au cours d'un parcours en descendant, un état a son successeur déjà "stack", alors nous sommes en présence d'une boucle : le successeur devient "root", et nous l'ajoutons à la liste L de l'état considéré. Par la suite, quand nous remontons, nous propageons les hypothèses, sauf si l'état sur lequel nous repassons est dans la liste des hypothèses propagée ; auquel cas nous l'enlevons de cette liste puisque c'est lui la racine de la boucle : autrement dit nous le déchargeons des hypothèses.

En définitive, le fait qu'un état ait une liste L non vide signifie qu'il est synchronisé *sous les hypothèses* L. De même, tout état dont la liste L est vide ou ne contient aucun état "stack" est forcément bien synchronisé car toutes les hypothèses ont été déchargées. Pour ne pas les confondre avec les états non encore visités, pour lesquels L est également vide, nous ajoutons l'indicateur suivant :

- "visited" : états du système complètement traités.

Un même état peut donc être "stack" et "root", "visited" et "root", mais pas "stack" et "visited".

Considérons par exemple les quatre étapes suivantes (ordonnées de haut en bas) de déroulement du parcours en profondeur d'un système de transitions étiquetées :

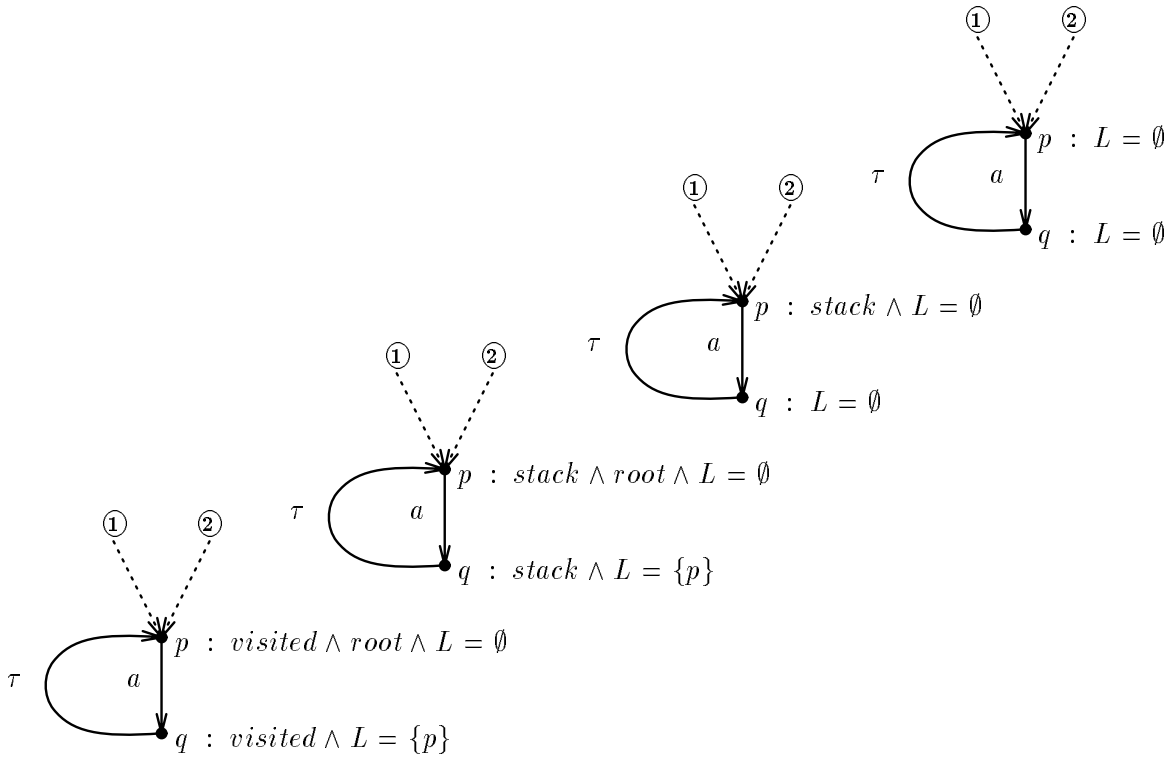


Figure 7.7: Exemple de parcours d'une boucle

Nous explicitons les étapes illustrées par la figure 7.7 :

- Au départ, aucun des deux états p et q n'est marqué.
- En arrivant à l'état p à partir de la branche ① : nous le marquons "stack".
- En arrivant à son successeur q : nous le marquons "stack", puis nous cherchons ses successeurs ; son seul successeur est l'état p qui est déjà "stack" ; donc nous marquons p "root" et nous ajoutons p à la liste des hypothèses de q.
- Nous remontons en propageant les hypothèses $\{p\}$; l'indicateur "stack" devient "visited" ; au moment où nous passons sur l'état p, nous le déchargeons des hypothèses à propager, et nous ajoutons les hypothèses restantes, ici $\emptyset$, aux hypothèses de p ; finalement la liste des hypothèses de l'état p est vide.
- Par la suite, en arrivant sur l'état p par la branche ②, nous interrompons le parcours puisque p est bien synchronisé.

7.4.2 Stabilité

Reste le problème de la stabilité : nous avons vu à la section 7.2 que les règles 7.3.R4 à 7.3.R6 forcent à parcourir toute boucle dans le système de transitions étiquetées plusieurs fois, jusqu'à ce que le système soit stable. C'est ce que nous faisons dès qu'il y a eu récriture d'un branchement. Pour détecter de telles récritures, nous introduisons l'indicateur *rewritten*, qui est passé en paramètre variable à la fonction *synchro_minimale*, et qu'elle positionne à **vrai** dès qu'elle opère une récriture.

7.4.3 Définitions préliminaires

La fonction *parcours_produit*, chargée de déterminer si deux états q_1 et q_2 du même système de transitions étiquetées se bisimulent ou pas (algorithme 7.2), est donnée dans la section 7.5. Elle retourne en résultat **vrai** si $q_1 \sim_o q_2$, et **faux** si $q_1 \not\sim_0 q_2$. Dans la mesure où elle est également susceptible de récrire un branchement, nous lui passons en paramètre variable l'indicateur *rewritten* qu'elle positionne à **vrai** à chaque récriture.

L'algorithme est donné sous une forme récursive. Le premier appel se fait par *synchro_minimale* $(q_0, rewritten)$, q_0 étant l'état initial de S, et *rewritten* un booléen valant **faux** au départ. Le résultat renvoyé est la liste des hypothèses qui doivent être rattachées à l'état q_0.

Pour tout état, nous accédons aux informations "stack", "root" et L qui lui sont rattachées au moyen respectivement des fonctions *stack*, *root* et *hypotheses*.

D'autre part, la fonction *synchronized* permet de détecter si un état est bien synchronisé, c'est-à-dire qu'il est "visited" et qu'aucun des états de L n'est "stack".

Nous introduisons les fonctions de base suivantes, valables pour tout système de transitions étiquetées S :

Définition 7.15 (représentation dynamique de S)

- Pour tout branchement p, $succ\_g\,(p)$ retourne la branche gauche de p.
- Pour tout branchement p, $succ\_d\,(p)$ retourne la branche droite de p.
- Pour tout état p qui n'est pas un branchement p, $succ\_g(p)$ retourne le successeur immédiat de p.
- Pour tout p, $act(p) = \{a \in A | \exists q \in Q : p \xrightarrow{a} q\}$.
- Pour tout p, $succ(p) = \{q \in Q | \exists a \in A : p \xrightarrow{a} q\}$. Ainsi $|succ\,(p)|$ désigne le nombre des successeurs immédiats de p.
- Pour tout p, $succ_{\tau^\star}(p) = \{p' | p \xrightarrow{\tau^\star} p'\}$
- Pour tout p, $act_{\tau^\star a}(p) = \{act(p') | p' \in succ_{\tau^\star}(p)\}$
- Pour tout p, $succ_{\tau^\star a}(p) = \{(a, p'') \in succ(p') | p' \in succ_{\tau^\star}(p) \wedge a \neq \tau\}$

La fonction *parcours_produit* met à jour les deux ensembles $Good$ et $Wrong$ contenant respectivement les couples d'états bisimilaires et les couples non-bisimilaires. Par construction, $Good \cap Wrong = \emptyset$. Nous pouvons donc expliciter les fonctions de caractérisation des branchements en fonction de ces ensembles :

Définition 7.16 (caractérisation des branchements)

- $tau\_test\,(p) = (|succ\,(p)| = 2 \wedge act\,(p) = \{\tau\} \wedge (succ\_g\,(p), succ\_d\,(p)) \notin Wrong \cup Good)$
- $true\_test\,(p) = (|succ\,(p)| = 2 \wedge act\,(p) = \{\tau\} \wedge (succ\_g\,(p), succ\_d\,(p)) \in Wrong)$
  $\vee\ (|succ\,(p)| = 2 \wedge act\,(p) \neq \{\tau\})$
- $reduced\_test\,(p) = (|succ\,(p)| = 2 \wedge act\,(p) = \{\tau\} \wedge (succ\_g\,(p), succ\_d\,(p)) \in Good)$
- $unreduced\_test\,(p) = (|succ\,(p)| = 2 \wedge act\,(p) = \{\tau\} \wedge (succ\_g\,(p), succ\_d\,(p)) \in Wrong)$

7.4.4 L'algorithme de synchronisation minimale

L'algorithme de synchronisation minimale des branchements est donc :

Algorithme 7.1

fonction *synchro_minimale* $(q_0, rewritten)$;
debut
  si $(synchronized\,(q_0))$ **alors**
    (* l'état initial est déjà bien synchronisé : l'algorithme est fini *)
    retourner $(\emptyset)$;
  fsi
  $L := hypotheses\,(q_0)$;
  $S := succ\,(q_0)$;
  $stack\,(q_0) :=$ **vrai** ;

```
(* premier cas : aucun successeur *)
si (|S| = 0) alors
      (* il n'y a rien à faire *)
fsi

(* deuxième cas : une transition simple *)
si (|S| = 1) alors
      q1 := succ_g (q0) ;
      si (¬synchronized (q1)) alors
            si (stack (q1)) alors
                  root (q1) := vrai ;
                  L := L ∪ {q1} ;
            sinon
                  rewritten1 := faux ;
                  L' := synchro_minimale (q1, rewritten1) ;          ①
                  si (root (q0)) alors
                        tantque (rewritten1) faire
                              rewritten1 := faux ;
                              L' := synchro_minimale (q1, rewritten1) ;          ②
                        ftantque
                  sinon
                        rewritten := rewritten1 ;
                  fsi
                  L := L ∪ L' ;
            fsi
      fsi
      L := L − {q0} ;
fsi

(* troisième cas : un branchement binaire *)
si (|S| = 2) alors
      q1 := succ_g (q0) ;
      si (¬synchronized (q1)) alors
            si (stack (q1)) alors
                  root (q1) := vrai ;
                  L := L ∪ {q1} ;
            sinon
                  rewritten1 := faux ;
                  L' := synchro_minimale (q1, rewritten1) ;
                  si (root (q0)) alors
                        tantque (rewritten1) faire
                              rewritten1 := faux ;
                              L' := synchro_minimale (q1, rewritten1) ;
                        ftantque
                  sinon
                        rewritten := rewritten1 ;
                  fsi
                  L := L ∪ L' ;
            fsi
      fsi
      q2 := succ_d (q0) ;
      si (¬synchronized (q2)) alors
            si (stack (q2)) alors
```

```
                    root (q2) := vrai ;
                    L := L ∪ {q2} ;
                sinon
                    rewritten2 := faux ;
                    L'' := synchro_minimale (q2, rewritten2) ;
                    si (root (q0)) alors
                        tantque (rewritten2) faire
                            rewritten2 := faux ;
                            L' := synchro_minimale (q2, rewritten2) ;
                        ftantque
                    sinon
                        rewritten := rewritten2 ;
                    fsi
                    L := L ∪ L'' ;
                fsi
            fsi
            si (tau_test (q0)) alors
                rewritten3 := faux ;
                bisimul := parcours_produit ((q1, q2), rewritten3) ;        ③
                si (¬bisimul) alors
                    récrire (c, c')τq1 + τq2 en cq1 + c'q2 ;
                fsi
                rewritten := ¬bisimul ou rewritten3 ;
            fsi
            L := L − {q0} ;
        fsi

        visited (q0) := vrai ;        ④
        stack (q0) := faux ;
        retourner (L) ;
fin
```

7.4.5 Complexité

Pour les calculs de complexité en temps et en mémoire, nous adoptons les notations suivantes :

- n est le nombre d'états de S,
- n_r est le nombre de boucles de S, et
- n_τ est le nombre de τ-tests de S.

Nous rappelons que, pour toute fonction f, nous notons $\mathcal{T}(f)$ et $\mathcal{M}(f)$ ses coûts de calcul en temps et en mémoire.

Complexité en temps : D'une part tous les états visités sont marqués en ④. Et d'autre part chaque état non visité donne lieu à un premier appel récursif, puis éventuellement à d'autres appels si il y a eu des récriture, respectivement en ① et ② dans le cas d'une transition simple. Il y a n_τ branchements donc tous les états sont visités au pire n_τ fois. Pour le traitement de chaque état, les seules opérations non-constantes sont l'union des listes d'hypothèses et la

suppression d'une hypothèse. L'union est en $O(n_r^2)$, puisque la taille maximale de ces listes est $O(n_r)$ et qu'à chaque insertion il faut s'assurer auparavant de la non appartenance. La suppression est en $O(n_r)$. A cela il faut rajouter le coût des appels à la fonction *parcours_produit* en ③, sachant qu'il y en a au pire n_τ. Donc le coût du parcours est $\mathcal{T}(synchro\_minimale) = O(n \times n_\tau \times max(n_r^2, n_r)) + n_\tau \times \mathcal{T}(parcours\_produit)$.

Complexité en mémoire : A chaque état de S, nous attachons trois marqueurs plus une liste d'hypothèses dont la taille est n_r. Donc le surcoût occasionné par rapport à la place nécessaire pour mémoriser l'automate OC est $O(n \times (3 + n_r)) = O(n \times n_r)$, à quoi il faut ajouter le coût en mémoire de la fonction *parcours_produit*.

En résumé, nous avons :

- $\mathcal{T}(synchro\_minimale) = O(n \times n_\tau \times n_r^2) + n_\tau \times \mathcal{T}(parcours\_produit)$
- $\mathcal{M}(synchro\_minimale) = O(n \times n_r) + \mathcal{M}(parcours\_produit)$

7.5 Mise en œuvre de l'auto-bisimulation

Il s'agit à présent d'implémenter le système 7.2. Nous travaillons toujours sur la représentation $S = \{Q, q_0, A, T\}$ du programme. Il faut effectuer un parcours en profondeur du produit synchrone $S \otimes S$ avec comme état initial le couple donné par la règle 7.2.R2.

7.5.1 Principe

Pour éviter un coût en temps de calcul exponentiel, l'idéal serait de procéder comme pour l'algorithme 7.1 en marquant les états du produit synchrone. Mais cela nous obligerait à construire en entier le produit synchrone $S \otimes S$, ce qui poserait des problèmes d'explosion de la taille mémoire. La solution consiste alors à construire le produit synchrone "à la volée" et à mémoriser les informations sur les états dans des ensembles séparés. C'est cette solution qu'avait retenu L.Mounier pour implémenter ses relations de bisimulations dans [45].

7.5.2 Définitions préliminaires

Au cours d'une exécution de la fonction *synchro_minimale*, nous aurons plusieurs appels à la fonction *parcours_produit*. Aussi entre deux appels successifs, nous conservons les couples d'états certainement non bisimilaires ainsi que les couples d'états certainement bisimilaires. Nous utilisons donc les structures de données suivantes :

- (q_{01}, q_{02}) est l'état initial du produit synchrone.
- *rewritten* est l'indicateur de récriture des branchements (cf. algorithme 7.1).
- *StState* est la pile des états de la séquence d'exécution courante du produit synchrone.

- $StTrans$ est la pile des transitions restant à examiner à partir de chacun des états de $StState$.
- $StNsucc$ est la pile des nombres de successeurs de chacun des états de $StState$.
- $StNgood$ est la pile des nombres de successeurs de chacun des états de $StState$ non présents dans $Good$.
- $StNwrong$ est la pile des nombres de successeurs de chacun des états de $StState$ non présents dans $Wrong$.
- $Good$ est l'ensemble rémanent des couples d'états bisimilaires : ce sont les états (p, q) tels que $\emptyset \xrightarrow{+} (p, q)$.
- $Wrong$ est l'ensemble rémanent des couples d'états non bisimilaires : ce sont tous les états du produit synchrone qui sont dans une séquence élémentaire menant à l'état $fail$. Nous représentons arbitrairement par $fail$ tout état (p, q) tel que p et q ne se bisimulent pas. Par construction, $Good \cap Wrong = \emptyset$.
- $Visited$ est l'ensemble non rémanent des états déjà visités précédemment au cours de l'exécution courante, et qui ne sont pas déjà dans $Good$ ou dans $Wrong$. A chaque état de $Visited$ correspond une liste $hypotheses$ qui sert à mémoriser les dépendances induites par les boucles du produit synchrone.

Les deux piles $StNgood$ et $StNwrong$ permettent de propager l'information sur l'équivalence ou la non-équivalence de façon sûre. Nous manipulons ces structures grâce aux primitives habituelles ($empiler$, $sommet$, $longueur$ ...).

Pour accéder aux successeurs d'un état dans le produit synchrone, nous définissons la fonction $succ\_produit$ dont nous donnons l'algorithme à la section 7.5.4.

7.5.3 L'algorithme d'auto-bisimulation

Algorithme 7.2

fonction $parcours\_produit\ ((q_{01}, q_{02}), rewritten)$;
debut
  $StState := \emptyset$; $StTrans := \emptyset$; $StNsucc := \emptyset$; $StNgood := \emptyset$; $StNwrong := \emptyset$; $Visited := \emptyset$;
  si $((q_{01}, q_{02}) \in Wrong)$ **alors**
    retourner (**faux**) ;
  fsi
  si $((q_{01}, q_{02}) \in Good)$ **alors**
    retourner (**vrai**) ;
  fsi
  $L := succ\_produit\ ((q_{01}, q_{02}))$;
  si $(L = \{fail\})$ **alors**
    $Wrong := Wrong \cup \{(q_{01}, q_{02})\}$;
    retourner (**faux**) ;
  fsi

$empiler\ (1, StNsucc)$;
$empiler\ (1, StNgood)$;
$empiler\ (1, StNwrong)$;
$empiler\ ((q_{01}, q_{02}), StState)$;
$empiler\ (L, StTrans)$;
$empiler\ (longueur\ (L), StNsucc)$;
$empiler\ (longueur\ (L), StNgood)$;
$empiler\ (longueur\ (L), StNwrong)$;
tantque $(StState \neq \emptyset)$ **faire**
    $(q_1, q_2) := sommet\ (StState)$;
    $H := hypotheses\ ((q_1, q_2))$; ⓪
    si $(sommet\ (StTrans) \neq \emptyset)$ **alors**
        (* il reste des successeurs de (q_1, q_2) à visiter *)
        choisir (q'_1, q'_2) dans $sommet\ (StTrans)$;
        $sommet\ (StTrans) := sommet\ (StTrans) - \{(q'_1, q'_2)\}$;
        si $((q'_1, q'_2) \in Good$ **ou** $q'_1 = q'_2)$ **alors**
            $sommet\ (StNgood) := sommet\ (StNgood)\ -\ 1$; ①
            $sommet\ (StNsucc) := sommet\ (StNsucc)\ -\ 1$;
        sinon si $((q'_1, q'_2) \in Wrong)$ **alors**
            $sommet\ (StNwrong) := sommet\ (StNwrong)\ -\ 1$; ②
        sinon si $((q'_1, q'_2) \in StState)$ **alors**
            (* (q'_1, q'_2) est la racine d'une boucle *)
            $H := H \cup \{(q'_1, q'_2)\}$; ③
            $sommet\ (StNsucc) := sommet\ (StNsucc)\ -\ 1$;
        sinon si $((q'_1, q'_2) \in Visited)$ **alors**
            (* (q'_1, q'_2) a déjà été traité précédemment *)
            $H' := explore\ (hypotheses\ ((q'_1, q'_2)))$; ④
            si $(H' = \{wrong\})$ **alors**
                $Wrong := Wrong \cup \{(q'_1, q'_2)\}$;
                $sommet\ (StNwrong) := sommet\ (StNwrong)\ -\ 1$;
            sinon si $(H' = \{good\})$ **alors**
                $Good := Good \cup \{(q'_1, q'_2)\}$;
                $sommet\ (StNsucc) := sommet\ (StNsucc)\ -\ 1$;
                $sommet\ (StNgood) := sommet\ (StNgood)\ -\ 1$;
            sinon
                $H := H \cup hypotheses\ ((q'_1, q'_2))$;
                $sommet\ (StNsucc) := sommet\ (StNsucc)\ -\ 1$;
            fsi
        sinon (* $(q'_1, q'_2) \notin Good \cup Wrong \cup StState \cup Visited$ *)
            (* (q'_1, q'_2) est un nouveau successeur *)
            $L' := succ\_produit\ ((q'_1, q'_2))$;
            si $(L' = \{fail\})$ **alors**
                $Wrong := Wrong \cup \{(q'_1, q'_2)\}$; ⑤
                $sommet\ (StNwrong) := sommet\ (StNwrong)\ -\ 1$;
            sinon
                $empiler\ ((q'_1, q'_2), StState)$; ⑥
                $empiler\ (L', StTrans)$;
                $empiler\ (longueur\ (L'), StNsucc)$;
                $empiler\ (longueur\ (L'), StNgood)$;
                $empiler\ (longueur\ (L'), StNwrong)$;
            fsi
        fsi
    sinon (* $sommet\ (StTrans) = \emptyset$ *)

(* tous les successeurs de (q_1, q_2) ont été visités *)
si $(status\ (sommet\ (StTrans)) = ok)$ **alors** ⑦
  (* ni q_1 ni q_2 n'était un τ-test *)
  $Visited := Visited \cup \{(q_1, q_2)\}$;
  $H := H - \{(q_1, q_2)\}$; ⑧
  si $(sommet\ (StNsucc) = 0)$ **alors**
    (* tous les successeurs de (q_1, q_2) se bisimulent *)
    $depiler\ (StState, StTrans, StNsucc, StNwrong)$;
    $sommet\ (StNsucc) := sommet\ (StNsucc) - 1$;
    si $(sommet\ (StNgood) = 0$ **ou** $H = \emptyset)$ **alors**
      (* nous sommes sûrs que $q_1 \sim_o q_2$ *)
      $depiler\ (StNgood)$;
      $sommet\ (StNgood) := sommet\ (StNgood) - 1$;
      $Good := Good \cup \{(q_1, q_2)\}$;
    fsi
  sinon
    (* tous les successeurs de (q_1, q_2) ne se bisimulent pas *)
    $depiler\ (StState, StTrans, StNsucc, StNgood)$;
    si $(sommet\ (StNwrong) = 0)$ **alors**
      (* nous sommes sûrs que $q_1 \not\sim_o q_2$ *)
      $depiler\ (StNwrong)$;
      $sommet\ (StNwrong) := sommet\ (StNwrong) - 1$;
      $Wrong := Wrong \cup \{(q_1, q_2)\}$;
    fsi
  fsi
sinon (* $status\ (sommet\ (StTrans)) = tau$ *)
  (* q_1 ou q_2 était un τ-test *)
  si $(unreduced\_test\ (q_1))$ **alors**
    $(q_{11}, q_{12}) = (succ\_g(q_1), succ\_d(q_1))$;
    récrire $(c, c')\tau q_{11} + \tau q_{12}$ en $cq_{11} + c'q_{12}$;
    $rewritten :=$ **vrai** ;
  fsi
  si $(unreduced\_test\ (q_2))$ **alors**
    $(q_{21}, q_{22}) = (succ\_g(q_2), succ\_d(q_2))$;
    récrire $(c, c')\tau q_{21} + \tau q_{22}$ en $cq_{21} + c'q_{22}$;
    $rewritten :=$ **vrai** ;
  fsi
  $L'' := succ\_produit\ ((q_1, q_2))$; ⑨
  si $(L'' = \{fail\})$ **alors**
    $depiler\ (StState, StTrans, StNsucc, StNgood, StNwrong)$;
    $Wrong := Wrong \cup \{(q_1, q_2)\}$;
  sinon
    $depiler\ (StTrans, StNsucc, StNgood, StNwrong)$;
    $empiler\ (L'', StTrans)$;
    $empiler\ (longueur\ (L''), StNsucc)$;
    $empiler\ (longueur\ (L''), StNgood)$;
    $empiler\ (longueur\ (L''), StNwrong)$;
  fsi
fsi
fsi
ftantque

si $((sommet\ (StNwrong) = 0)$ **ou** $(sommet\ (StNsucc) \neq 0))$ **alors**
retourner (**faux**) ;
sinon si $((sommet\ (StNgood) = 0)$ **ou** $(sommet\ (StNsucc) = 0))$ **alors**
retourner (**vrai**) ;
fsi
fin

7.5.4 La fonction *succ_produit*

Cette fonction calcule pour tout couple (p, q) la liste de ses successeurs immédiats dans le produit synchrone.

Au cas où p ou q soit un τ-test de la forme $\tau q_1 + \tau q_2$, les règles 7.2.R6 et 7.2.R7 imposent de d'abord se poser la question $q_1 \overset{?}{\sim}_o q_2$, avant de conclure sur (p, q). Ceci est réalisé par l'intermédiaire de la fonction *succ_produit* qui attache l'indicateur *tau* à sa liste résultat si un des deux arguments est un τ-test, et l'indicateur *ok* dans le cas contraire. La valeur de cet indicateur peut par la suite être testée par la fonction *status*. C'est ce que nous faisons en ⑦ : si le résultat est différent de *ok*, alors éventuellement nous récrivons le τ-test $(c, c')\tau q_1 + \tau q_2$ en $cq_1 + c'q_2$. Puis nous appelons à nouveau la fonction *succ_produit* en ⑨ et selon le résultat nous empilons ou non la nouvelle liste de successeurs, mais en conservant l'état courant de la pile *StState*.

La fonction *succ_produit* explore donc de façon exhaustive tous les cas de figure possibles. Si aucun de ces cas n'est rencontré, alors c'est qu'il n'y a pas de successeurs et elle renvoie le résultat $\{fail\}$. Les deux premiers cas correspondent à un τ-test à droite ou à gauche : elle positionne alors le *status* de la liste résultat à *tau*. Dans tous les autres cas, elle le positionne à *ok*.

Algorithme 7.3

fonction *succ_produit* $((p, q))$;
debut
si $(tau\_test\ (p))$ **alors**
$L := \{(succ\_g\ (p), succ\_d\ (p))\}$;
$status\ (L) := tau$;
retourner (L) ;
fsi
si $(tau\_test\ (q))$ **alors**
$L := \{(succ\_g\ (q), succ\_d\ (q))\}$;
$status\ (L) := tau$;
retourner (L) ;
fsi
si $(true\_test\ (p)$ **et** $true\_test\ (q)$ **et** $act\ (p) = act\ (q))$ **alors**
$L := \{(succ\_g\ (p), succ\_g\ (q)), (succ\_d\ (p), succ\_d\ (q))\}$;
$status\ (L) := ok$;
retourner (L) ;
fsi
si $(reduced\_test\ (p)$ **et** $reduced\_test\ (q))$ **alors**
$L := \{(succ\_g\ (p), succ\_g\ (q))\}$;
$status\ (L) := ok$;

```
        retourner (L) ;
    fsi
    si ((reduced_test (p)) ∨ (|succ (p)| = 1 et act (p) = {τ})) alors
        L := {(succ_g (p), q)} ;
        status (L) := ok ;
        retourner (L) ;
    fsi
    si ((reduced_test (q)) ∨ (|succ (q)| = 1 et act (q) = {τ})) alors
        L := {(p, succ_g (q))} ;
        status (L) := ok ;
        retourner (L) ;
    fsi
    si (|succ (p)| = 1 et |succ (q)| = 1 et act (p) = act (q)) alors
        L := {(succ_g (p), succ_g (q))} ;
        status (L) := ok ;
        retourner (L) ;
    fsi
    retourner ({fail}) ;
fin
```

Le coût en temps de la fonction *succ_produit* est constant puisque le facteur de branchement de S est 2.

7.5.5 Manipulation des listes d'hypothèses

Pour chaque état (q_1, q_2), nous extrayons la liste H des hypothèses qui lui est attachée en ⓪. Les opérations à effectuer dépendent de ses successeurs immédiats (q_1', q_2'), que nous traitons un à un :

- Si (q_1', q_2') est dans $Good$ ou est tel que $q_1' = q_2'$, alors nous décrémentons les sommets des piles $StNsucc$ et $StNgood$ en ①.
- Sinon si (q_1', q_2') est dans $Wrong$, alors nous décrémentons le sommet de la pile $StNwrong$ en ②.
- Sinon si (q_1', q_2') est dans $StState$, alors c'est la racine d'une boucle. Nous décrémentons le sommet de la pile $StNsucc$ et nous l'ajoutons aux hypothèses H en ③.
- Sinon si (q_1', q_2') est dans $Visited$, alors il a déjà été exploré précédemment et une liste H' d'hypothèses lui est attachée. Nous appelons en ④ la fonction *explore* :
  - Si elle retourne $\{wrong\}$ alors $q_1' \not\sim_o q_2'$ et donc $q_1 \not\sim_o q_2$. De plus, nous décrémentons le sommet de la pile $StNwrong$.
  - Sinon si elle retourne $\{good\}$ alors $q_1' \sim_o q_2'$ et donc $q_1 \sim_o q_2$. De plus, nous décrémentons les sommets des piles $StNsucc$ et $StNgood$.
  - Sinon elle retourne une liste d'hypothèses que nous ajoutons à la liste H des hypothèses déjà attachée à (q_1, q_2). De plus, nous décrémentons le sommet de la pile $StNsucc$.

Nous donnons le fonctionnement et le programme de la fonction *explore* plus loin.

- Sinon (q_1', q_2') est un nouvel état et nous calculons la liste L' de ses successeurs immédiats :
  - Si cette liste est vide c'est que $q_1' \not\sim_o q_2'$ et nous insérons (q_1', q_2') dans $Wrong$ en ⑤.
  - Sinon nous empilons (q_1', q_2') dans $StState$ et L' dans $StTrans$ en ⑥.

Enfin, à chaque fois qu'un état est complètement traité, nous le dépilons de $StState$, nous l'insérons dans $Visited$ et nous l'enlevons de la liste d'hypothèses qui lui est attachée en ⑧. Par la suite, si un état (p, q) complètement traité a une liste d'hypothèses vide, c'est forcément que $\emptyset \xrightarrow{+} (p, q)$. Dans ce cas, $p \sim_o q$ et nous l'insérons dans $Good$.

Nous explicitons à présent la fonction *explore*. Elle est appelée quand le successeur de l'état courant est dans $Visited - (Wrong \cup Good \cup StState)$. La figure 7.8 illustre un exemple de parcours du produit synchrone où un tel cas peut se produire :

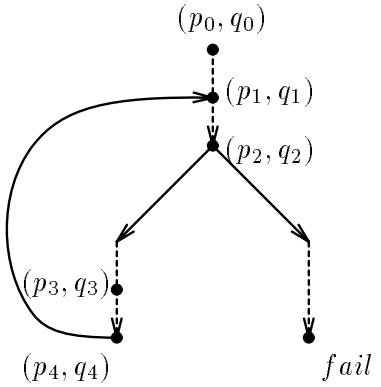


Figure 7.8: Exploration de la liste des hypothèses

- Nous partons de (p_0, q_0).
- En (p_2, q_2) nous explorons d'abord la branche gauche.
- En (p_4, q_4) le successeur immédiat est (p_1, q_1) qui est dans $StState$. Donc (p_1, q_1) est une racine de boucle. Nous faisons alors l'hypothèse que $p_4 \sim_o q_4$ sous réserve que $p_1 \sim_o q_1$. Donc $hypotheses\ ((p_4, q_4)) = \{(p_1, q_1)\}$.
- En dépilant nous propageons cette liste d'hypothèses. Donc $hypotheses\ ((p_3, q_3)) = \{(p_1, q_1)\}$.
- En (p_2, q_2) nous explorons la branche droite qui mène à $fail$. Donc $p_2 \not\sim_o q_2$ et par suite $p_1 \not\sim_o q_1$.

Si au cours du même parcours du produit synchrone nous aboutissons à (p_3, q_3), alors nous exploitons la liste de ses hypothèses grâce à la fonction *explore*. Il s'agit donc d'explorer la liste d'états H de la façon suivante :

- Si au moins un des états est dans $Wrong$ alors nous retournons $\{wrong\}$.
- Sinon si tous sont dans $Good$ alors nous retournons $\{good\}$.
- Sinon pour chaque état non présent dans $Good$:
  - s'il appartient à $StState$ nous l'insérons à la liste résultat, et
  - s'il appartient à $Visited$ nous explorons la liste des hypothèses qui lui est attachée.

La fonction *explore* retourne ou bien $\{good\}$, ou bien $\{wrong\}$, ou bien une liste d'hypothèses. Nous présentons son algorithme sous une forme récursive :

Algorithme 7.4

```
fonction explore (H) ;
debut
    resultat := ∅ ;
    tantque (H ≠ ∅) faire
        choisir (p1, p2) dans H ;
        H := H − {(p1, p2)} ;
        si ((p1, p2) ∈ Wrong) alors
            retourner ({wrong}) ;
        sinon si ((p1, p2) ∈ Good) alors
            (* rien *)
        sinon si ((p1, p2) ∈ StState) alors
            resultat := resultat ∪ {(p1, p2)} ;
        sinon
            (* nécessairement (p1, p2) ∈ Visited *)
            r := explore (hypotheses ((p1, p2))) ;
            si (r = {wrong}) alors
                retourner ({wrong}) ;
            sinon si (r = {good}) alors
                (* rien *)
            sinon
                (* nécessairement r est une liste *)
                resultat := resultat ∪ r ;
            fsi
        fsi
    ftantque
    si (resultat = ∅) alors
        retourner ({good}) ;
    sinon
        retourner (resultat) ;
    fsi
fin
```

7.5.6 Complexité

Il reste à calculer les coûts en temps et en mémoire de la fonction *parcours_produit*. En pratique, nous mémorisons tous les états du produit synchrone dans une structure de données unique, notée $States$. A chaque élément (p, q) de $States$, nous associons les informations suivantes :

- Le ou les ensembles auxquels (p,q) appartient parmi $StState$, $Good$, $Wrong$, $Root$ et $Visited$. Ces informations peuvent par la suite être manipulées grâce à la fonction $status$.
- Pour les éléments de la séquence courante (ceux tels que $status\,(p,q) = StState$), la liste $Trans$ des successeurs de (p,q) restant à examiner.
- Pour les éléments déjà visités (ceux tels que $status\,(p,q) = Visited$), la liste $Hypotheses$ des hypothèses attachées à (p,q).

En outre, un chaînage des éléments de $States$ permet d'ordonner la pile $StState$ à partir de son sommet. Enfin $States$ est implémentée par une table de hachage ouverte [37]. L.Mounier qui a utilisé pour ses algorithmes de vérification "à la volée" ce type de structure [45] a montré que les empilements et les dépilements peuvent être effectués en $O(1)$. Par contre la recherche et l'insertion dans un ensemble est en $O(N/H)$ où H est la taille de la table de hachage et N le nombre d'états déjà insérés dans la table. Nous notons toujours n le nombre d'états et n_r le nombre de boucles de S. Nous notons en outre $\dot{n}$ le nombre d'états du produit synchrone $S \otimes S$: $\dot{n} = O(n^2)$.

Complexité en temps : Tout couple (p,q) est marqué $Visited$ une fois qu'il a été traité et tout couple $Visited$ n'est jamais réempilé. De plus, à chaque itération, les seules opérations non constantes sont la manipulation de la liste H des hypothèses qui est en $O(n)$, les insertions dans $Wrong$, $Good$ ... qui sont en $O(N/H)$, et l'appel à la fonction $succ\_produit$. Donc le coût en temps est : $\mathcal{T}(parcours\_produit) = O(n^2 \times max(n, N/H, \mathcal{T}(succ\_produit))$.

Complexité en mémoire : C'est le coût de la structure $States$, soit $\dot{n}$ fois la taille maximale d'un état. Toutes les informations attachées à un état peuvent être mémorisées dans une taille constante, exceptée la liste $Hypotheses$ dont la taille est $O(n_r)$. Donc le coût en mémoire est : $\mathcal{M}(parcours\_produit) = O(n^2 \times n_r)$.

En résumé, nous avons :

- $\mathcal{T}(parcours\_produit) = O(n^2 \times max(n, N/H))$
- $\mathcal{M}(parcours\_produit) = O(n^2 \times n_r)$

7.5.7 Complexité finale

Nous récapitulons maintenant les coûts théoriques de la synchronisation minimale des branchements. D'une part

$$\mathcal{T}(synchro\_minimale) = O(n \times n_\tau \times n_r^2) + n_\tau \times O(n^2 \times max(n, N/H))$$

Et d'autre part

$$\mathcal{M}(synchro\_minimale) = O(n \times n_r) + O(n^2 \times n_r)$$

En conclusion, nous avons :

- $\boxed{\mathcal{T}(synchro\_minimale) = O(n^3 \times n_\tau)}$
- $\boxed{\mathcal{M}(synchro\_minimale) = O(n^2 \times n_r)}$

7.5.8 Un exemple d'exécution

Afin de bien comprendre le fonctionnement des divers algorithmes, nous présentons un exemple non trivial de système de transitions étiquetées, pour lequel nous allons synchroniser minimalement les branchements :

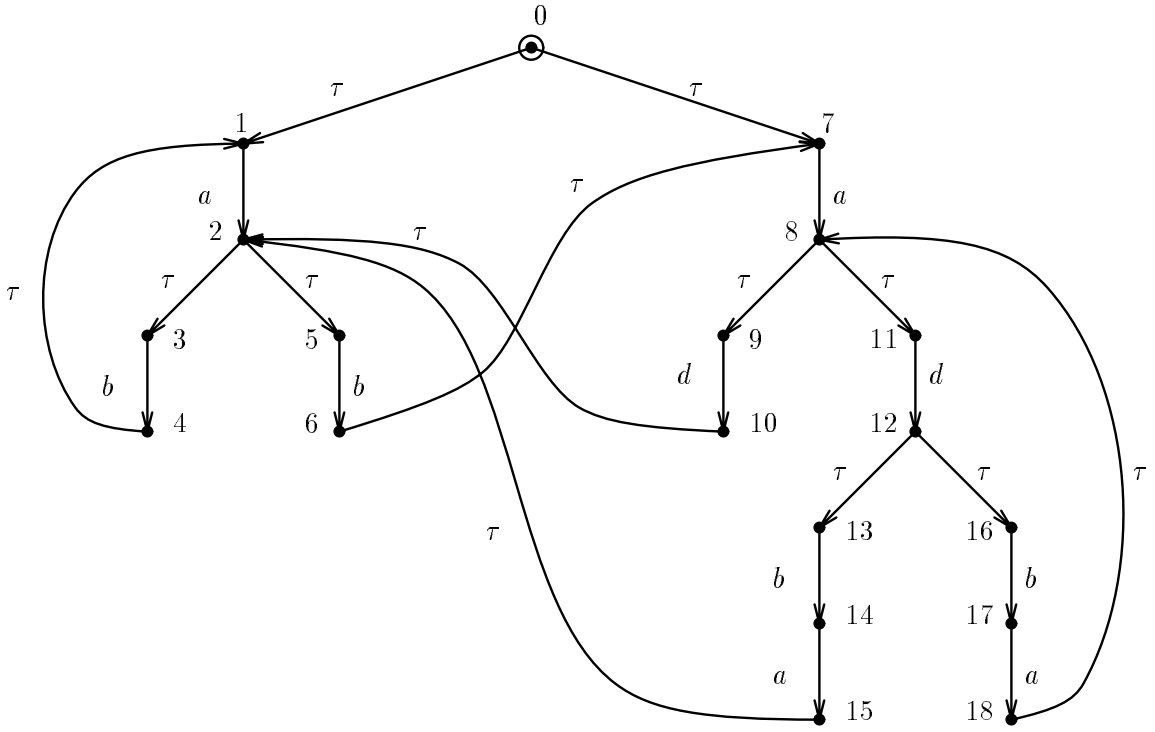
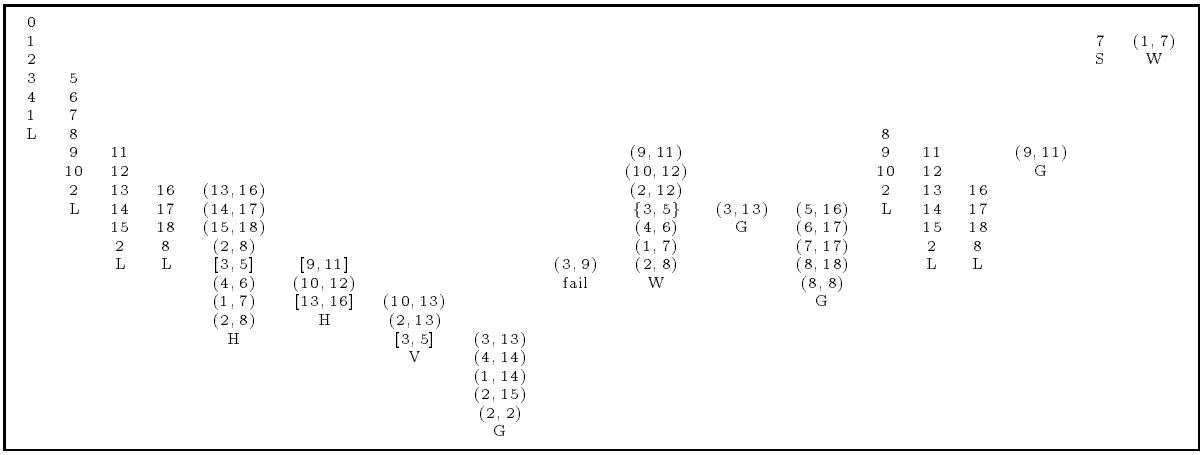


Figure 7.9: Système de transitions étiquetées indéterministe

Nous représentons les piles L et H verticalement. Les empilements se font en descendant, et les dépilements en remontant. Un état (p, q) de la pile H est entre crochets si $status(\{(p, q)\}) = tau$. Il faut alors rappeler la fonction *succ_produit* pour le prédécesseur de (p, q) au moment où un tel état est dépilé.



"L" indique que l'état considéré est déjà "stack" et qu'il doit être rajouté à la liste des hypothèses de son prédécesseur. "S" indique que l'état considéré est bien synchronisé. "H" indique que l'état considéré est déjà présent dans la pile $StState$ de l'algorithme 7.2. De tels états ne sont pas empilés à nouveau. "$fail$" indique que l'état n'a pas de successeurs dans le produit synchrone. "G" (resp. "W", resp. "V") indique que ou bien l'état considéré est dans l'ensemble $Good$ (resp. $Wrong$, resp. $Visited$), ou bien il doit y être inséré.

Nous remarquons un deuxième parcours de la boucle commençant en 8, du à la récriture du τ-test $(c, c')\tau 13 + \tau 16$, et imposé par la condition de stabilité (cf. algorithme 7.1). En outre, nous appelons la fonction $parcours\_produit$ successivement pour trois τ-tests :

- $(13, 16)$ pour lequel le résultat est **faux** puisque le successeur de $(3, 9)$ est $fail$. C'est pourquoi nous récrivons le τ-test $(c, c')\tau 13 + \tau 16$ en $c13 + c'16$.
- $(9, 11)$ pour lequel le résultat est **vrai** puisque $(3, 13)$ et $(5, 16)$ aboutissent respectivement par une séquence élémentaire à l'état $(3, 13) \in Good$ et à l'état $(8, 8)$. Au cours de ce parcours, nous récrivons le τ-test $(c, c')\tau 3 + \tau 5$ en $c3 + c'5$ puisque $(3, 5)$ aboutit par une séquence élémentaire à l'état $(2, 8) \in Wrong$.
- $(1, 7)$ pour lequel le résultat est trivialement **faux** puisque $(1, 7) \in Wrong$. C'est pourquoi nous récrivons le τ-test $(c, c')\tau 1 + \tau 7$ en $c1 + c'7$.

7.6 Redéterminisation et resynchronisation

Pour tout système de transitions étiquetées q, $g(q)$ est, par construction, bien synchronisé, au sens où les seuls branchements indéterministes restant sont tels que leurs deux branches se bisimulent (définition 7.5). Toutefois, il reste à résoudre le problème de la resynchronisation, c'est-à-dire de la transformation d'un système de transitions étiquetées indéterministe en un équivalent déterministe.

7.6.1 Systèmes de transitions étiquetées bien synchronisés

Nous donnons la propriété suivante :

Proposition 7.2
Si p est un *stes*, alors il existe un *sted* p' tel que $p \sim_o p'$.

Preuve :

Nous commençons par prouver que si p et q sont des *stes*, alors $[\![p]\!] = [\![q]\!] \Rightarrow p \sim_o q$. Par définition, $\sim_o$ est la plus grande relation d'équivalence telle que :

$$p \sim_o q - \begin{cases} \text{si } p \stackrel{\tau^\star a \tau^\star}{\rightarrow} p' \text{ alors } \exists q' \text{ tel que } q \stackrel{\tau^\star a \tau^\star}{\rightarrow} q' \text{ et } p' \sim_o q' \\ \text{si } p \stackrel{\tau^\star}{\rightarrow} p' \text{ alors } \exists q' \text{ tel que } q \stackrel{\tau^\star}{\rightarrow} q' \text{ et } p' \sim_o q' \end{cases}$$

Nous raisonnons par induction structurelle : il suffit de prouver que :

$$[\![p]\!] = [\![q]\!] - \begin{cases} \text{si } p \stackrel{\tau^\star a \tau^\star}{\rightarrow} p' \text{ alors } \exists q' \text{ tel que } q \stackrel{\tau^\star a \tau^\star}{\rightarrow} q' \text{ et } [\![p']\!] = [\![q']\!] \\ \text{si } p \stackrel{\tau^\star}{\rightarrow} p' \text{ alors } \exists q' \text{ tel que } q \stackrel{\tau^\star}{\rightarrow} q' \text{ et } [\![p']\!] = [\![q']\!] \end{cases}$$

Comme de plus dans le cas général, $[\![p]\!] = [\![q]\!] \Leftarrow p \sim_o q$, il suffit de prouver l'implication dans le sens $\Rightarrow$. D'après la définition 7.5 des *stes*, il n'y a que trois cas possibles :

- $[\![p]\!] = \{\varepsilon\}$: alors la seule transition que peut effectuer p est $p \stackrel{\tau^\star}{\rightarrow} p'$ et nécessairement $[\![p]\!] = [\![p']\!]$; nous prenons alors $q' = q$ et nous avons $q \stackrel{\tau^\star}{\rightarrow} q'$ avec $[\![p]\!] = [\![p']\!] = [\![q']\!] = [\![q]\!]$.
- $[\![p]\!] = aL$: alors p effectue l'une des deux transitions suivantes :
  - $p \stackrel{\tau^\star}{\rightarrow} p'$ et nécessairement $[\![p]\!] = [\![p']\!]$; nous prenons alors $q' = q$ et nous avons $q \stackrel{\tau^\star}{\rightarrow} q'$ avec $[\![p]\!] = [\![p']\!] = [\![q']\!] = [\![q]\!]$.
  - $p \stackrel{\tau^\star b \tau^\star}{\rightarrow} p'$ et nécessairement $b = a$ et $[\![p']\!] = L$; comme $[\![p]\!] = [\![q]\!] = aL$, ou bien q effectue la transition $q \stackrel{\tau^\star}{\rightarrow} q'$ et nous sommes ramenés au cas précédent, ou bien q effectue la transition $q \stackrel{\tau^\star d \tau^\star}{\rightarrow} q'$ et nécessairement $d = a$ et nous avons $[\![p']\!] = L = [\![q']\!]$.
- $[\![p]\!] = cL + c'L'$: alors p effectue l'une des deux transitions suivantes :
  - $p \stackrel{\tau^\star}{\rightarrow} p'$ et nécessairement $[\![p]\!] = [\![p']\!]$; nous prenons alors $q' = q$ et nous avons $q \stackrel{\tau^\star}{\rightarrow} q'$ avec $[\![p]\!] = [\![p']\!] = [\![q']\!] = [\![q]\!]$.
  - $p \stackrel{\tau^\star b \tau^\star}{\rightarrow} p'$ et nécessairement $b = c$ et $[\![p']\!] = L$ ou $b = c'$ et $[\![p']\!] = L'$; comme $[\![p]\!] = [\![q]\!] = cL + c'L'$, ou bien q effectue la transition $q \stackrel{\tau^\star}{\rightarrow} q'$ et nous sommes est ramenés au cas précédent, ou bien q effectue la transition $q \stackrel{\tau^\star d \tau^\star}{\rightarrow} q'$ et nécessairement $d = c$ et nous avons $[\![p']\!] = L = [\![q']\!]$ ou $d = c'$ et nous avons $[\![p']\!] = L' = [\![q']\!]$.

De plus, un comportement étant un ensemble de traces *maximales* (définition 4.4, page 53), dans les trois cas au moins une des transitions est effectuée. Nous concluons donc que si p et q sont des *stes*, alors $[\![p]\!] = [\![q]\!] \Rightarrow p \sim_o q$. Finalement, pour tout *stes* p, il existe un *sted* q tel que $[\![p]\!] = [\![q]\!]$ et donc tel que $p \sim_o q$.

7.6.2 Mise en œuvre de la redéterminisation

Le principe consiste à regrouper tout couple d'états reliés par une τ-transition, puis à effacer les boucles de τ-transitions. Toutefois, au lieu d'appliquer de telles transformations irréversibles à la structure de l'automate OC, nous nous contentons pour chaque τ-test de choisir une des deux branches et d'ignorer les actions invisibles. Le seul problème est alors de faire attention aux branches situées dans des boucles de τ-transitions.

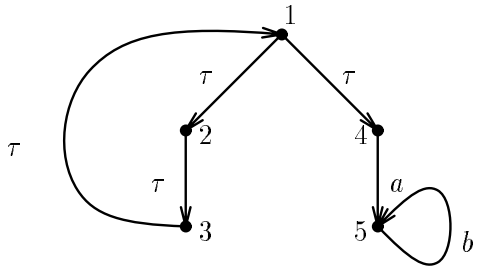


Figure 7.10: Système de transitions étiquetées bien synchronisé

Dans la figure 7.10, les deux *stes* 2 et 4 sont équivalents. Pourtant, si nous choisissons la branche gauche (l'état 2), alors le système de transitions étiquetées réduit n'exécute plus aucune action visible (figure 7.11 (a)) :

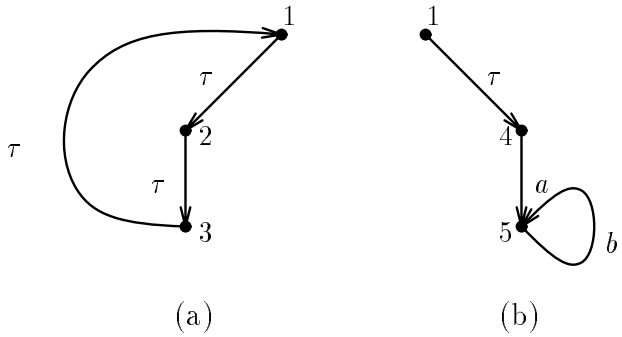


Figure 7.11: Deux systèmes de transitions étiquetées réduits

Par contre, si nous choisissons la branche droite, alors le système de transitions étiquetées réduit est bien équivalent à celui que nous aurions obtenu en regroupant tout couple d'états reliés par une τ-transition et en effaçant les boucles de τ-transitions (figure 7.11 (b)).

Nous utilisons donc une procédure de choix chargée d'explorer les deux branches d'un τ-test afin de savoir si une des deux branches exécute une action visible ou non.

Nous définissons pour cela la fonction *tau_explore* qui détermine la liste des successeurs d'un *stes* par une séquence de τ-transitions. Elle calcule donc, pour un état p de S donné, l'ensemble $succ_{\tau^\star}$ (définition 7.15). Pour cela elle effectue un parcours en profondeur des successeurs p,

en mémorisant dans l'ensemble Tau_set les successeurs déjà visités, afin de ne les visiter qu'une seule fois :

Algorithme 7.5

fonction $tau\_explore(p)$;
debut
    $L := \emptyset$;
    si $(p \notin Tau\_set)$ **alors**
        $Tau\_set := Tau\_set \cup \{p\}$;
        pour tout $(\tau, q) \in succ(p)$ **faire**
            (* appel récursif *)
            $L := L \cup tau\_explore\ (q)$;
        fpour
        $L := L \cup \{p\}$;
    fsi
    retourner (L) ;
fin

La procédure de choix utilise la fonction $tau\_explore$, uniquement pour le successeur droit. En effet, si la branche droite est "mauvaise", alors nous choisissons systématiquement la gauche :

- ou bien la branche gauche est "bonne",
- ou bien elle est également "mauvaise" auquel cas le choix est arbitraire et de toutes façons indifférent.

De plus, l'ensemble Tau_set est initialisé avec le τ-test que nous voulons déterminiser. Ainsi la fonction $tau\_explore$ arrêtera-t-elle son parcours avant ce τ-test :

Algorithme 7.6

fonction $choix\ (p)$;
debut
    $Tau\_set := \{p\}$;
    $L\_d := tau\_explore\ (succ\_d\ (p))$;
    si $(\{act(p') | p' \in L\_d\} \neq \{\tau\})$ **alors**
        retourner $(droite)$;
    sinon
        retourner $(gauche)$;
    fsi
fin

7.6.3 Resynchronisation des programmes réparti minimalement

Nous nous posons à présent le problème de la synchronisation de plusieurs systèmes de transitions étiquetées implantés sur des sites distincts, et dont les structures de contrôle diffèrent. Nous avons le choix entre la resynchronisation faible *totale* et *si besoin* (section 5.3). La resynchronisation *totale* n'est pas satisfaisante dans la mesure où elle ne préserve pas l'équivalence observationnelle.

En effet, elle force la synchronisation de tous les composants du programme réparti, ceci à tous les cycles. Au contraire, la resynchronisation *si besoin* est par construction adaptée au cas des programmes répartis minimalement. En effet, elle n'ajoute un message non valué dans une branche d'un programme que si il y a déjà un message valué dans cette même branche et dirigé dans le sens inverse. Par conséquent elle préserve l'équivalence observationnelle.

7.7 Un exemple complet : le programme `filtre`

Nous concluons ce chapitre par un exemple complètement traité de répartition minimale. Un petit programme réalisant un filtre nous servira de support :

state 0

`if (ck) then`
`  y:=x;`
`  output(y);`
`  output(x);`
`else`
`  output(x);`
`endif`
`goto 0;`

Nous choisissons de répartir ce programme sur deux sites, selon les directives suivantes : `x` et `ck` sur le site 0, et `y` sur le site 1. Nous montrons les deux programmes répartis obtenus l'un *avec* la synchronisation totale des branchements, et l'autre *sans* synchronisation des branchements (le cas *avec* est donné à titre de comparaison) :

avec		*sans*		
site	state 0	site	state 0	
(0)	`put(1,ck);`	()	`if (ck) then`	①
(0,1)	`if (ck) then`	(0)	`  put(1,x);`	
(0)	`  put(1,x);`	(1)	`  y:=x;`	
(1)	`  y:=x;`	(1)	`  output(y);`	
(1)	`  output(y);`	(0)	`  output(x);`	
(0)	`  output(x);`	()	`else`	
(0,1)	`else`	(0)	`  output(x);`	
(0)	`  output(x);`	()	`endif`	
(0,1)	`endif`	(0,1)	`goto 0;`	
(0,1)	`goto 0;`			

Construisons les deux *sten* dans le cas *sans*. Nous avons quatre actions : `put(1,x)`, `y:=x`, `output(y)` et `output(x)`, que nous désignons respectivement par les étiquettes a, b, d et e. Le test ① n'est exécuté par aucun site et est donc représenté par un τ-test. Toute action qui n'est pas exécutée par le site considéré est représentée par l'étiquette τ. La seule exception concerne

les `put`. En effet un `put(i,z)` dans une branche d'un test sur le site `j` implique la présence d'un `z:=get(j)` dans la même branche sur le site `i`. Comme les `get` ne sont pas encore placés (section 7.1.4), un tel `put` représenté par l'étiquette a sur son site "propriétaire" apparaît sur le site "destinataire" sous l'étiquette a'. Enfin les actions implicites `go 0` et `go 1` correspondent aux étiquettes $go0$ et $go1$ respectivement (voir à ce sujet la section 4.1). Donc les *sten* correspondant aux programmes des sites 0 et 1 sont :

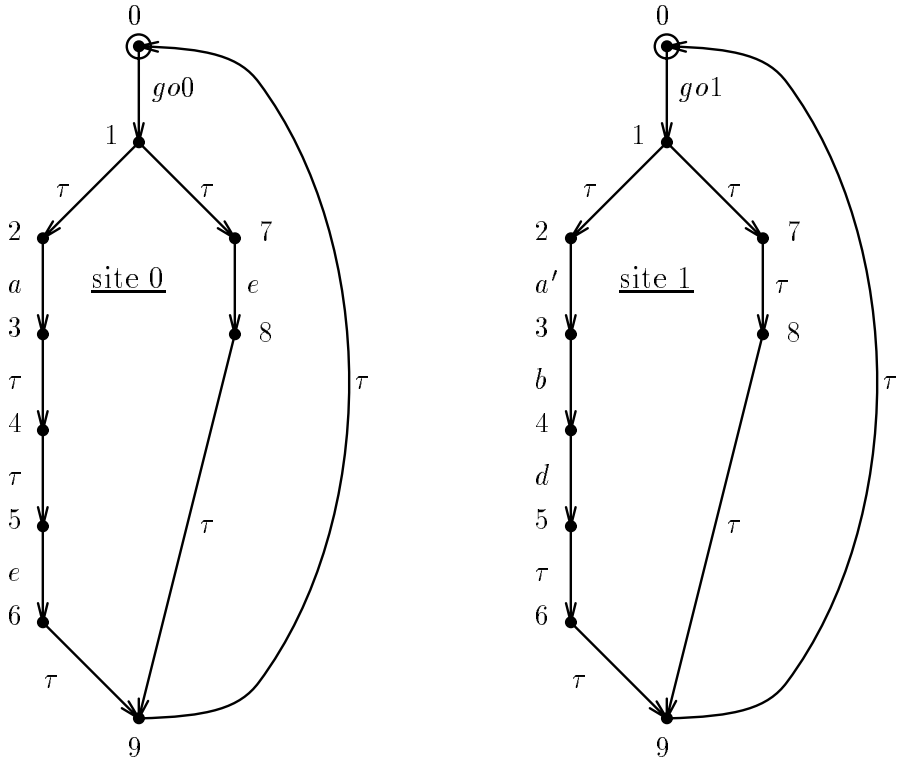


Figure 7.12: Filtre réparti sur deux sites

Le branchement ne peut pas être supprimé sur le site 0 car dans la branche `then` nous effectuons l'action a alors que dans la branche `else`, nous effectuons l'action e. Sur le site 1, cela dépend de si la variable `y` est une entrée ou une variable interne. Cela influe en effet sur la sémantique du `go 1`. Nous avons vu à la section 4.1 que cette action spéciale est une attente de valeurs des entrées de la part de l'environnement. Un programme n'ayant pas d'entrée est donc exécuté en boucle infinie sans jamais se synchroniser avec l'environnement. Nous distinguons donc les deux cas suivants :

- `y` est une variable interne : alors le `go 1` est une τ-action et le branchement peut être supprimé.
- `y` est une variable d'entrée : alors le `go 1` est une action "visible" et le branchement ne peut pas être supprimé.

Toutefois, un fragment de programme réparti doit toujours rester réactif, que ce soit à son environnement, ou à un autre fragment de programme. Pour être réactif, il suffit qu'un programme aie des entrées (dans ce cas il est réactif à l'environnement), ou recoive des valeurs venant d'un autre programme (dans ce cas il est réactif à ce programme). Pour assurer cela, nous ne supprimons définitivement un branchement sur un site donné que si le programme de ce site reste réactif même après cette suppression. Nous vérifions donc que le programme du site en question, ou bien a des entrées, ou bien reçoit des messages venant d'un autre site : c'est le cas du programme du site 1 grâce au `put(1,x)` effectué par le site 0.

Nous donnons à présent les diagrammes d'exécution de la répartition minimale selon le même principe qu'à la section 7.5.8 :

site 0			site 1 variable d'entrée			site 1 variable interne			
0			0			0			
1			1			1			
2	7	(2,7)	2	7	(2,7)	2	7	(2,7)	
3	8	$fail$	3	8	(2,8)	3	8	(2,8)	
4	9		4	9	(2,9)	4	9	(2,9)	
5	0		5	0	(2,0)	5	0	(2,0)	
6	L		6	L	$fail$	6	L	(2,1)	
9			9			9		[2,7]	(2,2)
0			0			0		H	G
L			L			L			

Donc le branchement n'est minimisé que dans un seul des trois cas. Il faut alors choisir la branche du test à exécuter. Pour cela, nous appliquons l'algorithme 7.6, qui fait appel à la fonction *tau_explore*. L'exécution de cette fonction est donnée dans le tableau suivant :

nœud du *stes*	Tau_set	L_d
début	{1}	{0,9 8,7,1}
7	{1,7}	{0,9 8,7}
8	{1,7,8}	{0,9,8}
9	{1,7,8,9}	{0,9}
0	{1,7,8,9,0}	{0}
1	fin	$\emptyset$

A la fin de l'exécution, $L\_d = \{0, 9, 8, 7, 1\}$ contient les successeurs de 7 par des τ-transitions. Donc $act_{\tau^*a}(7) = \{act(p) | p \in L\_d\} = \{\tau\}$. Autrement dit, la branche droite est mauvaise et il faut choisir la gauche.

Finalement, le programme réparti après placement des `get` est le suivant :

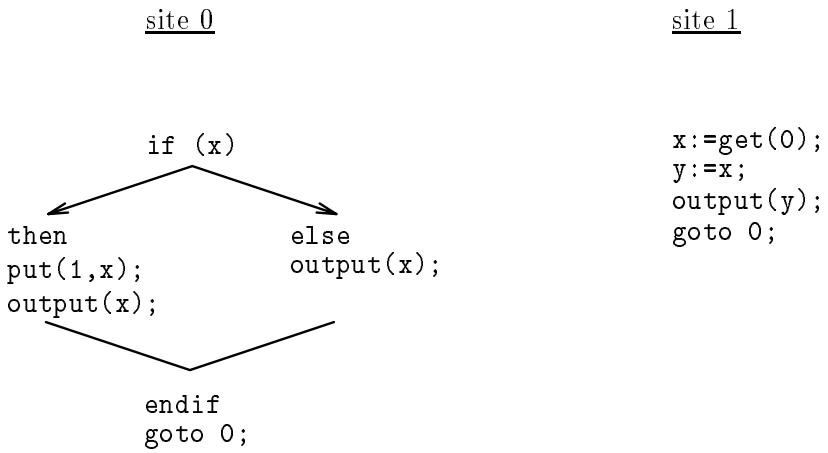


Figure 7.13: Filtre réparti minimalement

Maintenant le programme de la figure 7.13 n'est pas correctement synchronisé puisqu'il n'y a qu'un seul échange de message, dirigé du site 0 vers le site 1. L'algorithme de resynchronisation faible *si besoin* donne pour le programme `filtre` le résultat suivant :

site	state 0
(0)	`if (ck) then`
(1)	`  put_void(0);`
(0)	`  put(1,x);`
(1)	`  y:=x;`
(1)	`  output(y);`
(0)	`  output(x);`
(0)	`else`
(0)	`  output(x);`
(0)	`endif`
(0,1)	`goto 0;`

Après placement des `get`, nous obtenons le programme final suivant :

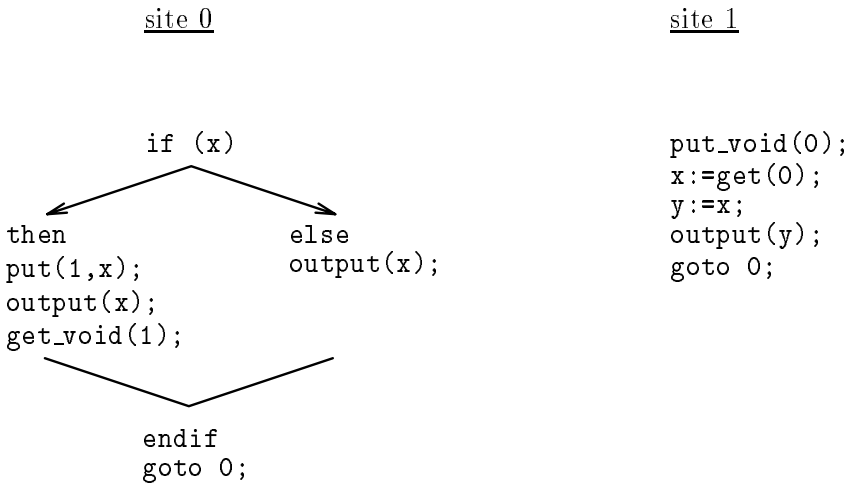


Figure 7.14: Filtre réparti minimalement et resynchronisé

7.8 Récapitulation sur la répartition minimale

La répartition fonctionnelle (chapitre 3) met en œuvre la synchronisation *totale* des branchements. Nous avons présenté dans ce chapitre une méthode de synchronisation *minimale* des branchements.

Cette méthode utilise un critère d'équivalence observationnelle pour établir si les deux branches d'un branchement ont des comportements équivalents. Un tel branchement est alors supprimé.

Nous avons donc définis formellement deux algorithmes :

- Un algorithme d'équivalence observationnelle entre deux systèmes de transitions étiquetées. Cet algorithme explore le produit synchrone de ces deux systèmes, en vérifiant qu'ils exécutent des actions équivalentes. Nous avons utilisé pour sa mise en œuvre des techniques de bisimulation "à la volée" qui évitent l'explosion combinatoire pouvant survenir lors du calcul du produit synchrone.
- Un algorithme de réduction des tests chargé d'appeler la procédure d'équivalence observationnelle pour chaque test, et de supprimer les tests dont les deux branches sont observationnellement équivalentes.

Pour chaque algorithme nous avons discuté les choix de mise en œuvre et donné les coût théoriques en temps et en mémoire. L'étude complete du programme `filtre` a montré que cette solution est satisfaisante.

Pour conclure nous récapitulons les étapes de la synchronisation minimale des branchements :

- placer les `put` correspondant aux actions autres que les branchements (section 3.4),
- placer les `put` pour les tests qu'on ne peut pas supprimer (chapitre présent),
- placer les `put` de resynchronisation (chapitre 5),
- éliminer les `put` redondants (chapitre 6),
- et enfin placer les `get` (section 3.5).

Ainsi, l'algorithme de placement des `get` n'est effectué qu'une seule fois.

Chapitre 8

Relâchement du synchronisme

La répartition fonctionnelle impose comme contrainte que tous les programmes répartis aient la même structure de contrôle. La répartition minimale que nous venons d'étudier permet de s'affranchir de cette contrainte. Nous étudions dans un premier temps les conséquences que cela entraîne, notamment le relâchement de l'hypothèse de synchronisme qui est à la base de tous les langages synchrones. Puis nous présentons une nouvelle sémantique de LUSTRE, qui permet d'expliquer, dans le cadre de ce langage "flots de données", cette perte du synchronisme fort. Enfin nous montrons comment on peut, grâce à la répartition minimale, faire de la répartition non plus fonctionnelle mais dirigée par les horloges.

8.1 Perte du synchronisme

La sémantique temporelle des langages synchrones a été étudiée dans la section 5.1. Nous comparons dans cette section les conséquences, sur le comportement temporel des programmes répartis, des deux méthodes de répartition présentées aux chapitres 3 et 7.

La répartition fonctionnelle avec synchronisation totale des branchements (chapitre 3) provoque une perte du synchronisme mise en évidence au chapitre 5. La resynchronisation, faible ou forte, permet en ajoutant des échanges de messages non valués entre les composants d'un programme réparti, de retrouver dans une certaine mesure le synchronisme, et donc de préserver la sémantique temporelle du programme centralisé :

- Dans le cas de la resynchronisation forte, chaque site ne commence son $n+1^{\text{ième}}$ cycle que lorsque tous les autres ont achevé leur $n^{\text{ième}}$ cycle.
- Dans le cas de la resynchronisation faible, chaque site ne commence son $n+2^{\text{ième}}$ cycle que lorsque tous les autres ont achevé leur $n^{\text{ième}}$ cycle.

La resynchronisation forte est illustrée par la figure 8.1 : nous donnons un exemple de déroulement temporel ainsi que le graphe de synchronisation (une flèche en noir entre deux sites indique que ces deux sites sont fortement synchronisés).

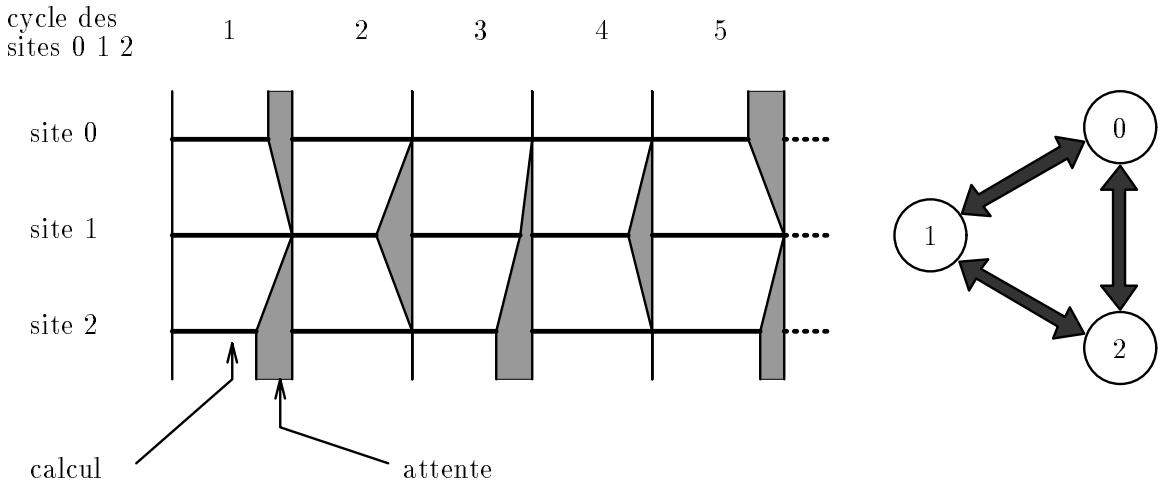


Figure 8.1: Synchronisation forte

Avec la resynchronisation faible, nous avons deux possibilités selon le graphe de synchronisation (défini à la section 5.3). Ou bien ce graphe est complet (resynchronisation faible *totale* : figure 8.2) et il y a au plus un cycle de décalage entre tout couple de sites (une flèche en gris clair entre deux sites indique que ces deux sites sont faiblement synchronisés) :

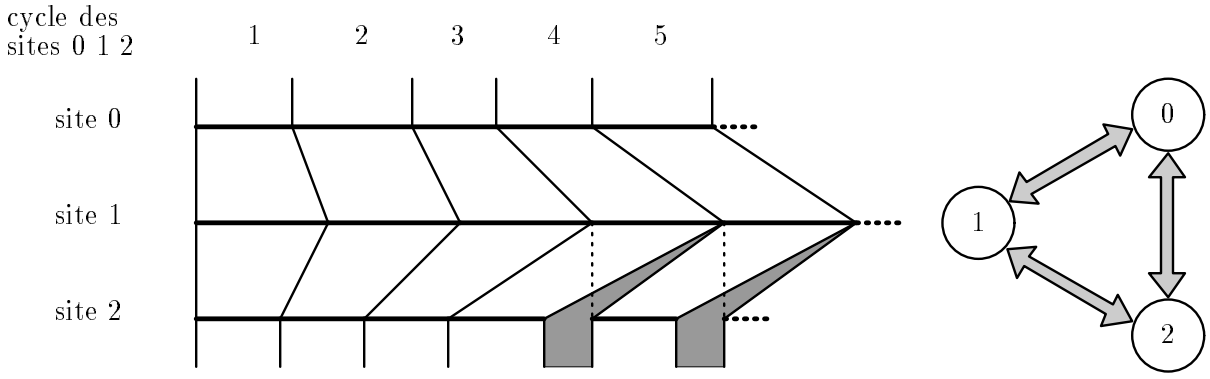


Figure 8.2: Synchronisation faible *totale*

Ou bien ce graphe n'est pas complet (resynchronisation faible *si besoin* : figure 8.3) et il y a au plus un cycle de décalage entre tout couple de sites adjacents. Il peut donc y avoir n cycles de décalage entre deux sites, n étant le diamètre du graphe de synchronisation. Nous voyons donc apparaître une notion de pipe-line au niveau des désynchronisations entre les sites. Par exemple, dans la figure 8.3, le site 1 est synchronisé avec le site 0 et avec le site 2, mais les sites 0 et 2 ne sont pas synchronisés entre eux :

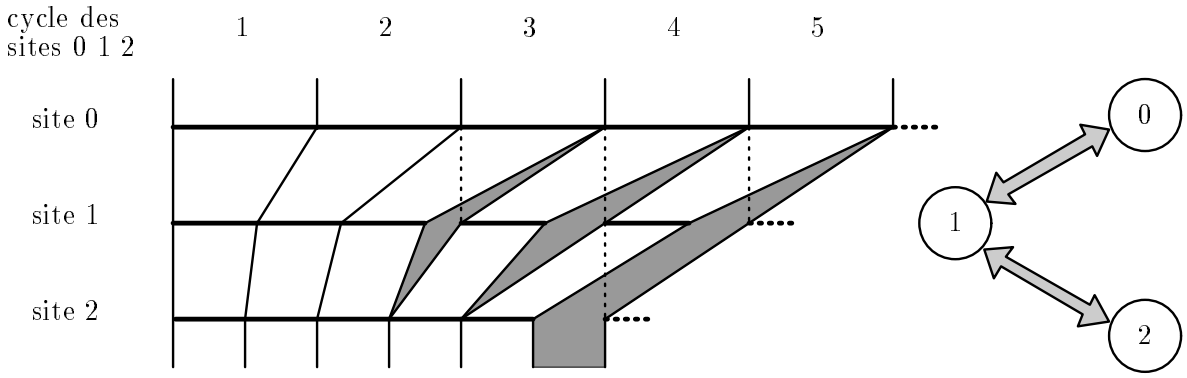


Figure 8.3: Synchronisation faible *si besoin*

Maintenant, si le programme a été réparti minimalement, au sens où les branchements ont été projetés avec l'opérateur de synchronisation minimale (section 7.2), alors les programmes de certains sites peuvent avoir une horloge plus lente que celles des autres. C'est le cas du programme `filtre` de la figure 7.13, page 138 : le $n^{\text{ième}}$ cycle du programme du site 1 est le cycle du programme du site 0 où la variable `x` vaut `true` pour la $n^{\text{ième}}$ fois. Nous avons vu à la section 7.6.3 que c'est forcément la resynchronisation faible *si besoin* qui doit être utilisée (figure 8.4) :

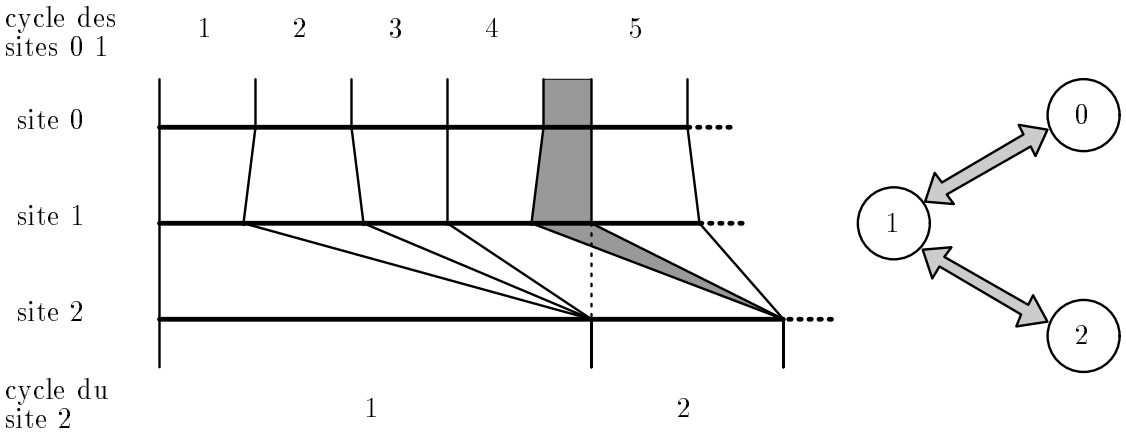


Figure 8.4: Répartition minimale

En conclusion, les cycles des composants d'un programme réparti où les branchements sont totalement synchronisés sont, quoi qu'il arrive, décalés d'un nombre borné de cycles, mais ce n'est plus le cas si les branchements ont été synchronisés minimalement. Nous introduisons dans la section suivante un nouveau formalisme permettant de comprendre cette dissociation des cycles.

8.2 Sémantique asynchrone de LUSTRE

Nous cherchons à résoudre le problème suivant : comment comprendre les désynchronisations que l'on désire introduire en exécution répartie ? Nous donnons ici une sémantique originale pour le langage LUSTRE qui permet d'expliquer de telles désynchronisations.

8.2.1 Généralisation du modèle

L'idée est de considérer l'ensemble des déroulements possibles du réseau flots de données LUSTRE. Ces déroulements sont pris chacun comme un ordre partiel. Cela permet d'individualiser chaque instant, et de ne regarder que ses relations d'ordonnancement nécessaires avec les instants des autres flots.

Par exemple, l'ordre partiel étiqueté (*lpo* défini par la définition 4.10 page 58)

$$(X \rightarrow u_0 \rightarrow X \rightarrow u_1 \rightarrow X \rightarrow u_2 \rightarrow \cdots)$$

signifie que le flot X véhicule successivement les valeurs $u_0, u_1, u_2, \ldots$

Ce modèle permet également de spécifier des ordonnancements entre différents flots.

Par exemple, le *lpo* suivant signifie que les flots X et Y véhiculent respectivement les valeurs x_0, x_1 et y_0, y_1, et que l'instant de la valeur y_0 *précède* celui de la valeur x_1 :

$$\begin{pmatrix} X & \rightarrow & x_0 & \rightarrow & X & \rightarrow & x_1 & \rightarrow & X & \ldots \\ & & & & & \nearrow & & & & \\ Y & \rightarrow & y_0 & \rightarrow & Y & \rightarrow & y_1 & \rightarrow & Y & \ldots \end{pmatrix}$$

Un tel *lpo* modélise également le comportement d'un programme réparti sur deux sites, les cycles du site 0 étant représentés par le flot X, et ceux du site 1 par le flot Y. La dépendance temporelle entre Y et X est alors interprétée comme une dépendance fonctionnelle : c'est une communication asynchrone par file d'attente entre les deux sites.

Le *lpo* suivant indique qu'il y a une communication du site 1 vers le site 0, à chaque cycle :

$$\begin{pmatrix} \text{site 0 :} & X & \rightarrow & x_0 & \rightarrow & X & \rightarrow & x_1 & \rightarrow & X & \ldots \\ & & \nearrow & & & & \nearrow & & & & \\ \text{site 1 :} & Y & \rightarrow & y_0 & \rightarrow & Y & \rightarrow & y_1 & \rightarrow & Y & \ldots \end{pmatrix}$$

Un tel *lpo* constitue donc un modèle d'exécution de deux sites non resynchronisés puisque les communications sont toutes dirigées vers le site 0. Pour modéliser deux sites faiblement resynchronisés, il suffit de rajouter des dépendances du flot X vers le flot Y à chaque cycle.

Ce modèle généralise le modèle d'ensembles de traces de valeurs, présenté dans [5] pour SIGNAL. D'autre part, il a été reconnu comme permettant de généraliser le modèle de Kahn [36] aux réseaux non déterministes, tout en évitant les problèmes d'anomalie ou d'équité [24]. Il est cependant trop général, en ce sens qu'il permet de représenter des réseaux non exécutables en temps et mémoire bornées. Nous allons le restreindre en considérant les ensembles d'ordres qui peuvent s'exprimer par des expressions régulières.

8.2.2 Concaténation et expressions régulières

Comme nous manipulons des programmes à nombre de flots fini, nous considérerons des ordres partiels de largeur finie. Il nous faut toutefois une notion de concaténation. Nous choisissons la définition 4.13 introduite à la section 4.3, page 59.

Par exemple :

$$\begin{pmatrix} a & \to & X \\ & \nearrow & \\ b & \to & Y \end{pmatrix} . \begin{pmatrix} X & \to & c \\ & \nearrow & \\ Y & \to & d \end{pmatrix} = \begin{pmatrix} a & \to & X & \to & X & \to & c \\ & \nearrow & & & & \nearrow & \\ b & \to & Y & \to & Y & \to & d \end{pmatrix}$$

Dans la suite, l'opérateur de concaténation sera traditionnellement noté "." et le *lpo* vide sera noté "ε". Le passage aux ensembles donne ensuite un sens à l'opérateur d'union ensembliste binaire noté "+" et n-aire noté "$\sum$". Nous étendons l'opérateur "." aux ensembles de *lpo* : $L.L' = \{l.l' \mid l \in L, l' \in L'\}$. Nous définissons alors l'opérateur "$\star$" par :

Définition 8.1 (opérateur $\star$ sur les *lpo*)

- Pour tout ensemble de *lpo* L, $L^0 = \varepsilon$.
- Pour tout ensemble de *lpo* L et pour tout entier n, $L^{n+1} = L.L^n$.
- Pour tout ensemble de *lpo* L, $L^\star = \sum L^n$.

Ainsi, $L^\star$ désigne l'ensemble de *lpo* constitué d'un nombre quelconque et *fini* d'éléments de L concaténés. La sémantique en ordres partiels de LUSTRE décrit donc des flots *finis*, ce qui est a priori plus simple que des flots infinis.

Munis de ces opérateurs, nous pouvons écrire des expressions régulières sur les *lpo*.

8.2.3 Sémantique des opérateurs

Nous donnons la sémantique en ordre partiel des opérateurs LUSTRE. Ce sont les quatre opérateurs temporels (`pre`, `->`, `when` et `current`), plus un opérateur binaire classique `f`.

8.2.3.1 Opérateur binaire classique : Z = f(X,Y)

Le flot `f(`X`,`Y`)` ne véhicule une nouvelle valeur $f(u, v)$ que quand le flot X (resp. Y) lui fournit une nouvelle valeur u (resp. v) :

$$\left(\sum_{u,v} \begin{pmatrix} X & \to & u & \to & X & & \\ & & & \searrow & & & \\ & & Z & \to & f(u,v) & \to & Z \\ & & & \nearrow & & & \\ Y & \to & v & \to & Y & & \end{pmatrix} \right)^\star$$

$\sum_{u,v}$ dénotant ici l'union ensembliste (le "`ou`") sur toutes les valeurs possibles de u et v.

8.2.3.2 Opérateur de retard : Y `=` `pre` X

Le flot `pre` X commence par véhiculer la valeur *nil* (initialisation), puis il ne véhicule une nouvelle valeur u que quand le flot X lui en fournit une nouvelle (phase courante) :

$$\underbrace{(Y \to nil \to Y)}_{\text{initialisation}} . \underbrace{\left(\sum_u \begin{pmatrix} X & \to & u & \to & X & \\ & & & \searrow & & \\ & & Y & \to & u & \to & Y \end{pmatrix} \right)^\star}_{\text{phase courante}} . \underbrace{\sum_u (X \to u \to X)}_{\text{terminaison}}$$

La phase de terminaison assure que les deux flots ont la même longueur.

8.2.3.3 Opérateur d'initialisation : Z `=` X `->` Y

Le flot X `->` Y commence par véhiculer la valeur u dès que le flot X la lui a fournie (initialisation), puis il ne véhicule une nouvelle valeur v que quand le flot Y lui en fournit une nouvelle (phase courante) :

$$\underbrace{\sum_{u,v} \begin{pmatrix} X & \to & u & \to & X & \\ & & & \searrow & & \\ & & Z & \to & u & \to & Z \\ & & & \nearrow & & \\ Y & \to & v & \to & Y & \end{pmatrix}}_{\text{initialisation}} . \underbrace{\left(\sum_{u,v} \begin{pmatrix} X & \to & u & \to & X & \\ & & & \searrow & & \\ & & Z & \to & v & \to & Z \\ & & & \nearrow & & \\ Y & \to & v & \to & Y & \end{pmatrix} \right)^\star}_{\text{phase courante}}$$

8.2.3.4 Opérateur de filtrage : Y `=` X `when` C

Le flot X `when` C ne véhicule une nouvelle valeur u que quand le flot C lui fournit la valeur $true$ et quand le flot X lui fournit sa nouvelle valeur u (recopie). Et quand le flot C véhicule des valeurs $false$, le flot X `when` C ne véhicule aucune valeur (filtrage) :

$$\left(\underbrace{\sum_u \begin{pmatrix} X & \to & u & \to & X & \\ & & & \searrow & & \\ & & Y & \to & u & \to & Y \\ & & & \nearrow & & \\ C & \to & true & \to & C & \end{pmatrix}}_{\text{recopie}} + \underbrace{\sum_u \begin{pmatrix} X & \to & u & \to & X \\ & & & & \\ C & \to & false & \to & C \end{pmatrix}}_{\text{filtrage}} \right)^\star$$

8.2.3.5 Opérateur de projection : Y `=` `current` X *avec* C *horloge de* X

Tant que le flot C ne véhicule que des valeurs $false$, le flot `current` X véhicule des valeurs *nil* (initialisation). Puis à chaque nouvelle valeur $true$ du flot C, le flot `current` X véhicule la

nouvelle valeur u du flot X (mémorisation), et continue à véhiculer cette même valeur u tant que le flot C ne véhicule que des valeurs $false$ (projection) :

$$\overbrace{\left(\begin{array}{cccccc} & & Y & \rightarrow & nil & \rightarrow & Y \\ & & & \nearrow & & & \\ C & \rightarrow & false & \rightarrow & C & & \end{array}\right)^{\star}}^{\text{initialisation}}.$$

$$\left(\sum_u \left(\underbrace{\left(\begin{array}{cccccc} X & \rightarrow & u & \rightarrow & X & \\ & & & \searrow & & \\ & & Y & \rightarrow & u & \rightarrow & Y \\ & & & \nearrow & & & \\ C & \rightarrow & true & \rightarrow & C & & \end{array}\right)}_{\text{mémorisation}} . \underbrace{\left(\begin{array}{cccccc} & & Y & \rightarrow & u & \rightarrow & Y \\ & & & \nearrow & & & \\ C & \rightarrow & false & \rightarrow & C & & \end{array}\right)^{\star}}_{\text{projection}}\right)\right)^{\star}$$

8.2.4 Sémantique asynchrone d'un programme LUSTRE

Un programme LUSTRE est le résultat de la composition parallèle d'opérateurs élémentaires. Pour définir le comportement d'un programme, il faut donc définir la mise en parallèle de deux flots X et Y représentés chacun par un ensemble de comportements qui sont tous des *lpo*.

Nous supposons que ces deux flots sont corrects du point de vue des horloges, c'est-à-dire que les vérifications statiques sur leurs horloges ont déjà été effectuées. Ces vérifications sont décrites formellement dans [18] et [31].

En notant V_X et V_Y les vocabulaires respectifs des *lpo* de X et de Y, nous avons :

Définition 8.2 (composition parallèle de deux *lpo*)
$X \,[\!]\, Y = \{z : \exists x \in X, \exists y \in Y$ tels que $x \preceq z/V_X \wedge y \preceq z/V_Y\}$ où $\preceq$ est l'ordre "plus partiel que" sur les *lpo* donné par la définition 4.11, page 59.

Intuitivement, $X \,[\!]\, Y$ est l'ensemble des *lpo* obtenus en mettant en commun les valeurs des flots de X et de Y de même nom. Par exemple, `f(X,Y) [] g(Y,Z)` est l'ensemble des sur-ordonnancements de :

$$\left(\sum_{u,v,w} \left(\begin{array}{ccccccc} X & \rightarrow & u & \rightarrow & X & & \\ & & & \searrow & & & \\ & & S & \rightarrow & f(u,v) & \rightarrow & S \\ & & & \nearrow & & & \\ Y & \rightarrow & v & \rightarrow & Y & & \\ & & & \searrow & & & \\ & & T & \rightarrow & g(v,w) & \rightarrow & T \\ & & & \nearrow & & & \\ Z & \rightarrow & w & \rightarrow & Z & & \end{array}\right)\right)^{\star}$$

Maintenant la sémantique asynchrone que nous venons de présenter pose le problème de l'exécution : nous ne pouvons pas garantir l'exécution en mémoire bornée.

Par exemple, pour l'opérateur binaire classique Z `= f(`X`,`Y`)`, les deux flots X et Y peuvent aller aussi vite qu'ils le veulent sans que `f` ait le temps de consommer toutes leurs valeurs. Si les flots X, Y et Z représentent respectivement les sites 0, 1 et 2, cela signifie que les communications sont toutes dirigées vers le site 2, et par conséquent que ce programme réparti n'est pas synchronisé.

Autrement dit cette sémantique est beaucoup trop asynchrone pour être utilisable. Nous donnons une seconde sémantique en ordres partiels de LUSTRE, que nous appelons "naturelle", et qui est plus structurée que la sémantique asynchrone.

8.3 Sémantique naturelle de LUSTRE

Cette sémantique s'obtient en ajoutant des dépendances entre les flots, afin d'assurer que les files d'attente de communication sont bornées. Elle présente les avantages suivants :

- garantie de mémoire bornée,
- pipe-line maximal (au sens défini à la section 8.1),
- et interprétation naturelle des horloges comme des intervalles de temps insécables.

8.3.1 Sémantique des opérateurs

Nous donnons la sémantique naturelle des opérateurs LUSTRE dans le même ordre que précédemment. La sémantique d'un programme LUSTRE reste définie par l'opérateur de composition des *lpo* de la section 8.2.4.

8.3.1.1 Opérateur binaire classique : Z `= f(`X`,`Y`)`

Par rapport à la sémantique asynchrone de l'opérateur binaire `f`, nous avons ajouté des dépendances du flot Z vers les flots X et Y, après le calcul de $f(u,v)$. Ainsi la consommation de deux valeurs u et v pour le calcul d'une valeur $f(u,v)$ du flot Z autorise la fabrication de deux nouvelles valeurs des flots X et Y. Il ne faut jamais mémoriser plus d'une valeur pour chaque flot, et l'exécution peut donc bien se faire en mémoire bornée :

$$\left(\sum_{u,v} \begin{pmatrix} X & \to & u & & & & X \\ & & & \searrow & & \nearrow & \\ & & Z & \to & f(u,v) & \to & Z \\ & & & \nearrow & & \searrow & \\ Y & \to & v & & & & Y \end{pmatrix} \right)^{\star}$$

8.3.1.2 Opérateur de retard : Y `= pre` X

Par rapport à la sémantique asynchrone de l'opérateur `pre`, nous avons ajouté une dépendance du flot Y vers le flot X, après le calcul de `pre(`u`)`. Ainsi la consommation d'une valeur u par le

flot Y autorise la fabrication d'une nouvelle valeur du flot X :

$$(Y \to nil \to Y).\left(\sum_u \begin{pmatrix} X & \to & u & & & & X \\ & & & \searrow & & \nearrow & \\ & & Y & \to & u & \to & Y \end{pmatrix}\right)^{\star}.\sum_u (X \to u \to X)$$

8.3.1.3 Opérateur d'initialisation : Z = X -> Y

Par rapport à la sémantique asynchrone de l'opérateur `->`, nous avons ajouté des dépendances du flot Z vers les flots X et Y, après le calcul de la nouvelle valeur du flot Z. Ainsi la consommation de deux valeurs u et v pour le calcul d'une valeur du flot Z autorise la fabrication de deux nouvelles valeurs des flots X et Y :

$$\sum_{u,v} \begin{pmatrix} X & \to & u & & & & X \\ & & & \searrow & & \nearrow & \\ & & Z & \to & u & \to & Z \\ & & & \nearrow & & \searrow & \\ Y & \to & v & & & & Y \end{pmatrix}.\left(\sum_{u,v} \begin{pmatrix} X & \to & u & & & & X \\ & & & \searrow & & \nearrow & \\ & & Z & \to & v & \to & Z \\ & & & \nearrow & & \searrow & \\ Y & \to & v & & & & Y \end{pmatrix}\right)^{\star}$$

8.3.1.4 Opérateur de filtrage : Y = X when C

Les modifications que nous avons apportées par rapport à la sémantique asynchrone de l'opérateur `when` sont un peu plus délicates que pour les opérateurs précédents.

Etudions le cas d'une variable LUSTRE X filtrée par une horloge lente C puis reprojetée sur l'horloge de base :

cycle de base	1	2	3	4	5	6	7
C	`true`	`false`	`false`	`true`	`false`	`true`	`true`
X	X_1	X_2	X_3	X_4	X_5	X_6	X_7
cycle de C	1			2		3	4
Y = X when C	X_1			X_4		X_6	X_7
current Y	X_1	X_1	X_1	X_4	X_4	X_6	X_7

Pour calculer `current` Y, nous avons besoin de la valeur de Y à chaque début de cycle de l'horloge C. Mais ce n'est plus vrai si nous introduisons un retard :

C	`true`	`false`	`false`	`true`	`false`	`true`	`true`
X	X_1	X_2	X_3	X_4	X_5	X_6	X_7
cycle de C	1			2		3	4
Y = X when C	X_1			X_4		X_6	X_7
Z = 0 -> pre Y	0			X_1		X_4	X_6
current Z	0	0	0	X_1	X_1	X_4	X_6

Pour calculer `current (0 -> pre` Y`)`, nous avons uniquement besoin de la valeur de Y à chaque fin de cycle de l'horloge C. C'est ce qui permet de relâcher le synchronisme entre deux sites,

quand l'un est cadencé à une horloge "rapide" et l'autre à une horloge "lente". Par conséquent, c'est le flot C qui, quand il vaut `true`, commande un filtrage et donc réclame une nouvelle valeur du flot X. Ainsi le flot lent Y peut calculer sa valeur n'importe quand entre deux instants où son horloge C vaut `true`. Ceci permet de comprendre l'exécution répartie des tâches de longue durée.

Par rapport à la sémantique asynchrone de l'opérateur `when`, nous avons donc ajouté des dépendances du flot Y vers les flots C et X, avant le calcul de la nouvelle valeur de Y. Ainsi, *chaque* nouvelle valeur u, u et $true$, respectivement des flots X, Y et C, dépend des *trois* valeurs précédentes :

$$\left(\sum_u \begin{pmatrix} X & & u & \to & X & \\ & \searrow\nearrow & & \searrow & & \\ Y & \to\to & & & u & \to & Y \\ & \nearrow\searrow & & \nearrow & & \\ C & & true & \to & C & \end{pmatrix} + \sum_u \begin{pmatrix} X & & & u & \to & X \\ & \searrow\nearrow & & & & \\ & \nearrow\searrow & & & & \\ C & & & false & \to & C \end{pmatrix}\right)^\star$$

8.3.1.5 Opérateur de projection : Y = `current` X avec C horloge de X

Par rapport à la sémantique asynchrone de l'opérateur `current`, nous avons ajouté des dépendances du flot Y vers les flots C et X, après le calcul de la nouvelle valeur du flot Y. Ainsi en phase de mémorisation, la consommation de deux valeurs u et $true$ pour le calcul d'une valeur u du flot Y autorise la fabrication de deux nouvelles valeurs des flots C et X. De même, en phase de projection, la consommation d'une valeur $false$ pour le calcul d'une valeur u du flot Y autorise la fabrication d'une nouvelle valeur du flot C :

$$\begin{pmatrix} & Y & \to & nil & \to & Y \\ & & \nearrow & & \searrow & \\ C & \to\ false & & & & C \end{pmatrix}^\star .$$

$$\left(\sum_u \left(\begin{pmatrix} X & \to & u & & & X \\ & & & \searrow & \nearrow & \\ & & Y & \to & u & \to & Y \\ & & & \nearrow & & \searrow & \\ C & \to & true & & & & C \end{pmatrix} . \begin{pmatrix} & Y & \to & u & \to & Y \\ & & \nearrow & & \searrow & \\ C & \to\ false & & & & C \end{pmatrix}^\star\right)\right)^\star$$

8.4 Répartition dirigée par les horloges

Dans la section 8.3, nous avons présenté une nouvelle sémantique de LUSTRE permettant de comprendre les désynchronisations introduites par la répartition minimale des programmes. Nous étudions à présent une application originale, toujours de la répartition minimale : la répartition dirigée par les horloges. L'idée est de compiler normalement le programme LUSTRE, puis d'écrire les directives de répartition en fonction des horloges de ses entrées/sorties. Après cela, le répartiteur de code `oc2rep` produira automatiquement des exécutables - un pour chaque horloge - désynchronisés. Nous illustrons tout d'abord cette méthode avec un exemple.

8.4.1 Le programme LUSTRE

```
node exemple (ck:bool; x:int)
          returns
          ((y:int) when ck);
let
        y = filtre(x when ck);
tel.

node filtre (a:int)
       returns
       (b:int);
let
        b = calcul(a);
tel.
```

La fonction `calcul` représente une tâche de longue durée qui doit être exécutée chaque fois que l'horloge `ck` (l'horloge de `y`, `a` et `b`) vaut `true`.

8.4.2 Le programme OC

La compilation en automate nous donne l'automate OC suivant :

```
state 0
if (ck) then
     y:=calcul(x);
     output(y);
else
endif
goto 0;
```

8.4.3 Les directives de répartition

Le programme LUSTRE comporte trois entrées/sorties :

- `ck` et `x` sur l'horloge de base,
- `y` sur l'horloge `ck`.

En règle générale, nous attribuons un site à chaque horloge, et donc à chaque variable sur cette horloge, mais cela peut varier en fonction des besoins. Dans tous les cas, la répartition est dirigée par la demande : c'est l'utilisateur qui écrit les directives de répartition.

Dans le cas présent, nous désirons répartir le programme sur deux sites, et les directives sont :

```
site 0 : ck x  -- horloge ce base
site 1 : y     -- horloge ck
```

8.4.4 Répartition minimale

Nous trouvons par bisimulation que sur le site 1 les deux branches du test sont équivalentes, ce qui permet de supprimer ce test. Voici ce que nous obtenons sans resynchronisation :

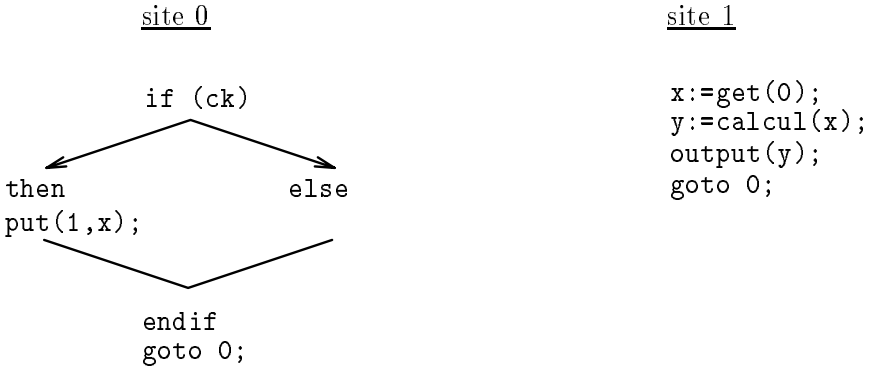


Figure 8.5: Filtre réparti avant resynchronisation

Clairement, si le producteur (site 0) va plus vite que le consommateur (site 1), nous pouvons avoir une explosion de la file d'attente servant à l'échange des valeurs de `x`. Pour remédier à cela, nous appliquons au programme la resynchronisation faible *si besoin* (section 5.3.2 page 81). Nous obtenons alors :

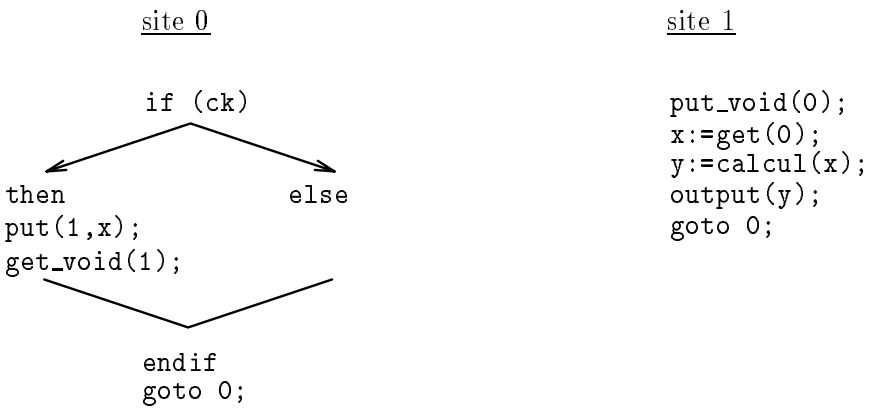


Figure 8.6: Filtre réparti après resynchronisation faible *si besoin*

La figure 8.7 illustre la synchronisation relâchée des programmes ainsi répartis : le programme du site 1 reste suspendu en attente du message du site 0, message qui arrive chaque fois que l'horloge

`ck` vaut `true`. Le comportement du programme du site 1 est donc bien celui d'un processus à horloge lente, et synchronisé sur le processus rapide.

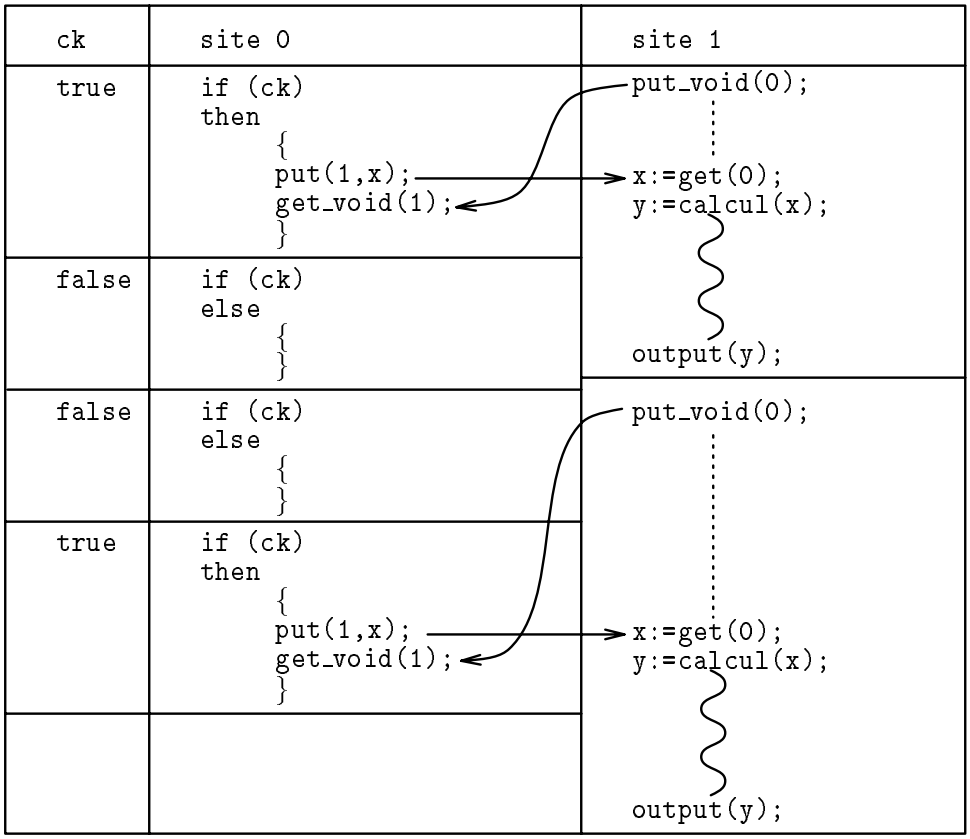


Figure 8.7: Exemple d'exécution du programme réparti

8.4.5 Algorithme de répartition par les horloges

La répartition est toujours dirigée par l'utilisateur. Pour répartir un programme synchrone par les horloges, l'utilisateur doit donc localiser sur un même site les variables ayant la même horloge.

Il est ainsi possible d'intégrer dans un programme la gestion des tâches de longue durée, non pas comme des procédures asynchrones mais comme des processus lents activés par un processus rapide.

Remarquons que contrairement aux tâches asynchrones d'ESTEREL, le processus rapide subit une contrainte de synchronisation de la part du processus lent.

8.5 Conclusion

Nous avons dans ce chapitre apporté des réponses au problème central : comment expliquer les désynchronisations introduites en mode réparti ?

En effet, la répartition minimale (chapitre 7) provoque des différences entre les structures de contrôle des fragments d'un programme réparti. Même après l'application de l'algorithme de resynchronisation (chapitre 5), tous ces fragments n'ont pas la même horloge logique. C'est précisément ce qui permet de prendre en compte, entres autres, les tâches de longue durée.

Nous avons alors considéré l'ensemble des déroulements des réseaux flots de données LUSTRE, déroulements considérés chacun comme un ordre partiel. Pour chaque instant d'un flot de données, nous n'avons pris en compte que ses relations d'ordonnancement avec les instants des autres flots. Pour la concaténation des ordres partiels nous avons opté pour la définition 4.13 déjà utilisée dans le cadre de la preuve de l'algorithme de parallélisation.

Nous avons donc proposé une nouvelle sémantique de LUSTRE, que nous avons appelé "sémantique asynchrone", définie par des ordres partiels. Un programme LUSTRE y est représenté par un ensemble d'ordres partiels, chacun étant un comportement possible de ce programme. Mais cette sémantique ne garantit pas l'exécution en mémoire bornée. Nous avons donc proposé une seconde sémantique plus stricte, que nous avons appelé "sémantique naturelle". Pour l'obtenir, nous avons appliqué le principe de la resynchronisation faible, c'est-à-dire que nous avons ajouté aux ordres partiels des opérateurs des dépendances supplémentaires. L'étude de la sémantique naturelle de l'opérateur `when` permet alors de comprendre le relâchement du synchronisme entre deux sites.

Chapitre 9

Exécution répartie

Le dernier problème qui reste à traiter concerne l'exécution des programmes OC que nous venons de répartir. Nous expliquons dans un premier temps comment s'exécute un programme OC centralisé, et comment est résolu le problème de l'interface entre l'environnement asynchrone et le programme synchrone. Puis nous détaillons la façon de mettre en œuvre les primitives de communication nécessaires à une exécution répartie. Enfin nous étudions en détail le problème posé par l'interface entre l'environnement asynchrone et le programme synchrone dans le cas de programmes répartis.

Le principal souci est ici de trouver une solution *répartie* la plus proche possible de la solution centralisée. Nous proposons deux solutions dont nous discutons le degré de fidélité par rapport à la solution centralisée.

9.1 Exécution des programmes OC centralisés

Le format OC n'est qu'un format interne et en aucun cas un format exécutable. Actuellement, la chaîne d'exécution d'un programme OC passe par une phase de traduction vers un langage de haut niveau (C, Ada, ...), puis par une phase de compilation, ce qui est avantageux par rapport à une compilation directe pour des raisons de portabilité. La compilation produit du code particulièrement efficace puisque les programmes OC ont pour structure de contrôle un automate d'états fini, qui plus est minimal. Ceci leur confère une grande rapidité d'exécution et surtout permet de calculer leur temps de réaction à une sollicitation de l'environnement. C'est une propriété importante vu que les langages synchrones sont utilisés essentiellement pour des applications temps réel.

De façon comportementale, un programme synchrone est une machine transformant instantanément un vecteur d'entrées en un vecteur de sorties. Cette machine est plongée dans son environnement qui est par nature asynchrone : les stimuli qu'il envoie au programme n'ont a priori aucun ordre. En ce qui concerne les entrées, nous en distinguons deux types, les entrées continues et les entrées impulsionnelles :

- Une entrée continue se comporte de manière analogue au potentiel d'un fil électrique : on

peut en particulier toujours lire la valeur d'une entrée continue.

- Une entrée impulsionnelle est au contraire considérée comme un événement : l'entrée est présente ou ne l'est pas ; l'événement "l'entrée est présente" est donc assimilable à une impulsion de Dirac.

Enfin les circuits synchrones ont des entrées continues qu'ils échantillonnent sur un front d'horloge : nous pouvons donc les considérer comme des entrées impulsionnelles.

Les entrées arrivent sans aucun ordre préétabli. Pourtant le programme doit obéir à l'hypothèse de synchronisme, à savoir : *les entrées sont synchrones entre elles et avec les sorties.* Nous avons déjà étudié la sémantique temporelle des langages synchrones à la section 5.1. Il en découle que le programme a un comportement cyclique, chaque cycle représentant un instant logique. Il faut donc faire la liaison entre le temps physique de l'environnement et le temps logique du programme synchrone. Pour cela, il faut saisir les entrées venant de l'environnement et les fournir de façon présentable au programme, de telle sorte qu'elles lui apparaissent comme étant simultanées. Autrement dit, il faut une interface chargée de transformer les entrées venant de l'univers asynchrone en vecteurs d'entrées pour le programme synchrone.

9.1.1 Interfaces synchrone/asynchrone centralisées

La contrainte pesant sur l'interface synchrone/asynchrone est que le programme doit être réactif par rapport à son environnement. La solution la plus générale, déjà étudiée dans [26], est constituée de deux éléments (figure 9.1) :

- des gestionnaires d'événements spécialisés chargés de capter les événements d'entrée et de ranger les valeurs ainsi lues dans une file d'attente unique, ces événements apparaissant dans leur ordre chronologique ; autrement dit, les gestionnaires ont pour tâche de sérialiser les entrées.
- un générateur de vecteurs d'entrées qui :
  - ne fait rien tant que la file d'attente est vide,
  - extrait de la file d'attente le mot maximal appartenant au langage d'entrée du programme synchrone et élabore le vecteur correspondant,
  - et enfin active le programme synchrone.

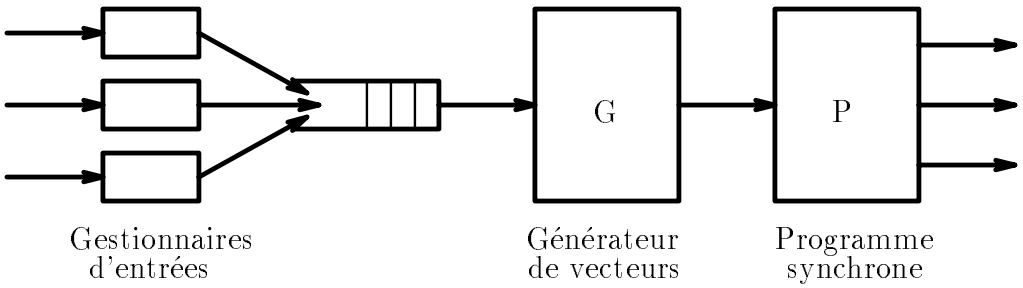


Figure 9.1: Interface synchrone/asynchrone centralisée

Le programme synchrone et le générateur de vecteurs s'échangent le contrôle : un seul à la fois est actif. Le programme exécute une transition de son automate dès qu'il reçoit un vecteur d'entrées, et rend la main au générateur après avoir fini sa transition. L'automate seul est un programme transformationnel, mais l'ensemble constitué de l'interface et du programme a bien un comportement réactif.

Enfin, le générateur doit respecter les relations existant entre les entrées (implications ou exclusions en ESTEREL, exclusions en ARGOS, contraintes d'horloges en LUSTRE et SIGNAL) quand il y en a. Les relations se sont donc pas des contraintes sur l'environnement, mais plutôt des contraintes à respecter par l'interface. Quand il n'y a aucune relation entre les entrées, le générateur élabore un nouveau vecteur d'entrées chaque fois que la file d'attente n'est pas vide en extrayant de cette file le préfixe le plus long ne contenant pas deux fois la même entrée.

Une autre solution a été proposée pour l'atelier SAGA [42]. Au lieu d'avoir une file d'attente, chaque gestionnaire met à jour un tampon avec la dernière valeur de son signal. Quant au générateur, il scrute les tampons et élabore des vecteurs d'entrées à partir de ce qu'il a lu. Cette solution est bien adaptée aux entrées continues, ce qui est le cas du projet CO3N4. Elle n'est toutefois pas satisfaisante dans le cas d'entrées impulsionnelles.

Enfin dans [48], M-A.Péraldi réalise une étude complète sur les interfaces synchrone/asynchrone. Elle présente deux modes d'exécution des programmes synchrones :

- Un **mode séquentiel** similaire à ce que nous venons de présenter.
- Un **mode parallèle** où des nouvelles valeurs des entrées arrivées pendant une exécution de l'automate peuvent être prises en compte par la transition courante.

9.1.2 Problèmes posés par les relations

Les relations sur les signaux d'entrée sont de deux types :

- les exclusions qui spécifient que deux (ou plus) entrées ne sont jamais présentes en même temps ;
- les implications qui indiquent que quand une entrée (maître) est présente, alors forcément une autre entrée (esclave) l'est également.

En ESTEREL il est possible de spécifier directement des relations sur les signaux d'entrées. Ces relations servent d'une part à optimiser le code OC. D'autre part elles apparaissent telles quelles dans le programme OC résultat pour des vérifications ultérieures. Les exclusions en ARGOS apparaissent également telles qu'elles dans le programme OC résultat. En LUSTRE nous distinguons les relations et les contraintes d'horloges. Les relations sont écrites sous forme d'assertions mais peuvent être plus compliquées que des simples exclusions ou implications. Aussi le compilateur LUSTRE ne produit pas de relation OC : il utilise ces assertions pour simplifier la structure de contrôle de l'automate OC [52]. Par conséquent, les assertions sont des contraintes sur l'environnement et l'interface du programme n'a pas à en tenir compte. Quant aux contraintes d'horloges, elles se traduisent en relations sur les entrées. Par exemple, si les entrées A et B ont

la même horloge, nous en déduisons la double implication $A => B$ et $B => A$. Enfin en SIGNAL les relations sont également des contraintes d'horloges sur les entrées.

9.1.3 Un exemple d'exclusion

Considérons l'exemple d'un programme OC à trois entrées A, B et C, liées par l'exclusion $A\#B$. Supposons que le générateur de vecteurs d'entrées trouve dans sa file d'attente les valeurs suivantes (dans l'ordre) : a, a, b, c, b. La présence de deux a successifs le force à élaborer un vecteur avec a comme seule entrée disponible. Si les entrées sont impulsionnelles, alors pour chaque entrée absente, le générateur complète le vecteur avec ε. Si ce sont des entrées continues, alors pour chaque entrée absente, le générateur complète le vecteur avec la valeur précédente de l'entrée. Dans le cas d'entrées impulsionnelles, il génère les vecteurs suivants :

- $(a, \varepsilon, \varepsilon)$ à cause de la présence du second a,
- $(a, \varepsilon, \varepsilon)$ à cause de la présence du b et de l'exclusion,
- (ε, b, c) à cause de la présence du second b.

Maintenant le générateur peut lire les entrées dans la file d'attente plus vite qu'elles n'arrivent : dans ce cas il peut parfois trouver la file d'attente vide. Nous décidons de représenter cette absence de valeur par le symbole $\perp$. Le générateur de vecteurs peut par exemple trouver dans sa file d'attente les valeurs suivantes (dans l'ordre) : a, a, b, $\perp$, c, b. Or le programme doit être réactif, c'est-à-dire qu'il doit réagir aux sollicitations de l'environnement. Un $\perp$ indique que deux sollicitations sont éloignées dans le temps, et par conséquent cela doit forcer le générateur à élaborer un nouveau vecteur avec les valeurs lues avant le $\perp$. Dans notre cas, il génère les vecteurs suivants :

- $(a, \varepsilon, \varepsilon)$ à cause de la présence du second a,
- $(a, \varepsilon, \varepsilon)$ à cause de la présence du b et de l'exclusion,
- $(\varepsilon, b, \varepsilon)$ à cause du $\perp$.

Nous en concluons que la programmation synchrone n'élimine pas totalement l'indéterminisme. Elle ne fait que le reporter au niveau de l'interface synchrone/asynchrone.

9.1.4 Un exemple d'implication

Le cas des implications est plus difficile puisque certaines successions de valeurs d'entrées peuvent provoquer une erreur, comme en atteste l'exemple suivant. Soit en effet un programme OC à trois entrées impulsionnelles A, B et C, liées par l'implication $A => B$. Si la séquence des valeurs lues par le générateur est b, $\perp$, a, c et c, alors ce dernier fabrique les vecteurs suivants :

- $(\varepsilon, b, \varepsilon)$ à cause de la présence du $\perp$,

- ERREUR à cause de l'absence de b alors que a est présent.

On serait ainsi forcé de considérer qu'un programme synchrone est une machine partielle. En fait, on estime plutôt que deux entrées impulsionnelles liées par une implication sont lues sur un unique port d'entrée, et que l'environnement les fournit au programme en même temps. Autrement dit, les implications sont perçues comme des contraintes sur l'environnement. Par exemple, si nous avons l'implication $A => B$, alors le gestionnaire chargé du port d'entrée de A et B lira ou bien la valeur b toute seule, ou bien les valeurs a et b en même temps. De son côté le générateur lira dans sa file d'attente b ou ab.

Si au contraire une implication lie deux entrées continues, alors quand le générateur lit dans sa file d'attente une valeur pour l'entrée maître seule, il complète avec la valeur précédente de l'entrée esclave.

En fin de compte, les seules relations dont doit tenir compte le générateur sont les exclusions.

9.1.5 Génération automatique des interfaces

Actuellement, le programme synchrone est le seul à être dans le "monde synchrone" :

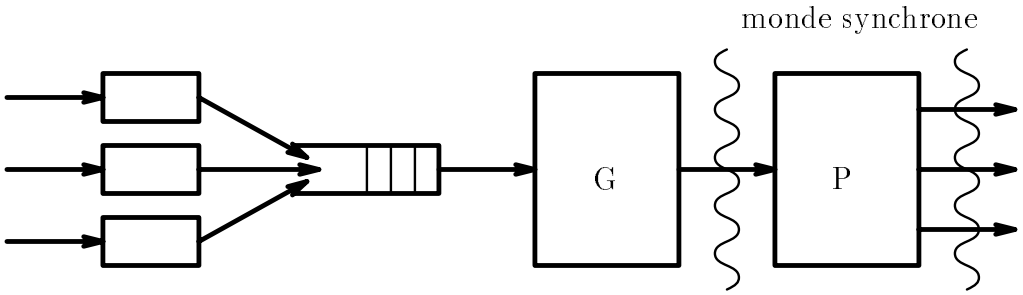


Figure 9.2: Interface non générique

Programmer de façon générique et dans un langage synchrone l'interface du programme permet donc de l'inclure dans le "monde synchrone" :

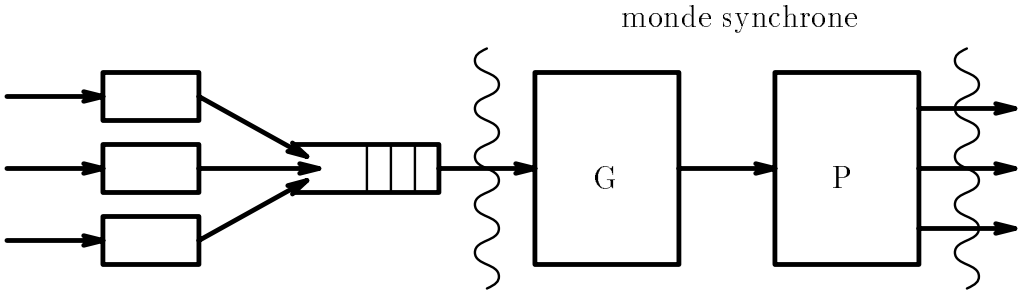


Figure 9.3: Interface générique

La difficulté de la programmation d'une telle interface consiste à se rappeler de ce qui a été lu précédemment dans la file d'attente. De façon intuitive, le générateur reçoit une par une les valeurs contenues dans la file d'attente, et dès que c'est nécessaire élabore un vecteur à partir de ces valeurs; il a donc un comportement cyclique. Plus précisément, pour chaque entrée il faut savoir :

- si elle est présente dans la file d'attente (c'est-à-dire si elle a ou non été reçue depuis le dernier vecteur généré),
- si elle doit être utilisée pour le prochain vecteur, et
- si elle a été consommée pour le dernier vecteur (dans le cas où un vecteur vient juste d'être élaboré).

Pour produire automatiquement les interfaces synchrone/asynchrone, nous avons retenu le langage synchrone LUSTRE, de façon purement arbitraire. Certes, le générateur étant lui-même un programme synchrone, il lui faut une interface synchrone/asynchrone. Mais cette fois-ci, l'interface se contente de lire les entrées dans sa file d'attente et de les envoyer séquentiellement au générateur. Elle est donc réduite au strict minimum. Les avantages sont ceux de la programmation synchrone : le générateur est déterministe et est implantable par un automate d'états fini.

Nous définissons donc un nœud LUSTRE `EVENT` chargé de déterminer pour chaque entrée quand elle doit être consommée par l'interface du programme synchrone. Puis nous définissons le nœud `INTERFACE` qui constitue l'interface du programme.

9.1.6 Le nœud `EVENT`

Pour chaque entrée `E`, nous déclarons trois booléens `PE`, `GOE` et `CE` indiquant respectivement la présence de l'entrée, son utilisation, et sa consommation. Le meilleur moyen de décrire le comportement de `PE` et `GOE` est de donner l'automate d'états fini qui les fabrique :

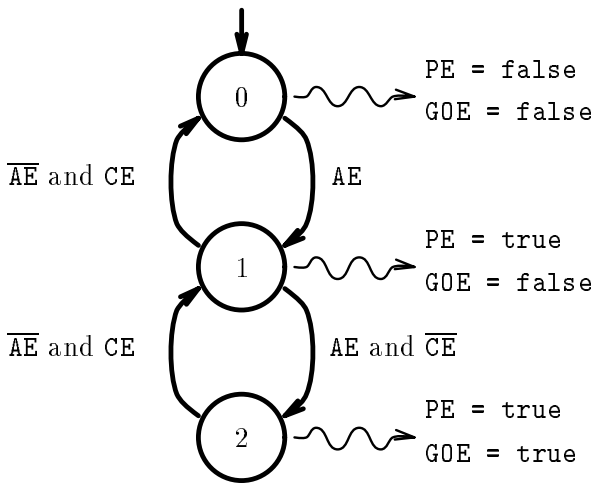


Figure 9.4: Automate calculant PE et GOE

Où AE signale que l'entrée E est arrivée. De l'automate de la figure 9.4 nous déduisons facilement le nœud LUSTRE qui calcule PE et GOE :

```
node EVENT
    (
        AE : bool;
        CE : bool
    )
    returns
    (
        PE : bool;
        GOE : bool
    );
let
    PE = false ->
         if (not pre PE and AE) then true
         else if (pre PE and not pre GOE and not AE and CE) then false
         else pre PE;
    GOE = false ->
          if (pre PE and not pre GOE and AE and not CE) then true
          else if (pre GOE and not AE and CE) then false
          else pre GOE;
tel.
```

9.1.7 Le nœud INTERFACE

Nous avons vu à la section 9.1.1 que l'interface d'un programme synchrone reçoit une entrée unique. Nous déclarons donc pour le nœud INTERFACE une seule entrée : DONNEE. Nous déclarons en outre en sortie le booléen GO signalant qu'il faut élaborer un nouveau vecteur d'entrée. Nous avons alors pour chaque entrée E libre de toute exclusion :

```
(PE,GOE) = EVENT(DONNEE = E, pre CE);
CE = PE and GO;
```

Dans le cas de deux entrées E1 et E2 exclusives, nous devons en plus tenir compte du fait que la présence de l'une provoque la fabrication d'un vecteur si l'autre a été lue auparavant dans la file d'attente :

```
CE1 = PE1 and GO and not(PE2 and AE1);
CE2 = PE2 and GO and not(PE1 and AE2);
```

Si plus que deux entrées sont mutuellement exclusives, cela complique bien entendu ces équations. Enfin, il reste à calculer GO, en l'absence d'exclusion :

```
    G0 =  G0E1
       or G0E2
       :
       or G0EN
       or BOTTOM;
```

où `BOTTOM` indique la présence de $\perp$ dans la file d'attente. Si nous avons l'exclusion `E1 # E2`, alors cela devient :

```
    G0 =  G0E1
       or G0E2
       :
       or G0EN
       or (PE1 and PE2)
       or BOTTOM;
```

Il reste à étudier comment activer le programme synchrone. Comme tout programme OC, chacune de ses entrées `ENT` est positionnée au moyen de la procédure `I_ENT`. Nous avons de plus une procédure `AUTOMATE` qui active l'automate du programme synchrone. Ces procédures sont écrites dans le langage cible (C, ADA, ...). Nous définissons donc la fonction LUSTRE externe `FAIRESORTIE` chargée d'effectuer toutes les opérations nécessaires à l'activation du programme synchrone. Finalement, nous avons le nœud principal suivant (toujours avec l'exclusion `E1 # E2`) :

```
node INTER
    (
        DONNEE : TYPE_ENTREE
    )
    returns
    (
        G0 : bool;
        (ACTIVE when G0) : bool
    );
var PE1, G0E1, CE1 : bool;
    PE2, G0E2, CE2 : bool;
     :
    PEN, G0EN, CEN : bool;
let
    (PE1, G0E1) = EVENT(DONNEE = E1, pre CE1);
    (PE2, G0E2) = EVENT(DONNEE = E2, pre CE2);
     :
    (PEN, G0EN) = EVENT(DONNEE = EN, pre CEN);
    CE1 = PE1 and G0 and not(PE2 and (DONNEE = E1));
    CE2 = PE2 and G0 and not(PE1 and (DONNEE = E2));
     :
    CEN = PEN and G0;
```

```
    ACTIVE = FAIRESORTIE((CE1, CE2, ... , CEN) when GO);
    GO =  GOE1
       or GOE2
       :
       or GOEN
       or (PE1 and PE2)
       or (DONNEE = BOTTOM);
tel.
```

9.2 Interfaces synchrone/asynchrone réparties

L'interface assure que le programme synchrone est réactif à l'environnement asynchrone dans lequel il est plongé (section 9.1.1). Dans le cas où le programme synchrone a été réparti fonctionnellement (chapitre 3) avec synchronisation totale des branchements (section 4.4.8), cela implique que chaque composant doit réagir en même temps et un même nombre de fois. Le second problème qui se pose consiste à fournir à chaque composant un vecteur d'entrées qui lui soit propre, tout en assurant que le comportement global soit le même que celui du programme centralisé.

9.2.1 Solution centralisée

Nous pouvons adapter une interface centralisée à un programme réparti : il suffit pour cela que l'interface partitionne les vecteurs d'entrées en autant de sous-vecteurs qu'il y a de sites. L'ensemble des ports d'entrée est partitionné en n ensembles, un pour chaque composant du programme réparti, cette partition résultant des directives de répartition. Chaque sous-vecteur est donc obtenu par projection du vecteur d'entrées sur l'ensemble des ports d'entrée du composant. Nous obtenons ainsi la situation suivante :

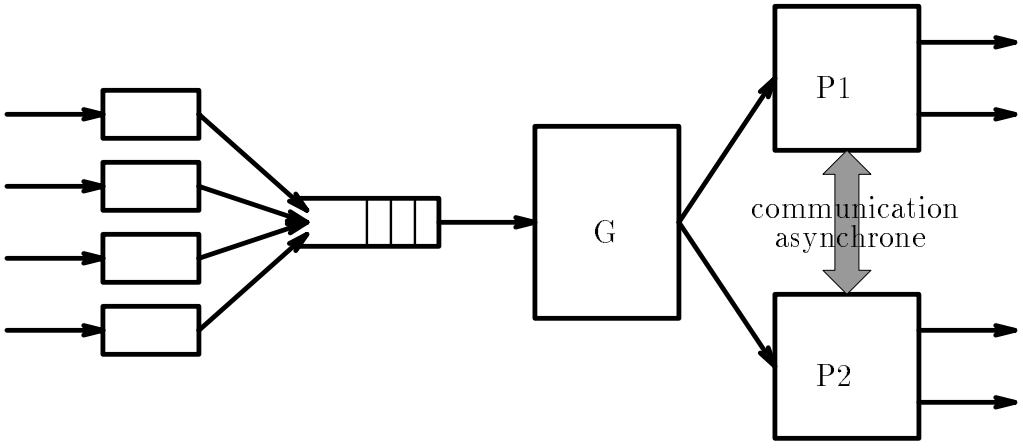


Figure 9.5: Interface centralisée avec un programme réparti

Supposons quatre entrées impulsionnelles - A, B, C et D - les deux premières appartenant au site 1 et les deux dernières au site 2. Supposons que les valeurs arrivent sur les ports d'entrée

puis dans la file d'attente de la façon suivante :

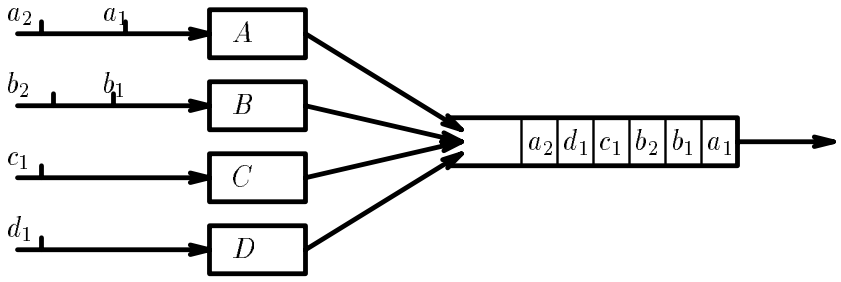


Figure 9.6: Interface avec quatre ports d'entrée

Le générateur de vecteurs produit alors successivement les deux vecteurs $(a_1, b_1, \varepsilon, \varepsilon)$ et (a_2, b_2, c_1, d_1). Par projection, nous obtenons d'abord les sous-vecteurs (a_1, b_1) et $(\varepsilon, \varepsilon)$, puis les sous-vecteurs (a_2, b_2) et (c_1, d_1).

Une telle interface garantit bien un comportement équivalent à celui du programme centralisé. Mais ce n'est pas une solution répartie. Aussi notre but est-il de répartir également l'interface du programme, afin que chaque composant du programme réparti soit complètement indépendant et donc implantable séparément.

9.2.2 Première solution répartie "naïve"

Nous pouvons envisager une première solution naïve utilisant le fait que les programmes répartis sont synchronisés (chapitre 5). Cela consiste à projeter le générateur sur les sous-ensembles des entrées de chaque site. Nous obtenons ainsi un générateur de vecteurs d'entrées pour chaque site :

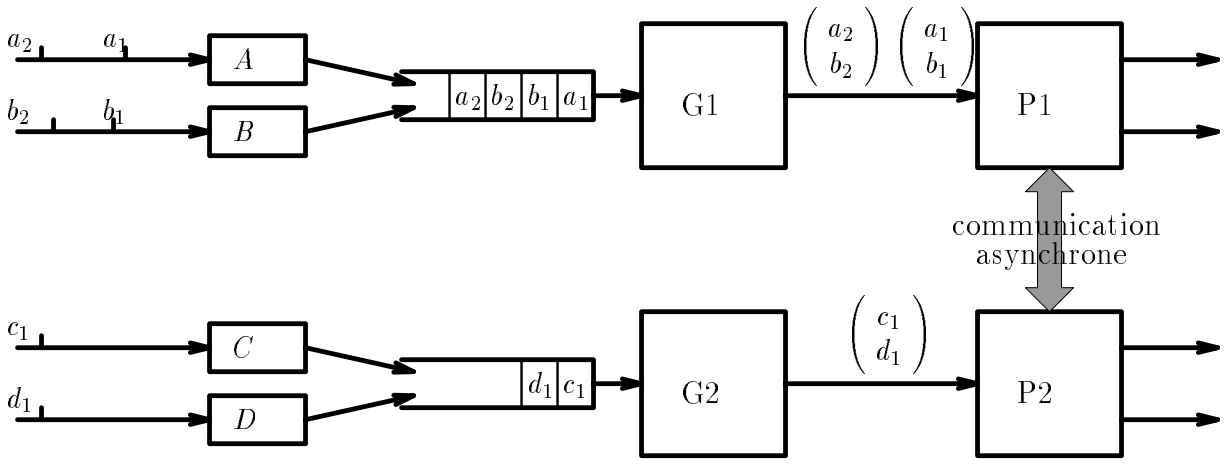


Figure 9.7: Interface répartie avec un programme réparti

Nous conservons les valeurs arrivant sur les ports d'entrée de la figure 9.6. Le premier vecteur d'entrées généré par le couple G1/G2 est donc (a_1, b_1, c_1, d_1). L'ensemble de ces deux interfaces réparties n'est donc pas équivalent à l'interface centralisée. De plus, l'idéal serait que G1 et G2 envoient des vecteurs d'entrées synchrones, ce qui n'est pas le cas à cause du retard des entrées $c1$ et $d1$ sur $a1$ et $b1$.

En fait, il faudrait forcer G2 à générer prématurément un vecteur au moment où G1 envoie à P1 le vecteur (a_1, b_1). Dans un tel cas, ou bien le générateur n'a pris en compte aucune nouvelle valeur de ses entrées et il doit produire un vecteur vide (ici le vecteur $(\varepsilon, \varepsilon)$), ou bien il a pu prendre en compte des valeurs pour une partie de ses entrées et il doit compléter son vecteur avec des ε.

Maintenant considérons l'exemple d'un programme à deux entrées A et B, sans relation entre elles, et réparties de la façon suivante : A sur le site 1 et B sur le site 2. Supposons que l'entrée A arrive beaucoup plus souvent que l'entrée B. La condition de réactivité impose que chaque générateur (G1 pour le site 1 et G2 pour le site 2) soit capable de prendre en compte toutes les entrées de son site. Dans la mesure où les deux composants du programme réparti (P1 et P2) sont synchronisés, ils effectuent exactement le même nombre de cycles. Deux cas peuvent se présenter :

- G1 et G2 s'exécutent à la même vitesse : celle de l'entrée la plus "rapide" A.
  Dans ce cas les deux composants P1 et P2 sont activés le même nombre de fois, même si P2 l'est avec un vecteur vide.
- G1 et G2 s'exécutent chacun à la vitesse de son entrée propre.
  G1 s'exécute à la vitesse de l'entrée "rapide" et G2 à la vitesse de l'entrée "lente". Dans ce cas P1 et P2 vont évoluer à la vitesse de l'entrée "lente" B, et il y aura accumulation des valeurs de l'entrée "rapide" A dans la file d'attente de G1.

Pour éviter le débordement de la file d'attente de G1, il faut forcer G2 à élaborer un vecteur en même temps que G1. Autrement dit, il faut que les programmes répartis évoluent au rythme de l'entrée "rapide" et non à celui de l'entrée "lente".

9.2.3 Seconde solution répartie "réactive"

Nous venons donc de mettre en évidence la nécessité d'une synchronisation entre les générateurs de vecteurs d'entrées d'un programme réparti, afin qu'ils produisent des vecteurs synchrones et si possible équivalents à ce qu'un générateur centralisé aurait produit. Plusieurs solutions sont envisageables pour aboutir à la synchronisation recherchée. Celle que nous présentons ici est simple et s'est montrée efficace lors des tests.

Nous supposons qu'un générateur est capable de savoir si son programme a reçu une donnée dans un des canaux asynchrones qui lui permettent de communiquer avec les autres sites. Si c'est le cas, il faut générer un nouveau vecteur d'entrées à chaque fois que :

- les entrées lues dans la file d'attente permettent d'élaborer un vecteur valide : dans ce cas nous activons le programme avec ce vecteur,

- les entrées lues dans la file d'attente ne permettent pas d'élaborer un vecteur valide (parce qu'il manque des valeurs) mais une donnée a été détectée dans un des canaux dirigé vers le programme : dans ce cas nous activons le programme avec le vecteur constitué des seules valeurs présentes.

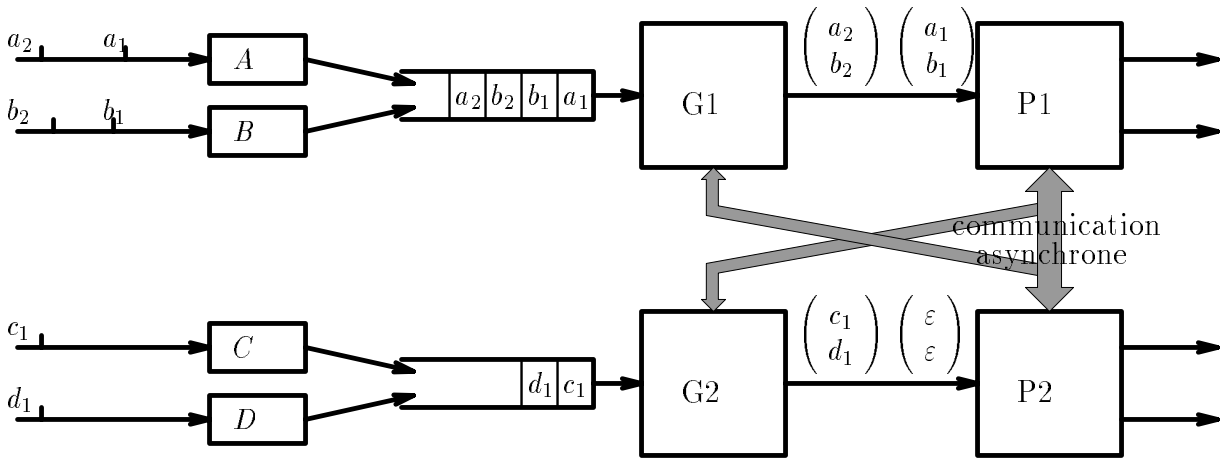


Figure 9.8: Interface répartie avec lien asynchrone

Cette solution est satisfaisante dans la mesure où elle génère des vecteurs équivalents à ce qu'un générateur centralisé aurait produit avec les mêmes entrées. Ce n'était pas le cas de la première solution répartie (section 9.2.2). Elle assure de plus que le programme réparti est réactif puisque dès qu'un composant réagit à un stimulus de l'environnement, tous les autres sont forcés à réagir, même si leurs entrées propres sont absentes.

Toutefois, l'indéterminisme, intrinsèque aux interfaces synchrone/asynchrone, empêche de démontrer l'équivalence de comportement entre une telle interface répartie et l'interface centralisée.

9.2.4 Les relations réparties

Reste le problème des relations sur les entrées, problème que nous avons déjà étudié dans le cas centralisé à la section 9.1.2.

- Soit l'implication $A => B$.
  Nécessairement les entrées A et B arrivent sur le même port, donc A et B doivent être localisées sur le même site.

- Soit l'exclusion $A\#B$.
  La solution que nous venons de proposer ne peut respecter une telle relation. En effet, si A et B appartiennent à deux sites distincts, alors rien n'empêche leur générateur respectif d'activer simultanément leur programme, l'un avec un vecteur contenant une occurrence de A, et l'autre avec un vecteur contenant une occurrence de B.

Le respect des exclusions exige par conséquent une synchronisation entre les générateurs beaucoup plus lourde que celle proposée ici, ce dont nous ne voulons pas pour des raisons d'efficacité.

9.3 Création des processus de communication

La méthode d'exécution d'un programme OC présentée dans la section précédente peut aussi être appliquée à des programmes OC répartis. Nous obtenons ainsi un exécutable pour chaque site spécifié dans les directives de répartition. Toutefois, nous avons vu au chapitre 3 que les programmes répartis utilisent des primitives de communication pour échanger des variables. Ces primitives sont des fonctions externes OC de deux sortes :

- les émissions se font au moyen de l'action `put(destination,valeur)`,
- les réceptions se font au moyen de l'action `variable:=get(source)`,

Le moyen utilisé est une file d'attente "fifo" unidirectionnelle. Nous avons donc besoin de deux files d'attente pour chaque paire d'exécutables, une dans chaque sens.

La solution que nous avons retenue pour mettre en œuvre une telle file d'attente est la "socket" UNIX qui respecte les deux contraintes requises : préservation de l'ordre et de l'intégrité des messages (hypothèse 3.1 page 39). Chaque "socket" est créée par deux descripteurs, un pour le processus qui écrit, et un pour celui qui lit.

Maintenant, l'implantation physique finale des exécutables est indépendante des directives de répartition : l'utilisateur peut par exemple vouloir tester l'exécution du programme réparti d'abord sur une machine centralisée, avant de passer à une architecture répartie. C'est pour cela que nous parlons de répartition en processus plutôt qu'en processeurs.

Or une "socket" reliant deux processus exécutés par la même machine doit être créée dans le domaine UNIX ("socket" locale), alors que si les deux processus sont exécutés par deux machines distinctes, elle doit l'être dans le domaine INTERNET ("socket" distante). Nous détectons au moment de l'exécution dans quel domaine doit être créée chaque "socket". Pour cela la ligne de commande comporte le nom de toutes les machines composant l'architecture d'exécution (voir le chapitre 10 pour plus de détails sur la ligne de commande du répartiteur de code).

9.4 Chaîne de compilation

La répartition en processus permet de faire de la répartition logicielle, indépendamment de l'implantation matérielle. La procédure de création des files d'attente, décrite à la section 9.3 est contenue dans le fichier `distinet.c`.

Comme tous les programmes OC, les programmes répartis par `oc2rep` nécessitent une interface chargée d'espionner l'environnement et d'activer l'automate chaque fois que les réactions de cet environnement l'exigent. Les directives de répartition ont donc pour effet de partitionner

l'ensemble des signaux et capteurs en autant de sous-ensembles qu'il y a site. Et pour chaque site il faut écrire une interface capable de scruter les signaux et capteurs qui lui incombent.

Nous illustrons l'enchaînement des étapes de la compilation répartie avec le programme `prog.oc`. Les directives de répartition sont dans `prog.rep`. Les deux interfaces sont respectivement dans `prog.2.0.main.c` et `prog.2.0.main.c`. Les fonctions réalisant les procédures de sortie de l'automate, appelées à chaque action `output`, sont dans `prog.2.0.h` et `prog.2.1.h`.

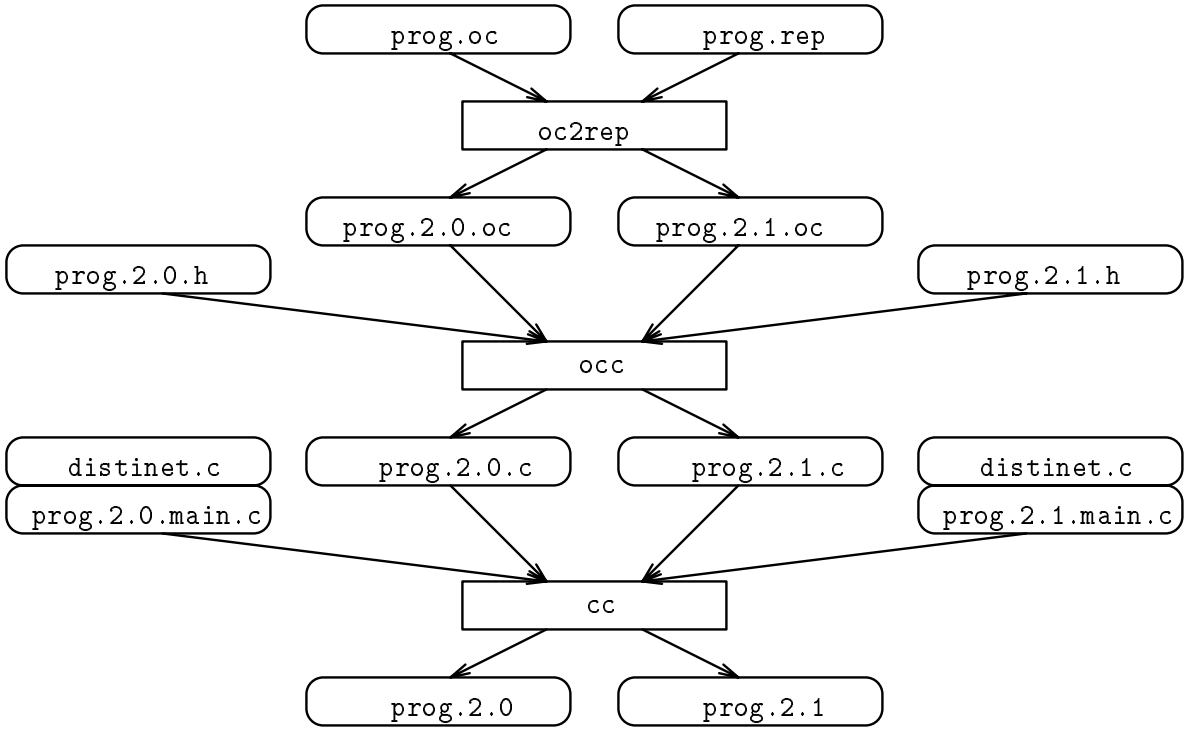


Figure 9.9: Chaîne de compilation des programmes synchrones

Enfin le chapitre 10 présente en détails le répartiteur de code `oc2rep`.

9.5 Conclusion

Nous avons donc étudié les problèmes liés à l'implémentation des programmes synchrones répartis. Le principal problème vient de ce que l'asynchronisme intrinsèque de l'environnement s'oppose au synchronisme du programme. Une interface synchrone/asynchrone est chargée de regrouper les événements d'entrées en vecteurs de telle sorte que les entrées apparaissent au programme comme étant simultanées. Elle réalise la liaison entre le temps physique de l'environnement et le temps logique du programme.

Après avoir présenté les interfaces synchrone/asynchrone centralisées, nous avons proposé une solution satisfaisante d'interface répartie. Cette étude nous a amené à conclure que les entrées liées par une quelconque relation doivent nécessairement être implantées sur le même site.

Chapitre 10

Le répartiteur `oc2rep`

Ce dernier chapitre expose nos réalisations pratiques. Le répartiteur `oc2rep` a été écrit en C++ et fonctionne actuellement sur station SUN-SPARC. Il représente 10000 lignes de code pour un exécutable de 900 Ko. Ce répartiteur crée des programmes OC répartis à partir d'un programme OC centralisé et de directives de répartition. Nous présentons successivement le répartiteur `oc2rep`, les directives de répartition, la ligne de commande et les options d'exécution, et enfin l'environnement d'exécution.

10.1 Le répartiteur

`oc2rep` opère comme un post-processeur des compilateurs des divers langages synchrones. Il prend en entrée :

- un fichier OC contenant la représentation interne sous forme d'automate du programme et suffixé par "`.oc`",
- et un fichier contenant les directives de répartition de l'utilisateur et suffixé par "`.rep`".

En sortie, il produit `n` fichiers OC, un pour chaque site de calcul. Chacun de ces fichiers contient un programme OC complet avec des appels aux primitives de communication. Celles-ci servent à un site donné à échanger avec les programmes des autres sites les valeurs nécessaires à ses calculs.

10.2 Les directives de répartition

Les directives de répartition permettent à l'utilisateur de spécifier répartition désirée. Elles doivent être regroupées dans un fichier dont la syntaxe est :

```
<fichier>       ::= sites: <entier> <listedesites>
```

```
<listedesites>  ::= <site> | <listedesites> <site>
<site>          ::= <entier> signals: <liste> sensors: <liste>
<liste>         ::= (<listedentiers>)
<listedentiers> ::= |<entier>|<listedentiers>, <entier>
```

Ce fichier indique pour chaque site la liste des signaux et des capteurs ("sensor" dans la terminologie OC) appartenant à ce site. La contrainte principale est que tout signal et tout capteur appartienne à un et un seul site. Il faut également au moins un signal ou capteur par site. Il eut été plus ergonomique de désigner les signaux par leurs noms, c'est-à-dire le nom des variables d'entrée ou de sortie du nœud LUSTRE d'origine, mais cela ne se serait pas étendu aux signaux locaux et aux exceptions (uniquement en ESTEREL) qui sont anonymes.

Pour placer une variable sur un site, il faut donc que ce soit une variable d'entrée/sortie, et il faut aller lire dans le code OC le numéro du signal correspondant ! Toutefois, une version ultérieure du répartiteur permettra de localiser directement les variables, sans passer par les signaux.

10.3 La ligne de commande

La ligne de commande a le format suivant :

```
oc2rep [ -sb st ] [ -a ] [ -e E ] [ -o ] nom.oc
```

Le répertoire courant doit contenir les deux fichiers `nom.oc`, programme OC à répartir, et `nom.rep`, contenant les directives de répartition. Les options offertes sont les suivantes :

- `-sb` pour que les programmes répartis soient synchronisés *si besoin* ; autrement dit, seuls les programmes des sites qui communiquent pour des échanges de variables sont synchronisés.
- `-st` pour que les programmes répartis soient synchronisés *totalement* ; ici tous les programmes répartis sont synchronisés.
- `-a` pour que les émissions de messages soient effectuées *au plus tôt* ; ainsi, une variable est émise juste après avoir été calculée ; la réception étant quant à elle effectuée *au plus tard*, cela permet de minimiser le temps d'attente imposé par la transmission à travers le réseau.
- `-e` pour que les émissions redondantes de messages soient supprimées localement dans chaque état ; si cette option est choisie, le répartiteur élimine l'émission d'une variable quand celle-ci n'a pas été recalculée depuis l'émission précédente ; attention, cette option est incompatible avec l'option précédente puisqu'il ne peut pas y avoir d'émission redondante quand les émissions de messages sont effectuées *au plus tôt*.
- `-E` pour que l'élimination des émissions redondantes soit globale, c'est-à-dire qu'elle se fasse en tenant compte des messages reçus dans les états précédant et non dans le seul état courant ; cette option n'est pas incompatible avec l'option `-a`.

- `-o` pour que les tests dont les deux branches sont observationnellement équivalentes (une fois effectuée la répartition) soient supprimés ; attention, il faut que les émissions soient placées *au plus tard*, c'est-à-dire que l'option `-a` ne soit pas sélectionnée.

La commande

```
oc2rep nom.oc
```

crée les fichiers OC suivants : `nom.n.i.oc` où `n` désigne le nombre total de sites, et `i` désigne les numéros des sites. Par convention, les sites sont numérotés de `0` à `n-1`. Si `oc2rep` ne peut pas localiser de façon univoque une action de calcul sur un site, alors il refusera de répartir le programme. Le remède consiste à faire sortir des variables locales pour en préciser la localisation. Il faut alors leur associer des actions de sortie vides.

Ensuite chaque fichier `nom.n.i.oc` doit être compilé séparément. Nous obtenons ainsi `n` processus que nous pouvons exécuter sur au plus `m` processeurs. Ceci est cohérent avec la distinction que nous avons faite à la section 9.3 entre répartition logicielle et implantation matérielle.

A la fin de la chaîne de compilation, nous obtenons `n` exécutables `nom.n.i`, un pour chaque site. La ligne de commande servant à invoquer l'exécutable du site `i+1` est alors :

```
nom.n.i processeur1 processeur2 ... processeurN
```

La connaissance des processeurs sur lesquels sont exécutés les autres programmes permet de créer à l'exécution les processus de communication (section 9.3).

10.4 L'environnement d'exécution

Les fichiers OC répartis font appel aux fonctions externes d'envoi et de réception de messages suivantes :

```
put_void(ss)
char* ss;

put_type(ss,x);
char* ss;
type x;

get_void(s)
char* s;

get_type(x,s)
type* x;
char* s;
```

où `type` peut être soit un type prédéfini OC (`integer`, `string`, `boolean`, `float` et `double`), soit un type externe du programme LUSTRE, et où `s` spécifie le site d'où recevoir le message, de la forme `_<entier>`.

L'exécution par le site `i` d'une action `put` consiste à mettre le message dans la file `i` de chacun des sites spécifiés par `ss`. L'exécution par un site d'une action `get` consiste à attendre si la file de numéro spécifié par `s` est vide, et sinon à retirer un message de cette file et à l'affecter à son argument `*x`, sauf dans le cas de `void` (les messages `void` étant non valués).

Conclusion

L'objet de cette thèse était d'aboutir à l'exécution répartie, sur une machine à mémoire distribuée, d'un programme synchrone.

L'approche synchrone s'inscrit dans le cadre de la programmation des systèmes dits *réactifs* dans le sens où ils subissent des contraintes temporelles fixées par leur environnement. D'une part, la classe des systèmes réactifs englobe la plupart des systèmes temps réels, ce qui impose des contraintes de sûreté de fonctionnement aux programmes. D'autre part, les travaux sur la compilation des langages synchrones ont conduit à proposer un format interne de représentation des programmes, caractérisé par une structure de contrôle en automate d'états fini et dont les transitions sont des programmes séquentiels. Ce format interne, OC, est le format de sortie des compilateurs ESTEREL, LUSTRE et ARGOS. En outre, les récents travaux sur le code commun des langages synchrones permettent d'ajouter à cette liste le langage SIGNAL.

L'existence du format OC, format de représentation interne des programmes synchrones au moyen d'automates, nous a conduits plus spécialement à étudier la répartition pour des programmes dont la structure de contrôle soit sous la forme d'un automate d'états fini.

Principaux apports

Les apports auxquels a donné lieu ce travail se situent dans plusieurs domaines : la répartition fonctionnelle, la répartition minimale, les fondements de la répartition, l'exécution répartie des programmes synchrones et les réalisations pratiques.

La répartition fonctionnelle (chapitres 3 à 6)

La première partie de ce travail porte sur la répartition fonctionnelle. Le principe consiste à sélectionner pour chaque site les calculs à faire en fonction des directives de répartition fournies par l'utilisateur, puis à faire communiquer harmonieusement les fragments ainsi créés. Notre algorithme de répartition fonctionnelle (chapitre 3) repose donc sur trois étapes : projection des actions sur leur site de calcul, placement des émissions et placement des réceptions.

Pour prouver formellement notre algorithme de répartition fonctionnelle (chapitre 4), nous avons modélisé les programmes OC par des systèmes de transitions étiquetées par les actions de ces programmes. Ceci a été aisé dans la mesure où leur structure de contrôle est un automate d'états

fini comportant dans chaque état du code purement séquentiel. Par la suite, nous avons modélisé les dépendances de données entre les actions par des ordres partiels étiquetés par les actions des programmes. Or ce sont ces dépendances qui donnent lieu en fin de compte aux communications entre les fragments de programmes répartis. Ainsi nous avons pu modéliser l'algorithme de répartition fonctionnelle comme une suite d'opérations à effectuer sur des systèmes de transitions étiquetés par des ordres partiels.

L'étude des programmes répartis fonctionnellement nous a permis de constater que d'une part l'équivalence temporelle avec le programme centralisé initial n'était pas respectée, et d'autre part certains messages échangés entre les fragments du programme réparti étaient redondants.

Nous avons résolu le problème de la non équivalence temporelle (chapitre 5) par l'ajout de communications supplémentaires, sans échange de valeur. Pour ces messages de synchronisation, nous avons défini plusieurs stratégies selon le degré de synchronisation que l'on désire obtenir. Dans chaque cas nous avons étudié le nombre de messages supplémentaires et le coût logiciel de la resynchronisation.

Nous avons remédié aux problèmes de redondance (chapitre 6) par des techniques d'analyse statique [1]. Toutefois, les particularités des programmes OC nous ont fait distinguer deux niveaux d'application pour l'élimination des messages redondants : les états de l'automate et le programme dans son ensemble. A cela il y a trois raisons :

- Les problèmes posés par ces deux types d'optimisation ne sont pas les mêmes ; code purement séquentiel dans les états et boucles dans l'automate.
- La notion d'état est liée à celle de réaction atomique du programme synchrone.
- Il est nécessaire de distinguer les messages valués, pour lesquels il faut faire une élimination globale, des messages de synchronisation, pour lesquels seule une élimination locale est autorisée.

Enfin nous avons procédé à la resynchronisation et à l'élimination des communications redondantes sur les programmes ne comportant que les émissions. En effet, le placement des réceptions se fait par simulation statique du contenu des files d'attente, et ne doit donc être effectué qu'avec la version définitive des programmes et de leurs émissions.

La répartition minimale (chapitre 7)

La seconde partie de ce travail concerne la minimisation des programmes répartis. La répartition fonctionnelle permet d'obtenir n programmes OC de telle sorte que le comportement de leur composition parallèle soit équivalent au comportement du programme centralisé initial. Bien que l'automate du programme centralisé soit minimal, nous avons constaté que les programmes répartis ainsi obtenus ne l'étaient plus. En effet la projection des actions sur leur site de calcul peut effacer des actions qui contribuaient à distinguer des états de l'automate initial. Ce principe d'effacer des actions nous a conduits, pour minimiser les programmes répartis, à appliquer un critère d'équivalence observationnelle qui permet justement de faire abstraction des actions invisibles [43]. Autrement dit, il a été possible de trouver, pour chaque site, un programme qui

soit observationnellement équivalent au programme initial de ce site, et qui soit en outre "plus petit". Notre idée a été de tirer parti de la congruence de l'équivalence observationnelle par rapport à la composition parallèle des programmes pour minimiser après coup les programmes répartis.

Le principe de la répartition minimale consiste à réduire les branchements. Pour cela nous remplaçons tous les branchements par des choix indéterministes. Puis pour chacun d'entre eux nous appliquons un critère d'équivalence observationnelle aux deux branches :

- Ou bien les deux branches du test sont observationnellement équivalentes et ce test peut être supprimé définitivement.
- Ou bien elles ne le sont pas et ce test doit être conservé.

Les fondements de la répartition (chapitre 8)

Le chapitre 8 permet de conclure ce travail en apportant des réponses au problème central : comment expliquer les désynchronisations introduites en mode réparti ? Pour ce faire, nous nous sommes limités au seul langage LUSTRE.

Tant que nous nous limitons à la synchronisation totale des branchements, il n'y a pas de problème : chaque fragment du programme réparti a la même structure de contrôle, et la resynchronisation permet d'assurer l'équivalence temporelle avec le programme centralisé, au prix certes d'échanges supplémentaires de messages. Ainsi tous les fragments ont la même horloge logique et il y a un nombre borné de cycles de décalage entre les horloges de deux sites donnés.

Au contraire, la synchronisation minimale des branchements provoque des différences entre les structures de contrôle des fragments. Même après la phase de resynchronisation, tous n'ont pas la même horloge logique. C'est précisément ce qui permet de prendre en compte, entres autres, les tâches de longue durée.

Notre idée a été d'assimiler les déroulements des réseaux flots de données LUSTRE à des ordres partiels. Ainsi pour un instant d'un flot donné, nous ne considérons que ses relations d'ordonnancement avec les instants des autres flots. Ce modèle d'ordres partiels, beaucoup trop général, a été restreint aux ensembles d'ordres pouvant être exprimés par des expressions régulières.

Nous avons donc proposé une sémantique originale de LUSTRE, asynchrone et définie par des ordres partiels. Nous avons alors appliqué le principe de la resynchronisation, c'est-à-dire ajouté des dépendances entre les flots pour garantir l'exécution en mémoire bornée. C'est la sémantique naturelle de LUSTRE. L'étude de la sémantique de l'opérateur de filtrage `when` permet alors de comprendre le relâchement du synchronisme entre deux sites.

L'exécution répartie (chapitre 9)

Nous nous sommes en dernier lieu intéressés aux problèmes posés par l'exécution des programmes synchrones répartis. Dans ce domaine, nos recherches ont conduit à deux résultats.

D'une part nous avons défini et mis en place un environnement permettant l'exécution répartie des programmes. Dans un souci de portabilité, c'est le système UNIX qui a été choisi pour servir de support. Cela nous a amenés à implémenter les files d'attente par des "sockets".

Et d'autre part nous avons adapté les interfaces synchrones/asynchrones au cas d'une exécution répartie. Ici notre principal souci a été de trouver une solution qui soit le plus proche possible de l'implémentation centralisée.

La première difficulté vient de ce que l'asynchronisme intrinsèque de l'environnement s'oppose au synchronisme du programme. C'est donc l'interface qui est chargée de regrouper les événements d'entrées en vecteurs de telle sorte que les entrées apparaissent au programme comme étant simultanées.

La seconde tient aux relations liant les entrées : contraintes d'horloges, exclusions et implications. L'étude des interfaces synchrones/asynchrones pour des programmes centralisés nous a amenés à constater que la programmation synchrone n'élimine pas totalement l'indéterminisme : elle ne fait que le reporter au niveau des interfaces.

Deux possibilités ont été envisagées dont seule la seconde nous a paru satisfaisante. Mais à cause du comportement indéterministe des interfaces, nous n'avons pas pu démontrer l'équivalence entre notre solution et l'interface centralisée. En outre cette étude nous a amenés à conclure l'impossibilité de répartir des relations entre entrées.

Le répartiteur `oc2rep` (chapitre 10)

Le répartiteur de programme `oc2rep` opère comme un post-processeur des compilateurs des divers langages synchrones. Etant donné un programme OC et un fichier de directives de répartition, il produit `n` programmes OC, un pour chaque site de calcul.

La notion de site de calcul doit en fait s'entendre comme processus plutôt que comme processeur. Rien n'empêche en effet d'exécuter un programme réparti sur 3 sites sur un processeur centralisé. Cela peut même s'avérer utile pour en tester le fonctionnement correct. Cela permet en outre de reconfigurer un système en fonction des ressources en unités de calcul. Autrement dit, la répartition logicielle des programmes OC n'est pas liée, *a priori*, à l'implantation matérielle finale.

Perspectives

Cette thèse est l'aboutissement des travaux sur la répartition des programmes OC. Quelques compléments nous semblent toutefois importants.

Dans l'immédiat, il nous parait souhaitable de procéder à une phase intensive de test du répartiteur `oc2rep`, et en particulier de l'algorithme de bisimulation. Cela nous permettrait de déterminer la complexité pratique des algorithmes. Un prototype industriel de répartiteur de code pour MERLIN GERIN est également envisagé.

Un autre axe de recherche concerne les interfaces synchrone/asynchrone réparties. La génération automatique des interfaces a déjà été envisagée mais il reste à résoudre le problème posé par les relations sur les entrées.

De plus, la localisation actuelle des actions sur les sites de calcul est simpliste : un site exécute une action de calcul si et seulement si il est le *propriétaire* de la variable située en partie gauche. Aussi des stratégies optimales de placement, utilisant les dépendances de données entre les actions, et minimisant le nombre de messages entre les sites, sont à étudier.

Il reste également à intégrer le répartiteur `oc2rep` au projet concernant le code commun des langages synchrones [47]. Il faudrait pour cela l'adapter à la nouvelle version du format interne OC.

Ce travail a donné lieu à des recherches en commun avec l'équipe PAMPA de l'INRIA-Rennes. Il serait donc bon de faire le lien avec les récents travaux de B.Caillaud sur la théorie de la parallélisation [14].

Enfin nous avons présenté dans le chapitre 2 une autre approche de la répartition, effectuée directement sur le code source et non plus sur le code objet. Il faudrait donc rapprocher ce travail des récentes recherches de B.Chéron et d'O.Maffeïs sur la mise en œuvre parallèle des programmes SIGNAL, au sein de l'équipe EP-ATR de l'INRIA-Rennes qui travaille sur ce langage [19, 40].

Bibliographie

[1] A.V. Aho, R. Sethi et J.D. Ullman. *Compilers, Principles, Techniques, and Tools*. Adison Wesley publishers, juin 1987.

[2] E.A. Ashcroft et W.W. Wadge. LUCID, *the data-flow programming language*. Academic Press, 1985.

[3] D. Austry et G. Boudol. Algèbre de processus et synchronisation. *Theoretical Computer Science*, 30:91–131, avril 1984.

[4] C. Bareau, B. Caillaud, C. Jard et R. Thoraval. Correctness of automated distribution of sequential programs. Dans *5th Internationnal PARLE Conference, LNCS 694*, pages 517–528, Munich, juin 1993. Springer Verlag.

[5] A. Benveniste, P. LeGuernic, Y. Sorel et M. Sorine. A denotational theory of synchronous communicationg systems. *Information and Computation*, 99(2):192–230, août 1992.

[6] J.-L. Bergerand et E. Pilaud. SAGA : A software development environment for dependability in automatic control. Dans *IFAC-SAFECOMP'88*. Pergamon Press, 1988.

[7] G. Berry. Real time programming: Special purpose or general purpose languages. Dans G. Ritter, éditeur, *IFIP Congress*, pages 11–17. Elsevier Science Publishers B.V. (North Holland), 1989.

[8] G. Berry. ESTEREL on hardware. *Philosophical Transaction Royal Society of London*, 339(87), 1992.

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

[10] G. Berry et G. Gonthier. The ESTEREL synchronous programming language, design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

[11] G. Berry, S. Ramesh et R.K. Shyamasundar. Communicating reactive processes. Dans *20th ACM Symposium on Principles of Programming Languages*, pages 85–89, janvier 1993.

[12] F. Boussinot et R. de Simone. The ESTEREL language. *Another Look at Real Time Programming, Proceedings of the IEEE*, 79(9):1293–1304, 1991.

[13] B. Buggiani, P. Caspi et D. Pilaud. Programming distributed automatic control systems: a language and compiler solution. Rapport Technique SPECTRE L4, LGI/IMAG, Grenoble, juillet 1988.

[14] B. Caillaud. Distribution asynchrone d'automates. Thèse, Université Rennes 1, 1994.

[15] D. Callahan et K. Kennedy. Compiling programs for distributed memory multiprocessors. *Journal of Supercomputing*, 2(2):151–169, juin 1988.

[16] P. Caspi. Clocks in data-flow languages. *Theoretical Computer Science*, 94:125–140, 1992.

[17] P. Caspi et A. Girault. Distributing finite transition systems. Dans *4th Internationnal PARLE Conference, LNCS 605*, pages 951–952, Paris, juin 1992. Springer Verlag.

[18] P. Caspi, D. Pilaud, N. Halbwachs et J. Plaice. LUSTRE : a declarative language for programming synchronous systems. Dans *14th ACM Symposium on Principles of Programming Languages*, pages 178–188, janvier 1987.

[19] B. Chéron. Transformations syntaxiques de programmes SIGNAL. Thèse, Université Rennes 1, septembre 1991.

[20] R. Cori et Y. Métivier. Recognizable subsets of partially abelian monoids. *Theoretical Computer Science*, 35:179–189, 1985.

[21] J.C. Fernandez. Aldebaran : un système de vérification par réduction de processus communicants. Thèse, Université Joseph Fourier, Grenoble, 1988.

[22] J.C. Fernandez et L. Mounier. Verifying bisimulations "on the fly". Dans J. Quemada, J. Manas et E. Vasquez, éditeurs, *Proceedings of the 3rd international conference on formal description techniques FORTE '90*. North Holland, Amsterdam, novembre 1990.

[23] G.C. Fox. Domain decomposition in distributed and shared memory environments. Rapport de recherche C3P-392, California Institute of Technology, 1987.

[24] H. Gaifam et V. Pratt. Partial order models of concurrency and the computation of functions. Dans *LICS*, pages 72–85. IEEE, 1987.

[25] N. Ghezal, S. Matiatos, P. Piovesan, Y. Sorel et M. Sorine. Un environnement de programmation pour multiprocesseur de traitement du signal. Rapport de recherche 1236, INRIA, 1990.

[26] A. Girault. Etude du relachement du synchronisme en programmation synchrone. Rapport de DEA, LGI/IMAG, Grenoble, septembre 1990.

[27] G. Gonthier. Sémantiques et modèles d'exécution des langages réactifs synchrones ; application à ESTEREL. Thèse, Université de Paris VI, 1988.

[28] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993.

[29] N. Halbwachs, P. Caspi, P. Raymond et D. Pilaud. Programmation et vérification des systèmes réactifs à l'aide du langage flot de données synchrone LUSTRE. *Technique et Science Informatique*, 10(2), 1991.

[30] N. Halbwachs, P. Caspi, P. Raymond et D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9), septembre 1991.

[31] N. Halbwachs et F. Lagnier. Sémantique statique du langage LUSTRE. Rapport non publié, février 1991.

[32] D. Harel. Statecharts: a visual approach to complex systems. *Science of Computer Programming*, 8(3), 1987.

[33] D. Harel et A. Pnueli. On the development of reactive systems. Dans *Logic and Models of Concurrent Systems,* NATO *Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems*. Springer Verlag, 1985.

[34] C.A.R. Hoare. Communicationg sequential processes. *Communication of the ACM*, 21:666–677, 1978.

[35] M. Jourdan, F. Maraninchi et A. Olivero. Verifying quantitative real-time properties of synchronous programs. Dans *International Conference on Computer-Aided Verification*, Elounda, juin 1993. LNCS 697.

[36] G. Kahn. The semantics of a simple language for parallel programming. Dans *IFIP 74*. North Holland, 1974.

[37] D.E. Knuth. *The Art of Computer Programming*, volume III : Sorting and Searching of *Computer Science and Information Processing*. Addison-Wesley, Reading, Massachussets, 1973.

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

[39] P. LeGuernic, T. Gautier, M. LeBorgne et C. LeMaire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.

[40] O. Maffeïs. Ordonnancements de graphes de flots synchrones ; application à la mise en oeuvre de SIGNAL. Thèse, Université Rennes 1, janvier 1993.

[41] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. Dans *CONCUR'92*. LNCS 630, Springer Verlag, août 1992.

[42] Merlin-Gerin. *Manuel utilisateur* SAGA. Document interne.

[43] R. Milner. A calculus of communicating systems. *LNCS*, 92, 1980.

[44] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3):267–310, juillet 1983.

[45] L. Mounier. Méthodes de vérification de spécifications comportementales: étude et mise en œuvre. Thèse, Université Joseph Fourier, Grenoble, janvier 1992.

[46] J.P. Paris. Exécution de tâches asynchrones depuis ESTEREL. Thèse, Université de Nice, 1992.

[47] J.P. Paris et al. Les formats communs des langages synchrones. Rapport de recherche 157, INRIA, juin 1993.

[48] M.A. Péraldi. Conception et réalisation de systèmes temps-réel par une approche synchrone. Thèse, Université de Nice-Sophia Antipolis, 1993.

[49] M. Pitel. SAGA : un atelier logiciel pour la conception de systémes temps réel réactifs. Dans *Journées internationnales du génie logiciel*, 1990.

[50] J.A. Plaice et J-B. Saint. *The* LUSTRE-ESTEREL *portable format*, 1987. Rapport non publié.

[51] P. Raymond. Compilation séparée de programmes LUSTRE. Rapport technique SPECTRE L5, VERIMAG, Grenoble, juin 1988.

[52] P. Raymond. Compilation efficace d'un langage déclaratif synchrone : le générateur de code lustre-v3. Thèse, INPG, Grenoble, novembre 1991.

Summary :

Synchronous programming has been introduced to facilitate the design and programming of reactive systems (systems that react continuously to their environment, the latter being unable to synchronize itself with the system). These systems are very often distributed, be it for reasons of physical implementation, performance enhancement or fault tolerance. Besides, the works on synchronous language compilation have led to represent programs by means of finite state automata: this is the OC format.

Thus this work deals with the automatic distribution of OC programs. The main difficulty is to ensure the functional and temporal equivalence between the initial centralized program and the distributed program, as well as to prove formally this equivalence. We also take pains to minimize locally the control structure of each distributed program. In order to achieve this, we design an original "on the fly" test reduction algorithm, using bisimulation techniques.

On the other hand, we completely define the execution environment of distributed programs. Our main concern here is to achieve the most faithful solution to the centralized execution.

Finally, so as to explain the desynchronizations due to the distribution, we propose an original semantics of the synchronous language LUSTRE, semantics which is defined by partial orders.

Key words :

reactive systems, synchronous programming, finite state automata, automatic distribution, "on the fly" bisimulation, distributed execution, partial orders semantics of LUSTRE.

Résumé :

La programmation synchrone a été proposée pour faciliter la conception et la programmation des systèmes réactifs (systèmes dont le rôle est de réagir continûment à leur environnement physique, celui-ci étant incapable de se synchroniser avec le système). Ces systèmes sont très souvent répartis, que ce soit pour des raisons d'implantation physique, d'amélioration des performances ou de tolérance aux pannes. En outre, les travaux sur la compilation des langages synchrones ont conduit à utiliser une représentation interne des programmes sous forme d'un automate d'états fini : c'est le format OC.

Ce travail porte donc sur la répartition automatique des programmes OC. La principale difficulté est d'assurer l'équivalence fonctionnelle et temporelle entre le programme centralisé initial et le programme réparti, et de prouver cette équivalence, ce qui est indispensable dans le domaine du temps réel critique. Nous nous attachons également à minimiser localement la structure de contrôle de chaque programme réparti. Pour cela nous développons un algorithme original de réduction des tests "à la volée" utilisant des techniques de bisimulation.

D'autre part nous définissons complètement l'environnement d'exécution des programmes répartis. Ici notre principal souci est de fournir une solution la plus proche possible de l'exécution centralisée.

Enfin dans le but d'expliquer les désynchronisations introduites par la répartition, nous proposons une sémantique originale du langage synchrone LUSTRE, sémantique définie par des ordres partiels.

Mots clés :

systèmes réactifs, programmation synchrone, automates d'états finis, répartition automatique, bisimulation "à la volée", exécution répartie, sémantique de LUSTRE en ordres partiels.