Un système formel de transformation de programmes pour leur exécution sur machines parallèles

Xiaobo Yu

▶ To cite this version:

Xiaobo Yu. Un système formel de transformation de programmes pour leur exécution sur machines parallèles. Modélisation et simulation. Institut National Polytechnique de Grenoble - INPG, 1992. Français. NNT : . tel-00341778

HAL Id: tel-00341778
https://theses.hal.science/tel-00341778

Submitted on 26 Nov 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THESE

présentée par

YU Xiaobo

pour obtenir le titre de **DOCTEUR**

de l'INSTITUT NATIONAL POLYTECHNIQUE de GRENOBLE

(Arrêté ministériel du 23 novembre 1988)

Spécialité : **Informatique**

UN SYSTEME FORMEL DE TRANSFORMATION DE PROGRAMMES POUR LEUR EXECUTION SUR MACHINES PARALLELES

Date de soutenance : 12 Novembre 1992

Composition du jury :

Président	Yves	CHIARAMELLA
Rapporteurs	Paul Philippe	FEAUTRIER JORRAND
Examinateur	Traian	MUNTEAN

Thèse préparée au sein du Laboratoire de Génie Informatique

à mes parents,

REMERCIEMENTS

C'est avec grand plaisir, au terme de ce travail, que je remercie ceux grâce à qui cette thèse a pu voir le jour.

Yves Chiaramella, Professeur à l'Université Joseph Fourier, pour s'être intéressé à mes recherches et me faire l'honneur de présider mon jury de thèse.

Philippe Jorrand, Directeur de Recherche au CNRS, et Paul Feautrier, Professeur à l'Université de Paris VI, pour avoir accepté d'être rapporteurs de cette thèse. Par leurs lectures attentives de mon manuscrit et leurs remarques avisées, ils ont largement contribué à son amélioration.

Traian Muntean, Directeur de Recherche à l'Institut National Polytechnique de Grenoble, pour avoir dirigé mes premiers pas dans la recherche et sû m'initier et me guider sur des chemins nouveaux. Grâce aux nombreuses discussions que nous avons eues, nous avons pu proposer une approche originale dans le domaine du parallélisme.

Je n'oublie pas chacun des membres de l'équipe Systèmes Massivement Parallèles, pour leur aide et leurs discussions constructives.

Marinella Khaldy et Véronique Amalric à CEGELEC, qui ont lu maintes fois le manuscrit à la chasse de mes trop nombreuses fautes d'orthographe et de rédaction.

Enfin, mes remerciements à An, pour l'encouragement qu'elle m'a donné.

Un système de transformation des programmes pour leur exécution sur machines parallèles

Xiaobo YU

Mots clés : programmation transformationnelle, programmation parallèle, système de transformation, lois algébriques de transformation, vérification formelle, placement de programmes sur machine parallèles à mémoire distribuée

Abstract

In parallel programming for massively parallel machines, the necessary software support for exploiting efficiently the available physical ressources has been a field of urgent research and development. In particular, interest is increasing in the transformational approach for supporting the parallel program development process.

We propose a program transformation system which provides formal methods and techniques for correct manipulation of parallel programs. The system consists of a collection of transformation rules represented by a rich and powerful set of algebraic laws of Occam and transformation strategies for achieving special goals. The transformation rules and strategies render the system an interactive environment for automatic program manipulation with controlled strategies.

Several important aspects of parallel programming in a distributed memory machine environment have been studied and investigated in our transformational approach, from program adaptation to a parallel architecture, program optimisation such as parallelism extraction and enhancement, distribution of global data structures, to the implementation of a process/processor mapping strategy.

Several examples of application are shown.

Résumé

Nous proposons un système de transformation des programmes parallèles qui fournit des méthodes et techniques formelles pour leur manipulation correcte. Le système est composé d'un ensemble de règles de transformation (lois algébriques d'Occam) et des stratégies de transformation.

Plusieurs aspects de la programmation parallèle dans un environnement distribué sont étudiés dans notre approche de transformation. Ils concernent notamment l'adaptation d'un programme à une architecture parallèle, l'optimisation des programmes parallèles au travers de l'extraction et de l'augmentation du parallélisme, la distribution des structures de données globales et la mise en oeuvre d'une stratégie de placement de processus/processeurs.

Quelques exemples d'application sont donnés.

Un système de transformation des programmes pour leur exécution sur machines parallèles

Xiaobo YU

Sommaire

1 **Introduction et buts de la thèse** 1

1.1 Evolution de la théorie de la programmation 1

1.2 Objectifs du travail : transformation algébrique des programmes pour exécution sur machine parallèle 3

1.2.1 Motivations de la transformation des programmes 3

1.2.2 Problèmes de la programmation des architectures parallèles 5

1.2.3 Notre approche transformationnelle et quelques notations 8

1.2.4 Organisation et interface utilisateur 11

1.3 Comparaison des systèmes de transformation 14

1.3.1 Systèmes pour la synthèse de programmes séquentiels 14

1.3.2 Systèmes pour l'exécution parallèle des programmes 18

1.4 Plan de l'ouvrage 21

2 **Une sémantique algébrique de transformation** **22**

2.1 Choix du langage 22

2.1.1 Le langage Occam et son modèle de programmation 23

2.1.2 Modèles formels d'Occam pour la transformation 26

2.2 Une sémantique de transformation Occam 29

2.2.1 Système Oxford de transformation 29

2.2.2 Interfaces utilisateur du système de transformation 34

2.2.3 Lois d'Occam 37

2.3 Vérification formelle des programmes parallèles 48

2.3.1 Formes normales 48

2.3.2 La preuve formelle d'équivalence de programmes 55

2.3.2.1 Programmes finis 55

2.3.2.2 Programmes infinis et leur approximation syntaxique 57

2.3.3 Exemple d'application de la vérification formelle 61

2.4 Conclusions 66

3 **Optimisation des programmes parallèles** **67**
3.1 Analyse statique d'un programme 68
3.1.1 L'espace d'état d'un programme et son calcul 68
3.1.2 Normalisation des variables et des canaux 73
3.2 L'optimisation des programmes parallèles 75
3.2.1 Techniques de transformation dans la compilation optimisée 76
3.2.2 Détection et extraction du parallélisme 78
3.2.2.1 Parallélisation des programmes séquentiels 79
3.2.2.2 Parallélisme réel 90
3.2.3 Un exemple du changement de la forme de communications 93

4 **Transformations de programmes pour leur placement sur des machines parallèles** **97**
4.1 Placement des programmes sur machines parallèles 97
4.1.1 Définition du problème 97
4.1.2 Supports pour le placement par transformation 99
4.2 Réalisation d'une stratégie de placement processus/processeur 100
4.2.1 Routage de messages et multiplexage de canaux 100
4.2.2 Implantation distribuée des structure de données globales 106
4.2.3 Transformations pour réduire le degré de parallélisme 113
4.2.3.1 Un exemple de la projection de réseau systolique 113

5 **Une extension de la sémantique algébrique** **122**
5.1 Lois d'ajustement du parallélisme 123
5.2 Une forme abstraite des programmes parallèles 128
5.3 Un exemple de l'augmentation du parallélisme 135

6 **Conclusions et perspectives** **141**
6.1 Conclusions 141
6.2 Perspectives : synthèse automatique des programmes parallèles 144

Annexe A1 Lois d'Occam élémentaires 148
Annexe A2 Lois d'ajustement du parallélisme 163
Annexe B1 Programme systolique à deux dimension : C = A x B 172
Annexe B2 Programme après la projection : C = A x B 174
Annexe C1 Programme de flot de données 177
Annexe C2 Parallélisation complète 180
Annexe D Exemples de l'utilisation du système de transformation 182
D1 Lancement d'une session de transformation 182
D2 Quelques exemples 185
D3 Performances 192
D4 Un listage séléctif des programmes sources de transformation 193

Bibliographie **199**

Liste des figures

Chapitre 1

Figure 1.1 La programmation des machines parallèles par transformation 9

Chapitre 2

Figure 2.1 Architecture du Système de transformation 30
Figure 2.2 Processus d'analyse et de la transformation 32
Figure 2.3 La vérification d'équivalence de deux programmes 48
Figure 2.4 Un système totalement vérifié 61

Chapitre 3

Figure 3.1 L'extraction du parallélisme à l'intérieur d'une construction séquentielle 80
Figure 3.2 La multiplication de deux matrices par un programme systolique 94

Chapitre 4

Figure 4.1 Graphe du programme (a) et graphe des processeurs (b) 100
Figure 4.2 Un placement possible 100
Figure 4.3 Le routage de messages et le multiplexage de canaux 103
Figure 4.4 Le processus routeur 104
Figure 4.5 Distribution des structures de données globales 108
Figure 4.6 Distribution des données 110
Figure 4.7 La composition de matrices - réseau systolique à deux dimensions 114

Chapitre 5

Figure 5.1 La transformation des programmes pour leur exécution parallèle 123
Figure 5.2 Forme abstraite des programmes parallèles pour leur exécution parallèle 129
Figure 5.3 Graphe du processus de la forme de treillis 131
Figure 5.4 La composition de matrices par un programme de flot de données 136

chapitre 6

Figure 6.1 L'impact de l'approche transformationnelle sur la conception logiciel 142

Chapitre 1

Introduction et buts de la thèse

1.1 Evolution de la théorie de la programmation

La théorie de la programmation est liée étroitement aux langages utilisés. Au début des premiers ordinateurs, il fallut écrire des programmes dans le seul langage qu'une machine interprète, son langage machine. Son emploi étant fastidieux et les risques de fautes élevés, on les a rapidement remplacés par les langages d'assemblage, n'offrant que les structures et les opérations du langage machine.

Le premier langage évolué, Fortran, marquait de façon profonde l'histoire de la programmation par le choix des structures du langage, le premier compilateur. De très nombreux langages (par exemple, Algol 60, Cobol, APL, etc) se sont très vite développés, à partir de deux principales caractéristiques (sauf le langage LISP, qui s'est fondé sur la définition récursive de fonctions) : (1) l'instruction d'affectation, ou modification du contenu d'une case de la mémoire de la machine ; (2) la notion de séquence entre les instructions, de la structure de boucle, d'une rupture de séquence par l'instruction <GO TO>, conditionnelle ou non.

Les moyens de construction de programmes étaient rudimentaires, voire inexistants. De nombreux chercheurs se sont penchés alors sur le problème de la création des programmes [Knuth68, Gries 81]. En ce qui concerne la théorie de la programmation, l'étape la plus importante a été l'introduction des assertations inductives par Floyd [Floyd67], système axiomatisé par Hoare [Hoare69]. On l'a d'abord considéré comme un outil de preuve de programmes. L'étude de la preuve de la correction de programmes a conduit à la découverte d'une nouvelle méthode de construction de programmes : la programmation structurée avec le principe de la dérivation par raffinement par étapes. L'idée élémentaire peut être illustrée par l'exemple suivant [Dijk76, Gries81, Arsac83].

Un exemple simple de développement de programmes

Considérons le problème : écrire un programme P qui calcule le maximum z de deux entiers donnés, x et y . Donc le programme P satisfait :

(1.1) $\{Q : \text{Vrai}\}\ P\ \{R : z = \max(x, y)\}$.

où Q est la pré-condition et R est la post-condition,

La condition R doit être raffinée par sa définition, pour le développement. La variable z contient le maximum de x et de y si elle satisfait :

(1.2) $R : z \geq x \wedge z \geq y \wedge (z = x \vee z = y)$.

Maintenant le problème est quelle commande pourrait être exécutée pour établir (1.1). Comme (1.2) contient <z = x>, l'affectation <z := x> est une possibilité. Pour déterminer la condition sous laquelle l'exécution de la commande <z := x> établira (1.2), nous calculons simplement la condition la plus faible [Dijk76] :

$$\begin{aligned} wp(\text{" z:= x"}, R) &= x \geq x \wedge x \geq y \wedge (x = x \vee x = y) \\ &= x \geq y \end{aligned}$$

Notre premier programme donc pourrait être :

si $x \geq y \rightarrow z := x$ **finsi**

Ce programme fonctionne comme nous l'espérons, à condition qu'il ne se comporte pas comme STOP. C'est-à-dire, au moins une garde doit être prête dans n'importe quel état initial défini par la condition Q. Mais Q, qui est Vrai, l'implique pas $x \geq y$. Donc, au moins une garde de plus est nécessaire.

Nous pouvons remarquer qu'une autre façon possible d'établir (1.2) est l'exécution de la commande <z := y>, et qu'il est évident que <y ≥ x> est la garde qui manque. Cette considération nous donne le programme suivant :

```
(1.3)     si   x ≥ y → z := x
               y ≥ x → z := y
          finsi
```

Maintenant au moins une garde est toujours vraie, donc c'est le programme désiré.

1.2 Objectifs du travail : transformation algébrique des programmes pour exécution sur machine parallèle

1.2.1 Motivations de la transformation des programmes

Grâce à l'introduction de la discipline de la programmation structurée et des recherches sur la vérification et la construction formelle de programmes [DHD72, Dijk76], de nombreux systèmes permettant de faciliter et d'automatiser la construction correcte de programmes ont été proposés [MW79, PS83, D[illegible]a89, BK89].

Bien que les activités créatives de la conception soit toujours réalisées manuellement, des systèmes de développement logiciel, qui offrent des supports automatiques, sont devenus indispensables [PS83]. Leur amélioration passe par l'intégration des systèmes de transformation de programmes. Les outils existants vont des plus simples (comme éditeurs structurés) aux systèmes interactifs puissants de transformation, voire même de synthèse automatique de programmes pour un domaine spécifique d'application [PS83, Leng88].

La transformation de programmes est utilisée pour atteindre des buts différents. Le plus commun est celui de support général pour la modification de programmes. Ceci inclut l'adaptation et l'optimisation de programmes à une architecture particulière, l'implantation efficace de structures de données. Selon les exigences et les buts à atteindre, la transformation de programmes vise un ou plusieurs des objectifs suivants:

- La vérification d'un programme par rapport à sa spécification fonctionnelle. Un cas particulier est la preuve d'équivalence des programmes. Il existe des méthodes différentes pour vérifier l'effet prévu d'un programme:

(1) La méthode classique de test ou d'échantillon. Cette méthode consiste à observer le fonctionnement ou les sorties d'un programme sur des données représentatives. Elle a un inconvénient parce que "cette méthode peut prouver qu'un programme contient des erreurs, mais pas le contraire" [Dijk76].

(2) La méthode de la logique à prédicat, ou la logique axiomatique telle que la logique de Hoare [DHD72].

(3) La méthode de transformation en changeant la représentation syntaxique des programmes équivalents vers une forme normale [HR85, Gold88].

- Le changement d'un programme clair mais probablement inefficace en un programme efficace mais probablement obscur [MW79]. Ce genre de transformations joue un rôle essentiel dans la phase d'optimisation d'un compilateur, par exemple [ASU86].

- La restriction de la représentation syntaxique d'un programme à des buts particuliers. Par exemple, ce type de transformation est utilisé dans la CAO où un programme est transformé successivement vers une représentation syntaxique réalisable sur des composants VLSI [Inmos88].

- La modification ou adaptation d'un programme à un environnement d'exécution particulier, basée sur des méthodes formelles. Ceci est notre objectif principal où nous cherchons à adapter un programme à une architecture parallèle, en vue d'obtenir son exécution efficace.

- La synthèse automatique de programmes : la construction automatique, correcte d'un programme à partir de sa spécification formelle. C'est un aspect du plus prometteur de l'application de l'approche transformationnelle.

1.2.2 Problèmes de la programmation des architectures parallèles

Pour satisfaire des besoins de plus en plus croissants en puissance de traitement exprimés par des applications d'horizon aussi divers que le calcul scientifique, l'industrie, l'économie, des puissantes machines dites supercalculateurs, telle le CRAY, le Cyber 205, l'ILLIAC IV, etc ont été développées.

En dépit de leur puissance de calcul, outre les limitations usitées, ces machines, souffrent également de limitations dues à la complexité architecturale : leur non extensibilité.

Nous avons vu apparaître une nouvelle classe de machines alternatives, les machines massivement parallèles (où le nombre de processeurs est facilement extensible) [Dong88, Trel88], qui permettent d'atteindre des performances équivalentes à celle des supercalculateurs. Grâce aux possibilités qu'offre l'intégration VLSI, ces machines sont devenues une réalité.

Elles se distinguent des supercalculateurs par un coût moindre (processeurs de base moins sophistiqués), et un grand nombre de processeurs avec des configurations très variées.

Une classe des machines parallèles, celle à mémoire distribuée, locale (MIMD sans mémoire commune) qui nous intéresse, est caractérisée par des processeurs faiblement couplés et communiquant uniquement par échange de messages. Les processeurs sont connectés par liens de communications, suivant certains types de topologies fixes ou dynamiques : on y trouve des réseaux de transputers tels que Computing Surface, T_Node ; Multiprocesseurs Cm*, iPSC/2, etc.

<u>**Exécution d'un programme sur les architectures parallèles sans mémoire commune**</u>

Un programme réalisant un algorithme d'application, est composé d'un ensemble de processus s'exécutant en parallèle et communiquant uniquement par échange de messages à travers de canaux de communications. Un processus correspond à une partition de l'algorithme, ou un traitement sur une partition des données du problème à résoudre. Les processus se synchronisent par des communications.

Ce modèle d'exécution pose un défi pour la programmation des architectures parallèles, dont les problèmes seront présentés dans la section suivante.

Problèmes de la programmation parallèle

Bien que les machines parallèles offrent une puissance potentielle pour l'exécution des programmes parallèles, il existe de nombreuses difficultés pour leur implantation et exécution efficace. Ces difficultés sont dues au modèle de calcul et à la configuration distribuée d'architectures parallèles :

L'espace fragmenté d'adresses, c'est-à-dire, chaque processus peut avoir l'accès uniquement à la mémoire locale du processeur sur lequel il s'exécute. Un programmeur est forcé d'effectuer une décomposition des structures de données en différentes partitions, chaque partition appartenant à un seul processus. Toutes interactions entre différentes partitions de données doivent être formulées explicitement, en utilisant les constructions d'échange de messages dans le langage cible.

Par conséquent, la programmation des machines distribuées est beaucoup plus difficile que celle séquentielle, sur les points suivants :

(1) Elle exige une conception distribuée de l'algorithme. Il existe différentes approches pour la conception d'un algorithme distribué, en fonction des caractéristiques du problème à résoudre [Hey89, Feaut89], dont nous pouvons citer comme exemples :

- Parallélisme algorithmique. Cette approche consiste à décomposer l'algorithme en un grand nombre de tâches, qui peuvent être exécutées en parallèle. Cette approche est très générale, et souvent utilisée dans le calcul distribué.

- Parallélisme géométrique. Cette approche consiste à distribuer les données d'un algorithme sur un certain nombre de processus, telle que la structure géométrique des données soit conservée.

Elle est efficace pour résoudre des problèmes de maillage, qui possèdent un parallélisme massif de données. L'avantage de cette approche est que la communication inter-processus est relativement simple, régulière et locale. Par exemple, des opérations portent sur certaine dimension de la structure de données : une propagation des données dans une direction ; une rotation des données tout au long d'une direction spécifique ; effectuer une opération (comme la somme) sur les données d'une direction.

(2) Efficacité de l'exécution d'un programme parallèle sur une architecture distribuée.

Supposons qu'on ait déjà un programme parallèle (écrit avec un langage qui permet l'expression explicite du parallélisme), les performances lors de son exécution sur une architecture parallèle dépendent principalement du placement de ses processus composants sur les processeurs [KM89, TM90]. Avant son placement sur les processeurs, il faut d'abord : adapter son degré de parallélisme au nombre de processeurs disponibles ; adapter les communications inter-processus dans le programme à la structure de communication de l'architecture cible. Le but principal de cette adaptation est de minimiser le temps d'exécution du programme, sous des contraintes imposées, comme les limitations physiques de la machine.

Il en suit un problème de portabilité car un programme, qui a une exécution efficace sur une architecture spécifique, n'est pas forcément efficace pour une autre machine.

1.2.3 Notre approche transformationnelle et quelques notations

Les langages et les outils d'aide à la programmation des machines parallèles permettent à l'utilisateur d'exprimer le parallélisme d'un algorithme, aussi de spécifier sa configuration d'exécution grâce à des primitives spécifiques à travers certains langages parallèles (C, FORTRAN ou Occam, etc) [TDS88, INT89]. Cette tâche étant fastidieuse et difficile pour une manipulation manuelle de la part de l'utilisateur, et le manque de support logiciel pour la configuration d'exécution des programmes forment la principale difficulté pour l'exploitation des machines parallèles [Hey89].

Les méthodes et outils de transformation que nous proposons dans cette thèse cherchent d'une part, à automatiser l'extraction du parallélisme et d'autre part, à offrir une assistance (par ajustement du degré de parallélisme) au placement automatique. Ceci permet en outre une programmation transparente de l'architecture cible.

Le principe de notre approche consiste à adapter par transformations successives un programme donné à une architecture parallèle (Voir figure 1.1) :

(1) A partir d'un programme correct, nous effectuons une analyse statique, en vue de collecter des informations du programme nécessaires à l'extraction du parallélisme (§3). Les informations obtenues suite à cette analyse sont stockées dans les processus parallèles, et représentées sous forme d'annotation de l'utilisation des canaux et des variables.

(2) En fonction du nombre des processeurs et leur connectivité dans l'architecture parallèle cible, nous effectuons une série de transformations source-à-source qui adapte (réduit ou augmente) le degré de parallélisme et/ou la forme des communications inter-processus (§ 4.1).

(3) L'implantation distribuée des données globales et le routage de messages, lors de son exécution (§4.2).

(4) Sur la base des techniques de transformation ci-dessus, nous transformons le programme en une forme abstraite (présentée dans le chapitre 5) qui possède un grand degré de parallélisme et une forme régulière des communications inter-processus.

Notons qu'en pratique, ces quatre phases interagissent et se coordonnent. Fonctionnellement, nous pouvons les présenter de la façon suivante.

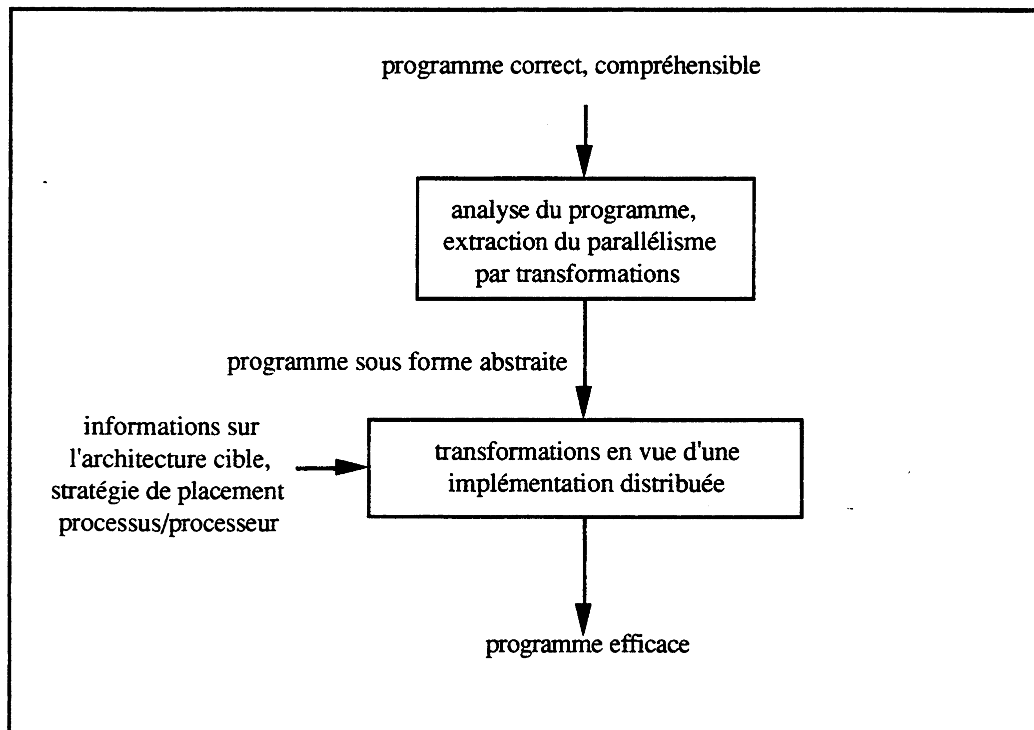

Figure 1.1 La programmation des machines parallèles par transformation

Quelques notations de transformation

Le terme *programme* est interprété dans un sens classique : c'est la description d'une méthode de calcul exprimée dans un langage formel. Nous définissons dans cette section les termes d'un système de transformation dans un langage descriptif, informel. Dans le contexte du langage Occam que nous adoptons comme exemple de langage parallèle, leur définitions seront données d'une façon formelle et précise.

Définition 1.1 Une *transformation* de programmes est une relation entre deux programmes P et P'. Cette relation est correcte (ou valide) si pour une sémantique S nous avons : S (P, P').

Parmi les relations sémantiques possibles, la plus importante dans ce contexte est l'équivalence. Elle exprime le comportement et les propriétés d'un programme qui ne sont pas affectés par son implantation.

Définition 1.2 Une *règle de transformation* est un "mapping" partiel d'un programme vers un autre, tel qu'un élément du domaine de départ et son image constituent une transformation correcte.

Elle peut être représentée sous deux formes : les procédures algorithmiques, ou une paire de programmes (régle de réécriture, par exemple). Pour une règle de transformation, il peut y avoir des prédicats comme pré-condition pour restreindre le domaine d'application.

Les règles de transformation peuvent être classées en deux catégories : règles élémentaires (complétude pour la sémantique associée) et règles engendrées (une séquence ou une dérivation de règles élémentaires).

Définition 1.3 La *programmation transformationnelle* est une méthodologie de la construction de programmes par applications successives de règles de transformation.

La programmation transformationnelle a deux caractéristiques : la construction et l'amélioration (l'optimisation) de programmes.

Il existe différentes techniques pour cette méthodologie. Nous pouvons citer comme exemples : le processus synthétique, la dérivation et la construction d'un programme correct à partir de sa spécification formelle ; le processus "analytique" qui a pour but de vérifier certaines propriétés (par exemple, l'équivalence) de programmes ou d'améliorer leur efficacité.

Nous pouvons remarquer que la programmation transformationnelle a été influencée par la programmation structurée, et elles ont des points communs, comme par exemple : le principe du "*divide and conquer* " et du raffinement par étapes successives. Néanmoins, la programmation structurée est un processus qui conserve l'organisation (structure de l'algorithme) originale, tandis que la programmation transformationnelle aboutit probablement à un programme final avec une organisation complètement différente.

Définition 1.4 Un *système de transformation* est un système réalisé pour supporter la programmation transformationnelle. Il est souvent constitué d'un ensemble de règles de transformation, et des stratégies spécifiques de transformation en fonction des objectifs à atteindre.

1.2.4 Organisation et interface utilisateur

Supports systèmes pour l'utilisateur

Un système de transformation, général ou spécifique, a un composant pour effectuer des transformations. Ce composant est généralement composé d'une certaine facilité pour maintenir une collection de transformations disponibles (catalogue de règles), et un mécanisme pour l'appliquer. Fréquemment, le système en question permet aux utilisateurs d'étendre cette collection par la définition de nouvelles règles, mais la plupart des systèmes supposent ces nouvelles règles d'être prouvées et vérifiées par l'utilisateur. Certains systèmes offrent cependant une aide pour la sélection des règles dans la collection.

Presque tous les systèmes de transformation sont interactifs. Même ceux qui sont "totalement automatisés" demandent à l'utilisateur des entrées initiales et d'intervenir en cas d'événements inattendus.

L'évaluation de programmes peut être supportée de différentes façons qualitatives : le système peut avoir certaine facilité d'exécution, comme interpréteur ou compilateur. Parfois le système offre des outils d'analyse de programmes, surtout en ce qui concerne l'aspect statique.

Organisation des transformations

Un système de transformation a une collection pré-définie de règles de transformation, qui est extensible dans un certain sens. En principe, il existe deux méthodes pour maintenir les transformations :

- L'approche de catalogue. Un catalogue de règles est une collection structurée linéaire ou hiérarchique des règles, reliées à un aspect particulier du processus de développement logiciel. Il peut contenir, par exemple, des règles sur (1) la connaissance de données ; (2) l'optimisation basée sur des caractéristiques du langage, comme l'élimination des structures récursives ou la fusion des boucles.

- L'approche des ensembles génératifs. Un petit ensemble puissant de transformations élémentaires est utilisé comme une base pour la construction de nouvelles règles. Si des transformations élémentaires sont couplées avec un langage restreint, une certaine complétude restreinte peut être atteinte.

Formes de règles de transformation

Une règle de transformation, comme un "mapping" (partiel) d'un programme en un autre, peut être représentée de deux façons :

- Sous forme algorithmique, qui prend un programme comme l'entrée et produit un nouveau comme la sortie. Les règles de ce type accomplissent l'analyse de flots de contrôle et de données, comme des stratégies de transformation.

- Comme une paire ordonnée de deux programmes (règle de réécriture).

Typiquement, les règles de ce type accomplissent les manipulations suivantes :

(1) relier des structures du langage (règles syntaxiques), comme l'exemple suivant, qui est une définition récursive d'une boucle :

```
WHILE condition
   P
------------------------
IF
   condition
       SEQ
          P
          WHILE condition
             P
   TRUE SKIP
```

(2) décrire des propriétés algébriques, reliant différentes structures du langage, par exemple :

1+ (IF b x ELSE y) = IF b THEN (1+ x) ELSE (1+ y)

(3) exprimer des connaissances du domaine sous forme de propriétés de types de données :

pop(push(s, x)) = s, pour une pile illimitée s.

Performance d'un système de transformation

Tous les systèmes de transformation appliquent des règles de transformation avec succès. La différence entre eux se situe au niveau de la quantité de travail qu'ils exigent de la part de l'utilisateur, en cas de comportement inattendu.

Evidemment, l'implantation la plus simple est celle qui demande à l'utilisateur d'être responsable de chaque étape de transformation. Telle approche est pratique si le système fournit certaines facilitées pour former des règles de transformation compactes et puissantes.

Des systèmes automatiques, par contre, accomplissent la sélection et l'application de règles adéquates d'une façon autonome, en utilisant des heuristiques ou d'autres considérations stratégiques. De tels systèmes fonctionnent de manière satisfaisante seulement pour des domaines restreints d'application.

Entre ces deux extrêmes il existe des systèmes semi-automatiques, qui, pour des tâches pré-définies, sont autonomes et sous le contrôle de l'utilisateur, dans d'autres cas. Souvent de tels systèmes offrent aux utilisateurs l'assistance nécessaire dans l'évaluation du critère de décisions.

Langages

Le langage utilisé influence considérablement la capacité d'un système de transformation. Bien que certains systèmes soient conceptuellement indépendants d'un langage particulier, leur implantation est liée à un éventuellement.

Selon les buts particuliers d'un système, différents langages peuvent être utilisés. Nous classons les langages en deux catégories : ceux de spécification, qui supportent l'expression formelle des problèmes, et ceux de la programmation, qui sont utilisés pour formuler des solutions d'un problème.

Des langages de spécification varient, du langage naturel formalisé, à ceux formels purement descriptifs, comme la notation mathématique ou la logique à prédicat.

Domaines d'application

Du point de vue pratique, l'aspect le plus intéressant et le plus important d'un système de transformation, est la classe de problèmes qu'il est capable de traiter. Le fait que cette classe soit souvent trop limitée est la principale critique de l'approche transformationnelle [Dijk76]. Tous les systèmes d transformation existants à ce jour, issus des instituts de recherche, doivent être considérés comme outils expérimentaux, donc avec une capacité restreinte.

1.3 Comparaison des systèmes de transformation

L'idée de la transformation de programmes est issue des recherches et la mise en oeuvre des techniques d'optimisation (essentiellement transformation) dans les compilateurs [ASU86]. Les systèmes de transformation existants accomplissent principalement deux caractéristiques de transformation : la construction automatisée et/ou l'amélioration des programmes. On peut trouver dans [PS83, DHKL85, Bau89] une présentation et classification des systèmes existants de transformation.

1.3.1 Systèmes pour la synthèse de programmes séquentiels

Nous remarquons que la plupart des systèmes de transformation réalisés, généraux ou particuliers à un domaine d'application, accomplissent essentiellement la synthèse de programmes de style fonctionnel. Ceci n'est pas étonnant parce que les programmes, formulés dans un langage fonctionnel, ont des propriétés formelles, faciles à manipuler comme des objets algébriques [Back78]. En ce qui suit, nous présentons les caractéristiques et comparons des systèmes représentatifs de cette classe de systèmes de transformation.

SAVE/TI d'ISI

Dans ce système, il y a deux phases caractéristiques [BGW76] : la phase SAVE (synthèse de spécifications formelles à partir des spécifications informelles) et la phase TI (Implantation Transformationnelle) qui est chargée de la dérivation de programmes à partir d'une spécification formelle par transformation.

Le développement d'un programme à partir d'une spécification est considéré comme une série de transformations : le remplacement d'une construction de la spécification par une construction algorithmique ; ou la simplification et l'optimisation des constructions algorithmiques. Le résultat est un programme formulé dans un sous-ensemble du langage de spécification GIST, qui sera traduit automatiquement par le système en un langage de type fonctionnel LISP.

La principale activité de transformation est la sélection de règles de transformation appropriées qui existent dans un catalogue pré-défini. Le programmeur peut étendre le catalogue par la définition de nouvelles règles en cas de besoin. Les règles de transformation expriment la connaissance sur les structures de données telles que les ensembles, la substitution des

structures de données par leur représentation et les structures du langage telles que les boucles, etc.

Le système est réalisé en LISP : un mécanisme interactif pour la sélection des règles ; un catalogue de règles pré-définies ; un mécanisme pour la traduction d'un programme résultat en un programme du type LISP. Les problèmes traités dans le système sont : un éditeur de texte, le problème des "Huit reines", etc.

Ce système est représentatif de l'approche de catalogue de règles, tandis le système suivant est représentatif de l'approche générative.

Le système d'Edinburgh

Le système [BD77] prend comme entrée des fonctions formulées en équations explicites de NPL, un langage d'équations récursives de premier ordre, et produit des programmes aussi en NPL, mais moins complexes : (1) la synthèse ; (2) l'amélioration des fonctions.

Il existe seulement six règles de transformation dans le système : la définition, l'instanciation, "folding" et "unfolding", l'abstraction, et lois (règles de réécriture sur les structures de données).

Le fonctionnement du système est caractérisé par la méthode "fold/unfold" : la règle de définition permet l'introduction d'une nouvelle fonction, ensuite, une opération "unfold" sur le coté droite de la fonction en vue d'une manipulation approfondie, finalement une opération "fold" permet la définition d'une nouvelle fonction équivalente mais plus efficace.

Le système PSI de Stanford

C'est un système LISP pour la synthèse de programmes efficaces, qui fonctionne en trois phases [Green77] :

- Le développement d'un modèle de programme. Une spécification formelle est obtenue suite à un dialogue avec l'utilisateur, qui ensuite sert à l'extraction des informations et la mise à jour d'un modèle partiel de programme. Les 200 environ de règles de transformation expriment la connaissance d'équivalence des modèles de programmes.

- Le codage. A partir d'un modèle de programme, un programme résultat en LISP sera obtenu.

- Efficacité. Cette dernière phase consiste à optimiser le programme résultat en LISP.

DEDALUS de Stanford

Le système accomplit deux activités [MW79] :

(1) La dérivation automatique des programmes LISP à partir d'une spécification formulée en notation mathématique et logique sous forme LISP: le raffinement par transformation pour atteindre le but exprimé dans la spécification formelle.

(2) La preuve de la correction et de la terminaison des programmes qui ne sont pas récursifs mutuels.

Le projet CIP de Münich

A part des caractéristiques d'un système de synthèse, le système [Bau89] adopte un langage algorithmique abstrait (ALGOL-like), dont un spécification formelle est décrite par les types abstraits algébriques. Les règles de transformation sous forme de schéma relient des modèles équivalents de programme. Dans la dérivation d'un programme, le système dépend de l'utilisateur pour la sélection de règles de transformation.

PDS de Harvard

C'est un environnement de programmation [Chea81] qui est composé d'outils utilisateurs. Ces outils permettent la définition, le test, le maintenance et la dérivation des programmes à partir d'une spécification abstraite. Le système contient une basé de données sur la connaissance d'un programme pour ses différentes versions intermédiaires. Les règles de transformation pour la dérivation sont fournies par l'utilisateur.

Les exemples de programmes traités dans le système sont : une évaluateur symbolique, le système PDS lui-même, etc.

MENTOR d'INRIA

C'est un environnement interactif de programmation [DHKL80] qui permet : la conception, la mise en oeuvre, la documentation, la validation, le maintenance et la modification des programmes. Bien que le système est conceptuellement indépendant des langages, la réalisation du système se limite aux programmes PASCAL.

Le système prend un programme PASCAL comme entrée, le traduit en une représentation d'arbres abstraits syntaxiques, sur laquelle la transformation s'effectue. Les règles de transformation sont sous forme de règles de réécriture d'arbres abstraits.

Par exemple, le système offre une normalisation des programmes PASCAL par le réarrangement des déclarations, en respectant leur portée et la transformation source-à-source comme la propagation des constantes, l'élimination des constructions récursives, etc.

Insuffisance de ces systèmes pour notre objectif

Ils visent la synthèse et la dérivation des programmes séquentiels, principalement fonctionnels. Ils sont utiles et suffisants pour l'aspect séquentiel dans notre cadre de la programmation parallèle, mais ne traitent pas l'aspect parallèle.

1.3.2 Systèmes pour l'exécution parallèle des programmes

Systèmes pour l'exécution parallèle des boucles

Parmi les systèmes de transformation, il existe une autre classe dans l'objectif de l'adaptation des programmes (principalement en FORTRAN, PASCAL ou C) à des machines vectorielles ou parallèles. Souvent des techniques de transformation sont intégrées dans un compilateur spécifique à un super-calculateur. Par exemple, le système Loveman [Love77], dont les idées de transformation sont implantées dans le compilateur IVTRAN (compilateur FORTRAN pour la machine ILLIAC-IV). A part des transformations "classiques" d'optimisation, ces systèmes contiennent des règles spéciales destinées à l'optimisation des structures de boucles (le déroulement et la fusion de boucles, l'interaction entre des boucles et conditionnels). Le plus intéressant est la transformation sur les boucles en vue d'obtenir leur exécution sur une machine parallèle avec un nombre fini de processeurs.

Les FOR-boucles emboîtées sous la forme suivante sont largement utilisées dans les programmes (par exemple en FORTRAN) tels que la multiplication de matrices, l'élimination de Gausse, le calcul du plus court chemin d'un graphe, etc. Elles dépensent une grande quantité de temps de calcul d'un programme, et font l'objet principal de parallélisation en vue de minimiser leur temps d'exécution.

```
FOR i1 = l1 TO u1 DO
        i2 = l2 TO u2 DO
           ...
              in = ln TO un DO
                 Le calcul
              END
           END
        ...
END
```

La parallélisation des FOR-boucles est basée sur la relation de dépendance entre les statements et leur niveau d'itération (distance). La relation de dépendance est représentée [SC91] par un ensemble de vecteurs d'entier : vecteurs de distance. Une transformation d'une FOR-boucle est modélisée comme une transformation linéaire dans l'espace de vecteurs d'itération. Une approche simple consiste à exécuter en parallèle une FOR-boucle pour toutes les itérations qui n'ont pas de dépendance. D'autres méthodes plus élaborées sont [Love77, Feaut89, SC91, WL91, KM91] :

(1) La méthode de "Coordination" : la transformation d'un ensemble de boucles emboîtées en boucles parallèles asynchrones.

(2) La méthode "Hyperplane" : la transformation d'un ensemble de boucles emboîtés en boucles parallèles synchrones. Elle consiste à transformer une FOR-boucle à dimension *n* en Hyperplane de temps à dimension *t* et hyperplane d'espace à dimension *s* , où n = s + t. Cela représente une partition de la boucle, dont les différentes parties (dans le même hyperplane de temps) peuvent être exécutées en parallèle.

(3) La méthode "Strip-mining" : l'adaptation d'une boucle parallèle à une machine avec un nombre fixe de processeurs.

Ces méthodes ne conviennent pas à notre objectif par la raison suivante: elles visent principalement des machines parallèles avec la mémoire partagée (les structures de données sont partagées par les itérations en parallèle), et la parallélisation se limite aux boucles.

Le système de transformation Oxford

Les premiers travaux de recherche dans le domaine de la transformation de programmes Occam ont été réalisés par Inmos et l'Université d'Oxford sur la base des méthodes formelles de CSP [Hoare78]. Ces efforts ont abouti à la première version expérimentale de "Oxford Occam transformation system" [HR85, Gold87, 88]. Le système réalise toutes les lois Occam et une stratégie de la vérification d'équivalence des programmes dans [HR85]. Pour l'application pratique, le système a été appliqué à la vérification du fonctionnement correct de certains composants VLSI, dans l'unité de calcul flottant du processeur T800 [Inmos88]. Le système sera présenté dans le chapitre 2.

Sur la base théorique de ce système, nos travaux de recherche consistent à étudier un autre aspect de l'application pratique de l'approche transformationnelle sur la programmation parallèle des machines distribuées, dans le but de l'optimisation de programmes (surtout l'extraction et l'adaptation du parallélisme), et de la mise en oeuvre d'une stratégie de placement processus/processeur.

Ces aspects nous amènent à :

(1) étendre la sémantique définie dans le système Oxford :

- Enrichissement de la collection des lois de transformation (nous nous plaçons également dans le cadre du langage Occam), surtout des lois qui permettent l'extraction du parallélisme, l'implantation des structures de données globales, le changement de la forme de communications inter-processus ;

- Définition d'une forme abstraite des programmes parallèles pour faciliter leur placement, ainsi la stratégie nécessaire de transformation.

(2) étudier des algorithmes pour l'extraction du parallélisme d'un programme et la réalisation de son placement, en vue d'obtenir une exécution efficace sur une machine donnée.

1.4 Plan de l'ouvrage

Le deuxième chapitre est consacré à la présentation d'une sémantique algébrique de transformation de programmes Occam. Ce système Oxford est bâti sur les lois algébriques Occam et une stratégie de transformation des programmes en une forme normale. Aussi présentée est son application sur la vérification formelle.

Dans le chapitre 3, nous étudions les techniques et méthodes d'analyse statique et d'optimisation de programmes parallèles par transformation. Les aspects importants de l'optimisation, telles que la détection et l'extraction du parallélisme, ainsi que les stratégies nécessaires de transformation seront étudiés.

Dans le chapitre 4, nous appliquerons notre système à l'ajustement du degré de parallélisme, l'implantation distribuée des données globales et la réalisation d'une stratégie de placement processus/processeur.

Dans le chapitre 5, nous proposons une extension du système Oxford de transformation, sur la base des résultats des chapitres précédents, pour la manipulation des programmes parallèles pour leur configuration d'exécution sur des machines parallèles.

Dans le dernier chapitre, nous tirons quelques conclusions et nous discutons des perspectives de recherches, sur des aspects prometteurs de l'approche transformationnelle, tels que la synthèse automatique de programmes.

Chapitre 2

Une sémantique algébrique de transformation

2.1 Choix du langage

Des langages pour la programmation d'architectures parallèles sont souvent développés sur la base des langages existants (comme C, FORTRAN, PASCAL, etc), avec l'extension qui offre la possibilité d'exprimer explicitement le parallélisme, la décomposition des structures de données et l'échange de messages. L'avantage de cette approche est que les programmes existants sont réutisables sur des nouvelles architectures.

Nos critères de choix du langage pour la transformation sont basés sur les considérations suivantes : une correspondance naturelle entre la structure du programme et une architecture parallèle (par exemple comme la capacité d'exprimer explicitement le parallélisme, l'implantation distribuée et locale de données ...), des caractéristiques formelles qui permettent la manipulation des programmes comme des objets algébriques (comme dans le cas des programmes en langages de style fonctionnel).

Sur ces critères, nous avons adopté le langage Occam pour notre système de transformation : le langage a été implanté, surtout sur des machines parallèles à base de Transputers. Il a des caractéristiques mathématiques qui permettent la manipulation des programmes, grâce à des modèles formels basés sur CSP [Hoare78].

Dans "le modèle mathématique pour Occam" [Rosc84] il existe une notion claire du "comportement logique", cela est lié aux aspects d'un programme qui ne sont pas affectés par une mise en oeuvre particulière (par exemple, son placement sur un réseau de processeurs, ou la vitesse relative du calcul et de la communication).

Cela nous permet de développer une algèbre de programmes Occam, dans laquelle il existe de nombreuses règles d'inférence sous forme de lois algébriques pour la transformation des programmes. Les transformations possibles concernent souvent des propriétés d'un programme ou l'efficacité de son implantation sur une architecture cible.

2.1.1 Le langage Occam et son modèle de programmation parallèle

Il existe (à ce jour) deux versions du langage. La première version appelée Occam 1, est un prototype ; la deuxième version est Occam 2 [Inmos84, 88]. Par rapport à la version 1, Occam 2 a des caractéristiques nouvelles pour la programmation pratique, surtout en ce qui concerne l'aspect séquentiel et la communication inter-processus. Par exemple, il a un ensemble riche de types de données, la définition de protocoles de communication, etc. Notre système est conçu abstraction faite de l'un ou l'autre version du langage. Notons que par souci d'espace, nous permettons l'écriture des programmes en un format libre, tandis que dans la définition d'Occam, la représentation textuelle est figée.

La Syntaxe d'Occam

En Occam, un programme est un processus composé, construit à partir de processus primitifs et des constructeurs qui permettent la formation des processus composés (séquentiels, parallèles, etc). Un *processus* est un programme, qui peut être considéré comme une boîte noire possédant des états internes, qui communique avec d'autres processus ou son environnement par échange de messages à travers des canaux de communication.

processus primitifs

- Affectation : variable := expression
  Le type de variable peut être simple ou tableau.

- Sortie : canal ! expression
  sortie d'une valeur sur un canal.

- Entrée : canal ? variable
  entrée d'une variable à partir d'un canal.

La communication est synchronisée : la sortie (l'entrée) ne peut avoir lieu jusqu'à ce que le processus d'entrée (de sortie) sur le même canal est prêt.

- Le processus SKIP ne fait rien, et termine immédiatement.
- Le processus STOP ne fait rien, mais ne termine jamais.

Processus composés

Des processus composés sont construits à partir de processus primitifs en utilisant certaines constructeurs. Parmi lesquels SEQ, IF, WHILE et CASE, ont le même sens et la même utilisation que dans les langages classiques. Ils permettent la construction des programmes séquentiels avec les structures conditionnelles et itératives.

Les constructeurs suivants permettent de décrire le parallélisme et le non-déterminisme.

- Processus alternatif

ALT (g1 P1, g2 P2, ...)

Le premier processus Pi dont la garde gi est prête est exécuté. Cette garde peut être une condition booléenne, une entrée ou le processus SKIP.

- Processus parallèle

PAR (P1, P2, ...)

Les processus composants P1, P2, ... s'exécutent en même temps. Le processus composé termine quand tous ses processus composants Pi terminent leur exécution.

Il y a des restrictions sur l'usage des canaux et des variables, dites la validité des processus parallèles : (1) pour les canaux : un seul lecteur, un seul écrvain ; (2) pour les variables : pas de variables partagées modifiées.

- Le réplicateur FOR : avec d'autres constructeurs, il permet de créer un tableau régulier de processus. Par exemple, la structure suivante crée un nombre *count* de processus parallèles :

```
PAR index = base FOR count
  processus
```

Implantation orientée

- Priorité PRI ALT et PRI PAR : pour discriminer l'ordre d'exécution de processus composants.

Les primitives PLACED PAR, PROCESSOR permettent le placement des processus parallèles sur les processeurs spécifiques, et PLACE AT l'allocation de mémoire pour les structures de données.

Les autres structures syntaxiques sont les suivantes :

- Les types de données

Occam 2 fournit un ensemble limité de types de données : entier INT (avec les différentes représentations INT16, INT32, INT64), octet BYTE, réel (REAL32 et REAL64), booléen BOOL et le type tableau. Le type <CHAN OF protocol> définit un canal de communication avec un protocole spécifique. Le type TIMER fournit une horloge.

Il existe aussi la possibilité de définir une procédure <PROC> ou une fonction <FUNCTION> classique.

La programmation parallèle en Occam

Les programmes séquentiels, classiques peuvent être exprimés par des affectations, combinés avec des constructions séquentielles SEQ, conditionnelles IF et itératives WHILE.

Les programmes parallèles sont formulés à partir des processus séquentiels avec les constructeurs PAR et ALT. Des entrées/sorties permettent la communication entre deux processus en parallèle. Les constructions de ALT expriment les choix multiples et non-déterministes. Ainsi le langage permet d'exprimer le parallélisme au niveau syntaxique, ce qui donne aux programmeurs une plus grande liberté d'expression. Un programme est indépendant de l'architecture de la machine sur laquelle il va s'exécuter, ce qui permet la programmation abstraite. En plus, dans une application réelle, le parallélisme, la synchronisation et la communication explicite expriment, d'une façon naturelle, le flot de contrôle, la structure de données du problème à résoudre.

Des programmes Occam ont été développés pour des domaines variés d'application, dont on y trouve une liste non exhaustive [Jons87, Burn89] :

- l'algorithme classique et distribué
- temps réel
- la simulation
- traitement d'image
- parallélisme géométrique : chaque processus exécute les même instructions sur une sous-région de données, appelé aussi la décomposition du domaine d'application.
- parallélisme algorithmique : chaque processus réalise une partie de l'algorithme.
- réseaux de neurones

2.1.2 Modèles formels d'Occam pour la transformation

La principale force d'Occam est dans la rigueur mathématique de sa sémantique. Grâce à la connection entre Occam et CSP [Hoare78], la notation mathématique pure de CSP a pu être appliquée et adaptée à Occam [Rosc84]. Surtout les propriétés algébriques qui permettent la manipulation correcte des programmes CSP sont valables pour des programmes Occam.

Une sémantique dénotationnelle d'Occam

Il existe deux modèles sémantiques pour Occam : dénotationnelle et algébrique ; ils offrent effectivement une façon de décider si deux programmes sont équivalents ou non.

Une sémantique dénotationnelle est composée d'un certain nombre de fonctions, dont chacune affecte une notion abstraite de sens (valeur sémantique) à chaque structure syntaxiquement bien-formée (objet syntaxique) d'une catégorie du langage. La valeur d'un objet composé est complètement déterminée par la valeur du constructeur et celles de ses composants. Deux objets sont équivalents s'ils ont la même valeur pour certaine fonction.

La sémantique dénotationnelle développée dans [Rosc84] utilise un modèle de théorie des ensembles relié au modèle "traces/divergence" de CSP. Donc tous les processus, qui n'ont pas le même ensemble de séquences possibles de communication avec leur environnement, sont différenciés, ainsi ceux qui n'ont pas les mêmes valeurs de leur variables libres à la terminaison.

La sémantique dénotationnelle définit une relation d'équivalence de programmes Occam qui peut être une base pour la transformation de programmes. Il est fastidieux de calculer la valeur sémantique d'une longue séquence de processus équivalents. Aussi elle convient plutôt à l'analyse de relation entre un programme Occam et sa spécification, donc la vérification formelle (par exemple, le comportement d'un programme telle que la terminaison, la divergence et le blocage, etc).

Une sémantique algébrique d'Occam

A partir du modèle de sémantique dénotationnelle, une sémantique algébrique peut être dérivée [HR85], qui prend forme de lois algébriques. Ces lois, qui relient des formes équivalentes d'un programme, constituent des règles dans le système de transformation présenté dans la section suivante.

Définition 2.1 : une *loi* Occam est une règle reliant deux programmes P et Q :

$$P = Q$$

où P et Q sont deux programmes avec des structures syntaxiques probablement différentes. Ils sont équivalents si et seulement s'ils ont la même valeur sémantique dans le modèle [Rosc84].

Une telle loi signifie que les comportements de P et de Q sont identiques pour un observateur dans un environnement, où il ne peut pas détecter la structure interne des programmes P et Q. Pour tout contexte de processus C[.], C[P] et C[Q] sont aussi équivalents sémantiquement pourvu que les syntaxes de C[P] et de C[Q] soient correctes.

Ces lois sont d'utilisation agréable dans le sens où elles obéissent à la règle mathématique d'inter-substitutionalité, permettent aussi une induction structurale sur l'ensemble de formes syntaxiques :

$$\frac{x = y \quad y = z}{x = z} \qquad \frac{x = y \quad F(z) = G(z)}{F(x) = G(y)}$$

Ces lois peuvent être obtenues par deux méthodes : (1) la dérivation à partir de la sémantique dénotationnelle ; (2) la composition ou la dérivation des lois existantes.

Formellement, un modèle algébrique des programmes Occam est composé d'un ensemble de règles d'inférence sous forme de lois algébriques, et d'une relation d'équivalence entre les programmes. Cet ensemble de lois algébriques n'est pas exhaustif, mais complet pour la transformation des programmes équivalents en une forme identique (syntaxiquement). Ainsi ces lois décrivent précisément la sémantique de chaque opérateur Occam.

L'idée principale de la sémantique algébrique, dont le détail sera développé dans la section 2.3, peut être résumée comme suit :

- On transforme chaque programme fini (qui ne contient pas de constructions de boucle WHILE) en une forme normale. Sous cette forme normale, deux programmes sont sémantiquement équivalents si et seulement s'ils ont la même forme syntaxique. En montrant comment transformer chaque programme fini en sa forme normale, on a donc obtenu une procédure de décision pour la vérification de l'équivalence des programmes finis.

- Les lois ne sont pas assez puissantes pour transformer des programmes infinis (qui contiennent des constructions de boucle WHILE) en leur forme normale. Une technique basée sur l'approximation syntaxique permet la représentation d'un programme infini par un ensemble de programmes finis. Notons que cet ensemble peut être infini, en conséquence, l'équivalence de programmes infinis est indécidable, donc cette technique ne présente qu'une valeur théorique.

2.2 Une sémantique de transformation Occam

Des recherches sur les modèles formels de sémantique d'Occam et les méthodes formelles pratiques ont abouti à un système de transformation. Le système permet la manipulation des programmes Occam tout en conservant leur comportement.

2.2.1 Système Oxford de transformation

Une version du Système de Transformation Occam a été réalisée à l'Université d'Oxford, et présentée dans [Gold87, 88]. Toutes les lois dans [HR85], plus quelques autres, ainsi que la stratégie de transformation de programmes en leur forme normale, sont réalisées. Bien que le système soit encore simple et expérimental pour une utilisation réelle et pratique, il forme le noyau du mécanisme de transformation. Nos lois et stratégies nouvelles peuvent être facilement intégrées dans ce système.

Le système de transformation (Figure 2.1), réalisé dans la version ML de l'Université Edinburgh, dispose d'un environnement interactif de programmation pour un langage strictement fonctionnel, avec quelques constructions d'effet de bord. Les notions de type de données et de fonction ML ont facilité la réalisation des lois et stratégies de transformation.

En particulier, il existe un mécanisme pour produire et traiter les exceptions, qui sont utiles pour combiner des stratégies alternatives selon leur domaine d'applicabilité. Grâce à ce mécanisme d'exceptions et des fonctions d'ordre supérieur, des stratégies désignées par la syntaxe peuvent être réalisées.

Quelques exemples de l'utilisation du système de transformation sont donnés dans l'annexe D.

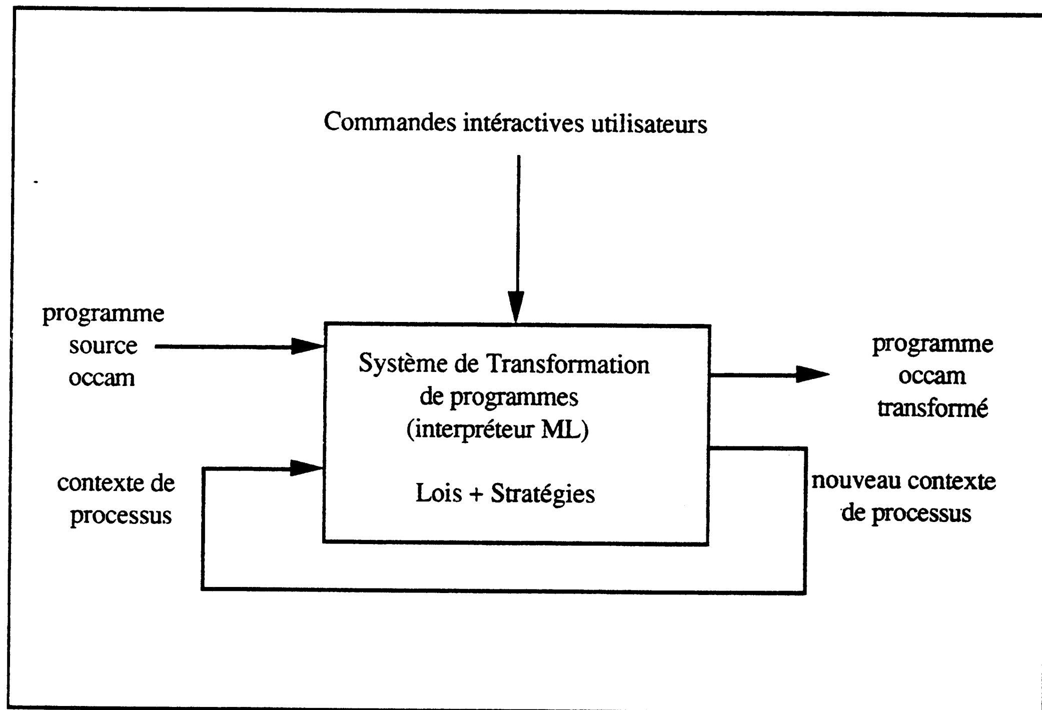

Figure 2.1 Architecture du Système de transformation

Classification du système de transformation

D'après des considérations d'un système de transformation dans la section §1.3 : le système est interactif, a une collection de règles de transformation, adopte un langage parallèle (Occam), permet la définition de nouvelles lois (par la programmation en ML ou une dérivation des lois existantes) et des stratégies (transformation automatique à but), a comme domaine d'application la programmation parallèle, une interface interactive d'utilisateur (E/S, contrôle d'affiche et de la profondeur dans la manipulation des programmes).

Mise en oeuvre du système

Suite à l'analyse syntaxique et sémantique (les premières deux phases dans la Figure 2.2), un programme source est représenté dans une syntaxe abstraite réalisée par des listes récursives (arborescentes) ML. Les codes ML du système entier peuvent être classifiées en trois niveaux conceptuels :

- Au niveau le plus bas, ce sont des types de données ML, correspondant à la syntaxe d'Occam, augmentée avec un processus divergent ZERO, des annotations (utilisation de variables et de canaux) dans les composants d'une construction parallèle, l'affectation multiple et des gardes de sortie dans une construction ALT. Notons que ces extensions sont purement pour le but de faciliter la manipulation formelle.

Le type élémentaire <process> représente des processus Occam, auxquels toutes les lois s'appliquent. Les types <fold> et <unfold>, complémentaire au type <processus>, permettent l'abstraction et la représentation d'un processus. Trois autres types de niveau plus bas <channel>, <variable> et <expression>, permettent la paramétrisation des lois, qui correspondent respectivement aux canaux de communication, aux variables et aux expressions arithmétiques ou logiques.

- Au niveau suivant, on trouve les lois élémentaires dans [HR85] exprimées comme des règles de réécriture sur la syntaxe abstraite, réalisées par des fonctions ML. Cet ensemble de lois est complet, dans le sens que n'importe quelle paire de programmes finis équivalents peuvent être vérifiée uniquement par ces lois.

- Au niveau le plus haut, on trouve un environnement qui supporte des opérations sur un processus et l'application des lois adéquates à ses composants. Cet environnement est réalisé par une pile du type <context>, un important type abstrait dans le système de transformation. Pour faciliter l'utilisation du système interactif de transformation, un composant d'un processus peut être sélectionné et traité en isolation comme le contexte courant, et plus tard, remplacé par son image suite à l'application des lois.

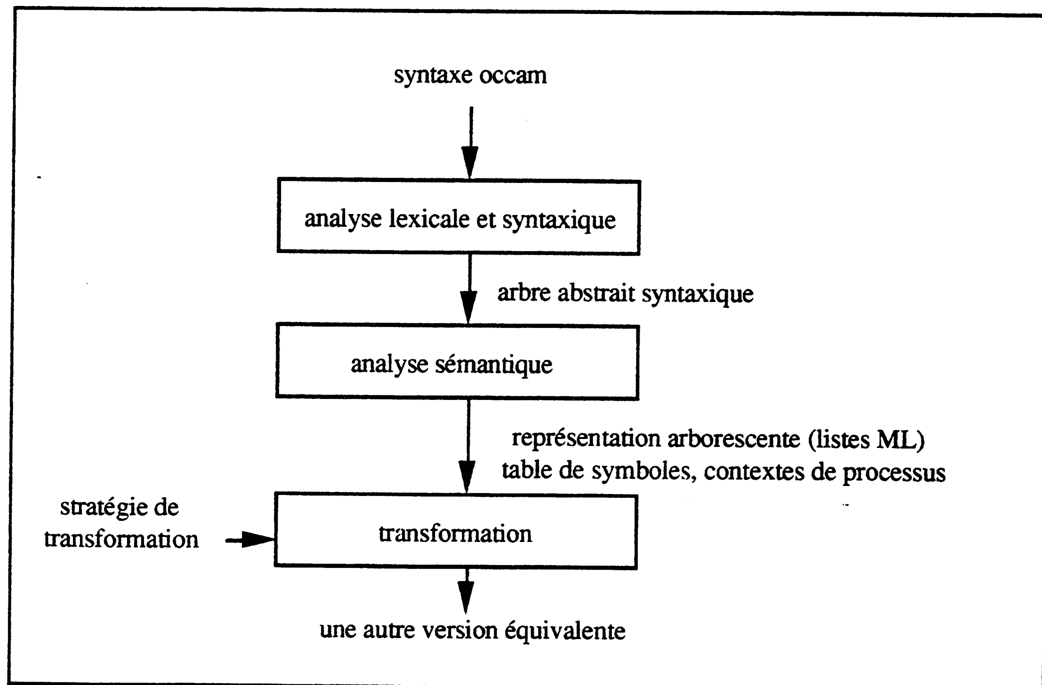

Figure 2.2 Processus d'analyse et de la transformation

Réalisation des stratégies de transformation

Il y a deux approches pour le codage des stratégies disponibles : le codage direct en ML, au niveau des lois, et une séquence de commandes, au niveau des contextes. La première méthode permet la manipulation efficace des structures détaillées d'un processus. Mais elle ne garantit pas une transformation valide, ce qui doit être prouvée par l'utilisateur. Par contre, des manipulations aux niveau de contexte sont restreintes aux combinaisons de stratégies connues, elles sont aussi valides que ses composants.

- Combinaison des lois

Trois formes élémentaires de combinaison des lois Occam existent : THEN, ELSE et REPEAT, dont les définitions sont les suivantes : où f et g sont des lois, x un processus.

```
fun ( f THEN g ) x = g (f x);
fun ( f ELSE g ) x = (f x) handle ? => (g x);
fun REPEAT f x = ( REPEAT f (f x)) handle ? => x;
```

<THEN> est la composition simple de deux lois ; <ELSE> traite l'échec de l'application de la première loi ; <REPEAT> répète l'application de son argument loi.

La relation entre les trois formes de combinaison peur être formulée comme suit :

REPEAT f = (f THEN REPEAT f) ELSE I

où <I> est la fonction d'identité.

- Stratégies au niveau de processus

Ces stratégies typiquement prennent la forme d'analyse des cas sur la syntaxe de leur argument processus.

Un exemple de cette réalisation est la stratégie qui permet la transformation complète des programmes finis en leur forme normale <IF-ALT> (Cf. la section §2.3.1).

- Stratégies au niveau de contextes

A ce niveau, des transformations sont une séquence d'applications de lois sur le contexte courant. Elles sont garanties valides relative aux lois et opérations qui auront lieu. Plusieurs commandes sont offertes pour cet effet : la commande élémentaire est <apply_law>, qui applique une loi sur le contexte courant d'un processus ; une autre combinateur utile pourrait être <branch_apply>, qui prend une loi, l'applique à chaque processus composant du contexte courant.

Un exemple de ce type est la stratégie de la normalisation de l'utilisation des variables et des canaux d'un programme (Cf. la section §3.2).

2.2.2 Interfaces utilisateur du système de transformation

<u>Entrée/Sortie</u>

- Entrée

La commande fondamentale d'entrée est <parse>, qui prend le nom de fichier d'un programme source Occam, et retourne le type <process> du programme. Deux autres commandes <consider> et <contemplate> prennent le même argument que <parse>, et retournent respectivement le type <fold> et <context> (Cf. §2.2.1).

Les commandes suivantes permettent l'entrée (le nom d'un fichier, d'une variable, d'un processus etc) à partir de l'entrée standard (par exemple, clavier) :

read : unit -> string
read_string : unit -> string
read_int : unit -> int
read_var : unit -> variable
read_assign : unit -> processus

-Sortie

Les commandes de sorties concernent le type <processus> :

file_process : prend un nom d'un fichier et dans lequel met le texte du <processus> courant ;
file_fold : prend un nom d'un fichier et dans lequel met le <fold>courant ;
file : prend un nom d'un fichier et dans lequel met le <context> courant;

Les commandes <view_process>, <view_fold>, <view> fonctionnent comme <file_...>, mais placent les sorties sur la sortie standard (écran, par exemple).

- Contrôle utilisateur

Le système de transformation offre aux utilisateurs certaines facilités de contrôler son fonctionnement, en ce qui concerne l'affiche de sortie :

<ECHO> : détermine si la commande <parse> doit afficher une copie de texte ;
<PRINT_USING> : contrôle si les annotations d'une construction parallèle seront affichées ou pas.

Les commandes suivantes contrôlent une variable globale de profondeur du processus courant, qui représente le nombre des constructeurs qui l'entourent. Cette variable de profondeur influence l'effet des commandes comme <view> (afficher un processus du contexte courant), qui affiche uniquement les processus composants qui ont une profondeur inférieure à sa valeur :

<limitdepth n> affecte *n* à la variable ;
<unsetdepth> lui affecte <infinity> ;
<moredepth> et <lessdepth> augmente ou réduit sa valeur si elle a une valeur <finite>, sinon aucun effet.

La manipulation de programmes au niveau de contextes

A ce niveau, des lois opèrent sur le processus et ses processus composants du contexte courant. La manipulation des contextes est réalisée par application d'un certain nombre de commandes. Parmi ces commandes,

move_in *n* : sélectionne le *n*-ème composant d'un processus.

move_out : remplace le composant.

new_context : crée un nouveau contexte avec le processus argument comme le point d'intérêt.

zoom_in *n* : permet de concentrer sur le *n*-ième contexte de l'environnement du système.

apply *loi* : remplace le processus par son image lors de l'utilisation de *loi*.

view : visualise (affiche) le processus courant.

pan_out : valide le processus modifié, et le contexte immédiatement externe devient le contexte courant.

re_take : au contraire de <pan_out>, revient à l'état de la dernière d'opération de <zoom_in> pour annuler les modifications sur le processus courant.

modify_context : remplace le contexte courant par un nouveau contexte.

2.2.3 Lois d'Occam

Le système de transformation supporte la syntaxe d'Occam définie dans [Inmos88], avec des omissions et des additions suivantes.

Omissions

- l'aspect de temps : les constructions de TIMER sont traitées comme des entrées standard ;

- La configuration et les performances liées : priorité PRI PAR, PRI ALT, des primitives de placement PLACED PAR, PRI PLACED PAR etc.

Additions

- Le processus ZERO

On distingue le processus divergent (livelock), qui est défini ci-dessous, avec le processus STOP (deadlock).

ZERO = WHILE (TRUE SKIP)

ZERO représente le comportement imprévisible d'un programme, comme le blocage vivant, des erreurs ou exceptions arithmétiques etc. Pour une comparaison de ZERO et STOP, prenons un exemple : le premier processus représente une séquence infinie des communications internes d'un processus.

```
ZERO = CHAN of INT c:
      PAR (WHILE TRUE c ! ANY,
           WHILE TRUE c ? ANY)
```

STOP = IF (FALSE SKIP)

- Des sorties dans une garde de processus ALT

Des gardes de sortie apparaissent seulement pendant les phases intermédiaires de transformation, comme illustrées ci-dessous, elles seront éliminées éventuellement.

PAR (c ! ANY, d ? ANY)

ALT (c ! ANY d ? ANY, d ? ANY c ! ANY)

- On exige que chaque processus composant d'une structure parallèle indique, d'une façon explicite, l'usage des canaux globaux comme entrée ou sortie, et des variables libres en lecture ou en écriture. Notons qu'en pratique, c'est le système de transformation qui insère automatiquement ces informations lors de l'analyse sémantique et l'analyse d'espace d'états (la commande <parse>, Figure 2.2).

L'usage des canaux et des variables est formulé sous forme d'annotations suivante, et à ne pas confondre avec des déclarations :

PAR (Ui : Pi) -- Ui est l'annotation pour le processus Pi.

Quelques notations

Dans la description des lois algébriques ci-après, les notations suivantes sont utilisées comme un méta-langage :

P, Q	processus
v, x, y, z	variables simples
e, f	expressions générales arithmétiques ou logiques
b	expression booléen
g	gardes alternatives dans une construction ALT
c, d	canaux de communication
U, Ui	annotation de l'utilisation de canaux et de variables
$\underline{x}$	une liste de variables simples dont le type est le même que x
P[x/y]	la substitution de toute occurrence libre de y dans le processus P par x. De même pour l'expression $\underline{f[}\ \underline{e}\ /\ \underline{x}\ ]$.
libre(P)	l'ensemble de canaux et de variables libres du processus P
lié(P)	l'ensemble de canaux et de variables liés du processus P
IF $b_i\ P_i$,	i=1...n dénote IF($b_1\ P_1, \ldots, b_n\ P_n$). Similaire pour les autres opérateurs.

Nous adoptons deux formes suivantes pour formuler des lois Occam, elles expriment l'équivalence des processus P et Q : la première est une règle de réécriture, la deuxième est une forme de déduction.

$$P = Q \qquad \text{et} \qquad \frac{P}{Q}$$

Lois d'Occam

Les lois d'Occam dans le système de transformation sont classifiées en deux catégories : (1) les lois élémentaires, dans le sens où ces lois forment une complétude ; et (2) des lois orientée-application, dérivées de lois élémentaires. Contrainte de l'espace, nous présentons quelques lois qui expriment des propriétés élémentaires de différentes structures et leur effet de transformation. Une liste complète des lois élémentaires est dans l'annexe A1.

Notons que quand une loi P = Q est appliquée à un programme correct C[P], il est possible de le transformer en un autre syntaxiquement incorrect C[Q]. C'est à cause de la violation de condition de validité des processus PAR (Cf §2.1.1). Ces lois sont marquées avec le symbole *.

Lois de IF

IF ( ) = STOP <IF-STOP)

Une structure conditionnelle vide de IF se comporte comme STOP.

IF ($\underline{C1}$, IF($\underline{C2}$), $\underline{C3}$)
---------------------------- <IF-assoc>
IF ($\underline{C1}$, $\underline{C2}$, $\underline{C3}$)

Cette loi exprime l'associativité de IF.

IF (b_1 P, b_2 P, $\underline{C}$)
---------------------- <IF-v distrib>
IF (b_1 v b_2 P, $\underline{C}$)

Si deux booléens gardent le même processus, alors ils peuvent être combinés en une seule condition.

IF (FALSE P, $\underline{C}$)
--------------------- <IF-FALSE unit>
IF ($\underline{C}$)

Une garde FALSE ne peut jamais être activée, donc elle peut être supprimée. Cela représente l'élimination d'un code mort dans la compilation optimisée.

Lois de CASE

CASE ( ) = STOP <CASE-STOP unit>

S'il n'y a pas d'arguments, la construction de CASE est équivalent à STOP.

CASE (CASE (S1), S2)

CASE (S1, S2) <CASE assoc>

cette loi permet de supprimer les emboîtements de CASE.

CASE Si

CASE $S\pi(i)$ où π est une permutation de {1 ... n}. <CASE sym>

L'ordre des arguments dans la structure de CASE n'est pas important.

Lois de ALT

Dans les lois suivantes, une garde simple est sous forme : SKIP, c?x ou c!e.

ALT ( ) = STOP <ALT-STOP unit)

Une structure alternative vide de ALT se comporte comme STOP.

ALT (ALT (G1), G2)
———————————
ALT (G1, G2) <ALT assoc>

cette loi permet de supprimer les emboîtements de ALT.

ALT Gi = ALT $G\pi(i)$ où π est une permutation de {1 ... n}. <ALT sym>

L'ordre des arguments dans la structure de ALT n'est pas important.

ALT (b & g P, G)

IF (b ALT (g P, G), ¬b ALT(G)) <bool-gard elim>

Une structure de ALT avec des gardes contenant des composants booléens peut être réduite à la combinaison de IF et ALT, avec des gardes simples.

Lois de l'affectation

On permet l'affectation multiple sous forme: x := e, où x et e sont respectivement une liste de variables simples et une liste d'expressions. La longueur de x et de e est la même.

$<x_i \mid i=1 \ldots n> := <e_i \mid i=1 \ldots n>$

-- <assign sym>

$<x_{\pi(i)} \mid i= 1 \ldots n> := <e_{\pi(i)} \mid i=1 \ldots n>$

où π est une permutation de $\{1 \ldots n\}$.

L'ordre d'affectation d'expression/variable n'a pas d'importance.

L'affectation multiple est utile pour résumer la dépendance de données dans un bloc élémentaire, donc permet l'extraction du parallélisme dans une dimension locale.

Lois de SEQ

Les deux lois suivantes permettent de transformer toute occurrence de SEQ en une constructeur binaire.

SEQ ( ) = SKIP <SEQ-SKIP unit>

Si SEQ n'a pas d'arguments, il termine simplement.

SEQ (P, SEQ($\underline{P}$))

---------------------- <SEQ assoc>

SEQ(P, $\underline{P}$)

Cette loi exprime l'associativité de SEQ.

La relation entre SEQ et IF, ALT :

* SEQ(IF b_i P_i, $\underline{Q}$)

---------------------- $i=1 \ldots n$ <SEQ-IF distrib>

IF b_i SEQ(P_i, $\underline{Q}$)

SEQ distribue sur IF.

* SEQ(ALT g_i P_i, $\underline{Q}$)

------------------------ $i=1 \ldots n$ <SEQ-ALT distrib>

ALT g_i SEQ(P_i, $\underline{Q}$)

SEQ distribue sur ALT.

* SEQ($\underline{x}$:= $\underline{e}$, $\underline{x}$:= $\underline{f}$)

------------------------ <combin assign>

$\underline{x} := \underline{f}\,[\underline{e}\,/\underline{x}]$

La composition séquentielle de deux affectations à la même liste de variables simples peut être combinées en une seule, en tenant compte de l'effet de l'affectation précédente. Cela représente la propagation des valeurs.

Lois de PAR

Dans les lois suivantes, l'expression U (ou Ui, U*) dénote l'annotation des canaux et des variables, et à ne pas confondre avec des déclarations.

PAR ( ) = SKIP <PAR-SKIP unit>

$$\frac{\text{PAR } (U_1\text{: } P_1\text{, } U^*\text{: (PAR } U_j\text{: } P_j))}{\text{PAR } U_i\text{: } P_i} \quad i=1...n,\ j=2...n$$

<PAR assoc>

où l'annotation U* est l'union des $U_2, \ldots, U_n$. Notons que U* indique des canaux comme internes si ces canaux sont déclarés comme entrées et sorties entre U_i. Cette loi exprime l'associativité de PAR.

$$\frac{\text{* PAR}(U_1\text{: IF } b_i\ P_i\text{, } U_2\text{: Q})}{\text{IF } b_i \text{ PAR}(U_1\text{: } P_i\text{, } U_2\text{: Q})} \quad i=1...n$$

<PAR-IF distrib>

à condition que $b_1 \mathrm{v} \ldots \mathrm{v}\ b_n$ = TRUE.
Le choix conditionnel peut être exécuté avant d'entrer dans une construction de PAR. La condition b_1 v. . .v b_n = TRUE garantit que le processus conditionnel puisse entrer dans PAR.

$$\frac{\text{* PAR}(U_1\text{: ALT } g_i\ P_i\text{, } U_2\text{: } \underline{x} := \underline{e})}{\text{ALT } g_j \text{ PAR}(U_1\text{: } P_j\text{, } U_2\text{: } \underline{x} := \underline{e})} \quad i=1...n,$$

<PAR-ALT expansion1>

où g_j sont des gardes simples.
$j \in X$, X est un ensemble d'indices $i \in \{1, \ldots, n\}$ telle que :
g_j = SKIP,
ou g_j = c!e et $c \in$ output(U_1) - input(U_2),
ou g_j = c?x et $c \in$ input(U_1) - output(U_2).

Cette loi permet la suppression des constructions PAR en les transformant en celles de ALT (avec le non-déterminisme).

Lois de déclarations

Ces lois permettent de transformer tout programme en une forme canonique : toutes les déclarations de variables et de canaux sont placées juste avant les constructions qui les utilisent (Cf. §3.4.1.2).

VAR x: P
---------------- $y \notin$ libre(P) <VAR rename>
VAR y: P[y/x]

On peut changer le nom d'une variable liée.

Il existe des lois similaires de déclaration pour les canaux :

CHAN c_1: (CHAN c_2: . . . CHAN c_n: P). . .)
-- <CHAN assoc>
CHAN c_1, . . ., c_n: P

VAR x_1: (VAR x_2: . . . VAR x_n: P). . .)
--- <VAR assoc>
VAR x_1, . . ., x_n: P

Une séquence de déclarations peut être combinée en une déclaration multiple.

VAR x_1: (VAR x_2: P)
------------------------- <VAR sym>
VAR x_2: (VAR x_1: P)

L'ordre de déclaration n'a pas d'importance.

VAR x: P
---------- pourvu $x \notin$ libre(P), <VAR elim>
P

Cela représente l'élimination d'une variable morte.

Lois de ZERO

On identifie ZERO avec tout processus qui est possible de diverger.

ZERO = WHILE TRUE SKIP <ZERO>

Lois de WHILE

WHILE b P

IF(b SEQ(P, WHILE b P), ¬b SKIP) <WHILE expansion>

C'est la définition récursive d'une boucle WHILE.

WHILE b_1 (WHILE b_2 P)

WHILE b_1 v b_2 IF(b_2 P, TRUE ZERO) <WHILE combine>

Les emboîtement de conditions booléens peuvent être combinés en un seul.

Transformations d'autres structures en Occam 2

- Type tableau

Nous traitons des variables de type tableau comme une liste de variables. Par exemple, une affectation de tableau sera transformée en l'affectation multiple sous la forme : $\underline{x} := \underline{e}$. A noter que le système de transformation n'assure pas l'effet d'affectation à des variables de type tableau. Par exemple, la phrase suivante ne peut pas être traitée avec la loi <combin assign> :

SEQ (i = 1, a[i] = 0, i = 5, a[i] = 1)

Une transformation similaire pour des tableau de canaux (avec le même protocole).

- Les types de données

Les lois sont valides pour les types primitifs de données : BOOL, BYTE, INT, REAL, <CHAN OF type> et le type de tableau : <[expression] type> ; ainsi les types tableau dynamique, variant et enregistrement, qui ne peuvent apparaître que dans des entrées et sorties de communications. Donc on utilise uniquement le mot VAR pour les déclarations de variables de types différents, à condition que le type d'une variable soit conservé pendant toutes les transformations d'un programme. Une variable de n'importe quel type doit obéir la loi suivante :

$$\frac{\mathrm{PAR}(c\ !\ x,\ c\ ?\ y)}{y := x}$$

- Canaux avec des protocoles de communications

- Protocoles simples

Dans les lois, on considère seulement les protocoles de type entier INT. Dans la manipulation des programmes qui contiennent des protocoles d'autres types de données, il faut garantir la même condition de l'affectation distribuée, c'est-à-dire :

$$\frac{\mathrm{PAR}\ (\ c\ !\ e,\ c\ ?\ x)}{x := e}$$

• Protocoles discriminés

Ce sont des protocoles sous formes :

```
PROTOCOL nom IS CASE (tag1; type1
                      tag2; type2)
```

On transforme les communications de protocoles discriminés, en constructions de ALT avec les communications de protocoles simples :

```
c ? CASE
  tag1; donne1
    P1
  tag2; donne2
    P2
------------------------------
CHAN OF type1 chanel1 :   -- type1 est le type de donne1
CHAN OF type2 chanel2 :   -- type2 est le type de donne2
ALT
  chanel1 ? donne1
    P1
  chanel2 ? donne2
    P2
```

- Procédure et fonction

On traite un appel d'une procédure par la substitution du texte de définition de la procédure, avec le renommage nécessaire des paramètres formels par les paramètres actuels.

Par exemple, le programme suivant qui compare les valeurs a et b, et met les valeurs minimum et maximum aux variables *min* et *max* respectivement.

```
PROC min.max (INT min, max, VAL INT a, b)
  IF
    a <= b
      min, max := a, b
    a >= b
      max, min := a, b
:
INT min1, max1 :
VAL INT data1, data2 :
... Traitement de data1, data2
min.max (min1, max1, data1, data2)
```

L'appel de la procédure *min.max* sera traité par la substitution du corps de la procédure, ce qui donne le programme équivalent, comme suit :

```
INT min, max :
VAL INT data1, data2 :
... Traitement de data1, data2
IF
   data1 <= data2
      min, max := data1, data2
   data1 >= data2
      max, min := data1, data2
```

La notion de fonction permet l'abstraction des valeurs. Il existe des fonctions anonymes et nommées en Occam 2. Nous traitons les fonctions d'une façon similaire comme la procédure, c'est-à-dire, la substitution d'un appel par le texte de sa définition.

Par exemple, le programme suivant sera transformé en un équivalent ci-dessous, avec la substitution des variables résultats dans la fonction.

```
SEQ
   ... Calculer [] INT data
   sum := ( INT sum.local
               VALOF
                  SEQ
                     sum.local := 0
                     SEQ i = 0 FOR SIZE data
                        sum.local := sum.local + data[i]
                  RESULT sum.local)
-------------------------------------------------------------
SEQ
   ... Calculer [] INT data
   SEQ
      sum := 0
      SEQ i = 0 FOR SIZE data
         sum := sum + data[i]
```

Notons qu'en Occam, il n'y a pas de définition récursive pour les fonctions et les procédures. Notre traitement ne convient pas aux fonctions et procédures récursives, s'il y en a.

2.3 Vérification formelle des programmes parallèles

2.3.1 Formes normales

Les lois présentées dans ce chapitre et la relation d'équivalence de programmes forment une sémantique algébrique pour Occam (Cf. §2.1). Dans ce modèle de sémantique, les programmes qui ont la même valeur sémantique peuvent être transformés en une même représentation syntaxique. Par le développement et la définition d'une forme normale syntaxique, et un algorithme de décision qui change chaque programme fini en cette forme normale (uniquement par l'application des lois dans le système de transformation), on obtient ainsi une procédure de vérification formelle d'équivalence de programmes finis (Voir la figure 2.3).

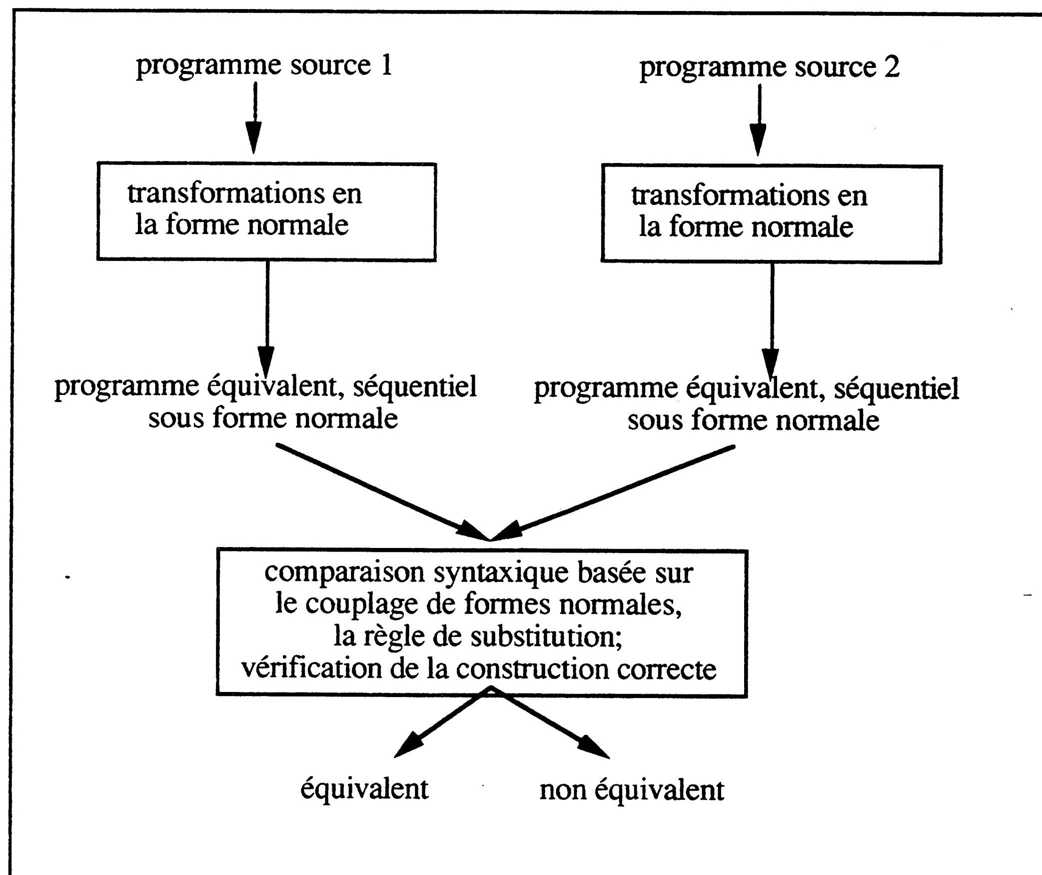

Figure 2.3 La vérification d'équivalence de deux programmes

La définition d'une forme normale consiste à identifier et représenter, d'une façon unique, le comportement d'un programme en ce qui concerne la terminaison, l'état final de ses variables libres, et le flot de contrôle (y compris le non-déterminisme, exprimé par des gardes SKIP dans une construction ALT).

Définition 2.2 *La forme* $\underline{x}$*-IF/ALT* est un programme sous une de ces formes suivantes :

ZERO -- le processus divergent.

$\underline{x} := \underline{e}$ -- l'affectation multiple de la liste de variables $\underline{x}$.

IF b_i P_i, i=1...n
où P_i est sous $\underline{x}$-IF/ALT forme et b_1 v. . .v b_n =TRUE,
$b_i \wedge b_j$ = FALSE, pour $i \neq j$.

VAR $x_1, \ldots, x_n$: ALT g_i P_i, i=1...n
où P_i est sous forme $\underline{x}$-IF/ALT,
g_i sont des gardes simples : SKIP , c ! e ou c ? x,
$x_1, \ldots, x_n$ sont les variables utilisées dans les gardes c ? x, et disjointes de $\underline{x}$ et de liés(P).

VAR x: P
où $x \in$ libre(P), $x \notin \underline{x}$, et P est sous forme $\underline{x}$-IF/ALT.

L'interprétation de cette définition est que sous cette forme, un programme fini est un processus divergent, sinon il peut être représenté par un arbre de décision qui branche, alternativement via <IF> et <ALT>, aux feuilles de l'arbre, qui sont des affectations multiples aux variables libres (le résultat du calcul) du programme. L'utilisation des variables est bien normalisée : une variable est soit libre, et affectée une valeur uniquement une fois à la fin du programme ; soit déclarée immédiatement avant une construction ALT, où elle apparaît dans une garde d'entrée.

Théorème 2.1 : Si la liste $\underline{x}$ contient toutes les variables auxquelles un programme fini P effectue ses entrées ou ses affectations, alors il existe un programme P′ sous la forme $\underline{x}$-IF/ALT, tel que libre(P) $\cup \underline{x} \in$ libre(P′), et P = P′ est prouvable dans le système de transformation.

Le principe de la démonstration de ce théorème est que chaque programme fini peut être transformé en la forme IF/ALT, en appliquant les lois existantes, avec l'analyse des cas sur la syntaxe possible du programme.

Une stratégie de la démonstration pour ce théorème est la suivante :

(1) Dans la première étape on transforme toutes les constructions SEQ et PAR en applications binaires, par les lois <SEQ-SKIP unit>, <PAR-SKIP unit>, <SEQ assoc> et <PAR assoc>. Ensuite on supprime l'emboîtement des constructions IF et ALT, en déplaçant les conditions booléennes dans les gardes des constructions ALT, avec les lois <ALT sym>, <bool gard elim>.

L'objectif des ces transformations est de réduire toutes les constructeurs en l'application binaire, afin de les appliquer des lois (toutes les lois exigent des constructeurs binaires).

(2) Le reste de cette stratégie est l'application récursive des transformations suivantes, sur toutes les formes syntaxiques possibles qu'un programme peut prendre.

Les processus primitifs sont évidents :

STOP = ALT( ) <ALT-STOP unit)

SKIP = $\underline{x} := \underline{x}$ <SKIP> et <assign ident>

$$\frac{x := e}{\underline{x} := \underline{x}[e/x]}$$ <assign sym> et <assign ident>

$$\frac{c \,!\, e}{ALT(c \,!\, e\ \underline{x} := \underline{x})}$$ <output> , <SKIP> et <assign ident>

$$\frac{c \,?\, x}{VAR\ y: ALT(c \,?\, y\ \ x := x\,[y/x])}$$

<input> <input rename>, <SKIP>, <assign ident>, <assign sym> et <combin assign>

Ensuite, on traite les formes possibles par induction structurelle, en appliquant récursivement des lois nécessaires :

IF b_i P_i, i=1...n
ALT g_i P_i, i=1...n
traiter Pi, supprimer l'emboîtement de IF, par <IF-ALT distrib>.

CASE c_i P_i, i=1...n
cette construction peut être transformée en une construction IF par <CASE-IF>.

SEQ (P, Q)
par les lois de <SEQ-Op distrib>, et les lois d'affectations multiples.

VAR y: P ou CHAN $C_1, \ldots, C_n$: P
appliquer la stratégie entière sur P.

PAR (U_1: P, U_2: Q)
par les deux lois d'expansion, cette construction parallèle peut être transformée en une construction ALT.

Chaque application des lois d'expansion diminue le nombre des constructions parallèles, donc le nombre des communications. La procédure est garantie de terminer quand il n'existe plus de construction parallèle. Les autres constructions peuvent être facilement traitées par les lois de distribution de IF, ALT. fi

Cette forme ne prend pas en compte l'équivalence due au non-déterminisme (par exemple, des communications internes) d'un programme, non plus à l'équivalence des programmes par le renommage de variables et de canaux de communication. Elle ne peut pas être donc la forme normale que nous cherchons.

Définition 2.3: *La b, x-forme normale* est un programme sous la forme :

IF bi Pi, i=1...n
où $b_1 \vee \ldots \vee b_n$ = TRUE et $b_i \neq$ FALSE pour tout i.
Les processus Pi sont sous formes distinctes b_i, $\underline{x}$-ALT.

Définition 2.4 *b, x-ALT forme normale* est un programme sous la forme :

ZERO ou

VAR $y_1, \ldots, y_n$: ALT g_i P_i , i=1...n

et il existe K, L, N : $0 \leq K \leq L \leq N$ tels que :

$1 \leq i \leq K$ -- implique que g_i est sous forme c?y_j ou c!e. P_i est sous la b,$\underline{x}$-forme normale.Tous les canaux d'entrée sont distincts et les variables d'entrée sont précisément :

$y_1, \ldots y_n \notin \underline{x}$.

liés(P_i) est disjoint de libre(P), de { $y_1, \ldots, y_n$ }, et de $\underline{x}$.

$K < i \leq L$ -- implique que g_i est SKIP. P_i est sous forme ALT g_j P_j , $j \in X_i$ où les Xi sont des sous-ensembles de $\{1, \ldots, K\}$, avec la propriété suivante :

si g_r = c ! e et g_s = c ! f, alors $s \in Xi \Leftrightarrow r \in X_j$.

$L< i \leq N$ -- implique que g_i est sous forme : SKIP, et P_i est $\underline{x} := \underline{e}_i$ où $\underline{e}_i$ est une liste d'expressions générales.

L'interprétation de cette forme normale est que les K premières gardes correspondent aux communications possibles du processus avec son environnement. Les K-L gardes correspondent aux combinaisons minimales de communications que le processus peut choisir d'une façon non-déterministe. Les dernières N-L conditions expriment les états finaux (représentés par des affectations multiples) du processus.

Deux programmes sous la forme normale $\underline{x}$-ALT sont équivalents s'il existe une permutation entre les trois composants abstraits. D'où on définit un couplage entre deux programmes sous la forme normale.

Définition 2.5 *Le couplage (matching) de forme ALT*:

soient $P = \text{VAR } x_1, \ldots, x_m$: ALT g_i P_i, $i = 1, \ldots, n$ avec :

$K< i \leq L \Rightarrow g_i = \text{SKIP}$ et $P_i = \text{ALT } g_j\, P_j, j \in X_i$

et $L< i \leq N \Rightarrow g_i = \text{SKIP}$ et $P_i = \underline{x} := \underline{e}_i;$

et $Q = \text{VAR } y_1, \ldots, y_m$: ALT h_i Q_i, $i = 1, \ldots, n$ avec :

$K^*< i \leq L^* \Rightarrow h_i = \text{SKIP}$ et $Q_i = \text{ALT } h_j\, Q_j, j \in Yj$

et $L^*< i \leq N^* \Rightarrow h_i = \text{SKIP}$ et $Q_i = \underline{x} := \underline{f}_i$

respectivement b et b*, $\underline{x}$-ALT formes.

Si N=N*, m=m*, K=K* et L=L*, alors un couplage entre P et Q est un quadruple :

<S, T, V, W> de bijections :

$S : \{1, \ldots, m\} \rightarrow \{1, \ldots, m\};$

$T: \{1, \ldots, K\} \rightarrow \{1, \ldots, K\};$

$V : \{K+1, \ldots, L\} \rightarrow \{K+1, \ldots, L\};$

$W : \{L+1, \ldots, N\} \rightarrow \{L+1, \ldots, N\}.$

telles que:

(a) si $g_i = c?x_j$ alors $h_{T(i)} = c?y_{S(j)}$;
si $g_i = c!e$ alors $h_{T(i)} = c!e^*$ pour certain e*.

(b) $Y_{V(i)} = \{T(j) \mid j \in X_i\}$

(c) e_{ij} et f_{ij} dénotent respectivement le j-ème composant des listes $\underline{e}_i$ et $\underline{f}_i$, alors

$$b \models e_{ij} < e_{kj} \Leftrightarrow b^* \models f_{w(i)j} < f_{w(k)j}$$
$$b \models e_{ij} = e_{kj} \Leftrightarrow b^* \models f_{w(i)j} = f_{w(i)j}$$
$$b \models e_{ij} > e_{kj} \Leftrightarrow b^* \models f_{w(i)j} > f_{w(i)j}$$

où b |= e signifie que dans la condition b, l'expression logique e est vrai.

Jusqu'à maintenant, dans le système de transformation, on ne prend pas en compte des expressions générales arithmétiques ou logiques, et leur équivalence. On laisse aux utilisateurs le soins d'indiquer si deux expressions sont équivalentes. La raison est que le problème de l'équivalence de deux expressions générales est indécidable [HR85].

Définition 2.6 *La Règle de substitution d'expressions générales.*

(a) Si e est une expression apparaissant dans P, et b |= e=e', alors si P', obtenu en remplaçant e par e', est correct , on a P = P'.

(b) Si b |= e=e' alors :

IF b ALT(c ! e P, $\underline{G}$)

IF b ALT(c ! e' P, $\underline{G}$)

(c) Si b |= e=e' alors :

IF b x:=e

IF b x:= e'

Sur l'hypothèse de cette règle de substitution, on peut définir l'équivalence des programmes sous forme normale.

Définition 2.7 *La relation d'équivalence de formes normales:*

a) Deux programmes sous b, $\underline{x}$-forme normale :

IF b_i P_i et
IF b_i' P_i', i=1...n

sont équivalents si et seulement si :

n = n',

et il existe une bijection :

s: $\{1,\ldots,n\} \longrightarrow \{1,\ldots,n\}$ telle que pour tout $i \in \{1, ..., n\}$, on a :

$b_i = b_{s(i)}$' et P_i est équivalent à $P_{s(i)}$'.

b) P et Q sous b, $\underline{x}$-ALT forme normale sont équivalents si et seulement s'ils sont soit ZERO, soit :

$P = VAR\ x_1,\ldots,x_n$: ALT $g_i\ P_i$, i=1, ..., n
avec $K< i \leq L \Rightarrow g_i = SKIP$ et $P_i = ALT\ g_i\ P_i$,
$L< i \leq N \Rightarrow g_i = SKIP$ et $P_i = x := \underline{e}i$;

$Q = VAR\ y_1,\ldots,y_n$: ALT $h_i\ Q_i$ i=1, ..., n
avec $K< i \leq L \Rightarrow h_i = SKIP$ et $Q_i = ALT\ h_j\ Q_j$ et
$L< i \leq N \Rightarrow h_i = SKIP$ et $Q_i = \underline{x} := \underline{f}_i$.

et il existe une permutation entre les composants de P et Q, c'est-à-dire, il existe un couplage entre P et Q (Cf. la définition 3.4).

Théorème 2.2 : Si la liste $\underline{x}$ contient toutes les variables auxquelles le programme P effectue ses entrées ou ses affectations, et si P n'évalue jamais une variable non-initialisée, alors il existe un programme P' sous *TRUE, $\underline{x}$-forme normale* tel que :

libre(P) $\cup\ \underline{x} \in$ libre(P'),

et P = P' peut être démontré avec les lois élémentaires et la régle de substitution d'expressions générales.

La démonstration de ce théorème est similaire que celui du théorème 2.1, dont la procédure est omise ici.

Un programme résultat sous la forme normale a une taille à peu près exponentielle par rapport à sa taille initiale. Donc cette représentation n'est pas pratique comme programme exécutable, mais il permet une comparaison directe des programmes équivalents à vérifier. Les méthodes et l'application pour la vérification formelle sont présentées dans la section suivante.

2.3.2 La preuve formelle d'équivalence de programmes

La vérification formelle d'un programme est critique et indispensable pour le développement des logiciels. Les méthodes et techniques formelles dans le système de transformation sont prometteuses pour cet aspect d'application pratique (par exemple, des systèmes vérifiés).

2.3.2.1 Programmes finis

La procédure de la transformation des programmes finis en leur forme normale peut être considérée comme une stratégie de preuve de l'équivalence. Sous cette forme, il suffit de comparer la forme syntaxique de deux programmes sous forme normale, sur la base de la règle de substitution d'expressions.

<u>**Réalisation dans le système de transformation**</u>

Dans le système de transformation, un type abstrait <normal_form> a été défini pour représenter la forme normale (arbre de décision IF/ALT). Sa manipulation (pour la transformation des programmes finis à leur forme normale) a été réalisée au niveau de processus (Cf. §2.2.1), dont les commandes disponibles sont les suivantes :

normal_df : process -> normal_form
normal_bf : (int * int) -> process -> normal_form

<normal_df> (par profondeur) change un programme fini complètement à la forme <IF/ALT> ; <normal_bf (w, s)> (par largeur) produira les premières <s> communications tout au long de chaque chemin dans l'arbre de décision (syntaxique) d'un programme infini, ne nécessitant que de dérouler <w> fois une structure composante <WHILE>. Ceci évite le comportement divergent du processus de transformation à cause du comportement divergent possible du programme en question (par exemple, boucles infinies).

Des résultats de ces deux commandes sont adéquates pour une manipulation plus profonde de la forme normale. Dans le cas qu'on souhaite utiliser des formes normales comme le type <process>, afin de les transformer ; ou bien générer une forme normale dans un contexte (qui contient des objets de type <process>). Ce qui demande une fonction pour transformer une forme normale à la représentation <process> :

convertnf : (liste de variables) -> normal_form -> process

La commande demande une liste de variables libres changées dans le programme. La composition des commandes <normal_df>, <normal_bf> avec cette fonction nous donne les commandes suivantes, qui concernent des lois reliant des programmes sous forme normale :

nf_df : loi
nf_df : (int * int) -> loi

où des arguments <int> ont le même sens comme dans la commande <normal_bf>.

Les autres commandes concernent des entrées/sorties :

print_nf : (string * normal) -> unit
file_nf : (string * normal) -> normal_form
show_nf : normal_form -> normal_form

<print_nf> et <show_nf> permettent l'affichage d'un programme sous forme normale, <file_nf> met le contenu dans un fichier.

2.3.2.2 Programmes infinis et leur approximation syntaxique

Approximation syntaxique

Cette technique permet la représentation d'un programme général (qui contient probablement des boucles WHILE) par un ensemble (probablement infini) de programmes finis. Les programmes finis de cet ensemble sont plus restreintes que le programme initial.

Définition 2.8 *L'approximation syntaxique.*
Un programme P est plus restreinte qu'un autre programme Q s'il existe la relation entre eux :

Q = ALT(SKIP P, SKIP Q)

Cette relation exprime que chaque comportement du programme P est aussi celui de Q. On dit dans ce cas-là P est une approximation syntaxique de Q.

Toutes les approximations syntaxiques d'un programme P forment un ensemble fini, dans le cas où P est fini ou contient des boucles avec un nombre constant d'itération; sinon il est infini.

On a un résultat disant que la sémantique algébrique d'un programme est déterminée complètement par son ensemble d'approximations syntaxiques [HR85]. Ceci étend le résultat pour les programmes finis (la complétude des lois dans le système de transformation) à ceux infinis.

Cette technique est intéressante pour des études théoriques. Mais elle ne peut pas être un candidat pour la procédure de la vérification formelle des programmes généraux. La raison est qu'elle demande potentiellement un nombre infini de comparaisons des paires d'approximations syntaxiques de deux programmes. Pour cette raison, elle n'a pas été réalisée dans le système de transformation.

D'autres techniques doivent être donc utilisées pour la vérification formelle des programmes généraux, comme celle basée sur le théorème de point-fixe unique.

Considération de l'efficacité de la vérification formelle

Sans mentionner les programmes infinis, même pour les programmes finis, la vérification d'équivalence de deux programmes d'une certaine complexité par la transformation en leur forme normale (Voir figure 2.3), pourrait devenir difficile de manipuler. Ceci est du au fait que la taille de représentation textuelle de la forme normale d'un programme, est exponentielle du programme original. Dans beaucoup de cas, l'efficacité de la vérification d'équivalence de deux programmes peut être améliorée, en transformant leur représentation syntaxique en un niveau approprié qui permettrait la comparaison directe. Les exemples suivants illustrent cette technique.

On peut remarquer que tous les programmes qui contiennent des structures WHILE ne sont pas nécessairement des programmes infinis, dans le cas où le nombre d'itération de la structure de boucle est constant (par l'analyse statique).

Une méthode pratique et complète pour la manipulation des programmes généraux n'a pas été trouvée. Néanmoins certaines manipulations sont possibles et réalisées, basées sur :

(1) Les lois existantes ;

(2) Le théorème de point-fixe unique. Cette approche de la preuve peut être accompagnée de l'utilisation des lois existantes, avec d'autres techniques comme le déroulement du corps d'une boucle.

L'utilisation du *théorème du point-fixe* peut être illustrée comme l'exemple suivant. Un cas particulier pour prouver l'équivalence de deux processus, par exemple,

chan c: PAR (WHILE b1 P1, WHILE b2 P2) et
WHILE b Q

où P1, P2 et Q sont des programmes finis. Ils sont équivalents s'ils communiquent avec l'environnement au moins une fois par chaque certain nombre fixé d'itérations.

Un exemple

Les deux programmes P et Q suivants représentent les versions différentes d'un "tampon double infini", P est une version conventionnelle; Q est une version parallèle, formé de deux "tampon simple infini" connectés par un canal intermédiaire :

WHILE TRUE (VAR ch: SEQ (in ? ch, out ! ch)).

P = VAR ch1, ch2: SEQ (in?ch1,
WHILE TRUE (SEQ(PAR (in ? ch2, out ! ch1),
PAR (in ? ch1, out ! ch2))))

Q = CHAN c: PAR (WHILE TRUE (VAR ch: SEQ (in ? ch, c ! ch)),
WHILE TRUE (VAR ch: SEQ (c ? ch, out ! ch)))

Une procédure de preuve est de transformer le programme Q en P, en utilisant la loi <WHILE expansion> et des lois de renommage de variables.

On transforme Q comme suit:

Q = CHAN c: PAR (WHILE TRUE (VAR ch1: SEQ (in ? ch1, c ! ch1)),
WHILE TRUE (VAR ch2: SEQ (c ? ch2, out ! ch2)))
<VAR rename>

= VAR ch1, ch2: CHAN c: PAR (WHILE TRUE SEQ (in ? ch1, c ! ch1),
WHILE TRUE SEQ (c ? ch2, out ! ch2))
<VAR-WHILE sym>

= VAR ch1, ch2: CHAN c:
PAR (SEQ(SEQ (in ? ch1, c ! ch1), WHILE TRUE SEQ (in ? ch1, c ! ch1)),
SEQ(SEQ (c ? ch2, out ! ch2), WHILE TRUE SEQ (c ? ch2, out ! ch2)))
<WHILE expansion>

= VAR ch1, ch2: CHAN c:
SEQ(in ? ch1,
PAR (SEQ (c ! ch1, WHILE TRUE SEQ (in ? ch1, c ! ch1)),
SEQ(SEQ (c ? ch2, out ! ch2), WHILE TRUE SEQ (c ? ch2, out ! ch2)))
<PAR-SEQ com>

= VAR ch1, ch2: CHAN c:
SEQ(in ? ch1, ch2 := ch1,
PAR (WHILE TRUE SEQ (in ? ch1, c ! ch1)),
SEQ(out ! ch2, WHILE TRUE SEQ (c ? ch2, out ! ch2)))
<PAR-SEQ com>

= VAR ch1, ch2:
SEQ(in ? ch1,
CHAN c: PAR (WHILE TRUE SEQ (in ? ch1, c ! ch1),
SEQ(out ! ch1, WHILE TRUE SEQ (c ? ch2, out ! ch2)))
la règle 3.4 et <PAR-SEQ com>

En répétant les étapes ci-dessus, on a:

```
Q = VAR ch1, ch2:
      SEQ(in ? ch1, PAR( out ! ch1, SEQ( in ? ch2, out ! ch2)),
         CHAN c: PAR (WHILE TRUE SEQ ( in ? ch1, c ! ch1)),
                              WHILE TRUE SEQ (c ? ch2, out ! ch2)))
```

c'est à dire:

```
Q = VAR ch1, ch2:
      SEQ(in ? ch1, PAR( out ! ch1, SEQ( in ? ch2, out ! ch2)), Q)
```

Pour le processus P, on déroule une fois le corps P' de la boucle dans P, on a:

```
P'    = WHILE TRUE SEQ(PAR (in ? ch2, out ! ch1), PAR (in ? ch1, out ! ch2))
      = SEQ(PAR (in ? ch2, out ! ch1), PAR (in ? ch1, out ! ch2), P')
```

Les points fixes sont les solutions de Q et P', c'est-à-dire :

Q est une séquence infinie suivante, on dénote une telle séquence avec le symbole * :

```
Q = *SEQ(in ? ch1, PAR( out ! ch1, SEQ( in ? ch2, out ! ch2)))
  = SEQ(in ? ch1, *SEQ(PAR( out ! ch1, SEQ( in ? ch2, out ! ch2)), in ? ch1))
```

Pour P', on a d'une façon similaire :

```
P' = *SEQ(PAR (in ? ch2, out ! ch1), PAR (in ? ch1, out ! ch2))
```

On peut remarquer que les deux séquences infinies sont équivalentes, et

P = SEQ (in ? ch1, P'),

on a donc le résultat : P = Q.

2.3.3 Exemple d'application de la vérification formelle

L'importance de la vérification formelle du comportement global d'un système entier est du à sa compléxité et son caractère critique d'application. La transformation en vue de la vérification formelle joue un rôle essentiel et critique dans chaque étape de la conception d'un système totalement vérifié, dont le principe est peut être illustré dans la figure 2.4 [MS89] :

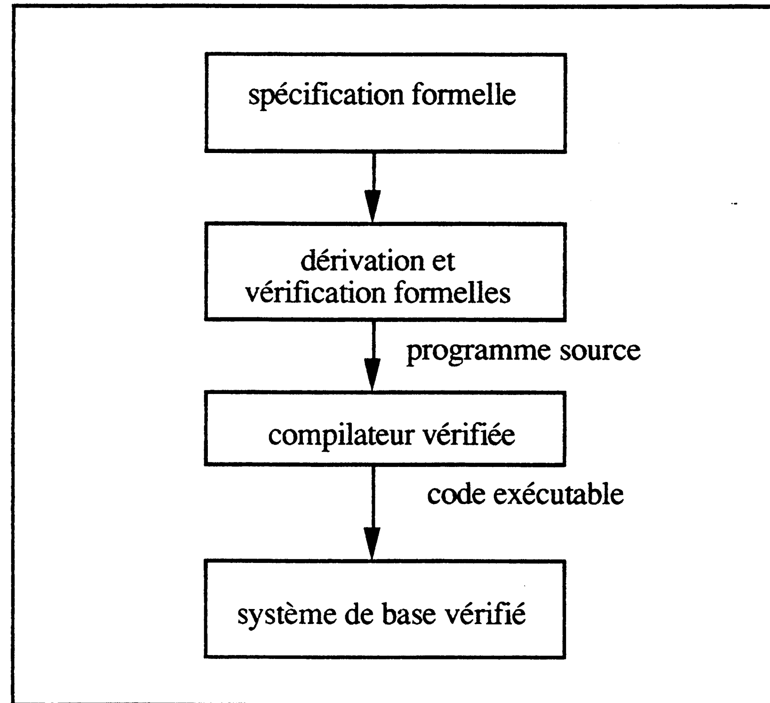

Figure 2.4 Un système totalement vérifié

Une technique de cette transformation est de réduire un programme à une syntaxe restreinte. Cette transformation s'applique surtout à la conception VLSI, où un programme est décomposé en modules VLSI efficaces et réalisables. Cela typiquement transforme un programme général en un sous-ensemble restreinte d'Occam avec les formes de communications qui sont connues d'avoir une réalisation efficace.

Nous donnons un exemple pour illustrer le principe et la stratégie de la démonstration d'équivalence de programmes Occam. Il s'agit de la vérification formelle de l'implantation correcte d'un composant VLSI [Inmos88].

Cette vérification est d'autant plus importante qu'il n'est pas pratique de valider par des tests exhaustifs due à la complexité de ces unités. L'utilisation des méthodes formelles réduit considérablement le temps de la conception, ainsi pour assurer l'implantation correcte.

Définition du problème

Il s'agit de la vérification de représentation correcte des valeurs dans un registre, dénoté comme Areg. La spécification du problème est la suivante : une opération est définie seulement quand le registre Areg contient une valeur valide en format <flottant>, et que le registre Error_Flag est mise à 1 seulement si Areg n'a pas une représentation correcte de format.

Vérification formelle et implantation correcte

Le processus de la conception est le suivant : premièrement on vérifie qu'une implantation en Occam de haut niveau est correcte par rapport à sa spécification formelle ; ensuite on transforme le programme Occam en une syntaxe restreinte (au niveau micro-codes) de bas niveau; en fin, le programme de bas niveau est traduit en micro-code, en vue d'une implantation VLSI.

La première et dernière étape étant en dehors de nos études [Inmos88], l'étape 2 est essentiellement une transformation du programme, dont le processus est illustré comme suit.

- Réalisation en Occam de haut niveau

```
P =

IF
   Areg.Exp < LargestINTExp
       SKIP
   (Areg.Sign = NEGATIVE) AND (Areg.Exp = LargestINTExp) AND
                                  (Areg.Frac = MSBit )
       SKIP
   TRUE
       SetError (ErrorFlag)
```

L'interprétation du programme est la suivante :

Le registre Areg a un format correct s'il a :

- un exposant inférieur à LargestINTExp;

- ou un signe négatif, un exposant égal à LargestINTExp et une fraction avec seulement le bit le plus significatif égal à 1.

Sinon il n'a pas un format correct, et le registre ErrorFlag est mis à 1.

- Réalisation en Occam de bas niveau

Le programme ci-dessus est transformé vers une représentation syntaxique plus restreinte : seulement des processus et des variables définis dans les micro-codes peuvent être utilisés :

```
Q =
SEQ
   AregSignNEGATIVE := (Areg.Sign = NEGATIVE)
   ExpZbus := (Areg.Exp - LargestINTExp)
   ExpZbusNeg := ExpZbus < 0
   IF
      AregSignNEGATIVE
         IF
            ExpZbus
               SKIP
            (Areg.Exp = LargestINTExp) AND (Areg.Frac = MSBit)
               SKIP
            NOT ((Areg.Exp = LargestINTExp) AND (Areg.Frac = MSBit))
               SetError(ErrorFlag)
      NOT AregSignNEGATIVE
         IF
            ExpZbusNeg
               SKIP
            NOT ExpZbusNeg
               SetError (ErrorFlag)
```

Le processus de transformation est effectué comme suit : la condition booléenne sur <Areg.Sign> est déplacée au début du programme. Il est facile de démontrer que ces deux programmes sont équivalents en transformant le programme Q en P :

Par les lois <SEQ assign> et <assign combine>, nous pouvons substituer les variables AregSignNEGATIVE, ExpZbusNeg par leur valeur respective dans le processus IF, ensuite on élimme ces variables locales (lois <var elim>, <assign-SKIP>), nous avons :

```
Q =
SEQ
   SKIP
   IF
      Areg.Sign = NEGATIVE
         IF
            (Areg.Exp - LargestINTExp) < 0
               SKIP
            (Areg.Exp = LargestINTExp) AND (Areg.Frac = MSBit)
               SKIP
            NOT ((Areg.Exp = LargestINTExp) AND (Areg.Frac = MSBit))
               SetError(ErrorFlag)
      NOT (Areg.Sign = NEGATIVE)
         IF
            (Areg.Exp - LargestINTExp) < 0
               SKIP
            NOT ((Areg.Exp - LargestINTExp) < 0 )
               SetError (ErrorFlag)
```

Nous supprimons la construction SEQ(<SEQ-SKIP>) et l'emboîtement des constructions IF(<IF assoc>), notons que :

(Areg.Exp - LargestINTExp) < 0 ¤ Areg.Exp < LargestINTExp
et Areg.Exp = LargestINTExp ¤ NOT (Areg.Exp < LargestINTExp)

on transforme Q comme suit : <IF-^ distrib2>

```
Q =
IF
   Areg.Sign = NEGATIVE
      IF
         Areg.Exp < LargestINTExp
            SKIP
         (Areg.Exp = LargestINTExp) AND (Areg.Frac = MSBit)
            SKIP
         NOT ((Areg.Exp = LargestINTExp) AND (Areg.Frac = MSBit))
            SetError(ErrorFlag)
   NOT (Areg.Sign = NEGATIVE)
      IF
         Areg.Exp < LargestINTExp
            SKIP
         Areg.Exp = LargestINTExp
            SetError (ErrorFlag)

=

IF
   (Areg.Sign = NEGATIVE) AND ((Areg.Exp < LargestINTExp) OR
              (Areg.Exp = LargestINTExp) AND (Areg.Frac = MSBit))
      SKIP
   (Areg.Sign = NEGATIVE) AND NOT ((Areg.Exp = LargestINTExp) AND
                          (Areg.Frac = MSBit))
      SetError(ErrorFlag)
   NOT (Areg.Sign = NEGATIVE) AND (Areg.Exp < LargestINTExp)
      SKIP
   NOT (Areg.Sign = NEGATIVE) AND (Areg.Exp = LargestINTExp)
      SetError (ErrorFlag)
```

Pour le programme P, nous le transformons comme suit, en appliquant les lois <IF-v distrib> et <IF priori>, nous avons :

```
P =
IF
   (Areg.Exp < LargestINTExp) OR
          ((Areg.Sign = NEGATIVE) AND (Areg.Exp = LargestINTExp) AND
                             (Areg.Frac = MSBit ))
      SKIP
   TRUE AND NOT
        ((Areg.Exp < LargestINTExp) OR
          ((Areg.Sign = NEGATIVE) AND (Areg.Exp = LargestINTExp) AND
                             (Areg.Frac = MSBit )))
      SetError (ErrorFlag)
```

Nous introduisons les abréviations suivantes pour simplifier la représentation textuelle de P et Q :

a = (Areg.Sign = NEGATIVE)
b = (Areg.Exp < LargestINTExp) = NOT (Areg.Exp = LargestINTExp)
c = (Areg.Frac = MSBit)

Alors P et Q peuvent être réécrits comme suit, en simplifiant les expressions logiques: où $\underline{b}$ = NOT b

P =
IF
  b OR $a\underline{b}c$
    SKIP
  $\underline{a}\underline{b}$ OR $\underline{b}\underline{c}$
    SetError (ErrorFlag)

Q =
IF
  $a(b$ OR $\underline{b}c)$
    SKIP
  ab OR $a\underline{c}$
    SetError(ErrorFlag)
  $\underline{a}b$
    SKIP
  $\underline{a}\underline{b}$
    SetError (ErrorFlag)

Par les lois <IF-priori> et <IF-v distrib>, nous pouvons transformer Q comme suit :

Q =
IF
  b OR $a\underline{b}c$
    SKIP
  $\underline{a}\underline{b}$ OR $\underline{b}\underline{c}$
    SetError (ErrorFlag)

A la fin, en comparant les expressions logiques qui gardent les deux processus SKIP et SetError(ErrorFlag) respectivement, nous avons le résultat évident :

P = Q.

2.4 Conclusions

Grâce à la rigueur de la sémantique d'Occam, des règles de transformation et des formes normales des programmes ont pu être développées. Sur cette base, une stratégie pour la preuve formelle d'équivalence des programmes parallèles a été réalisée. Ceci a une application significative dans la vérification formelle des systèmes logiciels.

Nous remarquons qu'en pratique, nous pouvons dans certain cas éviter la transformation complète d'un programme en sa forme normale, qui demande potentiellement un grand nombre d'étapes de transformation, comme illustré dans les exemples vus.

A ce jour, notre système ne s'applique pas aux programmes généraux, c'est-à-dire programmes infinis, sur les aspects suivants : les boucles WHILE, les expressions algorithmiques et logiques, les tableaux. Ils doivent être manipulés intuitivement avec des lois de substitution d'expressions, de renommage de variables etc.

Chapitre 3

Optimisation des programmes parallèles

A part des applications dans la vérification formelle des programmes parallèles (Cf. le chapitre 2), une autre application, la plus attractive, de l'approche transformationnelle, est l'optimisation en vue de leur implantation efficace sur une architecture parallèle. C'est un problème très complexe, et une grande quantité d'informations externes d'un programme devrait être indispensables pour son optimisation : statistiques d'exécution d'instructions, l'analyse statique des données d'entrée, etc [ASU86]. Néanmoins un certain degré de manipulation des programmes, basé sur les structures du programme et des heuristiques, est réalisable.

Pour améliorer la performance d'exécution d'un programme sur une architecture cible, une bonne stratégie de placement joue un rôle important pour minimiser son temps d'exécution [KM89, Gaud88, TM90]. Dans le placement, on suppose souvent que le parallélisme du programme est supérieur au nombre de processeurs disponibles. Donc pour une bonne performance d'un programme, un haut degré de parallélisme est crucial pour son adaptation à une architecture, sans connaître à priori, sa configuration.

Nous identifions deux niveaux d'optimisation d'un programme : machine-indépendant et machine-dépendant. Dans ce chapitre, nous appliquons le système de transformation en vue de l'optimisation de programmes, sur des aspects machine-indépendants, notamment la détection et l'extraction du parallélisme.

3.1 Analyse statique de programmes

3.1.1 L'espace d'état d'un programme et son calcul

Avant toute manipulation d'un programme, la connaissance de son espace d'état, c'est-à-dire, l'ensemble des variables et des canaux qu'il utilise, doit être acquit.

Ayant déterminé son espace d'état, il est trivial de détecter statiquement certaines propriétés d'un programme. Par exemple, la vérification de la construction correcte du programme :

- La validité (séparation de variables et de canaux) des constructions parallèles (Cf. §2.1.1).

- Appel d'une procédure ou d'une fonction tel que les paramètres actuels soient conformes aux paramètres formels en nature et nombre.

<u>**Espace d'états**</u>

Définition 4.1 Un identificateur x de variable ou de canal, apparaissant dans un programme P, est libre, si x n'apparaît pas dans la portée d'une déclaration. Sinon x est lié dans P.

Nous avons des notations suivantes pour l'espace d'état d'un programme P :

libre(P) = l'ensemble des variables et des canaux libres de P;
liés(P) = l'ensemble des variables et des canaux liés de P.

En fait, ces deux ensembles sont composés de deux parties respectivement, les variables et les canaux libres et liés. Nous les dénotons respectivement : librevar(P), librechan(P), liésvar(P) et liéschan(P).

Nous pouvons identifier trois ensembles pour un processus P :

librevar(P) l'ensemble des variables libres de P.

init(P) l'ensemble des variables d'initialisation dont les valeurs initiales déterminent le comportement de P.

final(P) l'ensemble des variables dont les valeurs finales sont dus aux actions de P. Par exemple, l'occurrence du but d'une affectation ou d'une entrée, ou un résultat d'un paramètre <VAR> dans un appel d'une procédure.

Nous pouvons aussi effectuer une répartition des canaux libres de P en trois ensembles :

input(P) l'ensemble des canaux d'entrée dans P.
output(P) l'ensemble des canaux de sortie dans P.
interne(P) l'ensemble des canaux internes de P.

L'intérêt de connaître l'espace d'état d'un programme peut être illustré par l'exemple suivant, considérons le programme :

SEQ (Q_1, P, Q_2)

où le processus P n'a pas de canaux externes en commun avec Q1 et Q2. Si nous dénotons <c ? x> et <c ! x> pour l'entrée et la sortie d'un ensemble de variables x à travers d'un ensemble de canaux c, et nous choisissons c1 et c2 d'une façon qu'il n'y ait pas de conflit avec les noms existants, alors le programme ci-dessus est équivalent à :

```
CHAN c1, c2 :
PAR
  SEQ (Q1, c1 ! init(P), c2 ? final(P), Q2)
  VAR libre(P) : SEQ (c1 ? init(P), P, c2 ! final(P) )
```

Cela peut être démontré par leur forme normale, mais informellement parce que les communications sur c1 et c2 effectivement synchronisent la construction parallèle. Cette transformation permet une mise en oeuvre d'une implantation distribuée d'une construction séquentielle.

<u>Calcul d'espace d'états d'un programme</u>

La procédure du calcul consiste à effectuer les étapes suivantes [ASU86, Gold88] :

(1) calcul de libre(P), final(P)

- Le calcul de librevar(P) et librechan(P) est facile. Il suffit de parcourir le programme en notant les variables et les canaux qui ne sont pas dans la portée d'une déclaration, en tenant compte des déclarations locales ;

- Les variables de valeur finale final(P) sont dans l'ensemble de variables libres et apparaissent comme des identificateurs de gauche dans les affectations.

Les canaux d'entrée et de sortie input(P) et output(P) peuvent être identifiés par leur apparence de type c?x ou c!e respectivement ; l'ensemble des canaux internes est vide après la normalisation d'utilisation des canaux.

Dans le système de transformation, ce calcul est élémentaire pour des stratégies de transformation, et réalisé comme une fonction :

process -> (liste de variables)

(2) calcul de init(P)

Le calcul exact de init(P) est impossible. Cela est du au flot de contrôle. Prenons un exemple :

```
P = SEQ
      IF b
            x:= 1
      NOT b
            SKIP
      Q
```

où $x \in$ init(Q).

Si la condition <b> est toujours vraie, alors $x \in$ init(P) ; et si le processus <SKIP> est remplacé par un processus <STOP> ou <ZERO>, alors P est équivalent à ZERO et $x \notin$ init(P).

Donc nous calculons plutôt un sur-ensemble de init(P), bien que cela perde l'efficacité dans le programme résultat de transformation en augmentant la volume de messages échangés, dans une transformation typique de programme :

```
          SEQ (P, Q)
---------------------------------------------
CHAN C:
PAR (SEQ (P, C ! init(P))
     VAR init(P): SEQ (P ? init(P), Q)
```

Une approximation de init(P) peut être déterminée en incluant simplement toutes les variables possibles dans librevar(P) $\cup$ final(P).

Une autre solution plus précise peut être calculée par induction structurale sur la syntaxe de P :

(1) Calculer libre(P) et final(P) ; toutes les variables dans libre(P) - final(P) appartiennent à init(P).

(2) Initialiser deux listes : *connu* = librevar(P) - final(P) et *possible* = final(P), nous continuons le calcul par induction structurale comme suit:

SKIP : pas de changement ;
STOP, ZERO : *connu, possible* = Ø ;

tous les autres processus atomiques : ajouter les variables à droite d'une affectation dans *connu;* enlever dans *possible* les variables apparaissant à gauche d'une affectation.

IF, ALT : traiter les gardes et les conditions booléennes comme processus primitifs, et appliquer la procédure récursivement aux processus composants. Nous avons alors :

connu = l'union de *connu* résultat, et
possible = l'union de *possible* résultat - *connu* nouveau

PAR(P, Q) : appliquer la procédure aux processus composants P et Q, nous avons :

connu = *connu*(P) $\cup$ *connu*(Q) et
possible = *possible* (P) $\cup$ *possible*(Q) - *connu* nouveau

WHILE (b P) : dérouler le corps de la boucle au moins une fois pour révéler des variables initiales potentielles, et traiter toute autre occurrence comme SKIP. C'est-à-dire, nous traitons la construction <WHILE b P> comme <IF (b P, ¬b SKIP)>.

Déclarations : appliquer l'analyse récursivement sur les processus dans leur portée et restaurer l'état des variables où la portée a été enlevée par des déclarations locales.

SEQ (P, Q) : répéter la procédure à chaque composant processus, utiliser le résultat d'une étape P (*connu* (P) et *possible* (P)), comme les valeurs initiales pour la suivante, Q, ce qui nous donne les résultats *connu_local* et *possible_local* :

$$liste_temp = possible_local \cap connu$$

Déplacer les variables dans *liste_temp* de la liste *possible* à la liste *connu* :

$$possible = possible - liste_temp$$
$$connu = connu \cup liste_temp$$

(3) A la fin, nous avons la relation suivante [Gold88] :

$$connu \subseteq init(P) \subseteq connu \cup possible$$

Donc nous pouvons prendre $connu \cup possible$ comme notre appriximation de *init* .

Nous pouvons raffiner le calcul de l'ensemble *connu* en considérant les branches conditionnelles d'une construction IF, en accumulant les conditions booléennes dans lesquelles des variables sont utilisées, et en jugeant si elles sont dans init(P) ou pas. Ceci exclut les variables dont les valeurs sont invariantes.

Par exemple, dans le programme :

```
P = SEQ (IF (y = 0 x := 0, y <> 0 SKIP),
         IF (y <> 0 x := 1, y = 0 SKIP),
         y := x)
```

La variable x ne sera pas inclue dans init(P), parce que dans l'affectation <y := x>, la valeur de y ne dépend pas en celle de x, tandis que l'algorithme ci-dessus l'includrait dans init(P).

3.1.2 Normalisation des variables et des canaux

Le but de la normalisation des variables et des canaux d'un programme est de localiser l'utilisation de ses canaux et de ses variables, en déplaçant leur déclaration immédiatement avant des constructions où ils sont utilisés. Cela est exigé dans notre représentation d'une construction parallèle dans le système de transformation, où les processus composants parallèles indiquent explicitement leur utilisation de variables et de canaux, sous la forme suivante (Cf. §2.2.3) :

PAR (U_i P_i), i = 1, ..., n

où l'annotation U_i dénote l'espace d'états du processus composant Pi.

Cela facilite aussi l'analyse statique et l'extraction des informations des processus pour les applications dans les chapitres qui suivent. Ces informations sont utiles et indispensables pour estimer les aspects quantitatifs du calcul ou des communications d'un programme, donc pour son placement en vue d'une implantation distribuée.

Définition 3.2 : *La normalisation des variables et des canaux* d'un programme est la localisation des déclarations des variables et des canaux. C'est-à-dire, les déclarations de canaux sont juste avant les constructions parallèles où ces canaux sont utilisés, et les affectations successives sont réduites à l'affectation multiple.

Un programme P peut toujours être transformé en cette forme dans le système de transformation Occam. La procédure consiste à appliquer les étapes suivantes par induction structurale sur P :

Pour des déclarations et des affectations, nous appliquons les lois <VAR assoc>, <VAR combin>, <CHAN assoc>, <CHAN combin> et <CHAN-VAR sym>, pour réduire des déclarations simples de variables et de canaux en déclarations multiples ; utiliser les lois <assign multiple>, <assign sym>, <assign combin> pour réduire des affectations simples en affectations multiples ;

S'il est possible, pour toutes des constructions de P, appliquer les lois <VAR elim> et <CHAN elim> pour supprimer les canaux et variables non utilisées ;

A chaque construction, nous appliquons récursivement des lois de distribution de variables ou de canaux <VAR-* distrib *> et <CHAN-* distrib *>. Nous appliquons les lois <CHAN rename> et <VAR rename> pour résoudre le conflit avec les variables ou des canaux locaux des processus composants, jusqu'à des déclarations sont placées juste avant les constructions qui les utilisent. Par exemple, les variables d'une construction de PAR sont repartitiées et placées dans les branches composantes qui les utilisent ;

A la fin, nous appliquons l'algorithme du calcul d'espace d'états au programme résultat, et ajoutons l'annotation d'espace d'états pour chaque processus composant parallèle.

Réalisation dans le système de transformation

Plusieurs commandes dans le système de transformation réalisent cette stratégie de la régularisation de l'utilisation de variables et de canaux. <pushCHANall>, <pushCHAN> et <pushCHANtoPAR> permettent la normalisation d'utilisation de canaux :

<pushCHANtoPAR> prend un contexte avec une déclaration de <CHAN>, déplace cette déclaration et toutes les intermédiaires, vers l'interne jusqu'à une construction PAR est rencontrée, sinon éliminées;

<pushCHAN> continue le processus jusqu'à touts les canaux rencontrés sont traités;

<pushCHANall> permet la localisation complète des déclarations de canaux.

Par exemple, la commande <pushCHANall> est réalisée comm suit :

si on combine des déclarations consécutives de canaux en une seule déclaration par la loi <CHAN assoc>. Après quoi répéter le processus récursivement ;
sinon si on a une déclaration <CHAN> immédiatement avant une déclaration <VAR>, ou une construction <SEQ>, <PAR>, <ALT>, <IF>, <WHILE> ;
alors distribuer la déclaration <CHAN> dans la construction interne ;
sinon éliminer la déclaration. Après quoi appliquer la procédure récursivement à tous les processus composants.

Des commandes similaires pour des déclarations de variables et leur normalisation sont fournies aussi dans le système de transformation : <pushVAR> et <pushVARall>.

3.2 L'optimisation des programmes parallèles

Une règle de transformation peut être appliquée dans différentes directions, qui se révèlent contradictoires, dépendant largement de l'application et des formes de programmes que nous souhaitons obtenir. Donc des stratégies de transformation sont nécessaires pour l'orientation de la manipulation de programmes dans le système de transformation.

Pour l'optimisation d'un programme parallèle, un aspect essentiel est la détection et l'extraction du parallélisme. Il devrait être nécessaire d'utiliser les informations d'un programme, c'est-à-dire son espace d'états, pour développer des stratégies qui combinent de petits segments de programmes séquentiels en structures parallèles. Et ensuite, le parallélisme sera déplacé vers la construction externe, séquentielle. Ces manipulations combinées avec une répartition appropriée de l'espace d'états du programme, nous fournissent une stratégie générale de transformer un programme Occam en un équivalent, avec le nombre augmenté des processus composants en parallèle.

Dans le cas général, les programmes obtenus ainsi ne sont pas forcément élégants, clairs ou efficaces. Les causes peuvent être : les communications entre les processus parallèles (telles que la pluspart de processus doivent interrompre leur calcul et attendre les autres pour communiquer). Donc certaines considérations particulières doivent être données pour raffiner la méthode générale pour obtenir un résultat plus satisfaisant. La mesure et la comparaison des temps d'exécution de programmes, ainsi des heuristiques doivent être utilisées pour déterminer sur quelles structures d'un programme il vaut la peine d'appliquer les transformations.

Pour évaluer l'exécution machine-indépendante d'un programme, nous avons besoin d'une hypothèse sur la machine sur laquelle le programme s'exécute.

Hypothèse de la Machine virtuelle

C'est une machine comme les machines massivement parallèles d'architecture MIMD sans mémoire commune. Nous supposons que la machine est capable de supporter n'importe quel programme parallèle et a une connectivité complète des processeurs. Nous supposons aussi que le coût de communication (par unité de message échangé) entre deux processeurs soit identique entre toute paire de processeurs.

Avec cette hypothèse de la machine virtuelle, le temps d'exécution d'un programme parallèle est déterminé uniquement par le poids (de calcul) des processus et par la quantité de messages échangés entre les processus [GL89, Hey89]. Ainsi un programme a une meilleur performance s'il a un haut degré de parallélisme et un équilibre de charge des processus en parallèle, dont l'idée sera développée dans le chapitre suivant.

3.2.1 Techniques de transformation dans la compilation optimisée

Des compilateurs accomplissent des transformations en vue d'améliorer la qualité de codes générés (intermédiaires ou machine) [ASU86]. Des transformations d'amélioration de codes doivent satisfaire, évidemment, (1) le critère de conserver la sémantique, et le plus important, (2) le critère d'amélioration : accélerer l'exécution d'un programme par une facteur mesurable en moyen.

Un plus grand gain peut être obtenu si nous arrivons à identifier les parties du programme fréquemment exécutées et rendre ces parties le plus efficace possible. Des statistiques de l'exécution d'un programme sur des données représentatives identifient avec précision les régions du programme, qui consomment le plus de temps. Puisque nous n'avons pas cette information, nous sommes amenés à l'approche de l'analyse statique et des heuristiques. Par exemple, des heuristiques et l'intuition nous indiquent que des boucles sont presque toujours des candidates d'optimisation, puisque les instructions dans une boucle sont exécutées potentiellemnt le plus grand nombre de fois.

Des méthodes d'analyse des programmes sont basées sur l'analyse de flot de contrôle et de flot de données [ASU86]. La première consiste à analyser des structures de contrôle telles que les conditionnelles, boucles etc. La deuxième consiste à collecter des informations, dont les variables sont utilisées dans un programme. Le processus de calcul de l'espace d'un programme expliqué dans la section §3.1 est une analyse de flot de données globale (distribuer des informations des variables à chaque structure de programme).

Après cette analyse, nous identifions les sources principales et les transformations nécessaires suivantes :

- Sous-expression commune : son extraction peut éviter le re-calcul de la même expression arithmétique.

- Propagation de copie de valeurs : l'évaluation d'une expression constante pendant la compilation et le remplacement d'occurrence de cette expression par sa valeur calculée.

- Elimination de codes morts : une variable est vivante si sa valeur est susceptible d'être utilisée après, sinon elle est morte. Pour un segment de programme la définition est identique. Cette transformation évite le calcul inutile, souvent apparu comme un résultat d'autres transformations.

- Optimisation des boucles : la transformation consiste à réduire le plus possible le nombre d'instructions dans une boucle. Pour ceci, les techniques suivantes sont utilisées : le déplacement des instructions constantes devant la boucle ; élimination de variables d'induction ; réduction de force d'opération arithmétique par transformation algébrique. Par exemple :

```
x ** 2  => x * x
2.0 * x => x + x
...
```

Relation avec des lois de transformation

Les lois de transformation expriment certaines techniques d'optimisation utilisées dans la compilation. Par exemple, la stratégie de transformation d'un programme en sa forme canonique, qui normalise l'utilisation de variables et de canaux, et combine des affectations en l'affectation multiple, accomplit la phase d'optimisation dans un bloc élémentaire du programme.

En effet, en intégrant les techniques de la compilation d'optimisation dans le système de transformation, nous pouvons envisager appliquer les techniques de transformation sur les programmes sources dans une phase d'un compilateur.

3.2.2 Détection et extraction du parallélisme

Notre objectif est de transformer des constructions séquentielles en constructions parallèles en vue de leur implantation distribuée. Cela consiste à pousser les constructions parallèles vers leur contextes externes, essentiellement des constructions séquentielles. Cette approche combinée avec l'analyse de l'espace d'état de processus (Cf. §3.1), nous donnent une stratégie pour transformer un programme en un autre équivalent de haut degré de parallélisme.

L'idée principale de transformations est la réalisation de la relation de dépendance de données (exprimée par des valeurs initiales init(P) et des valeurs résultats final(P)) par des communications explicites. Cela permet une réalisation distribuée des calculs séquentiels. L'ensemble des lois nécessaires pour ces transformations est dans l'annexe A2, il s'agit du changement de la forme des communications inter-processus et de l'extraction du parallélisme.

3.2.2.1 Parallélisation des programmes séquentiels

Etant donné un programme général, la procédure consiste à effectuer les trois étapes suivantes :

- Premièrement, nous transformons le programme en sa forme canonique. Cela a pour but de localiser ses variables et ses canaux, et l'affectation multiple résultat représente l'extraction du parallélisme dans une dimension locale (bloc élémentaire) du programme (les affectations composantes peuvent être exécutées en parallèle).

- Ensuite, nous transformons des constructions séquentielles (pour des processus de SEQ, IF, ALT, WHILE) en constructions parallèles. Pendant cette étape, nous obtenons des programmes distribués dont le calcul est séquentialisé par des communications entre eux.

Ces transformations peuvent être réalisées par application des lois <PAR-Op*> pour pousser le PAR vers l'extérieur de son contexte. Op est un des opérateurs SEQ, IF, ALT, WHILE, PAR.

Beaucoup de cas en ce qui concernent les constructions séquentielles similaires peuvent être trouvés pour ce type de transformations. La stratégie et des lois nécessaires sont : le calcul des variables init(), final() de chaque processus composant ; l'introduction des communications entre des processus, qui élimine la dépendance de données ; mettre des processus composant en parallèle. Des lois utiles pour ces transformations sont <PAR-Op distrib>, <VAR rename>, <CHAN rename>, <PAR-Op com*> etc.

- A la fin, sur la base des résultats précédents, nous utilisons des critères et les stratégies nécessaires (dans la section suivante) pour la détection et l'extraction du parallélisme réel.

Cette étape est la plus difficile et délicate. Il n'existe pas encore une stratégie générale et des cas spéciaux sont considérés. Par exemple, dans le cas où des processus composants sont des constructions parallèles, il est possible de pousser le PAR vers le contexte externe, par les lois <PAR-Op distrib>, et nous donnent un parallélisme réel.

Aspects techniques

Constructions parallèles à l'intérieur d'une construction séquentielle

Une fois le calcul d'espace d'état, ainsi que la normalisation d'un programme est effectué, il reste à découvrir des moyens de pousser l'opérateur <PAR>, probablement accompagné par ses déclarations de canaux, en dehors de la constructeur externe (l'étape 2 de la stratégie).

Si la constructeur externe est une autre <PAR>, des lois sur l'associativité de <PAR> peuvent être utilisées pour éliminer l'emboîtement de ces constructions parallèles, ayant d'abords déplacé ses déclarations de canaux vers l'externe. Cela pourrait exiger le renommage des noms de canaux pour éviter le conflit possible. Cette transformation peut être illustrée par la figure 3.1 :

où une ligne fleche représente la relation (ordre) d'exécution, une ligne pointillée représente la communication,

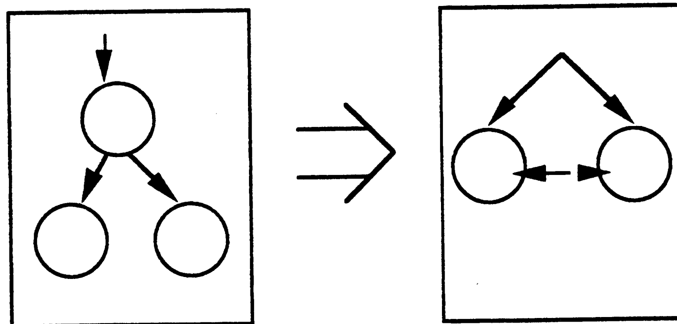

Figure 3.1 L'extraction du parallélisme à l'intérieur d'une construction séquentielle

Ici, nous donnons quelques lois d'exemple importantes pour cette manipulation, elles exigent une analyse de la dépendance de données et l'introduction de communicaions. Le reste des lois sera présenté dans la section suivante et dans l'annexe A2.

Les constructions IF et ALT

IF (b_0 P_0, ..., b_i (PAR (P, Q)), ...)

Nous calculons d'abords init(P) et final(P), et introduisons les canaux *c* et *d* sur lesquels se passent des communications de ces valeurs. Alors nous pouvons transformer ce programme en un autre équivalent :

```
CHAN c, d:
PAR
   VAR init(P): SEQ (c ? init(P), P, d ! final(P))
   IF (  b0 P0, ...,
         bi (SEQ (c ! init(P), Q, d ? final(P))
         ...)
```

Remarquons que lors de communication des valeurs init(P), P et Q sont en parallélisme réel.

Pour une construction ALT, nous avons d'une façon similaire :

```
      ALT (g0 G0, ..., gi (PAR (P, Q)), ...)
---------------------------------------------------
CHAN c, d:
PAR
   VAR init(P) : SEQ (c ? init(P), P, d ! final(P))
   ALT (g0 P0, ...,
        gi (SEQ (c ! init(P), Q, d ? final(P))
        ...)
```

où les canaux c et d sont choisi d'une façon qu'il n'ait pas de conflit avec les noms existants de canaux.

La construction SEQ

Dans la transformation d'une construction séquentielle en une construction parallèle, il faut éviter le blocage possible due aux communications avec l'environnement. On doit garantir aussi la syntaxe correcte du nouveau programme. Par exemple, quand nous transformons le programme comme suit :

(1) SEQ (P , PAR (Q, R))

```
          SEQ (P , PAR (Q, R) )
------------------------------------------------------------------
CHAN c, d:
PAR
    VAR init(R) :  SEQ (P, c ? init(R), R, d ! final(R))
    VAR final(R) : SEQ (c ! init(R), Q, d ? final(R))
```

- Si P et Q communiquent avec le reste du programme sur un même canal alors le programme obtenu est incorrect, parce que les processus composants parallèles effecturont des E/S sur le même canal avec l'environnement.

- Si le processus P n'a pas de de communication avec le reste du programme, alors nous pouvons transformer de la façon suivante, au prix de répéter l'exécution du processus P:

```
  SEQ (P, PAR (Q, R))
--------------------------------------
CHAN c:
PAR
  VAR librevar(P) ∪ librevar(Q):
  SEQ (P, Q)
  SEQ (P, R)
```

- Par l'associativité de PAR, nous avons d'une façon symmétrique :

```
          SEQ (P, PAR (Q, R))
------------------------------------------------------------
CHAN c, d:
PAR
  VAR init(R) : SEQ (P, c ? init(R), R, d ! final(R))
  VAR final(R) : SEQ (c ! init(R), Q, d ? final(R))
```

(2) SEQ (P, Q)

Dans ce cas général de la composition séquentielle, il existe des cas spéciaux où la structure peut être transformée en parallèle. Les loi suivantes doivent être utilisées à ce que P et Q communiquent avec l'environnement, l'ordre de communications doit être tel qu'un blocage ne se produise pas.

- Dans le cas où libre(P) $\cap$ libre(Q) = $\emptyset$, c'est-à-dire, il n'y a pas de dépendance de données et des canaux en commun entre P et Q, nous avons alors :

SEQ (P, Q)

PAR (P, Q)

- Dans le cas où final(P) $\cap$ init(Q) $\neq \emptyset$, c'est-à-dire, Q utilise des valeurs calculées par P, alors nous avons le résultat : où le canal C est choisi tel que C $\notin$ libre(P) $\cup$ libre(Q),

```
                SEQ (P, Q)
---------------------------------------------------------
CHAN C :
PAR
    VAR final(P) : SEQ (P, C ! final(P)),
    VAR final(P) :  SEQ (C ? final(P), Q(final(P) / init(Q))
```

L'exemple suivant illustre la technique de transformation pour faire apparaître le plus tôt possible les communications, tel que le calcul ne soit pas retardé. Considérons le processus suivant :

```
P =
PAR
  VAR x :
  SEQ
    C1 ? x
    PROC1 ( x )
  VAR y:
  SEQ
    C2 ? y
    PROC2 ( y )
```

Ce processus satisfait la condition de la loi <SEQ-PAR2> :

PAR (SEQ (P, Q), SEQ (P', Q'))

SEQ (PAR (P, P'), PAR (Q, Q'))

Donc nous pouvons l'appliquer et obtenons le nouveau processus suivant :

```
P =
SEQ
  PAR
    C1 ? x
    C2 ? y
  PAR
    PROC1 ( x )
    PROC2 ( y )
```

Le résultat signifie que si ce processus P est en parallèle avec un autre, disont Q, alors les E/S sur les canaux C1 et C2 de Q ne seront pas retardées par les calculs de P : PROC1 et PROC2.

La construction WHILE

Une boucle statique (dont le nombre d'itérations peut être déterminé par l'analyse statique) peut être transformée en un programme fini, en déroulant le corps de la boucle. Ensuite nous appliquons la procédure entière sur le programme obtenu.

Pour une boucle infinie, à part des techniques d'optimisation locales utilisées dans la compilation (§3.2.1), nous pouvons appliquer d'autres techniques comme la composition et la décomposition des boucles. Par exemple, nous pouvons dérouler au moins une fois le corps de la structure de boucle, afin de révéler ou de mettre en explicite les communications avec d'autres processus en parallèle, ce qui permet une manipulation approfondie.

Décomposition d'une boucle WHILE

Il existe un cas intéressant où un programme est sous forme :

WHILE e (PAR (P, Q))

Il est possible d'arranger une communication pour le changement des valeurs dans la condition booléenne de terminaison <e>, et de pousser le <PAR> vers l'externe. L'interprétation de cette transformation est qu'une série de compositions parallèles de P et de Q, est la même que la composition parallèle d'une série de P et d'une série de Q, avec la synchronisation nécessaire.

Pour transformer le programme, nous avons besoin de connaître librevar(P), init(P), final(P) et aussi l'ensemble de variables référencées A dans la condition e . Parce que le programme original (supposons qu'il soit en forme canonique) partitionne des variables entre P et Q, nous savons que seulement les variables dans A peuvent causer des problèmes.

Après la transformation nous obtenons le programme résultat suivant :

```
CHAN init, data, bool:
PAR
  VAR libre(P) ∩ A, loop:
  SEQ
    init ? init(P) ∩ A
    bool ? loop
    WHILE loop SEQ (P, data ! final(P) ∩ A, bool ? loop)
  SEQ
    init ! init(P) ∩ A
    bool ! e
    WHILE e SEQ (Q, data ? final(P) ∩ A, bool ! e )
```

où les canaux *init, data, bool* sont choisis de façon à ce qu'il y ait pas de conflit avec les noms de canaux existants.

Nous pouvons remarquer que :

- dans le cas où final(P) $\cap$ A = $\emptyset$, c'est-à-dire, le processus P ne change pas l'expression *e*, alors les communications sur le canal *data* peuvent être supprimées;

- si la condition e = TRUE, alors c'est une boucle infinie. Nous pouvons remplacer le message sur le canal *bool* par ANY, mettre la condition de terminaison de P à TRUE, et supprimer la communication initiale sur le canal *bool*.

Avec ces deux cas, nous obtenons le programme équivalent :

```
CHAN c, init, bool:
PAR
  SEQ
    init ? init(P) ∩ A
    WHILE TRUE SEQ (P, bool ? ANY)
  SEQ
    init ! init(P) ∩ A
    WHILE e SEQ (Q, bool ! ANY)
```

Un exemple pratique de la décomposition d'une boucle est l'a réalisation différente d'un tampon double infini :

```
VAR ch1, ch2:
SEQ
    .in?ch1,
    WHILE TRUE
        SEQ
            PAR (in ? ch2, out ! ch1)
            PAR (in ? ch1, out ! ch2)
---------------------------------------------          <infinite buffer>
CHAN c:
PAR
    WHILE TRUE
        VAR ch: SEQ ( in ? ch, c ! ch)
    WHILE TRUE
        VAR ch: SEQ (c ? ch, out ! ch)
```

Un exemple du réarrangement d'une boucle

Pour un programme composé de processus sous forme de "pipeline", chaque processus composant est lié à ses voisins par un seul canal, et répète indéfiniment une phase de traitement. Un exemple est le programme d'un compilateur [Capon89], qui est organisé comme un pipeline de plusieurs étapes de compilation, dont un segment typique d'une étape est le suivant :

```
WHILE TRUE
  SEQ
    in ? donnes
    ... consommer 'donnes' et produire 'resultats'
    out ! resultats
```

Nous remarquons que les entrée/sortie sur les deux canaux sont indépendantes. La boucle peut être réarrangée en vue d'augmenter le parallélisme avec le déroulement du corps de la boucle, avec l'application des lois suivantes :

```
            WHILE b SEQ(P, Q, R)
---------------------------------------------------------------          <WHILE-SEQ2>
IF (b SEQ(P, WHILE b SEQ(Q, R, P), Q), ¬b SKIP)
```

Si les variables libres de P ne sont jamais changées par Q.

```
SEQ (P, R)
--------------                                                           <SEQ-PAR1>
PAR (P, R)
```

Si librevar(P) $\cap$ librevar(R) = $\varnothing$.

Grâce à la transformation précédente (telle que R et P sont maintenant ensemble), nous obtenons le processus résultat suivant, qui après la première entrée, met en parallèle les deux E/S. Cela signifie qu'après la première entrée de données, la forme de communications inter-processus du programme devient homogène, touts les processus composants effectuent l'échange de messages à la fin du traitement. Remarquons que les E/S en parallèle sur *out* et *in* ne causent pas un blocage, parce que tout les processus composants sont sous cette forme, donc l'ordre d'E/S est maintenu :

```
SEQ
  in ? data
  WHILE TRUE
    SEQ
      ... consommer 'data' et produire 'result'
      PAR
        out ! result
        in ? data
```

D'autres combinaisons des constructions syntaxiques

Le reste de la stratégie de l'extraction du parallélisme concerne le réarangement de différentes constructions, surtout la distribution des constructeurs séquentielles sur une construction parallèle. Cela n'exige pas nécessairement l'introduction de communications.

Une structure de ALT avec des gardes contenant des composants booléens peut être réduite à la combinaison de IF et ALT avec des gardes simples :

$$\frac{\text{ALT} (b \,\&\, g\ P, \underline{G})}{\text{IF} (b\ \text{ALT} (g\ P, \underline{G}), \neg b\ \text{ALT}(\underline{G}))}$$

La relation entre des constructions IF et ALT :

$$\frac{\text{IF } b \text{ ALT } g_i\ P_i}{\text{IF } b \text{ ALT } g_i\ (\text{IF } b\ P_i),\ i=1...n}$$

A condition qu'aucune variable dans b n'ait été changée par g_i.

Une construction SEQ distribue sur une construction IF et ALT :

$$\frac{\text{SEQ (IF } b_i\ P_i,\ Q)}{\text{IF } b_i \text{ SEQ } (P_i,\ Q),\ i=1 \ldots n}$$

$$\frac{\text{SEQ (ALT } g_i\ P_i,\ Q)}{\text{ALT } g_i \text{ SEQ } (P_i,\ Q),\ i=1 \ldots n}$$

Le choix conditionnel peut être exécuté avant d'entrer dans la construction de PAR. La condition <b_1 v. . .v b_n = TRUE> garantit que le processus conditionnel peut entrer dans la construction parallèle.

$$\frac{\text{PAR } (U_1\text{: IF } b_i\ P_i,\ U_2\text{: } Q)}{\text{IF } b_i \text{ PAR}(U_1\text{: } P_i,\ U_2\text{: } Q),\ i=1 \ldots n}$$

pourvu que b_1v . . .v b_n = TRUE.

Pour une construction séquentielle à l'intérieur d'une boucle, nous avons :

$$\frac{\text{WHILE } b \text{ SEQ}(P,\ Q)}{\text{IF } (b \text{ SEQ}(P, \text{ WHILE } b \text{ SEQ}(Q,\ P),\ Q),\ \neg b \text{ SKIP})}$$

Si les variables libres de P ne sont jamais changées par Q.

3.2.2.2 Parallélisme réel

Une attention particulière doit être portée au cas très possible, où le calcul d'un processus est interrompu ou retardé par des communications. Nous appelons ce phénomène *la séquentialisation des processus parallèles.*

Bien que les transformations selon les cas des structures du programme ci-dessus, permettent une implantation distribuée des constructions séquentielles, la séquentialisation des processus en parallèle n'est pas idéale puisque c'est le parallélisme réel que nous souhaitons obtenir. La séquentialisation des programmes parallèles est due au fait qu'un processus composant transfert ses résultats à un autre en parallèle, seulement après que celui-ci ait finit son calcul. Par exemple :

CHAN c: PAR (P, Q)
où P = SEQ (result := calcul1(), c ! result),
Q = SEQ (c ? var, calcul2(var))

ce programme peut être réduit par transformation (loi <I/O>) à :

SEQ (result := calcul1(), calcul2(result)).

Nous remarquons aussi que pour n'importe quel processus P, même si P est une construction séquentielle, nous pouvons toujours appliquer la loi <PAR-SKIP unit>. Ce qui nous donne un résultat qui n'a pas d'intérêt pour nous :

P = PAR (P, SKIP)

Dans n'importe quelle implantation, le temps d'exécution de ce programme ne peut pas être amélioré. Ce cas est vrai pour les transformations de programmes programmes parallèles séquentialisés.

Sur la base de transformations décrites auparavant, nous pouvons améliorer la stratégie de l'extraction du parallélisme systématiquement en considérant les cas, où il est possible d'extraire le parallélisme réel. Par exemple, dans les transformations de processus séquentiels en processus distribués, nous pouvons mettre des communications, au début ou à la fin d'un processus, dans des positions favorables possibles. La contrainte est que les communications concernées ne soient pas déplacées en dehors des communications externes avec l'environnement, et que les valeurs nécessaires doivent être calculées avant d'être communiquées.

Nous devons aussi établir des critères pour justifier l'introduction du parallélisme pour que les programmes obtenus aient une implantation améliorée. Ces critères concernent la mesure et l'estimation, finalement la comparaison de temps de calcul et de communications. Dans le cas où le coût

de la communication est supérieur que celui du calcul, il est même souhaitable de supprimer certain parallélisme.

• **Bloc élémentaire**

L'analyse de dépendances de données nous donne une méthode systématique pour extraire le parallélisme dans un bloc élémentaire (un segment de programme séquentiel [ASU86]). Nous utilisons des affectations multiples pour résumer les informations de dépendances de données et les propager dans une dimension locale.

Des lois concernées sont les suivantes :

$$\frac{\text{SEQ } (\underline{x} := \underline{e}, \text{ IF } b_i \, P_i)}{\text{IF } b_i \, [\underline{e}/\underline{x}] \text{ SEQ } (\underline{x} := \underline{e}, P_i), \; i=1...n}$$ <SEQ-IF assign>

$$\frac{\text{SEQ } (\underline{x} := \underline{e}, \text{ ALT } g_i \, P_i)}{\text{ALT } g_i \, [\underline{e}/\underline{x}] \text{ SEQ } (\underline{x} := \underline{e}, P_i), \; i=1...n}$$ <SEQ-ALT assign>

A condition qu'aucune variable dans $\underline{x}$ ou $\underline{e}$ ne soit l'entrée dans les gardes g_i.
Quand $\underline{x} := \underline{e}$ termine, la construction se comporte comme IF ou ALT, mais tient compte de l'effet de l'affectation.

$$\frac{\text{SEQ } (\underline{x} := \underline{e}, \; \underline{x} := \underline{f})}{\underline{x} := \underline{f} \, [\underline{e} / \underline{x}]}$$ <combin assign>

La composition séquentielle de deux affectations à la même liste de variables est réduite à une seule affectation en tenant compte de l'effet de l'affectation précédente.

$$\frac{\text{SEQ } (x := e, \; c \; ? \; x)}{c \; ? \; x}$$ <SEQ-com 1>

Il n'y a pas de sens d'affecter à une variable qui est utilisée comme entrée avant son utilisation.

$$\frac{\text{SEQ } (\underline{x} := \underline{e}, \; c \; ? \; y)}{\text{SEQ } (c?y, \; \underline{x} := \underline{e})}$$ <SEQ com2>

à condition que y ne soit pas libre dans $\underline{x} := \underline{e}$.

SEQ ($\underline{x}$:= $\underline{e}$, c ? f)
---------------------------- <SEQ com3>
SEQ (c! f ($\underline{e}/\underline{x}$), $\underline{x}$:= $\underline{e}$)

PAR ($\underline{x}$:= $\underline{e}$, $\underline{y}$:= $\underline{f}$)
-------------------------------------- <PAR assign>
$\underline{x}$ + $\underline{y}$:= $\underline{e}$ + $\underline{f}$

Deux affectations multiples en parallèle peuvent être combinées à une seule.

- **Structures composées**

La transformation sous forme suivante nous donne le parallélisme réel, à partir d'une structure composée :

où Op est une des opérateurs IF, ALT, SEQ, WHILE.

```
            Op ( ... PAR (P, Q), ...)
------------------------------------------------------------------
CHAN c, d :
PAR
      VAR init(P) : SEQ (c ? init(P), P, d ! final(P))
      Op (  ...
            VAR final(P) : SEQ (c ! init(P), Q, d ? final(P))
            ...)
```

3.2.3 Un exemple du changement de la forme de communications

Nous illustrons l'idée de changer la position d'E/S dans un processus afin d'améliorer la forme de communication, en minimisant le retard causé par E/S.

Considérons l'algorithme systolique suivant pour calculer la multiplication de deux matrices, où chaque noeud de la matrice résultat est représenté par un seul processus.

R = A * B où A et B sont de dimension M x M; un élément de R est calculé par :

r(i,j) = SUM a(i, k) * b(k, j), k=1..M

```
PROC Mult(CHAN OF INT N, S, E, W)
  INT Result:
  INT A, B :
  SEQ
    Result := 0
    SEQ i = 0 FOR M
      SEQ
        PAR
          N ? A
          W ? B
        Result := Result + ( A * B )
        PAR
          S ! A
          E ! B
```

Pour la génération de deux matrices A et B, les deux procédures suivantes peuvent être utilisées :

```
PROC GenerateA ([]CHAN OF INT Avalues)
PROC GenerateB ([]CHAN OF INT Bvalues)
```

Le programme suivant calcule C = A * B : pour une matrice de dimension M x M, m*(m+1) canaux sont nécessaires dans les deux sens verticale et horizontale. La figure 3.2 illustre le fonctionnement et la forme de communications du programme, où M = 4, une ligne flechée représente le sens de communication.

```
VAL M IS UnNombre :
[ M, M+1] CHAN OF INT Vert, Horz:
PAR
  PAR i = 0 FOR M
    PAR j = 0 FOR M
      Mult(Vert[i, j], Vert[(j, j +1 ],
           Horz[i, j], Horz[i+1, j] )
```

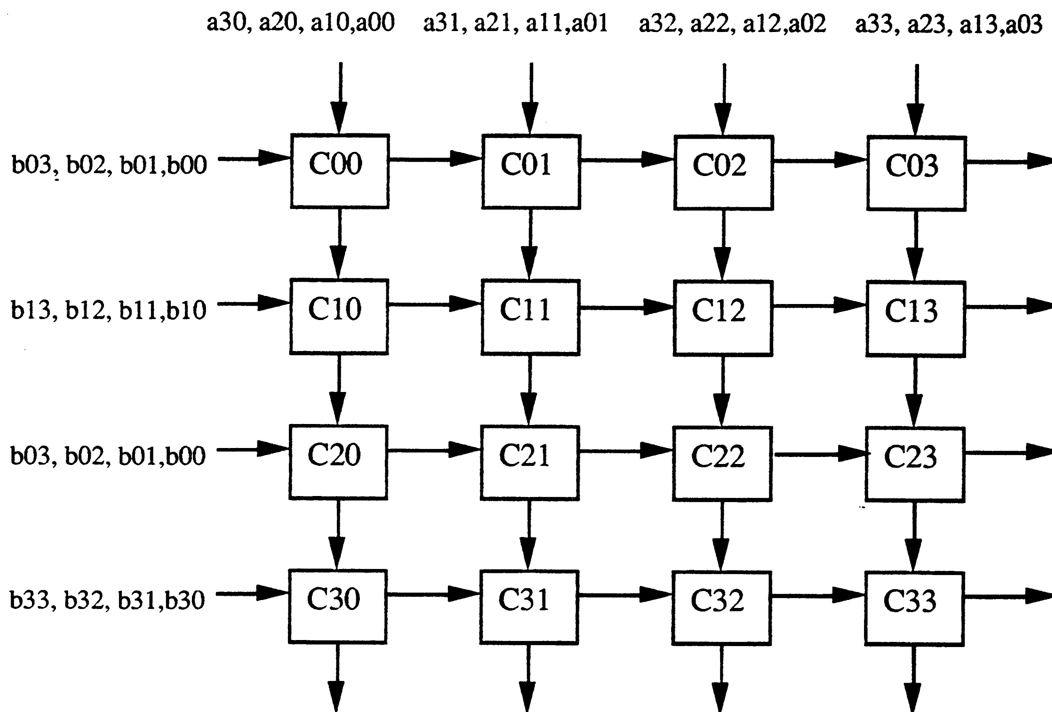

Figure 3.2 La multiplication de deux matrices par un programme systolique

Transformations en vue de changer la forme de communication

La procédure <PROC mult()> consiste à effectuer trois opérations séquentielles : entrer de nouvelle valeurs, mettre à jours la variable *Result;* sortir les valeurs. Il est désirable de mettre certaines structures séquentielles en parallèle, particulièrement les quatre entrées/sorties. Nous procédons les transformations suivantes :

Par la loi <SEQ-PAR *> nous transformons les derniers deux composants de SEQ en parallèle, et par loi <PAR assoc>, nous obtenons la structure répliquée comme suit :

```
SEQ i = 0 FOR M
   SEQ
      PAR
         N ? A
         W ? B
      PAR
         S ! A
         E ! B
         Result := Result + ( A * B )
```

La structure parallèle d'entrées sur les canaux N et W peut être transformée en appliquant les lois <Decl *>, <Input *> en :

```
INT Atemp, Btemp:
PAR
    SEQ
        N ? Atemp
        A := Atemp
    SEQ
        W ? Btemp
        B := Btemp
```

Par loi <SEQ-PAR *>, et en remarquant la condition de séparation de variables, le processus ci-dessus sous forme de PAR(SEQ, SEQ) peut être changé en SEQ (PAR, PAR) :

```
INT Atemp, Btemp:
SEQ
    PAR
        N ? Atemp
        W ? Btemp
    A, B := Atemp, Btemp
```

Nous obtenons la structure répliquée de SEQ comme suit :

```
SEQ i = 0 FOR M
    INT Atemp, Btemp:
    SEQ
        PAR
            N ? Atemp
            W ? Btemp
        A, B := Atemp, Btemp
        PAR
            S ! A
            E ! B
            Result := Result + ( A * B )
```

Nous déroulons la structure répliquée une fois comme suit :

```
SEQ i FOR n SEQ (P, Q, R)
= SEQ (P, SEQ i FOR n-1 SEQ (Q, R, P), Q, R)

où   P = PAR(N ? Atemp, W ? Btemp),
     Q =  A, B := Atemp, Btemp
     R = PAR(S ! A, E ! B, Result := Result + (A * B))
```

et mettons en parallèle des constructions possibles (ici, SEQ(Q, R, P)), nous avons le processus équivalent :

```
SEQ
  N ? A
  W ? B
  SEQ i = 0 FOR (M - 1)
    INT Atemp, Btemp:
    SEQ
      PAR
        S ! A
        E ! B
        N ? Atemp
        W ? Btemp
        Result := Result + ( A * B )
      A, B := Atemp, Btemp
  PAR
    S ! A
    E ! B
    Result := Result + ( A * B )
```

Remarques :

- Dans le processus résultat, après la première fois d'entrées sur N et W, le reste du traitement (E/S sur les quatre canaux, le calcul de *Result*) est en parallèle, ce qui accélère les communications entre l'ensemble de processus ;

- Dans cet exemple, le gain de la transformation paraît être minimal. Mais dans le cas où le temps du calcul est dominant, le gain peut être considérable.

Chapitre 4

Transformations de programmes pour leur placement sur des machines parallèles

4.1 Placement des programmes sur machines parallèles

Etant donné un programme, les performances de son exécution sur une architecture parallèle dépendent principalement du placement de ses processus composants sur les processeurs [KM89, Gaud88, MT91]. Bien que les outils d'aides existants permettent à l'utilisateur de spécifier le placement, cette tâche reste complexe à cause de la variété et de la complexité de la configuration d'une architecture cible.

Des outils et algorithmes d'allocation automatiques soulagent cette tâche complexe des utilisateurs. En plus ils permettent la portabilité des programmes entre différentes architectures parallèles.

4.1.1 Définition du problème

Soit un système parallèle formé d'un ensemble N de m processeurs $\{n_1, n_2, ..., n_m\}$ qui communiquent entre eux par un réseau d'interconnexion. Soit un programme composé d'un ensemble P de n processus $\{p_1, p_2, ..., p_n\}$ communiquant entre eux. Le problème d'allocation peut être formellement décrit par une fonction :

$\pi : P \rightarrow N$

Plusieurs critères entrent en considération pour l'évaluation d'un placement, comme par exemple: la longueur de chemins de communication, l'équilibre de charge des processeurs et des liens de communication. En général, un placement est optimal si le temps d'exécution du programme est minimisé. Ce temps dépend des temps d'exécution des processus sur les processeurs où ils sont placés, et des temps de communication entre eux.

Nous pouvons représenter les propriétés d'un placement par une fonction coût (exprime un critère) qui associe un coût à chaque placement :

Placement optimal : Un placement π est optimal si on a :

coût(π) <= coût(π')
pour tout placement π'.

Comme indiqué dans la littérature, le problème dans sa généralité est NP_complet [TM91]. Les stratégies d'allocation sont basées sur l'une des approches suivantes :

- La théorie des graphes (partition de coupe minimale d'un graphe, etc),
- Programmation mathématique,
- Des heuristiques.

On distingue deux types d'algorithmes d'allocation suivant l'instant où le placement est décidé : l'allocation statique (pendant la phase de la compilation) et l'allocation dynamique (pendant la phase d'exécution). Vue la nature statique de l'analyse et la manipulation des programmes dans notre approche transformationnelle, nous nous limitons à l'allocation statique du placement.

Allocation statique

Rappelons qu'une stratégie de placement consiste à partitionner les processus d'un programme suivant certains critères, et à affecter les partitions obtenues sur les processeurs d'une architecture spécifique.

Une stratégie générale, pour un programme P avec un degré de parallélisme n, et pour un système composé de N processeurs avec une connectivité complète, considère d'une façon suffisante seulement le critère de distribution de charge des processeurs. L'allocation statique des processus est effectuée comme suit :

- Construction du graphe de processus d'un programme P : l'estimation du poids de chaque processus (son temps d'exécution) et des communications inter-processus ;

- Répartition des processus selon certains critères. Le but principal de cette étape est de regrouper l'ensemble des processus en parallèle, en fonction du nombre disponible de processeurs, en tenant compte du critère de distribution des poids des partitions résultats, de la minimisation du coût de communications entre les partitions ;

Les processus d'une même partition seront placés sur le même processeur, et les différentes partitions s'exécutent en parallèle.

- Elimination des limitations physiques du système cible. Par exemple, l'allocation de liens physiques aux canaux de processus, le routage entre les processus résidant sur des processeurs qui ne sont pas directement connectés.

4.1.2 Supports pour le placement par transformation

Des algorithmes de placement processus/processeur étant en dehors de nos préoccupations ici, nous cherchons des supports logiciel qui facilitent le placement, et assurent la manipulation correcte des programmes lors de la phase d'allocation d'une stratégie de placement. La raison et la nécessité de transformation d'un programme, pour son exécution parallèle, est qu'un allocateur du placement ou le routage de messages peut seulement obtenir le changement sur les communications inter-processus, l'allocation processus/processeurs. Il ne peut pas changer la structure entière de la partition de l'algorithme et de la forme de communications.

On reconnaît des supports dans le système de transformation pour le problème de placement sur les aspects suivants :

- Dans une stratégie de placement, on suppose comme une condition pré-acquise que le parallélisme d'un programme (nombre de processus composants en parallèle définis dans les constructions externes) soit égal ou supérieur au nombre de processeurs. Le système de transformation peut offrir une aide par la détection et l'extraction du parallélisme potentiel d'un programme. Cet aspect est abordé dans le chapitre 3.

- Lors de la phase d'allocation d'une stratégie de placement (qui consiste à regrouper l'ensemble de processus), il existe des problèmes à résoudre dus aux limitations physiques du système cible, comme par exemple : le nombre limité de liens de communication par rapport à celui de canaux de communication des processus (regroupés dans une partition du placement) ; le routage de messages entre des processus placés sur des processeurs non directement connectés ; des structures de données globales (par l'héritage) entre des processus sur différentes partitions.

Ces problèmes peuvent être résolus par différentes méthodes et dans différents environnements, par exemple, par le système de base (OS). Notre solution consiste à éliminer ces contraintes en changeant les structures de programmes, telle que la forme de communications inter-processus, la relation de dépendance de données soit conforme à une architecture cible.

4.2 Réalisation d'une stratégie de placement

4.2.1 Routage de messages et multiplexage de canaux

Le routage de messages représente la communication entre des processus dans différentes partitions d'une stratégie de placement. Des supports logiciel ou système doivent être fournis tels que deux processus placés sur différents processeurs puissent communiquer : le routage de messages et le multiplexage de canaux.

Par exemple, supposons que nous avons un programme et une architecture, dont les graphes de connectivité sont les suivants (Figure 4.1) :

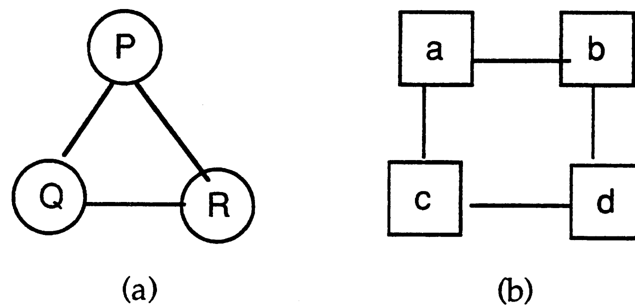

Figure 4.1 Graphe du programme (a) et graphe des processeurs (b).

Une stratégie de placement possible est de placer un processus parallèle sur un processeur individuel, dans notre cas : P sur a, Q sur b, R sur c. Les processeurs b et c ne sont pas connectés directement, la communication entre les deux processus Q et R est impossible.

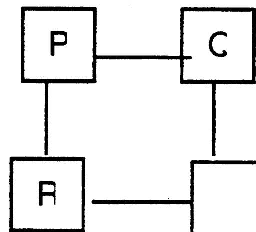

Figure 4.2 Un placement possible.

Solution par le système de base

Le système de base offre un mécanisme de support pour le routage de messages par des solutions suivantes [Jones88] :

- Routage de messages : un composant système prend en charge le routage de messages, par exemple, un serveur (centralisé ou distribué) sur chaque processeur transfère tous les messages vers un processeur voisin, selon un chemin de communication.

- Multiplexage de canaux : cette approche résoud le problème de limitation du nombre de liens physiques, par la gestion et l'utilisation partagée des liens physiques entre les canaux de communications.

- Reconfiguration dynamique : cette méthode a pour but d'éliminer d'une façon dynamique, la limitation de liens par l'adaptation de la configuration, surtout l'inter-connexion des processeurs, en fonction de la forme de communication inter-processus. L'allocation s'est faite d'une façon que des liens soient alloués aux processeurs qui les demandent, et une fois une communication est terminée, des liens seront libérés.

Support par transformation

Le routage de messages et le multiplexage de canaux pour l'exécution parallèle d'un programme peuvent être réalisés par transformation de programmes [Jones88, WC89, CLS89]. Cette transformation permet la réalisation du modèle de calcul parallèle (un grand degré arbitraire de la connectivité inter-processus) sur une machine physique. Elle a d'autres avantages, par exemple, le monitoring de communications sur les canaux (la collection des informations de leur utilisation, donc le comportement et la forme de communications inter-processus) etc.

Des méthodes de transformation proposées consistent à intégrer des composants logiciels dans un programme d'application, ces composants sont chargés du multiplexage des canaux, du routage de messages selon une politique de routage, etc.

L'introduction de ces composants logiciels peut causer des problèmes, comme le changement de la sémantique du programme, la perte de synchronisation, le blocage de communication. Le système de transformation offre un environnement qui permet la manipulation correcte et la vérification formelle des programmes.

Transformation pour le routage de messages et le multiplexage de canaux

Dans [OI88, LS88], une approche systématique de transformation pour l'adaptation de la connectivité inter-processus d'un programme à la connectivité de processeurs a été proposée. Le principe de l'approche est l'insertion d'un processus "routeur" sur chaque processeur, responsable de communications inter-processeurs pour l'ensemble de processus exécutant sur ce processeur: il est chargé de collectionner des messages provenant des processus émetteurs, et de les transférer aux processus récepteurs, ou à d'autres processus "routeur" dans le cas où des processus sont placés sur différents processeurs. En même temps, il est chargé du multiplexage de canaux, en tenant compte du nombre limité de liens physiques disponibles.

Prenons l'exemple vu au début de cette section (Figure 4.2), maintenant que nous avons un processus routeur sur chaque processeur, la communication entre les processus Q et R devient possible, via le routeur sur le processeur d, ou celui sur a (Figure 4.3).

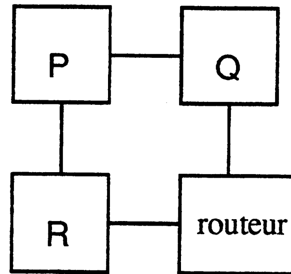

Figure 4.3 Le routage de messages et le multiplexage de canaux.

Chaque communication ayant la forme suivante :

PAR (P (c ! exp), Q (c ? var))

est transformé en :

PAR
P (SEQ (c ! exp, ack ? ANY))
Q (SEQ (c ? var, ack ! ANY))

La raison d'ajouter un canal complémentaire pour chaque canal de communication est de conserver la synchronisation de communication. Considérons le cas sans la communication sur le canal <ack>, chaque communication de sortie <c ! exp> aura lieu avec le processus routeur, et après quoi le processus P peut continuer son exécution (similaire pour la communication d'entrée), ce qui change la synchronisation de :

PAR (P(c ! exp), Q(c ? var))

Avec la communication supplémentaire sur un canal de synchronisation <ack>, le processus P ne peut continuer qu'après la réception de valeur <exp> par le processus Q.

Le processus routeur

Un ou plusieurs routeurs sont nécessaires pour le transfert d'un message du processus émetteur au processus récepteur, connectés par un canal (Figure 4.4). L'information de destination sera codée comme un "tag" dans chaque message pour que le routeur sache sa destination. Pour la simplicité, nous supposons que le routeur d'un processeur est chargé de tous ses liens. Chaque routeur doit être muni d'une table de routage, qui contient des informations de l'allocation et de routage pour chaque communication. Alors, les processus routeurs sont créés de la façon suivante : chaque processus composant est placé sur un processeur distinct,

PAR (P1, P2, ..., Pn)

est transformé en :

```
PAR
   CHAN C[] : PAR (P1', Routeur1)
   CHAN C[] : PAR (P2', Routeur2)
   ...
   CHAN C[] : PAR (Pn', Routeurn)
```

où un processus routeur a la forme suivante :

```
routeur =
ALT i FOR n (C[i] ? protocole
   SEQ
      décoder protocole
      envoyer le message dans protocole au routeur du processus destinataire
      attendre le signal de réception du routeur destinataire
      envoyer un signal au processus émetteur
```

La différence entre un processus composant Pi et le processus Pi' après la transformation est : les canaux dans un processus Pi sont changés en un vecteur de canaux locaux C[], qui est géré par le processus routeur $Routeur_i$, et toutes les communications sont transformées d'une façon comme ci-dessus. Quand un message est émis par un processus local, le routeur ajoute un tag dans le message, indiquant quel canal dans le programme source que le message utilise. Si le message est destiné au processus dans un autre processeur, le message sera envoyé à travers un lien physique, à un routeur intermédiaire dans le chemin de communication. Le message sera ainsi transféré au processus destinateur par des routeurs.

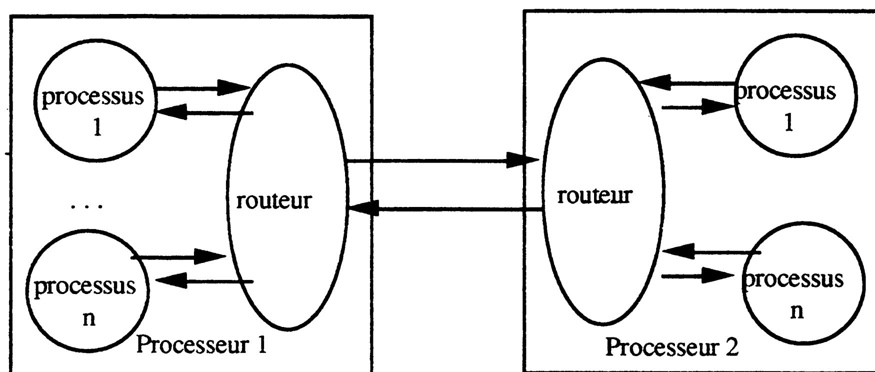

Figure 4.4 Le processus routeur.

La justification de transformation

L'introduction des processus routeur peut être justifiée dans le système de transformation par la transformation suivante :

```
          PAR
             SEQ (P1, c ! e, P2)
             SEQ (Q1, c ? v, Q2)
----------------------------------------------------------------
PAR
   CHAN lien, ack :
   PAR
      SEQ (P1', c1 ! e, ack1 ? ANY, P2')
      SEQ (c1 ? v, lien ! v, ack ? ANY, ack1 ! ANY)
   PAR
      SEQ (Q1', c2 ? v, ack2 ! ANY, Q2')
      SEQ (lien ? v, c2 ! v, ack2 ? ANY, ack ! ANY)
```

La preuve de cette transformation est triviale, nous l'avons omise ici.

Considérations de l'efficacité

Un canal supplémentaire de synchronisation est ajouté pour chaque canal, cela augmente la volume des messages échangés. Un processus routeur sur chaque processeur a besoin d'une table qui contient des informations de routage de messages, quand le nombre de processeurs devient important, la table de routage pourrait devenir difficile à mettre en oeuvre et à gérer.

4.2.2 Implantation distribuée des structures de données globales

Un problème que l'on rencontre dans le placement est le cas, où il existe des structures de données communes qui sont utilisées dans différents processus, et ceux-ci sont placés sur des processeurs distincts et non connectés directement.

Ce problème se présente souvent sous deux formes très proches :

- L'héritage de données. Une structure de données est partagée par les processus composants en parallèle à l'intérieur d'une structure séquentielle, et ils sont placés sur différents processeurs. Dans ce cas, la construction séquentielle est appelée le processus-père, et les processus composants les processus-fils.

- Structures de données globales [RS91]. Un ensemble de processus en parallèle prend en charge du traitement d'une partition de la structure de données globale, et ils sont placés sur différents processeurs.

La solution du problème peut être réduite à l'implantation distribuée de structures de données.

Un exemple du premier cas est le programme général suivant (notons que la variable A est partagée uniquement en lecture). Il est peut-être une solution naturelle pour un problème, surtout dans des programmes existants écrits pour une machine à mémoire commune. Il effectue une certaine initialisation avant d'entrer dans les constructions parallèles, celles-ci traitent une partie de données en même temps. Mais il est improbable que ce programme puisse être placé sur une architecture distribuée, à cause des données globales. La structure de données A, définie dans le processus-père P, est utilisée dans deux processus-fils qui résident sur des processeurs distincts (Cf. la figure 4.5 (a)).

```
INT []A:
SEQ
  Initialiser(A)          -- Processus P
  PAR
    P1(A)
    P2(A)
```

Nous remarquons qu'en réalité, une telle structure exprime des propriétés du programme, comme la compréhensibilité, l'absence de blocage, la dépendance de données. Un programmeur devrait pouvoir écrire des programmes qui sont clairs mais physiquement difficile à réaliser, et transformer le programme d'une façon que le parallélisme soit déplacé vers l'externe, rendant possible une implantation multi-processeur.

La solution par mécanisme système

Une solution par le système de base détecte et passe dynamiquement les valeurs d'héritage, à partir des processus-pères aux processus-fils. Cette approche est efficace pour résoudre les valeurs d'héritage qui sont dynamiques, par exemple une partie d'une structure de données. Ce problème peut être résolu en passant toute la structure de données, mais les communications introduites peuvent être trop coûteuses pour être pratiques.

La solution par transformation

Nous souhaitons que ce genre de manipulation soit explicite dans la syntaxe des programmes pour faciliter le placement sur une architecture distribuée. La solution que nous proposons ici est l'élimination de la dépendance de données entre des processus, en introduisant des communications entre les processus concernés.

Dans le cas où les valeurs d'héritage sont dynamiques (la partie précise de la structure de données ne peut pas être déterminée par l'analyse statique), nous pouvons passer toute la structure de données d'héritage. Bien que cela impose un coût supplémentaire de communication, ce passage de valeurs est tout de même nécessaire pendant l'exécution du programme. Cette solution statique et explicite peut faciliter l'analyse de flot de données.

Le problème de variables d'héritage est réduit à comment diffuser les valeurs de ces variables vers les processus fils qui les utilisent. Dans notre exemple, si P1 et P2 se situent dans des processeurs distincts, le problème est comment passer la valeur de A, à partir du processus P à P1 et P2.

En résumé, notre stratégie de transformation est la suivante :

(1) transformer les programmes pour que le parallélisme potentiel dans les constructions séquentielles puisse être poussé à l'extérieur, donc permet une implantation distribuée.

(2) introduire des communications explicites pour remplacer la dépendance de données.

Elimination de l'héritage de données

Le principe de transformation est de collecter toutes les valeurs d'héritage d'un processus-fils, et de lui diffuser ces données par communication explicite, à partir du processus-père. Dans l'exemple ci-dessus, supposons que le processus père P, les deux processus parallèles P1 et P2 se situent dans trois processeurs distincts, nous résolvons l'héritage des données A en introduisant une communication. Par conséquent, nous déclarons des variables locales nécessaires pour chaque processus fils. Le programme résultat de la transformation est le suivant (Figure 4.5 (b)):

```
[] CHAN OF INT ToP1, ToP2:
PAR
  INT [] A:
  SEQ                     -- processus P
    initialiser(A)
    PAR
      ToP1 ! A
      ToP2 ! A

  INT [] heritage:        -- variable locale
  SEQ                     -- processus P1
    ToP1 ? heritage
    P1(heritage)

  INT [] heritage:        -- variable locale
  SEQ                     -- processus P2
    ToP2 ? heritage
    P2(heritage)
```

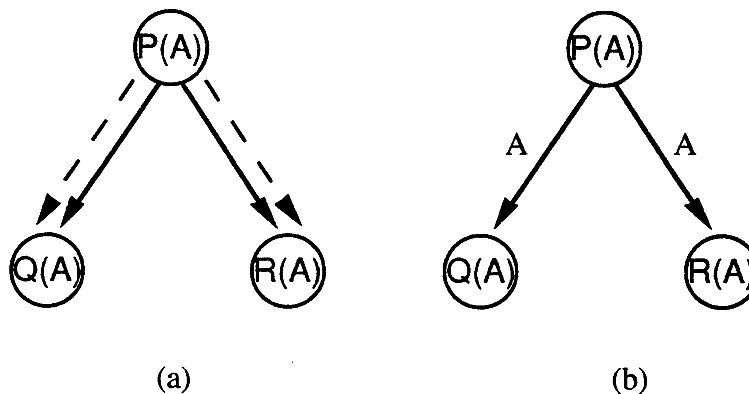

Figure 4.5 Distribution des structures de données globales.

La justification de cette transformation est l'application de lois <PAR-SEQ com*> et des lois de déclaration <VAR rename>, <VAR-Op sym> etc.

Considération de l'efficacité

(1) Nous pouvons remarquer que la communication introduite est indispensable dans l'implantation distribuée du programme, mais peut être optimisée pour minimiser son coût. Par exemple, si les valeurs utilisées par P1 ou P2 sont une partie de la structure A, et cette partie peut être déterminée statiquement, notée respectivement A1 et A2. Alors nous pouvons passer , au lieu de la structure entière A, A1 et A2 aux processus P1, P2 respectivement. Nous avons le processus P résultat suivant :

où A1, A2 $\subseteq$ A,

```
[]CHAN OF INT ToP1, ToP2:
PAR
  INT []A:
  SEQ                          -- processus P
    initialiser(A)
    PAR
      ToP1 ! A1
      ToP2 ! A2
```

(2) Dans le cas où les valeurs d'héritage sont dynamiques (par exemple, des indices de A sont passées par communication, ou l'entrée de l'utilisateur), d'une sorte que la détermination de A1 et A2 est très difficile, nous pouvons résoudre le problème en passant toute la structure de données A.

Et nous avons le résultat suivant :

Théorème 4.1 *Elimination d'héritage implicite de programmes.* Etant donné un programme P et le contexte d'héritage, dont les valeurs sont utilisées par les processus-fils en parallèle d'un placement, P peut être transformé en un programme équivalent P', tel que libre(P) $\in$ libre(P') et P = P' peut être prouvé dans le système de transformation, et qu'il n'existe pas d'héritage implicite dans P'.

La démonstration de ce théorème est similaire à la procédure de transformations décrite ci-dessus.

Un exemple d'implantation distribuées de données

Cet exemple a pour but d'illustrer la stratégie de transformation en vue de distribuer une structure de données globale. Le programme illustratif suivant est pris dans [May86], où d'une façon intuitive, une autre version équivalente de ce programme est donnée pour éliminer la synchronisation globale, ce qui impose la même difficulté d'une mise en oeuvre distribuée de communication globale. En plus, les deux structures de données globales : maître[n], esclave[n] posent un problème pour une implantation sur une architecture sans mémoire commune.

Nous utilisons le système de transformation pour démontrer qu'il est possible de résoudre ce problème par la transformation formelle et automatique. Aussi, nous utilisons cet exemple pour montrer la transformation nécessaire pour une stratégie de placement.

```
VAR maître[n], esclave[n]:
WHILE TRUE
   SEQ                                   -- processus PP
      PAR i = 0 FOR n                    -- processus Pi
         esclave[i] := f(maître[i])
      PAR                                -- processus Q
         input ? maître[0]
         PAR i = 0 FOR n-1               -- processus Ri
            maître [i+1] := esclave[i]
         output ! esclave[n]
```

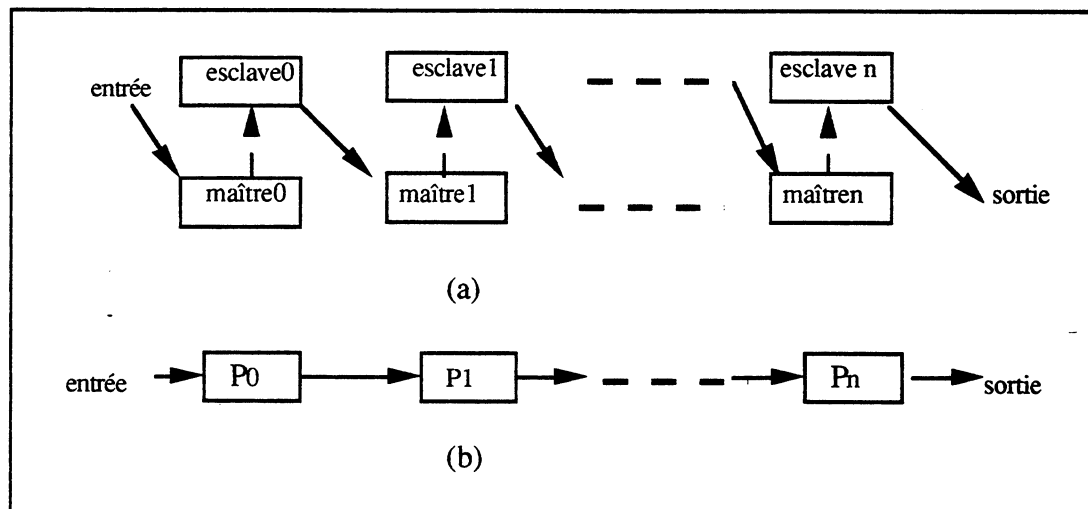

Figure 4.6 Distribution des données.

Ce programme décrit le calcul d'un algorithme systolique avec la synchronisation globale. A chaque itération de la boucle, tous les calculs (les lignes pointillés dans la figure 4.6 (a)):

```
PAR i = 0 FOR n
  esclave[i] := f(maître[i])
```

commencent et terminent en même temps, après quoi les valeurs calculées circulent tout au long du pipeline (les lignes de trait plein dans le graphe) :

```
PAR i = 0 FOR n-1
  maître [i+1] := esclave[i]
```

Nous voulons transformer le programme en une forme abstraite adéquate pour une une implantation distribuée. La procédure de transformation est la suivante :

En appliquant les lois (PAR-SEQ com*), nous transformons le processus R comme suit, en introduisant des canaux et des communications, en vue de réaliser l'échange de valeurs:

```
maître [i+1] := esclave[i]
```

par la communication explicite :

```
PAR i = 0 FOR n-1               à      CHAN c[n]:
  maître [i+1] := esclave[i]           PAR i = 1 FOR n-1  -- processus R'
                                         SEQ
                                           c[i] ? maître[i]
                                         c[i+1] ! esclave[i]
```

Nous renommons le canal input, output en c[0], c[n+1] respectivement. Alors le processus Q peut être réécrit comme suit :

```
CHAN c[n+1]:    --processus Q'
PAR i = 0 FOR n
SEQ
  c[i] ? maître[i]
  c[i+1] ! esclave[i]
```

En appliquant les lois (PAR-SEQ com*) une nouvelle fois, nous combinons les processus P et Q' pour obtenir le nouveau programme (le remplacement d'une variable esclave[i] par une variable locale *v* est fait par les lois d renommage, avec une justification intuitive) :

```
CHAN [n+1]:
PAR i = 0 FOR n
  VAR r:
  SEQ
    c[i] ? maître[i]
    r := f(maître[i])
    c[i+1] ! r
```

Remarquons que dans le programme PP, la phrase WHILE TRUE peut être placé avant la déclaration <VAR maître[n], esclave[n]>. En supprimant la déclaration esclave[n] par la loi (VAR decl*), et en poussant la déclaration <VAR maître[n]:> après la déclaration de canaux, nous avons pour le programme entier :

```
WHILE TRUE
CHAN [n+1]:
VAR maître[n]:
PAR i = 0 FOR n
   VAR r:
   SEQ
      c[i] ? maître[i]
      r := f(maître[i])
      c[i+1] ! r
```

D'après la loi <Decl-distrib> nous pouvons affecter une déclaration <VAR maître[n]:> dans le processus composant parallèle qui l'utilise.

```
WHILE TRUE
CHAN [n+1]:
PAR i = 0 FOR n
   VAR maître[i], r:
   SEQ
      c[i] ? maître[i]
      r := f(maître[i])
      c[i+1] ! r
```

Finalement, en renommant chaque maître[i] en une variable interne, disont *d*, nous obtenons la version finale :

```
WHILE TRUE
CHAN [n+1]:
PAR i = 0 FOR n
   VAR d, r:
   SEQ            -- Pi
      c[i] ? d
      r := f(d)
      c[i+1] ! r
```

Nous avons dérivé dans le système de transformation une version de ce programme qui est équivalente à celle dans [May86], dont le modèle de calcul peut être représenté par la figure 4.6 (b).

4.2.3 Transformations pour réduire le degré de parallélisme

Dans une stratégie de placement d'un programme, où le nombre de processus parallèles est bien supérieur au nombre de processeurs disponibles, les processus seront regroupés dans un certain nombre de partitions. Les méthodes générales de transformation basées sur l'analyse géométrique sont expliquées dans [Quint86].

Néanmoins il existe des cas où nous pouvons effectuer une transformation du programme, tel que le nombre de processus soit égal au nombre de partitions, tout en respectant les critères d'optimisation du placement. L'intérêt de cette transformation est que l'on demande souvent que le placement de processus d'un programme sur les processeur soit identique pour éviter la dégradation de sa performance, due à la simulation de canaux de communications [LS89].

Une autre avantage de cette manipulation du programme est qu'il est possible de supprimer ou d'alléger le problème d'allocation de sources (par exemple des liens physiques) pendant la transformation.

4.2.3.1 Un exemple de la projection d'un programme systolique

Dans l'implantation du calcul des réseaux systoliques, on distingue les problèmes suivant [Leng88, Quint86] :

(1) La dimension du réseau systolique dépasse celle de l'architecture cible dont différentes solutions existent : le changement de conception du réseau systolique ; la projection pour réduire la dimension du programme.

(2) Le nombre de canaux dépasse le nombre de liens d'un processeur : c'est un problème traité dans la communication globale (§4.1), dont une solution est le multiplexage et le routage de messages.

Le problème (1) nous intéresse particulièrement, parce que le principe est l'adaptation d'un programme dans un placement, dont une solution est la projection d'un réseau systolique.

Dans [LS89], une méthode est illustrée pour la conversion d'un programme à deux dimensions en un équivalent à une dimension. Leur justification de transformations utilisées est basée sur une relation entre deux systèmes d'états de transition, ce qui permet le remplacement d'un système

par un autre dans un environnement restrictif, où ces deux systèmes satisfont un certain prédicat. Comme indiqué, il existe une distance entre la méthode et leur application, et souvent il faut de l'intuition pour l'utilisation et le contrôle du processus de transformations. Suivant cette méthode, nous appliquons l'approche transformationnelle à cet exemple.

Définition du problème

Etant donnés deux matrices A et B, notre but est de calculer la composition C :

C = A * B où A, B et C sont des matrices à dimension n x n.

Le calcul peut être implanté par de façons différentes. Nous le réalisons par un programme systolique dont le modèle du calcul peut être illustré par la figure suivant :

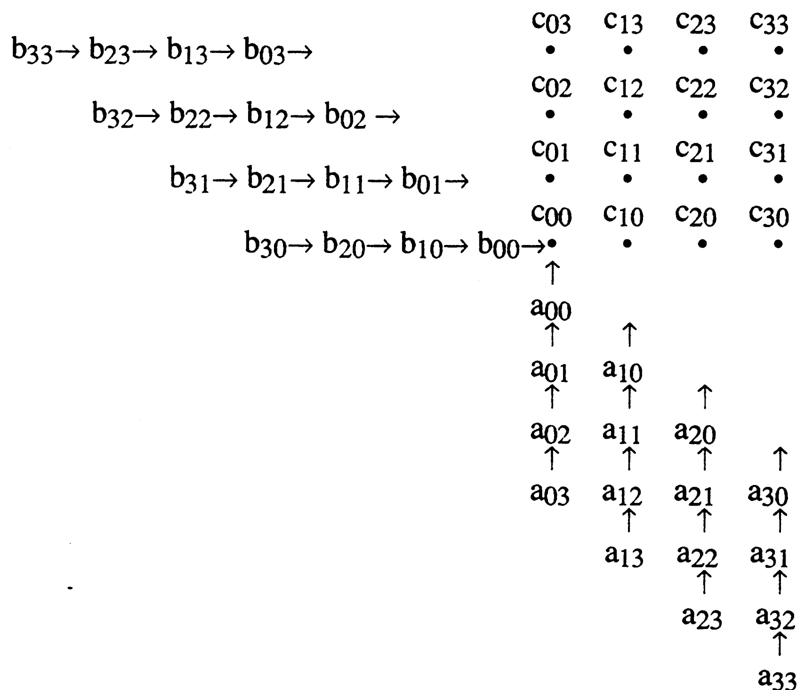

Figure 4.7 La composition de matrices - réseau systolique à deux dimensions.

Le programme systolique est composé de trois types de processus (cellules de traitement) : les directions de flot de données sont de gauche à droite et de bas en haut :

- Les cellules de calcul c_{ij} : elles acceptent le flot C, qui vient de gauche, calculent des éléments de C, transmettent les flots A et B, et transfert le flot C vers la droite ;

- Les cellules d'entrée a_{ij} et b_{ij} : elles se situent à gauche, injectent d'abords le flot C et ensuite le flot B ; celles d'en bas injectent le flot A ;

- Les cellules de sortie (omises ici) : les cellules à droite, extraient le flot B, et ensuite, celles d'en haut extraient le flot A.

Le programme complet à deux dimensions est donné dans l'annexe B1, dont un segment typique (le calcul des éléments de C) est le suivant :

```
             -- CELLULES DE CALCUL
19           PAR col = 0 FOR n
20             PAR row = 0 FOR n

                 -- entrée du flot C
21               VAR AElement, BElement, CElement:
22               SEQ

23                 Right[col, row] ? CElement

24                 SEQ unused = 0 FOR (n - 1) - col
25                   VAR tmp:
26                   SEQ
27                     Right[col, row] ? tmp
28                     Right[col+1, row] ! tmp

                 -- le calcul de cij
29                 SEQ k = 0 FOR n
30                   SEQ

31                     PAR
32                       Up[col, row] ? AElement
33                       Right[col, row] ? BElement
34                     CElement := CElement + AElement * BElement

                     -- transférer les flots A et B en haut et à droite
35                     PAR
36                       Up[col, row+1] ! AElement
37                       Right[col+1, row] ! BElement
```

La stratégie de transformation

Le principe de la transformation du programme est la projection de la dimension verticale (a_{ij}) à la dimension horizontale, dont la procédure est la suivante :

Premièrement, nous combinons les cellules d'une colonne (a_{ij} où j est fixé) en une, et par conséquent, les canaux d'une colonne en un. Ensuite ces cellules et canaux sont placés dans la dimension horizontale (le flot B), donc la direction d'entrée/sortie sera changée aussi. Le détail est omis ici (Cf. [LS89]). Nous cherchons plutôt les lois et les transformations nécessaires pour la projection, en particulier, les transformation de constructions parallèles en celles séquentielles. Ces transformations permettent la conversion de la direction de flots de données de verticale à horizontale : en commutant les boucles de communications sur *col* et *row* ; en remplaçant les canaux *Up* par *Right;* en éliminant les communications sur *Up* dans le processus de calcul; en convertissant A en un vecteur et en le déclarant par colonne de matrice.

Les lois

- Les lois de la *projection de cellules*

```
(1.1)       PAR (PAR i=0 FOR n (c[i] ? x[i]),
                 PAR i=0 FOR n (c[i] ! e[i]))
            -----------------------------------------
            PAR (SEQ i=0 FOR n (c[i] ? x[i]),
                 PAR i=0 FOR n (c[i] ! e[i]))

(1.2)       PAR (PAR i=0 FOR n (c[i] ? x[i]),
                 PAR i=0 FOR n (c[i] ! e[i]))
            -----------------------------------------
            PAR (SEQ i=0 FOR n (c[i] ? x[i]),
                 SEQ i=0 FOR n (c[i] ! e[i]))

(1.3)       PAR (PAR i=0 FOR n (c[i] ? x[i]),
                 SEQ i=0 FOR n (c[i] ! e[i]))
            -----------------------------------------
            PAR (SEQ i=0 FOR n (c[i] ? x[i]),
                 PAR i=0 FOR n (c[i] ! e[i]))

(1.4)       PAR (PAR i=0 FOR n (c[i] ? x[i]),
                 SEQ i=0 FOR n (c[i] ! e[i]))
            -----------------------------------------
            PAR (SEQ i=0 FOR n (c[i] ? x[i]),
                 SEQ i=0 FOR n (c[i] ! e[i]))
```

Les processus dans ces lois donnent le même résultat :

$\underline{x} := \underline{e}$

où $\underline{x}$ et $\underline{e}$ sont une liste de variables et d'expression respectivement, d'une longueur de n.

Ces quatre lois peuvent être démontrées par les lois <PAR-SEQ com*> par induction sur n. Nous en donnons la démonstration de (1.1). Celle des trois autres lois est similaire.

Démonstration de loi (1.1):

Dans le cas n = 0 et 1 : la formule peut être réécrite comme :

PAR (SKIP, SKIP) = PAR (SKIP, SKIP) et

PAR (c[0] ? x[0], c[0] ! e[0]) = PAR(c[0] ? x[0], c[0] ! e[0])

Trivial.

Supposons que la formule soit vraie pour n, nous avons alors pour n+1 :

PAR (PAR i=0 FOR n+1 (c[i] ? x[i]), PAR i=0 FOR n+1 (c[i] ! e[i]))

=> <replic unroll>

PAR (PAR i=0 FOR n (c[i] ? x[i]), c[n] ? x[n]),
PAR i=0 FOR n (c[i] ! e[i]), c[n] ! e[n]))

=> <PAR-PAR com1>

PAR (SEQ(PAR(PAR i=0 FOR n (c[i] ? x[i]),
PAR i=0 FOR n (c[i] ? e[i])),
PAR(c[n] ? x[n], c[n] ! e[n])))

=> (par l'hypothèse)

PAR (SEQ (PAR (SEQ i = 0 FOR n (c[i] ? x[i]),
PAR i=0 FOR n (c[i] ! e[i])),
PAR(c[n] ? x[n], c[n] ! e[n])))

=> <I/O simple> et <PAR-PAR com1>

PAR (SEQ i=0 FOR n+1 (c[i] ? x[i]),
PAR i=0 FOR n+1 (c[i] ! e[i]))

Fin de la démonstration.

- La loi de la *projection de flot*

```
PAR( PAR( SEQ i=0 FOR n (c[i] ? x[i], x[i] := fi(x[i]), c[i+1] ! x[i])),
      SEQ( c[0] ! e, c[n] ? z))
--------------------------------------------------------------------------------
PAR( SEQ( c[0] ? x, SEQ i=0 FOR n (x := fi(x))),
      SEQ( c[0] ! e, c[n] ? z))
```

Ces processus donnent le résultat :

z = f(e)

où *f* est la composition fonctionnelle de *f1, . . . fn.*

La démonstration de cette loi est la même que pour la loi de la section précédente. Il s'agit d'implantation distribuée de la synchronisation globale.

- La loi de la *réflection de flot*

```
PAR (PAR i= 0 FOR n (SEQ (c[i] ? x[i], x[i] := fi(x[i]), d[i] ! x[i])),
        PAR( PAR i=0 FOR n (c[i] ! xin[i]), PAR i=0 FOR n (d[i] ? xout[i])))
------------------------------------------------------------------------------------------
PAR (PAR(        SEQ i=0 FOR n (c[0] ! xin[i]),
                 SEQ i=0 FOR n (d[n] ? xout[i]))
      SEQ(
          SEQ j=i+1 FOR (n-i)+1 (c[i] ? xtmp[i], c[i+1] ! xtmp[i]),
          x[i] := fi(x[i])),
          SEQ j=0 FOR i ( c[i] ? xtmp[i], c[i+1] ! xtmp[i]),
          c[i+1] ! x[i])))
```

Ces processus donnent le même résultat :

SEQ i=0 FOR n (xout[i] := fi[xin[i])

- La justification de la *projection de canaux* est par cette loi :

```
PAR( SEQ i=0 FOR n (c[i] ? x[i]),
      SEQ i=0 FOR n (c[i] ! e[i]))
-------------------------------------------
PAR( SEQ i=0 FOR n (c ? x[i]),
      SEQ i=0 FOR n (c ! e[i]))
```

Ces processus donnent le même résultat: $\underline{x} := \underline{e}$, où $\underline{x}$ et $\underline{e}$ une longueur de n.

- La justification de la projection des éléments BElement[n] en BElement est par les lois de déclaration : <VAR rename>, <VAR-Op sym>.

L'application des lois

La procédure de la projection par l'application des lois ci-dessus au programme à deux dimension nous donne le programme équivalent à une dimension, dont le listage complet est dans l'annexe B2.

Nous appliquons les lois de la façon suivante :

Nous appliquons les lois (1.1)-(1.4) de la projection de cellules récursivement, à chaque colonne de cellules de droite à gauche. Par exemple, la substitution des constructions parallèles sur *row* par celles séquentielles (cf. Annexe B1, ligne 44 et Annexe B2, ligne 59), est le résultat de la transformation de (1.1) en (1.2);

Les flots de sortie sont transformés dans les colonnes par application des lois (1.2) et (1.3), et les flots d'entrée par (1.1) à (1.2). Cela remplace les constructions parallèles sur les rangs *row* par des constructions séquentielles.

Les transformations avancent comme suit :

Par application la loi de projection de flot, nous éliminons les lignes 32 et 36 dans le programme à deux dimension, en remplaçant le processus Q par le processus P. Les communications sur les canaux *Up* correspondent aux celles sur les canaux c[i], i = 0 ,..., n dans les lois. La transformation des variables x[i] dans P en une seule variable x dans Q, correspond au déplacement de déclaration de AElement[n] dans la construction parallèle sur *row* dans le programmes à deux dimension, au *col* dans le programme à une dimension. En fin, les différentes variables de AElement sont combinées en une vecteur de variables par la loi de <VAR rename> ;

En appliquant la loi de réflection de flot, nous entrelaçons les cellules de calcul avec l'acquisition et la restitution du flot A. Ainsi la réflection de canaux *Up* en canaux *Right*. La loi de projection de canaux est appliquée pour transformer les canaux de colonnes en un canal unique.

Après les transformations, le segment du calcul des éléments devient le suivant :

```
          -- CELLULES DE CALCUL
19        PAR col = 0 FOR n

20          VAR AElement[n], BElement :
21          SEQ

              -- entrée du flot A
22            SEQ row = 0 FOR n
23              SEQ

24                Right[col] ? AElement[row]

25                SEQ unused = 0 FOR (n-1)-col
26                  VAR tmp:
27                  SEQ
28                    Right[col] ? tmp
29                    Right[col+1] ! tmp

30            SEQ row = 0 FOR n
31              VAR CElement:
32              SEQ

                  -- entrée du flot C
33                Right[col] ? CElement

34                SEQ unused = 0 FOR (n-1)-col
35                  VAR tmp:
36                  SEQ
37                    Right[col] ? tmp
38                    Right[col+1] ! tmp

                  -- calcul des éléments de C
39                SEQ k = 0 FOR n
40                  SEQ

41                    Right[col] ? BElement

42                    CElement := CElement + AElement[k] * BElement

43                    Right[col+1] ? BElement

                  -- restitution du flot C
44                SEQ unused = 0 FOR col
45                  VAR tmp:
46                  SEQ
47                    Right[col] ? tmp
48                    Right[col+1] ! tmp

49                Right[col+1] ! CElement
```

```
                -- restitution du flot A
50              SEQ row = 0 FOR n
51                SEQ

52                  SEQ unused = 0 FOR col
53                    VAR tmp:
54                    SEQ
55                      Right[col] ? tmp
56                      Right[col+1] ! tmp

57                  Right[col+1] ! AElement[row]
```

Chapitre 5

Une extension de la sémantique algébrique

Pour l'application du système Oxford de transformation dans la programmation parallèle, dont les aspects techniques sont présentés dans les chapitres précédents, nous avons besoin d'une extension de la sémantique algébrique définie dans le chapitre 2. Cette extension concerne :

(1) De nouvelles lois pour l'extraction du parallélisme (§3 et ce chapitre), la distribution des structures de données et l'adaptation du degré de parallélisme (§4).

(2) Le développement et la définition d'une forme abstraite des programmes parallèles qui caractérise leur exécution sur une architecture parallèle, ainsi qu'une stratégie de transformation des programmes généraux en cette forme. Nous démontrerons qu'un programme sous cette forme abstraite facilite son placement sur une machine parallèle par son degré de parallélisme flexible et ses communications inter-processus faciles à adapter.

Notre approche de transformation peut être illustrée par la figure 5.1.

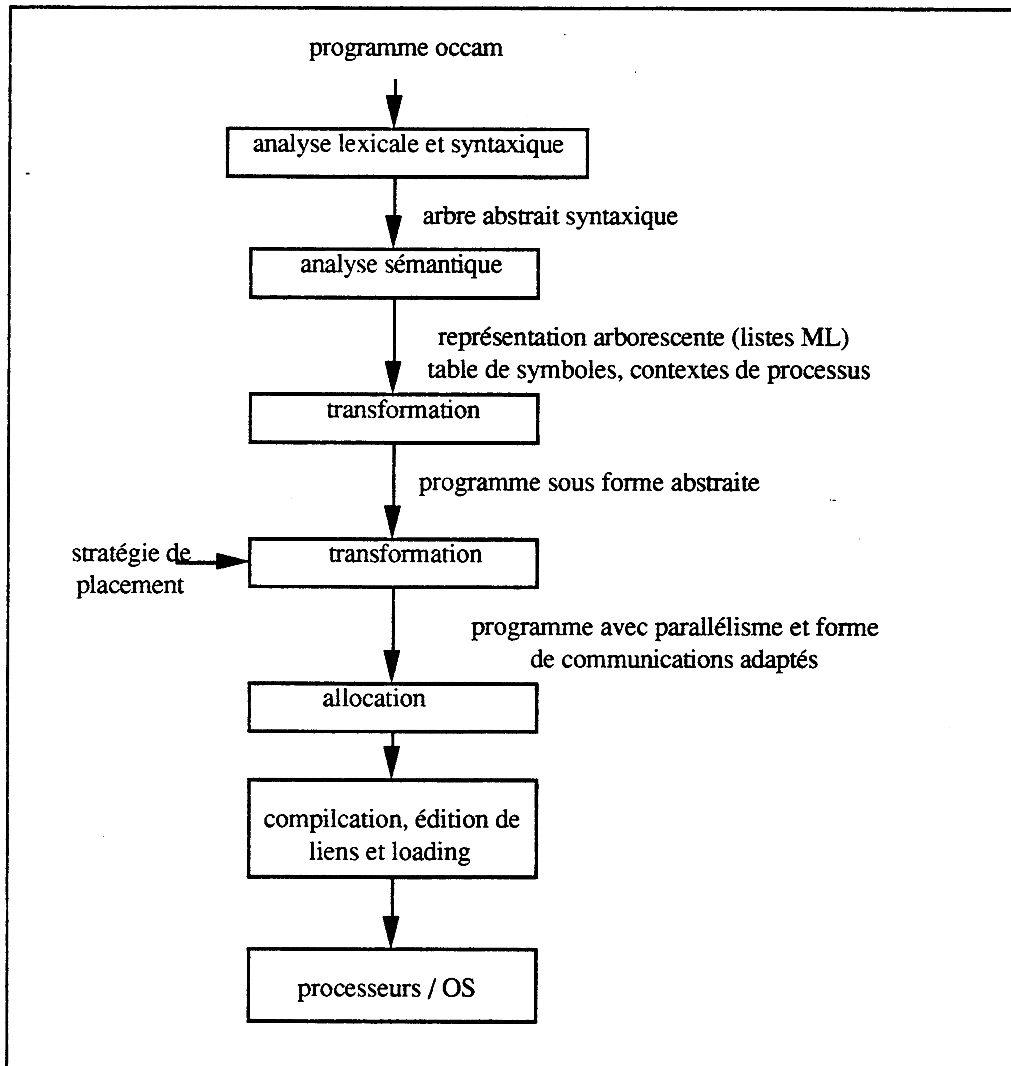

Figure 5.1 La transformation des programmes pour leur exécution parallèle.

5.1 Lois d'ajustement du parallélisme

Les lois dans cette section résument le résultat des techniques et méthodes d'analyse des programmes parallèles dans les chapitre 3 et 4. Elles expriment principalement les cas suivants : le changement de la forme de communications inter-processus, le changement et/ou l'extraction du degré de parallélisme, la distribution des structures de données, etc. En ce qui suit nous donnons quelques lois d'exemple, un listage des lois est dans l'annexe A2.

Le changement de la forme de communication

```
SEQ i=0 FOR N
    SEQ ( PAR(P1, P2),
        x := e1, y := e2
        Q[e1/x, e2/y])
-----------------------------------------------------------------        <PAR-SEQ pipe1>
PAR (   SEQ i = 0 FOR N SEQ (P1, c!e1),
        SEQ i = 0 FOR N SEQ (P2, d!e2),
        SEQ i = 0 FOR N SEQ (PAR(c?x, d?y), Q))
```

à condition que c ∉ librechan(P1), d ∉ librechan(P2).

Cette loi exprime l'effet de communications, et peut être démontrée en appliquant des lois <SEQ-PAR *> par induction sur i.

```
PAR( SEQ i=0 FOR N (c[i] ? x[i]),
     SEQ i=0 FOR N (c[i] ! e[i]))
-----------------------------------------        <PAR-SEQ assign2>
PAR( PAR i=0 FOR N (c[i] ? x[i]),
     PAR i=0 FOR N (c[i] ! e[i]))
```

Cette loi exprime la possibilité de mettre des communications séquentielles en parallèle.

Distribution des structures de données globales

La loi suivante exprime l'implantation distribuée des données globales :

```
VAR maître[N], esclave[N]:
WHILE TRUE
    SEQ
        PAR i = 0 FOR N esclave[i] := f(maître[i])
        PAR (entrée ? maître[0],
            PAR i = 0 FOR N-1 maître [i+1] := esclave[i]
            sortie ! esclave[N] )
-----------------------------------------------------------------        <SEQ-PAR distrib>
WHILE TRUE CHAN [N+1]:
    PAR i = 0 FOR N
        VAR d, r: SEQ ( c[i] ? d, r := f(d), c[i+1] ! r )
```

Extraction du parallélisme

Les lois suivantes permettent de déplacer l'opérateur PAR vers les constructions externes, donc sont très utiles pour augmenter le parallélisme.

SEQ(P,Q)
--------------- à condition que libre(P) $\cap$ libre(Q) = Ø. <PAR-SEQ paral1>
PAR(P, Q)

Cette loi doit être utilisée à ce que P et Q communiquent avec l'environnement, l'ordre de communications doit être maintenu pour qu'un blocage ne se produise pas. A remarquer cette loi représente le cas d'absence de la dépendance de données.

SEQ (P,Q)
------------------------------------ <PAR-SEQ paral2>
CHAN c:
PAR
VAR $\underline{x}$: SEQ(P, c!$\underline{x}$)
VAR $\underline{y}$: SEQ(c?$\underline{y}$, Q($\underline{y}$/$\underline{x}$)

A condition que le canal *c* ne soit pas libre dans P et Q, $\underline{y}$ ne soit pas libre dans Q ; $\underline{x}$ = final(P) $\cap$ init(Q).

Cette loi exprime un cas particulier de la parallélisation des processus séquentiels avec la dépendance de données. Selon les cas de dépendance de données entre les processus P et Q, il existe d'autres possibilités pour l'extraction du parallélisme (l'analyse approfondie des dépendances de données avec la technique des conditions de Bernstein étant hors de nos études, voir [CF89, Cytr89, Feaut91]).

Les lois suivantes permettent une mise en oeuvre distribuée des constructions séquentielles.

SEQ (P, Q)

CHAN C:
PAR
SEQ(P, C ! final(P)
VAR final(P) : SEQ (C ? final(P), Q(final(P) / init(Q))

à condition que final(P) $\cap$ init(Q) $\neq \emptyset$, C $\notin$ libre(P) $\cup$ libre(Q).

SEQ(P, PAR (Q, R), ...)

--- <SEQ-PAR conv2>

CHAN c, d:
PAR (SEQ (c ? init(Q), Q, d ! final(Q)),
SEQ (P, c ! init(Q), R, d ? final(Q)), ...)
où c,d $\notin$ libre(P) et libre(Q).

IF (b_0 P_0, ..., b_i (PAR(P,Q)), ...)

-- <IF-PAR conv>

CHAN c, d:
PAR (SEQ (c ? init(P), P, d ! final(P)),
IF (b_0 P_0, ...,
b_i (SEQ(c ! init(P), Q, d ? final(P)),...)
où c,d $\notin$ libre(P) et libre(Q).

ALT(g_0 G_0, ..., g_i (PAR(P,Q)), ...)

-- <ALT-PAR conv>

CHAN c, d:
PAR (SEQ (c ? init(P), P, d ! final(P)),
ALT(g_0 P_0, ...,
g_i (SEQ(c ! init(P), Q, d ? final(P)),...)
où c,d $\notin$ libre(P) et libre(Q).

La composition ou la décomposition des boucles

```
CHAN c:
PAR (WHILE TRUE SEQ (in?x, r := f1(x), c!r),
     WHILE TRUE SEQ (c?y, r' := f2(y), out!r'))
--------------------------------------------------------------------    <WHILE-PAR>
SEQ
    in?x
    WHILE TRUE
        SEQ
            PAR
                in ? y
                SEQ (r := f1(x), r' := f2(r), out!r')
            in ? x
            SEQ (r := f1(y), r' := f2(r), out!r')
```

Cette loi permet la fusion de deux boucles dans le cas où le pipeline de communications est la seule interaction entre ces deux boucles.

```
WHILE e (CHAN c:  PAR ( P, Q))
------------------------------------------    <WHILE-PAR conv>
CHAN c, init, data, bool:
PAR
    VAR libre(P) ∩ A, loop:
    SEQ
        init ? init(P) ∩ A
        bool ? loop
        WHILE loop SEQ ( P, data ! final(P) ∩ A, bool ? loop)
    SEQ
        init ! init(P) ∩ A
        bool ! e
        WHILE e
            SEQ ( Q, data ? final(P) ∩ A, bool ! e )
```

où A est l'ensemble des variables dans la condition *e*, les canaux *init, data, bool* $\notin$ libre(P) et libre(Q).

5.2 Formes abstraites des programmes parallèles

La forme normale définie dans le chapitre 2 joue un rôle de caractériser et représenter syntaxiquement la sémantique d'Occam. Mais un programme sous cette forme n'est guère adéquate pour une exécution sur une architecture distribuée, parce que tout parallélisme a été éliminé.

Selon le principe de la forme normale et le modèle d'exécution des programmes parallèles, nous essayons de trouver une forme syntaxique de programmes parallèles pour caractériser leur exécution. L'idée principale est de trouver des structures possibles de programmes, qui sont assez générales pour représenter des programmes pratiques, et en même temps suffisamment spécifique pour captiver leur modèle d'exécution parallèle.

Cela nous amène à chercher une approche systématique pour l'intégration de l'optimisation des programmes parallèles (notamment l'ajustement du parallélisme, §3) dans leur exécution efficace sur une machine spécifique (par estimation du temps d'exécution pour leur placement).

L'intérêt de la forme abstraite est qu'elle sert comme une forme intermédiaire dans notre approche systématique, pour la transition de programmes généraux à leur formes finales pour une implantation physique, en facilitant leur placement, donc leur allocation statique (Figure 5.2).

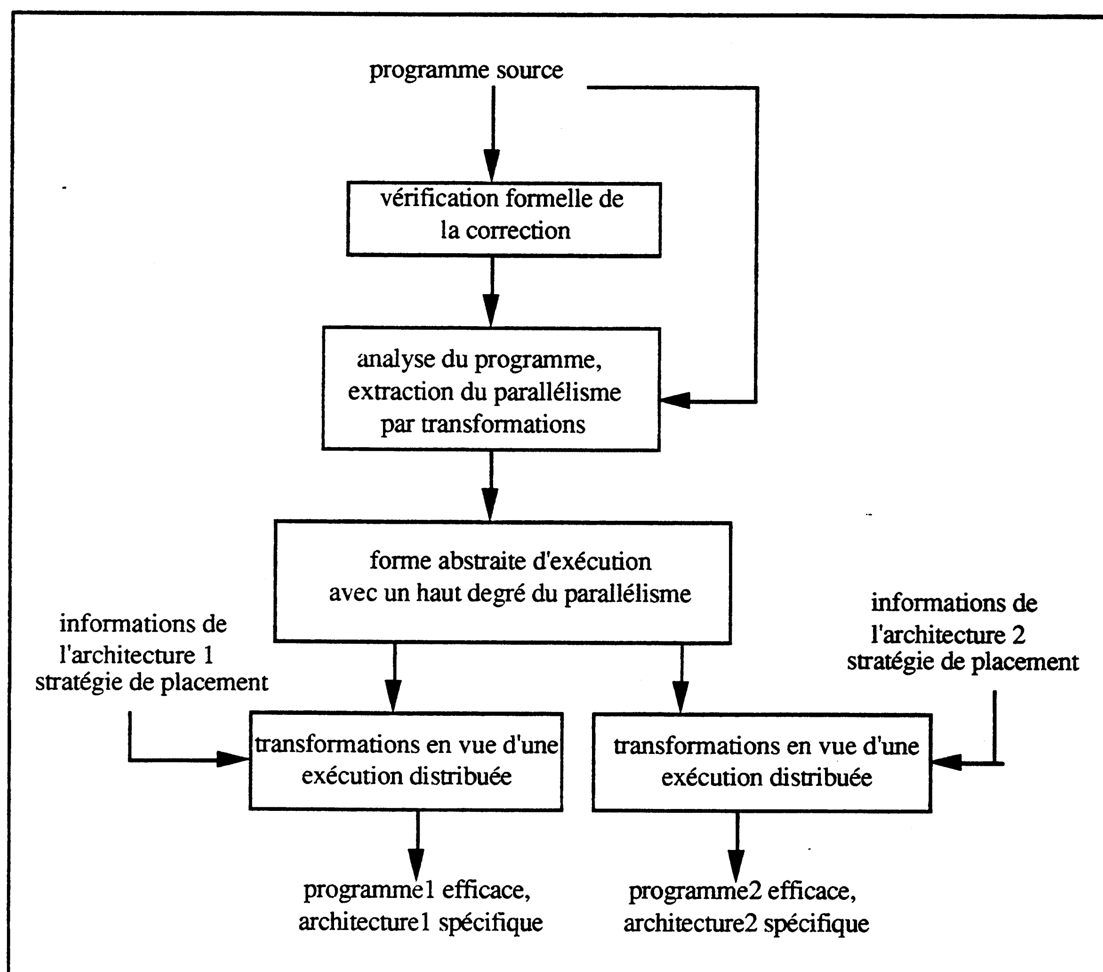

Figure 5.2 Forme abstraite des programmes parallèles pour leur exécution parallèle.

Mesure de temps d'exécution de programmes

Un programme peut avoir beaucoup de versions équivalentes dans le système de transformation, mais cela n'implique pas que leur temps d'exécution est le même. La mesure et l'estimation du temps d'exécution d'un programme parallèle est indispensable pour son placement (entre dans le critère de la distribution de la charge des processeurs). Elle est aussi un critère important pour comparer l'efficacité des formes de programmes pendant leur transformation.

Etant donné un programme P, nous notons Texe(P) son temps d'exécution.

Texe est composé de deux parties [TD89] : celui de calcul pur Tcalc(P), et celui de communications avec d'autre processus ou l'environnement Tcom(P).

Sur la machine virtuelle (Cf. §3.2), le coût de communication est le même pour n'importe paire de processeurs, et la connectivité de processeurs est complète. Tcom(P) peut donc être déterminé par la volume de messages échangés entre P et les processus partenaires ou l'environnement.

Définition des formes abstraites

Les formes abstraites sont développées d'après la classification et l'abstraction du parallélisme [Hey89], sur les aspects suivants : le contrôle de tâches, le flot de données, des caractéristiques des communications inter-processus, etc.

A part d'avoir un haut degré de parallélisme, les formes abstraites doivent posséder les caractéristiques suivantes : indépendantes de toute architecture spécifique ; représentatives des modèles généraux du calcul distribué.

Notons que nous permettons le placement des processus en parallèle à l'intérieur d'une construction séquentielle : dans ce cas, un programme a une forme suivante :

SEQ (PAR1, PAR2, ... PARn)

où la terminaison d'une étape PARi détermine un point d'allocation pour l'étape suivant PARi+1.

Le cas plus général est un programme avec une structure mixte de PAR et de SEQ, où un certain nombre de processus séquentiels déterminent des points d'allocation :

PAR
SEQ1 (PAR_{11}, PAR_{12}, ...)
SEQ2 (PAR_{21}, PAR_{22}, ...)
...
SEQn (PAR_{n1}, PAR_{n2}, ...)

Définition 5.1 : *Forme abstraite de treillis* définit un programme parallèle qui est sous la forme :

```
SEQ
 . U1 : PAR i = 0 FOR n1      -- commentaire : P1 désigne ce groupe de processus
        U1i : P1i
   ... d'autres Pj
   Um : PAR i = 0 FOR nm      -- Pm
        Umi : Pmi
```

Chaque processus composant P_{ij} d'une construction parallèle P_i peut contenir toutes les structures Occam. Le graphe d'un programme sous cette forme est le suivant, où la structure de treillis est définie par la relation structurale de contrôle père-fils entre les processus (Cf. la figure 5.3).

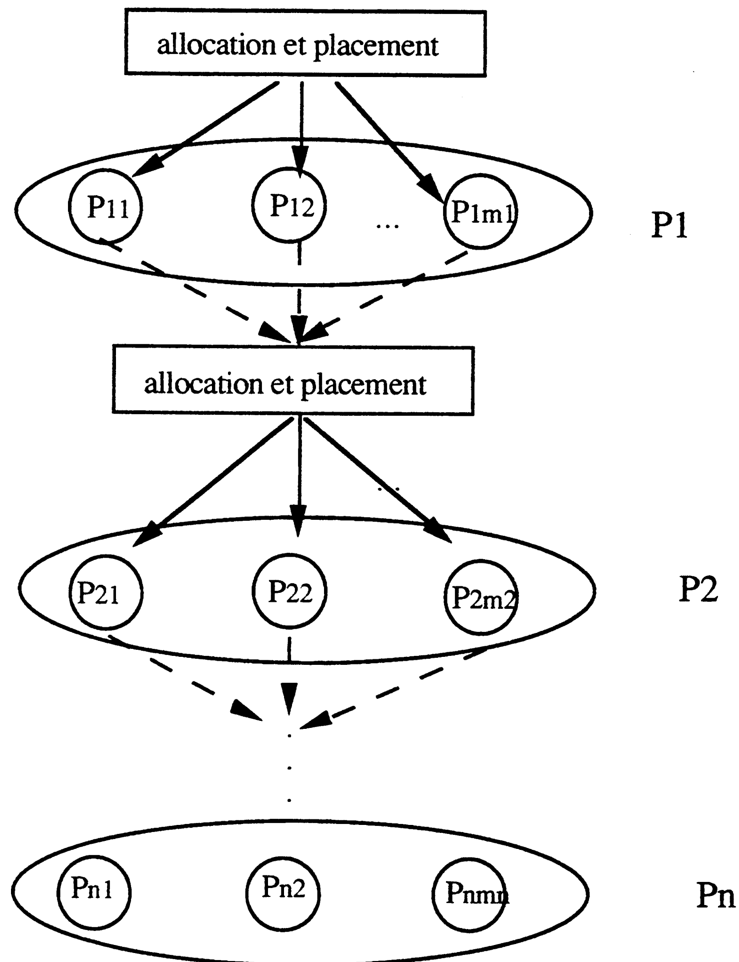

Figure 5.3 Graphe du processus de la forme de treillis.

L'estimation du temps de calcul d'un programmes sous cette forme peut se faire comme suit : où P = SEQ (PAR P_{ij}), i = 1, ..., n; j = 1,..., n_j

$$Texe(P) = \sum Texe(P_i) = \sum Tcalc(Pi) + Tcom(Pi)\ i=1, \ldots, m$$

$$Tcalc(P_i) = \max(Tcalc(P_{ij})), j = 1, \ldots n_j,$$

Tcom(Pi) = Volume de messages échangés entre Pij dans Pij', j ≠ j'.

Nous pouvons remarquer cette forme est adéquate pour des machines reconfigurables, comme Supernode [Hey89], où l'interconnexion des processeurs peuvent être adaptée ou pendant l'exécution d'un programme. Dans un placement pratique, cette forme de programme facilite l'allocation dynamique. Les différentes étapes de constructions parallèles correspondent aux différentes phases de l'allocation dynamique. Entre deux étapes, c'est le point de la reconfiguration.

Cas spéciaux de la forme générale

• Forme abstraite séquentielle. Dans le cas extrême, un programme sous la forme abstraite est une forme dégénérée, où chaque processus composant Pi est composé des construction séquentielles (des structures de IF, ALT, SEQ et WHILE) :

SEQ (P1, ..., Pn) où chaque processus Pi est séquentiel

Cette forme représente la version classique et séquentielle du programme. Sans quelques transformations de programmes telles que le parallélisme peut être introduit, cette forme de programme ne peut guère être adaptable pour exploiter le parallélisme physique. Pourtant, cette forme est appropriée pour s'exécuter sur un seul processeur, ou comme un processus composant dans une construction parallèle.

Pour cette forme de programme, son temps d'exécution est :

$$Texe(P) = Tcalc(P) = \sum Tcalc(Pi), i = 1, \ldots, n$$

puisque il n'existe pas de communications.

• La forme parallèle suivante, qui possède seulement une étape de construction parallèle.

```
Déclarations de variables :
SEQ
  "Initialisation"
  PAR (Ui : Pi)
  "Post-traitement"
```

Cette forme est le cas général dans des études du placement, où un programme parallèle est considéré composé d'un ensemble de processus composants en parallèle.

Cette forme parallèle a deux cas spéciaux suivants qui sont intéressantes, selon la forme de communication inter-processus.

• *Forme abstraite d'arbre* a la forme ci-dessus, mais avec la restriction que les processus composants en parallèle ne communiquent pas entre eux.

Cette forme de programme représente un algorithme, où un processus hôte distribue des tâches aux processus esclaves, et ceux-ci fonctionnent indépendamment. Le temps du calcul pour cette forme de programme peut être estimé par :

Texe(P) = max(Tcalc(Pi)) + Tcom(P), i = 1, . . . n.
Tcom(P) = volume de messages échangés avec l'environnement

• *Forme abstraite linéaire* est définie comme la forme abstraite parallèle, avec la restriction : un processus composant parallèle P_i ne peut communiquer qu'avec ses voisins $P_{i-1(\text{mod } n)}$ et $P_{i+1(\text{mod } n)}$.

La performance de cette forme de programmes est limitée par le processus le moins rapide de la structure linéaire.

Théorème 5.1 : chaque programme Occam P peut être transformé en un autre programme équivalent P' sous forme abstraite, et P = P', libre(P) $\subseteq$ libre(P') peut être démontré dans le système de transformation.

La démonstration de ce théorème est basée sur le résultat de la détection et de l'extraction du parallélisme dans le chapitre 3. Le point essentiel est la procédure suivante, et son application récursive aux structures possibles de programmes par induction structurale :

(1) D'abords nous transformons le programme P en sa forme canonique, afin de calculer son espace d'états; Ensuite nous appliquons la procédure de l'extraction du parallélisme dans la section §3.2 sur la forme canonique. Après quoi nous obtenons un programme équivalent P' avec un haut degré de parallélisme, dont les constructions externes sont des processus communicants en parallèle. A remarquer que le programme P' est en fait sous forme abstraite de treillis, avec une seule construction parallèle :

SEQ ("initialisation", PAR(U_i : P_i), "post-traitement")

(2) Ensuite, il suffit de démontrer que cette forme abstraite de treillis peut être transformée en d'autres formes abstraites :

- Pour la forme abstraite séquentielle, la procédure de transformation est similaire à celle qui transforme les programmes finis (sans boucles) en leur forme normale (§2.2). Puisque chaque pas de transformations diminue le nombre de constructions parallèles, donc des communications, la procédure se termine quand il n'existe plus de structures parallèles;

- Pour la forme abstraite d'arbre, il suffit de regrouper les processus qui communiquent en un, pour qu'il n'existe pas de communications entre les processus composants en parallèle.

- Pour la forme abstraite linéaire, nous pouvons appliquer l'algorithme suivant :

D'abords nous construisons le graphe du programme, où les noeuds sont les processus composants P_i et la connectivité du graphe est la communication entre les processus. Nous pouvons toujours sélectionnons une racine (qui n'a pas d'arête d'entrée, ou n'importe quel noeud dans le cas contraire) ; puis nous effectuons un tri topologique sur le graphe, ce qui est toujours possible. En regroupant les processus dans le même groupe résultat du tri, nous obtenons le résultat : un programme sous forme linéaire où une partition du graphe ne communique qu'avec ses voisins.

5.3 Un exemple de l'augmentation du parallélisme

Définition du problème

Le programme suivant accomplit la multiplication de matrices A et B à dimension n x n :

$S = A \times B$, $s_{ij} = SUM(a_{ik} * b_{kj})$; $i = 1... n$; $j = 1... n$; $k = 1... n$

Synthèse du programme

Un processus systématique de synthèse est présenté dans [GL89], l'idée principale est la suivante :

- effectuer une analyse des flot de données pour générer un graphe du programme (figure 5.4) qui contient :

(1) Une structure arborescente qui correspond au flot de contrôle.

(2) Les informations de flots de données.

Dans le graphe, les flèches en trait plein représentent la relation de contrôle entre les processus, les lignes pointillées représentent la relation de dépendance de données. Par exemple, le noeud "Plus" est contrôlé par la boucle "forall", et dépend des noeuds "Element" dans la relation de données.

- Ensuite, nous traduisons ce graphe en un programme parallèle. Les règles de traduction du graphe de flot de données sont les suivantes : pour chaque noeud, nous créons un processus parallèle, et la relation du processus avec les autres est déterminée par le graphe arborescent. Par exemple, un noeud qui accomplit la multiplication peut être transformé en Occam comme suit :

```
VAR x, y:
SEQ
  PAR
    C1 ? x
    C2 ? y
  C3 ! x * y
```

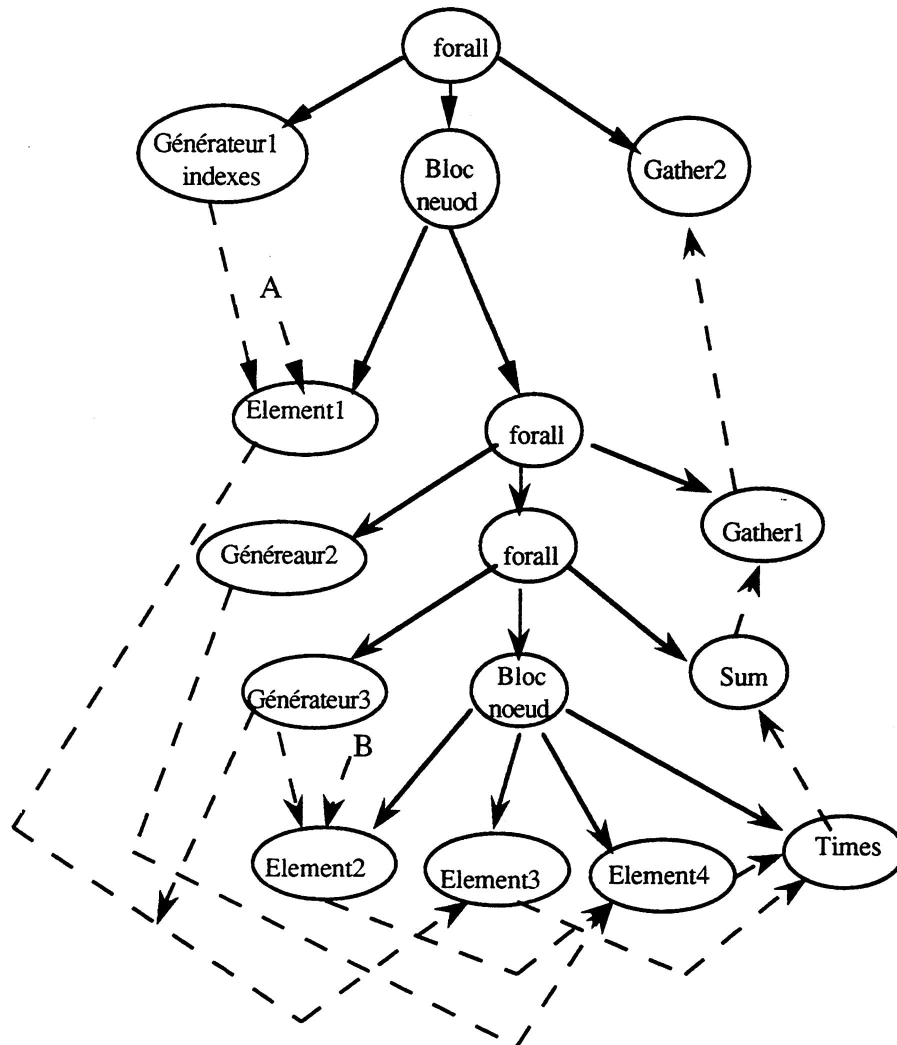

Figure 5.4 La composition de matrices par un programme de flot de données

D'après de ce graphe, nous obtenons le programme complet dans l'annexe C1. Par exemple, le noeud "Sum" est réalisé par le processus P9 suivant : il reçoit des valeurs calculées par le noeud "Times" à travers du calcul C8, calcule les éléments de la matrice C, et les transfert au noeud "Gather1" à travers du calcul C9.

```
-- P9 : Sum : faire la somme pour éléments de  C
VAR CElement, tmp:
SEQ  i = 0 FOR n
   SEQ j = 0 FOR n
      SEQ
         SEQ k = 0 FOR n
            SEQ
               C8 ? CElement
               tmp := tmp + CElement
         C9 ! tmp
```

Nous pouvons remarquer facilement que dans le programme C1 de style original, les processus en parallèle sont tous séquentialisés par des communications entre eux; les déclarations de matrices A, B et C en début du programme rendent difficile pour une implantation distribuée, à cause des structures de données globales : AElement[n,n], BElement[n,n] et CElement[n,n].

Normalisation de l'utilisation des variables et des canaux

Cette étape a pour but de localiser les variables globales. Nous localisons les déclarations de matrices dans les processus qui les utilisent respectivement, c'est-à-dire, celle de A est déplacée au processus P4, celle de B au processus P5 et celle de C au processus P11 (Gather2 qui collectionne les rangs de C) ; et nous ajoutons une déclaration :

VAR CElement[n]:

au processus P10 (Gather1 qui collectionne les éléments d'un rang de C) en tant que variables locales. Après cette étape, nous avons éliminé les structures de données globales. Ces transformations peuvent être justifiées facilement par des lois de la portée de déclaration. Des transformations similaires pour les déclarations de canaux peuvent être effectuées pour une implantation distribuée.

Transformation des boucles statiques en processus séquentiels

Tous les processus sous formes de constructions de WHILE peuvent être transformés en constructions de réplicateurs de SEQ, grâce au nombre statique d'itérations des boucles. Pour cette transformation, nous avons besoin de la loi suivante (avec l'application dans un seul sens, c'est-à-dire, de gauche à droite) :

```
VAR loop, c:
SEQ (    loop := 0, c := N,
         WHILE (loop < c) SEQ (P, loop := loop +1))
-------------------------------------------------------------        <WHILE-SEQ conv>
SEQ loop = 0 FOR N
   P
```

à condition que la variable *loop* n'est pas changée dans P.

Cette loi peut être démontrée en déroulant N fois la construction de WHILE. Cela transforme toute construction de WHILE en une construction séquentielle. C'est-à-dire, nous transformons un programme sous forme suivante :

```
VAR loop, limit:
SEQ
   limit := n
   loop := 0
   WHILE loop < limit
      SEQ
         C ? CElement[loop]
         loop := loop +1
```

en :

```
SEQ loop = 0 FOR n
   C ? CElement[loop]
```

Augmentation du parallélisme

Finalement, nous appliquons la stratégie de l'extraction du parallélisme, qui permettent de transformer les constructions séquentielles en celles de PAR, en introduisant des canaux et des variables locales. Cela permet de transformer des communications séquentielles en parallèles. Ces lois concernées sont sous forme suivante :

```
PAR(SEQ i=0 FOR n SEQ(P, PAR(c1 ! e1, c2 ! e2)),
      SEQ i=0 FOR n SEQ(PAR (c1 ? x, c2 ? y), d ! f(x,y)),
      SEQ i=0 FOR n VAR r: d ? r)
--------------------------------------------------------------------------------
CHAN c1[n], c2[n], d[n] :
PAR
      PAR i=0 FOR n (VAR e1, e2 : SEQ(P, PAR(c1[i] ! e1, c2[i] ! e2))),
      PAR i=0 FOR n
            VAR x,y: SEQ (PAR (c1[i] ? x, c2[i] ? y), d[i] ! f(x,y)),
      PAR i=0 FOR n VAR r: d[i] ? r
                                                                <pipe 1>
```

La démonstration de cette loi peut être effectuée par l'application de lois <PAR-SEQ com*> et le renommage nécessaire des variables concernées.

Par application de cette loi, nous obtenons le degré renforcé de parallélisme du programme, au prix d'une augmentation des communications et donc du nombre de canaux. Le programme résultat est donnés dans l'annexe C2. Il a un degré n^2 de parallélisme. C'est un bon résultat car le programme original composé de constructions séquentielles et de boucles WHILE paraîssait difficile à paralléliser.

Par exemple, le processus P9 (neoud Sum) ci-dessus est transformé en :

```
-- P6 : Sum : faire la somme pour éléments de  C

PAR  i = 0 FOR n
   PAR j = 0 FOR n
      VAR tmp:
      SEQ
         SEQ k = 0 FOR n
            VAR CElement:
            SEQ
               C4[i, j] ? CElement
               tmp := tmp + CElement
         C5[i, j] ! tmp
```

<u>**Placement des programmes sous forme abstraite**</u>

Il est connu que la difficulté de placer les processus du programme d'une façon optimale sur les processeurs, à cause de la complexité de l'inter-connexion de processus et de processeurs (§4). La forme abstraite rend le placement facile grâce à : leur structure de communication régulière, un haut degré de parallélisme, et des informations disponibles des processus composants (sous forme d'annotations dans une construction parallèle Ui : Pi) pour l'estimation de leur temps d'exécution.

Prenons le programme de la composition de matrices dans l'annexe C2, pour illustrer la réalisation d'une stratégie de placement d'un programme sous forme abstraite. La version du programme avec parallélisation complète est donnée dans l'annexe C2. Nous pouvons remarquer que le degré de parallélisme de cette version résultat de la parallélisation complète est n^2 (donné par la structure suivante).

```
-- P6 : Reduce : faire la somme pour éléments de  C

PAR  i = 0 FOR n
   PAR j = 0 FOR n
      VAR tmp:
      SEQ
         SEQ k = 0 FOR n
            VAR CElement:
            SEQ
               C4[i, j] ? CElement
               tmp := tmp + CElement
         C5[i, j] ! tmp
```

La réalisation d'une stratégie de placement selon les cas différents peut être la suivante :

- Dans le cas général où le nombre de processeurs N est inférieur au degré de parallélisme du programme, une stratégie de placement consiste à répartitionner les processus en parallèle en N groupes. Supposons que l'architecture cible soit composée de 4 processeurs avec une connectivité complète. Dans ce cas, une version du programme, avec un degré de parallélisme égal à 4, est suffisante pour exploiter ce parallélisme physique. Nous transformons donc le programme C2 en un programme qui sera composé de quatre processus parallèles. Une stratégie simple et naturelle est de regrouper en quatre les huit processus en parallèle du programme, avec leur poids homogènes pour obtenir un équilibre de charge des processeurs:

Remarquons que dans cette version du programme, le poids de chaque processus peut être estimé par le nombre de processus composants. Nous pouvons donc combiner les processus P1, P2, P3, P4 en une partition P1', les processus P5, P6 forment deux partitions P3' et P4' respectivement, et les processus P7, P8 en une partition P4'. Après ce regroupement, nous éliminons le parallélisme dans chaque partition de processus. Ceci consiste à transformer toutes les constructions parallèles en celles séquentielles. Par conséquent, nous éliminons les canaux inutiles, c'est-à-dire, seuls les canaux C0, C1, C2, C3 sont suffisants pour les communications entre ces quatre partitions.

- Si nous avons une machine composée d'un nombre N de processeurs, où N est comparable au degré de parallélisme du programme n^2. Dans ce cas, tous les processus en parallèle dans les constructions externes peuvent être placés sur des processeurs distincts.

Chapitre 6

Conclusions et perspectives

6.1 Conclusions

Il existe une demande croissante pour des logiciels sûres, sécures, et performants. Cette demande parvient des considérations économiques, d'exigences de sécurité (pour des applications critiques). L'approche transformationnelle peut certainement offrir une contribution importante vers ce but. Des méthodes formelles, intégrées dans la programmation transformationnelle, aident contre la tendance des programmeurs qui souvent sous-estiment la complexité du problème donné. En plus, cette approche a d'autres avantages : parce qu'une version initiale du programme, qui est normalement machine-indépendante et compréhensible, peut être adaptée pour exécuter, non seulement sur des machines classiques, mais aussi sur des architectures nouvelles (comme processeurs parallèles).

Parmi les approches courantes à la construction de logiciels formellement vérifiés, l'approche transformationnelle est certainement la plus avancée et la plus flexible. Elle couvre déjà plusieurs phases de la cycle de vie de génie logiciel. Elle est prometteuse de devenir un sujet standard en informatique, analogue à la méthode déductive en Mathématiques. L'impact de l'approche transformationnelle sur la conception et le développement logiciel peut être exprimé par sa comparaison avec les méthodes classiques (Figure 6.1).

Les systèmes de transformation existants sont encore expérimentaux et des problèmes qu'ils sont capable de résoudre sont encore plus ou moins "problèmes de jouet" [PS83]. Pourtant, des recherches sont en route pour vérifier la faisabilité de la méthodologie sur des problèmes pratiques, survenus de beaucoup de domaines d'application.

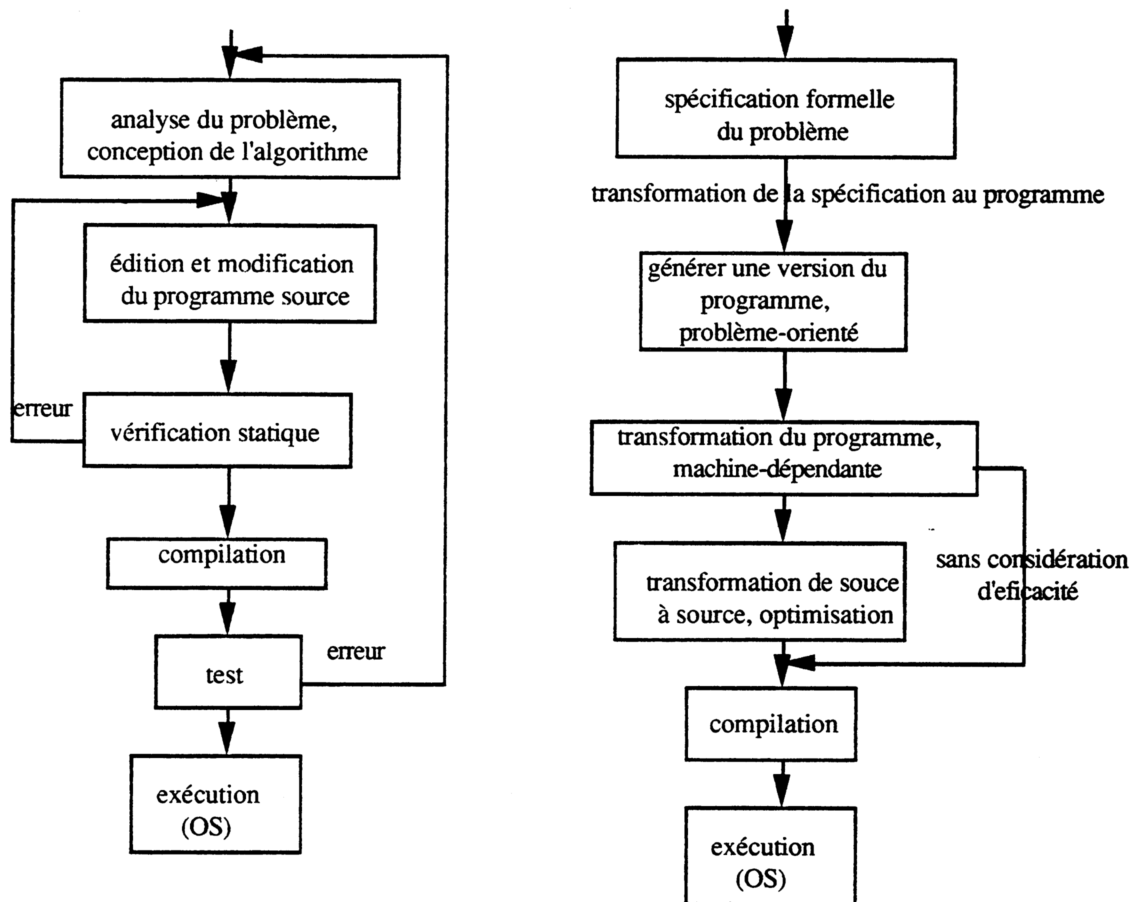

Figure 6.1 L'impact de l'approche transformationnelle sur la conception logiciel.
(a) La méthode classique. (b) La programmation transformationnelle

Nous constatons que la tendance des systèmes de transformation est d'éloigner des systèmes généraux et vers des sous-systèmes individuels, problème orienté, basés sur un petit ensemble de règles puissantes. Pour être maniable, la collection de règles de transformation doit être spécifique à des domaines particuliers. Il est seulement par cette restriction qu'un système puisse être suffisamment puissant pour devenir un outil pratique.

Ceci est aussi vrai pour notre système de transformation, où une solution générale à la programmation des architectures parallèles n'est pas envisageable, à cause des particularités du domaine des problèmes. Nous nous sommes accentués sur des problèmes spécifiques de la programmation parallèles des architectures parallèles. Nous avons pu traiter d'une façon systématique par la transformation, l'adaptation du parallélisme des programmes à celui de la machine physique, l'implantation distribuée de structures de données, etc.

Réalisation

Un cadre naturel pour la mise en oeuvre de notre système de transformation est le système de transformation Oxford, où les types de données et les fonctions ont été réalisés pour incorporer nos règles et stratégies de transformation. L'interface interactif utilisateur du système permet la vérification d'une nouvelle loi, l'automatisation de l'application d'une loi sur un segment de programme, et la visualisation du processus de transformation. Des exemples de l'utilisation du système pour la transformtion sont donnés dans l'annexe D.

Jusqu'à maintenant, sauf pour des stratégies de transformation spécifiques (dont leur application est automatisée), aucune facilité intelligente est offerte aux utilisateurs pour le choix des règles et la décision de leur application, en fonction du but de transformation et la forme du programme en question. Il devrait nécessaire de faire un effort sur cet aspect.

Un autre cadre naturel, plus important pour l'application pratique, est l'intégration de ces méthodes et techniques de transformation dans un compilateur d'optimisation pour les machines parallèles. Cela permettrait la portabilité des programmes pour différentes configurations des machines distribuées. Ceci est d'autant plus nécessaire et urgent, dû au manque de la facilité de l'adaptation du parallèlisme des programmes, en fonction de celui de la machine cible, dans les compilateurs existants et commercialisés.

6.2 Perspectives

Nous voudrions indiquer les aspects de recherche à poursuivre en vue de rendre le système plus puissant, spécifique et application-orientée. Il s'agit d'enrichir l'ensemble de règles de transformation par des lois de haut niveau, et de développer des stratégies problème-spécifiques.

Règles de transformation et stratégies orientées-application puissantes et spécifiques

Bien que l'ensemble des lois élémentaires est suffisant théoriquement pour des transformations générales, leur utilisation est fasditieuse, du au fait que ces lois expriment des propriétés locales d'un segment de programme. Il est nécessaire de développer et de découvrir un ensemble de lois spécifique à un domaine particulier d'application. Les lois concernées expriment la connaissance du domaine, et faciles à prouver à partir des lois élémentaires. Leur utilisation sera beaucoup plus facile et compréhensible, sur un niveau plus haut de la syntaxe. Ce type de règles de transformation est illustré dans l'exemple de l'augmentation du parallélisme (Cf. §5.3), l'implantation distribuée des structures de données (Cf. §4.2.2).

Une règle de transformation peut être appliquée dans les deux sens. Cela produit différents effets, par exemple, une réduction ou un renforcement du parallélisme, une structure de données plus ou moins distribuée. Donc des stratégies apropriées pour des applications, qui ont le même type de transformations, sont nécessaires pour devenir une méthode systématique, en vue d'automatiser des transformations.

temps réel

Il manque de traitement de la conception du temps dans le modèle de sémantique sur lequel l'ensemble des lois est dérivé. Il serait intéressant et pratique de développer des lois qui traitent le temps (comme l'opérateur de TIMER en Occam), pour des applications temps-critiques, responsives. Cela consiste à extraire et à découvrir l'équivalence des structures de programmes parallèles, qui en même temps satisfont une exigence du comportement temporel. Ce genre de transformations peut être appliqué à la vérification d'un systèmes de protocole de communications, par exemple.

Synthèses automatiques des programmes parallèles

Pour la construction correcte et automatique des programmes, de nombreux systèmes de synthèse de programmes ont été proposés [MW79]. L'objectif principal d'un système de synthèse automatique est d'offrir un environnement interactif, qui donne des techniques et méthodes formelles, pour automatiser une partie ou une phase de la dérivation et du développement d'un programme à partir de sa spécification. Le programme résultat est garantit de satisfaire sa spécification formelle.

Un composant essentiel d'un tel système est sans doute un mécanisme de transformation, qui est chargé de : (1) la dérivation et la vérification d'une version du programme correcte, problème-orienté, compréhensible, à partir de sa spécification formelle ; (2) l'adaptation du programme (source-à-source ou source-à-code exécutable) à un environnement spécifique. Un caractéristique de ce processus est une série de transformations successives et vérifiées.

- **Spécification formelle**

Dans un environnement interactif de développement de programmes, la spécification d'un problème est exprimée en terme du comportement d'entrée/sortie, des contraintes de temps et d'espace du calcul, d'exigence de performance. Elle est utile pour la séparation de l'algorithme du problème et son réalisation en un langage spécifique ou une architecture cible.

Une spécification est souvent exprimée par des notations mathématiques (algébriques, logiques ...), ce qui donne un cadre strict et formel pour d'une part, spécifier correctement et précisément les structures de données et les opérations associées, d'autre part, permettre une dérivation de programmes avec les règles de déduction (le raffinement par des transformations successives).

- **Dérivation formelle du programme**

Les approches et techniques existantes pour la dérivation formelle des programmes, varient des méthodes classiques pour les programmes séquentiels (principalement fonctionnels [MW79, Chea81, BD77, BGW76]), à la traduction automatique des programmes parallèls (par exemple, de style de flots de données, réseaux systoliques [LS89, GL89]) à partir d'une programme fonctionnel :

• Approche classique et conventionnelle de la dérivation des algorithmes dans un environnement du langage séquentiel ou fonctionnel.

Des analyses et transformations doivent être effectuées pour la détection et l'extraction du parallélisme. Un programme dérivé doit être repartié en fonction du parallélisme algorithmique ou géométrique (structure de données). Les formes abstraites des programmes, qui sont définies dans la section §5.2, jouent un rôle de représentations intermédiaires pour la transition d'un programme général à une forme adaptable pour des architectures distribuées.

• Algorithmes pour des architectures parallèles.

- La conception et la synthèse automatique des réseaux systoliques. Cette approche est très prometteuse car il existe des méthodes formelles dans les domaines spécifiques d'application, et les programmes résultats sont connus d'avoir un modèle de calcul efficace.

- La méthode de flots de données. Dans cette approche, par analyse de flots de données d'un programme fonctionnel, le parallélisme potentiel dans les opérations de donnée pourrait être révélé. Cela permet d'obtenir un certain degré raisonnable de parallélisme, comme illustré dans l'exemple de la composition de matrices dans la section §5.3. Cette méthode est une compromise entre les deux cas ci-dessus, générale pour la description des problèmes pratiques, et en même temps permet d'obtenir un haut degré de parallélisme.

Limitations de l'approche transformationnelle

A la fin, nous dégageons des limitations de l'approche de transformation:

- L'approche est basée sur l'analyse statique des programmes, donc hérite des limitations de celle-ci : elle ne convient pas bien pour la manipulation du comportement dynamique d'un programme. Par exemple, dans une transformation en vue de l'optimisation des programmes, il manque de traitement des structures dynamiques et de l'aspect dépendant des E/S.

- Domaines limités d'application. Ceci n'est pas vraiment une limitation de l'approche, mais plutôt l'état actuel de l'application des systèmes existants de transformation.

Annexe A1 Lois élémentaires

Lois de réplicateurs FOR, de PROC et de FONCTION

Les règles de déroulement des constructions répliquées <FOR> et l'appel de la procédure <PROC> sont données par les lois de transformations suivantes.

SEQ n=B FOR 0 = SKIP <SEQ-replic SKIP>

IF n=B FOR 0 = STOP <IF-replic STOP>

PAR n=B FOR 0 = SKIP <PAR-replic SKIP>

ALT n=B FOR 0 = STOP <ALT-replic SKIP>

Dans les deux lois suivantes, Op représente un des opérateurs SEQ, IF, PAR et ALT.

Op n=b FOR c P(n) = ZERO, pourvu que c < 0. <replic zero>

Op n=b FOR c P(n) = Op (P(b), P(b+1), . . . , P(b+c-1)), <replic unroll>
pourvu que c > O.

Supposons qu'il existe une définition de procédure où *fi* est une liste de paramètres formels :

PROC nom_de_procédure (f1, f2, ..., fn)
corps -- de la procédure

alors, un appel de cette procédure avec une liste d'arguments*ai* sous forme :

nom_de_procédure (a1, a2, ..., an) =
corps ([a1/f1], [a2/f2], ..., [an/fn]) <PROC unroll>
à condition que les types et le nombre d'arguments formels et actuels soient cohérents.

Lois de IF

IF ( ) = STOP <IF-STOP)
Une structure conditionnelle vide de IF se comporte comme STOP.

IF (C1, IF(C2), C3)
-------------------------- <IF-assoc>
IF (C1, C2, C3)

Cette loi permet la suppréssion de l'emboîtement de IF.

IF b_i P_i
---------- i=1...n où $b_i^* = \neg b_1 \wedge . . . \wedge \neg b_{i-1} \wedge b_i$ <IF priori>
IF b_i^* P_i

Cette loi exprime le fait que dans le processus IF b_i P_i, i=1...n, c'est la première garde booléenne évaluée à vrai qui active le processus correspondant. Donc un processus composant Pi s'exécute seulement si bi est vrai et $b_1, \ldots, b_{i-1}$ sont faux.

$$\frac{\text{IF } b_i\ P_i}{\text{IF } b_{\pi(i)}\ P_{\pi(i)}}$$ <IF sym>

i=1...n où π est une permutation de $\{1,...,n\}$,
à condition que $b_i \wedge b_j = \text{FALSE}$ pour $i \neq j$.
Si les gardes booléennes dans IF b_i P_i sont disjointes, alors l'ordre de la composition na pas d'importance.

$$\frac{\text{IF } (b_1\ P,\ b_2\ P,\ \underline{C})}{\text{IF } (b_1 \vee b_2\ P,\ \underline{C})}$$ <IF-v distrib>

Si deux booléens gardent le même processus successifs, alors ils peuvent être combinés en une seule condition.

$$\frac{\text{IF (FALSE } P,\ \underline{C})}{\text{IF } (\underline{C})}$$ <IF-FALSE unit>

Une garde FALSE ne peut jamais être activée, donc elle peut être supprimée.

$$\frac{*\ \text{IF } (\underline{C},\ b\ \text{STOP})}{\text{IF } (\underline{C})}$$ <IF-STOP unit>

Si aucune condition dans la structure de IF n'est vraie, alors le processus se comporte comme STOP. Donc on peut ajouter ou supprimer librement le processus conditionnel <b STOP> à la fin d'une construction de IF.

IF (TRUE P) = P <IF-TRUE unit>

$$\frac{\text{IF}(\underline{C},\ b\ \text{IF } b_i\ P_i)}{\text{IF } (\underline{C},\ \text{IF } b \wedge b_i\ P_i)} \quad i=1\ldots n$$ <IF-^ distrib1>

Cette loi avec <IF-assoc> permet de supprimer complètement l'emboîtement dans les structures de IF.

$$\frac{\text{IF}(\underline{C1},\ b\ \text{IF } b_i\ P_i,\ \underline{C2})}{\text{IF } (\underline{C1},\ \text{IF } b \wedge b_i\ P_i,\ \underline{C2})} \quad i=1\ldots n$$ <IF-^ distrib2>

à condition que $\vee\, b_i = \text{TRUE}$, i = 1 ... n.

Cette une loi généralise <IF-^ distrib1>. La condition $\vee\, b_i = \text{TRUE}$, i = 1 ... n garantit le processus IF $b_i \wedge b_i$ P_i , i = 1, ... , n ne se comporte pas comme STOP, dans le cas où b = FALSE et aucun b_i n'évalue à TRUE.

IF b_i P_i

------------ i=1...n <IF-ALT permut>

ALT b_i P_i

à condition que $b_i \wedge b_j$ = FALSE pour $i \neq j$.
Si les conditions dans les structures de IF sont disjointes, alors l'ordre des branches de IF importe peu. Cette loi est équivalent à <IF sym>.

IF b_i P_i

--------- i=1...n <IF-TRUE unit2>

P_k

à condition que b_k = TRUE et b_i = FALSE, $i < k$.
Cette loi est déduite de <IF-FALSE unit> et <IF-TRUE unit>.

Lois de CASE

CASE ( ) = STOP <CASE-STOP unit>

S'il n'y a pas d'arguments, la construction de CASE est équivalent à STOP.

CASE (CASE ($\underline{S1}$), $\underline{S2}$)

------------------------------ <CASE assoc>

CASE ($\underline{S1}$, $\underline{S2}$)

cette loi permet de supprimer les emboîtements de CASE.

CASE $\underline{Si}$

----------- <CASE sym>

CASE $\underline{S\pi(i)}$

où π est une permutation de {1 ... n}.
L'ordre des arguments dans la structure de CASE n'est pas important.

CASE ($\underline{S}$, ELSE P)

------------------------ <CASE-Def sym>

CASE (ELSE P, $\underline{S}$)

cette loi est un cas spécial de <CASE sym>.

CASE($\underline{S}$, s CASE s_i P_i)

-------------------------------- i=1...n <CASE-^ distrib1>

CASE ($\underline{C}$, CASE s ^ s_i P_i)

Cette loi avec <CASE-assoc> permet de supprimer complètement l'emboîtement dans les structures de CASE.

CASE($\underline{S1}$, s CASE s_i P_i , $\underline{S2}$)

------------------------------------ i=1...n <CASE-^ distrib2>

CASE($\underline{C}$, CASE s ^ s_i P_i , $\underline{S2}$)

à condition que $\vee s_i$ = TRUE, i = 1 ... n.

Cette une loi généralise <CASE-^ distrib1>. La condition $\vee s_i$ = TRUE, i = 1 ... n garantit le processus CASE s ^ s_i P_i, i = 1, ... , n ne se comporte pas comme STOP dans le cas où s = FALSE et aucun s_i n'évalue à TRUE.

IF b_i P_i
------------- $i=1...n$ <IF-CASE permut>
CASE b_i P_i

à condition que $b_i \wedge b_j$ = FALSE pour $i \neq j$.
Si les conditions dans les structures de CASE sont disjointes, alors l'ordre des branches de CASE importe peu. Cette loi est équivalente à <IF sym>.

Lois de ALT

Dans les lois suivantes, une garde simple est sous forme : SKIP, c?x ou c!e.

ALT ( ) = STOP <ALT-STOP unit)
Une structure alternative vide se comporte comme STOP.

ALT (ALT (G1), G2)
---------------------------- <ALT assoc>
ALT (G1, G2)
cette loi permet de supprimer l'emboîtement dans une construction ALT.

ALT Gi
------------ <ALT sym>
ALT $G\pi(i)$

où π est une permutation de {1 ... n}.
L'ordre des arguments dans la structure de ALT n'est pas important.

ALT ( ) = STOP <ALT-STOP unit>

S'il n'y a pas d'arguments, la construction ALT est équivalent à STOP.

ALT (b & g P, G)
-- <bool-gard elim>
IF (b ALT (g P, G), $\neg$b ALT(G))

Une structure ALT avec des gardes contenant des composants booléens peut être réduite à la combinaison des constructions IF et ALT avec des gardes simples.

ALT (SKIP P) = P <ALT-SKIP ident>
Puisque la garde SKIP est toujours prête.

ALT (c?x SKIP) = c?x <input>

ALT (c!e SKIP) = c!e <output>

ALT(g P, G)
---------------------- <ALT idem>
ALT (g P, g P, G)

Car l'ensemble d'alternatives ne change pas.

ALT(g P, g Q, G)

-- <gard distrib>

ALT(g ALT(SKIP P, SKIP Q),$\underline{G}$)

Cette loi exprime le fait que dans une construction ALT, si plusieurs gardes sont prêtes, alors le choix est non déterministe.

IF b ALT g_i P_i

------------------------ i=1...n, <IF-ALT distrib>

IF b ALT g_i (IF b P_i)

à condition qu'aucune variable dans b n'ait été changée par g_i.
Cette loi relie IF et ALT.

Les lois suivantes traitent le non déterminisme dans les structures de ALT.

ALT(SKIP ALT(g_1 P, $\underline{G1}$), g_2 Q, $\underline{G2}$)

-- <ALT-SKIP1>

ALT(SKIP ALT(g_1 P, g_2 Q, $\underline{G1}$), $\underline{G2}$)

à condition que g_1 = c?x et g_2 = c?y ou g_1 = c!e et g_2 = c!f.

ALT(SKIP c?x, SKIP c?y)

-------------------------------- <ALT-SKIP 2>

ALT(c?x SKIP, c?y SKIP)

ALT(SKIP ALT(SKIP P, $\underline{G1}$), $\underline{G2}$)

-- <ALT-SKIP 3>

ALT(SKIP P, $\underline{G1}$, $\underline{G2}$)

ALT(SKIP ALT($\underline{G1}$), SKIP ALT($\underline{G1}$, $\underline{G2}$), $\underline{G3}$))

-- <ALT-SKIP 4>

ALT(SKIP ALT($\underline{G1}$), $\underline{G2}$, $\underline{G3}$))

Lois de l'affectation

On permet l'affectation multiple sous forme : $\underline{x} := \underline{e}$, où $\underline{x}$ et $\underline{e}$ sont une liste de variables et une liste d'expressions respectivement, de la même longueur.

$< > := < >$

------------- <assign SKIP>

SKIP

L'affectation vide est sans effet.

$<x_i \mid i=1 \ldots n> := <e_i \mid i=1 \ldots n>$

-- <assign sym>

$<x_{\pi(i)} \mid i=1 \ldots n> := <e_{\pi(i)} \mid i=1 \ldots n>$

où π est une permutation de $\{1 \ldots n\}$.
L'ordre d'affectation d'expression/variable n'a pas d'importance.

* $\underline{x}$ + y := $\underline{e}$ + y

-------------------- <assign ident>

$\underline{x}$:= $\underline{e}$

L'affectation de valeur d'une variable à elle-même est sans effet.

Les lois suivantes concernent seulement les variables de type tableau, la distribution de données (souvent sous la forme de type tableau) dans les processus parallèles. On suppose que s=[v FROM b FOR c] est valide et e de même type.

s : = e

-------------------- <array-assign>

s[π(i)] := e[π(i)]

où π est une permutation de {b, . . . , (b+c)-1}.

L'ordre des affectations des éléments d'un tableau na pas d'importance.

Les lois qui relient l'affectation avec les autres constructions sont dans la section de lois des déclarations.

Lois de SEQ

Les deux lois suivantes permettent la transformation de toute occurrence de SEQ en une constructeur binaire.

SEQ ( ) = SKIP <SEQ-SKIP unit>

Si une construction SEQ n'a pas d'arguments, il termine simplement.

SEQ(P, $\underline{P}$)
---------------------- <SEQ assoc>
SEQ (P, SEQ($\underline{P}$))

Cette loi exprime l'exécution séquentielle des processus composants dans une structure SEQ.

* SEQ(IF b_i P_i, $\underline{Q}$)
---------------------- i=1...n <SEQ-IF distrib>
IF b_i SEQ(P_i, $\underline{Q}$)

SEQ distribue sur IF.

* SEQ(ALT g_i P_i, $\underline{Q}$)
------------------------ i=1...n <SEQ-ALT distrib>
ALT g_i SEQ(P_i, $\underline{Q}$)

SEQ distribue sur ALT.

* SEQ($\underline{x}$:= $\underline{e}$, IF b_i P_i)
------------------------------ i=1...n <SEQ-IF assign>
IF b_i [$\underline{e}$/$\underline{x}$] SEQ($\underline{x}$:= $\underline{e}$, P_i)

* SEQ($\underline{x}$:= $\underline{e}$, ALT g_i P_i)
---------------------------------- i=1...n <SEQ-ALT assign>
ALT g_i [$\underline{e}$/$\underline{x}$] SEQ($\underline{x}$:= $\underline{e}$, P_i)

à condition qu'aucune variable dans $\underline{x}$ ou $\underline{e}$ ne soit l'entrée dans les gardes g_i.
Quand $\underline{x}$:= $\underline{e}$ termine, la construction se comporte comme IF ou ALT, mais tient compte de l'effet de l'affectation.

* SEQ($\underline{x}$:= $\underline{e}$, $\underline{x}$:= $\underline{f}$)
----------------------- <combin assign>
$\underline{x}$:= $\underline{f}$ [$\underline{e}$ /$\underline{x}$]

La composition séquentielle de deux affectations à la même liste de variables peu être combinées en une seule affectation, en tenant compte de l'effet de l'affectation précédente.

SEQ(x := e, c ? x)
---------------------- <SEQ-com 1>
c ? x

Il n'a pas de sens d'affecter à une variable qui est utilisée comme le but d'entrée avant son utilisation.

ALT(g P)
------------ où g est SKIP, c ? x ou c ! e. <SEQ-ALT gard>

SEQ(g, P)

SEQ($\underline{x}$:= $\underline{e}$, c ? y)
----------------------- <SEQ com2>
SEQ(c?y, $\underline{x}$:= $\underline{e}$)

à condition que *y* ne soit pas libre dans $\underline{x}$:= $\underline{e}$.

SEQ($\underline{x}$:= $\underline{e}$, c ? f)
---------------------------- <SEQ com3>
SEQ(c! f ($\underline{e}$/$\underline{x}$), $\underline{x}$:= $\underline{e}$)

Lois de PAR

Un processus PAR invalide est équivalent au processus ZERO. Par exemple, ceux qui contiennent les constructions en parallèle où les conditions de séparation de canaux et de variables ne sont pas respectées.

On exige que chaque processus parallèle déclare d'une façon explicite des canaux comme entrée ou sortie, et des variables en lecture ou en écriture. Par exemple :

$P = U_i : P_i$, où U_i contient l'utilisation de canaux et de variables.

L'objet de ces exigence est de faciliter la détection de la validité de constructions PAR, de la portée et de la nature de canaux et variables utilisés pour l'analyse statique. Si l'utilisation de variables et de canaux est simple ou ne change pas, on omet d'indiquer explicitement les U_i pour les processus parallèles.

PAR ( ) = SKIP <PAR-SKIP unit>

Une construction parallèle vide termine simplement.

PAR U_i: P_i
-- $i=1...n$, $j = 2...n$ $(n > 0)$ <PAR assoc>
PAR (U_1: P_1, U^*: (PAR U_j: P_j))
où U^* est l'union de $U_2, \ldots, U_n$.

où U^* est l'union de $U_2, \ldots, U_n$. Notons que U^* déclare des canaux comme internes si ces canaux sont déclarés comme entrées et sorties entre U_i.

On utilise ces deux lois pour réduire PAR à une opérateur binaire.

PAR(U_1: P_1, U_2: P_2)
---------------------------- <PAR sym>
PAR (U_2: P_2, U_1: P_1)

L'ordre des composant processus de PAR n'est pas important.

* PAR(U_1: IF b_i P_i, U_2: Q)
---------------------------------- $i=1...n$ <PAR-IF distrib>
IF b_i PAR(U_1: P_i, U_2: Q)

à condition que $b_1 \vee \ldots \vee b_n$ = TRUE.
Le choix conditionnel peut être exécuté avant d'entrer dans une construction parallèle. La condition $b_1 \vee \ldots \vee b_n$ = TRUE garantit que le processus conditionnel puisse entrer dans PAR.

* PAR($\underline{x} := \underline{e}$, $\underline{y} := \underline{f}$)
------------------------------------ <PAR assign>
$\underline{x} + \underline{y} := \underline{e} + \underline{f}$
Deux affectations multiples en parallèle peuvent être combinées en une seule.

* PAR(U_1: ALT g_i P_i, U_2: $\underline{x} := \underline{e}$)

-- $i=1...n$, <PAR-ALT expansion1>

ALT g_j PAR(U_1: P_j, U_2: $\underline{x} := \underline{e}$)

avec j dans X où g_j sont des gardes simples,

où X est l'ensemble d'indices $i \in \{1,..., n\}$ tel que :

g_j = SKIP,

ou g_j = c!e et $c \in$ output(U_1) - input(U_2),

ou g_j = c?x et $c \in$ input(U_1) - output(U_2).

Si P = ALT g_i P_i , i=1...n et Q = ALT h_j P_j , j=1...m où g_i, h_j sont sous la forme : c?x, c!e ou SKIP, on a alors :

* PAR(U_1: P, U_2: Q)

------------------------------ <PAR-ALT expansion 2>

ALT k_r R_r

où les paires <k_r, R_r> sont toutes les formes possibles suivantes :

(i) R_r = PAR(U_1: P_i, U_2: Q) et

$k_r = g_i$ = SKIP

ou $k_r = g_i$ = c!e et $c \in$ output(U_1) - input(U_2)

ou $k_r = g_i$ = c?x et $c \in$ input(U_1) - output(U_2)

(ii) R_r = PAR(U_1: P, U_2: Q_j) et

$k_r = h_j$ = SKIP

ou $k_r = h_j$ = c!e et $c \in$ output(U_2) - input(U_1)

ou $k_r = h_j$ = c?x et $c \in$ input(U_2) - output(U_1)

(iii) R_r = SEQ(x := e, PAR(U_1: P_i, U_2: Q_j)) et

k_r = SKIP

g_i = c!e et h_j = c?x et $c \in$ input(U_2) $\cap$ output (U_1)

ou g_i = c?x et h_j = c!e et $c \in$ input(U_1) $\cap$ output(U_2)

(i) et (ii) expriment que P et Q avancent indépendamment, (iii) exprime l'effet de communications internes entre P et Q.

PAR(c!e, c?x)

------------------- <I/O simple>

x := e

Cette loi est la définition de communication sur un canal.

PAR(c!$\underline{e}$, c?$\underline{x}$)

------------------- <I/O>

$\underline{x} := \underline{e}$

Cette loi est la définition de communication sur un canal.

* PAR(U_1: P, U_2: SKIP)

---------------------------------- <PAR-SKIP unit>

P

Cette loi exprime l'effet de la terminaison des processus composants dans une structure parallèle.

Lois de la portée des déclarations

Ces lois avec les lois de renommage de variables et de canaux permettent de transformer tout programme en une forme canonique : toutes les déclarations de variables et canaux sont placées juste avant les constructions qui les utilisent, et les noms de variables internes de chaque processus parallèle composant sont différents de ceux des autres dans la même construction.

VAR x_1: (VAR x_2: . . . VAR x_n: P). . .)
--- <VAR assoc>
VAR x_1, . . ., x_n: P

Il n'est pas important que les variables soient déclarées dans une liste ou non.

VAR x_1: (VAR x_2: P)
-------------------------- <VAR sym>
VAR x_2: (VAR x_1: P)

L'ordre de déclaration n'a pas d'importance.

VAR x: P
---------- à condition que $x \notin$ libre(P). <VAR elim>
P

VAR x: P
---------------- $y \notin$ libre(P) <VAR rename>
VAR y: P[y/x]

On peut changer le nom d'une variable liée.

Les lois suivantes expriment le fait que si une variable n'apparaît pas libre dans les conditions ou les gardes d'une construction IF ou ALT, alors la déclaration de cette variable peut être placée hors de la structure.

ALT g_i (VAR x: P_i)
------------------------ i=1...n <VAR-ALT distrib>
VAR x: (ALT g_i P_i)

A condition que x ne soit libre dans aucune garde g_i.

IF b_i (VAR x: P_i)
--------------------- i=1...n <VAR-IF distrib>
VAR x: (IF b_i P_i)

A condition que x ne soit libre dans aucune b_i.

SEQ(VAR x: P, Q)
------------------------ $x \notin$ libre(Q). <VAR-SEQ sym1>
VAR x: SEQ(P, Q)

SEQ(P, VAR x: Q)
------------------------ $x \notin$ libre(P). <VAR-SEQ sym2>
VAR x: SEQ(P, Q)

PAR (U_1: (VAR x: P_1), U_2: P_2)
-- <VAR-PAR sym>
VAR x: PAR(U_1*: P_1, U_2: P_2)

A condition que x ne soit pas libre dans U_2, U_1* est l'union de U_1 et VAR: x. Quand une variable est placée hors d'une construction, le processus qui l'utilise doit la déclarer explicitement.

VAR x: WHILE b P
-------------------------- <VAR-WHILE sym>
WHILE b (VAR x: P)

A condition que x n'est pas libre dans b.

ALT(c?x P, $\underline{G}$)
--- <input rename>
VAR y: ALT(c?y SEQ(x := y, P), $\underline{G}$)

A condition que $x \neq y$ et y ne soit pas libre dans P et $\underline{G}$.
L'entrée à une variable x est de même effet que l'entrée à une nouvelle variable y et ensuite affectée à x.

* VAR x: (<x> + $\underline{y}$) := (<e> + $\underline{f}$)
-- <assign elim>
VAR x: ($\underline{y}$:= $\underline{f}$)

L'affectation à une variable est sans effet en dehors de sa portée .

VAR x: P
-------------------------------------- <VAR init>
VAR x: SEQ(VAR z: (x := z), P)

Il s'agit de l'usage de variables non-initialisées dans les expressions. La valeur d'une variable non-initialisée peut être remplacée par celle d'une autre.

D'une façon similaire, il existe des lois de déclaration pour les canaux comme pour les variables.

On simplifie la notion de déclaration de canaux <CHAN OF type> par <CHAN>, et on comprend que chaque déclaration de canaux implique son type (protocole).

CHAN c_1: (CHAN c_2: . . . CHAN c_n: P). . .)
-- <CHAN assoc>
CHAN c_1, . . ., c_n: P

Les canaux sont de même type.
Il n'est pas important que les canaux soient déclarés dans une liste ou non.
A remarquer que cette loi permet de réduire plusieurs déclarations de canaux à une déclaration multiple.

$$\frac{\text{CHAN } c_1\text{: (CHAN } c_2\text{: P)}}{\text{CHAN } c_2\text{: (CHAN } c_1\text{: P)}}$$ <CHAN sym>

L'ordre de déclaration n'a pas d'importance.

$$\frac{\text{CHAN } C_1, \ldots C_n\text{: P}}{\text{P}}$$ <CHAN elim>

A condition que $C_1, \ldots, C_n \notin$ libre(P).

$$\frac{\text{CHAN c: P}}{\text{CHAN d: P[d/c]}}$$ si d $\notin$ librechan(P). <CHAN rename>

On peut changer le nom d'un canal lié.

Les lois suivantes expriment le fait que si un canal n'apparaît pas dans les gardes d'une construction ALT, ou dans les autres constructions, alors la déclaration de ce canal peut être placée hors de la structure.

$$\frac{\text{CHAN } \underline{c}\text{: ALT } g_i \text{ (CHAN c: } P_i\text{)}}{\text{CHAN } \underline{c}'\text{: (ALT } g_i \text{ (CHAN } \underline{c}_i \text{ : } P_i\text{)}}$$ i=1...n <CHAN-ALT distrib1>

A condition que $\underline{c}_i$ ne soit pas libre dans g_j, $i \neq j$; $\underline{c} = \underline{c}' \cup \underline{c}_i$.

$$\frac{\text{ALT } g_i \text{ (CHAN c: } P_i\text{)}}{\text{CHAN c: (ALT } g_i \text{ } P_i\text{)}}$$ i=1...n <CHAN-ALT distrib2>

A condition que c ne soit libre dans aucune garde g_i.
Cette loi est un cas particulier de la loi ci-dessus.

$$\frac{\text{IF } b_i \text{ (CHAN c: } P_i\text{)}}{\text{CHAN c: (IF } b_i \text{ } P_i\text{)}}$$ i=1...n <CHAN-IF distrib>

$$\frac{\text{SEQ(CHAN c: P, Q)}}{\text{CHAN c: SEQ(P, Q)}}$$ <CHAN-SEQ sym1>

A condition que c $\notin$ librechan(Q).

$$\frac{\text{SEQ(P, CHAN c: Q)}}{\text{CHAN c: SEQ(P, Q)}}$$ <CHAN-SEQ sym2>

A condition que $c \notin$ librechan(P).

CHAN $\underline{c}$: SEQ P_i
------------------------ $i = 1,\ldots, n$ <CHAN-SEQ distrib>
SEQ (CHAN c_i : P_i)

A condition que $\cup \underline{c}i = \underline{c}$ et $\underline{c}_i$ ne soit pas libre dans P_j, $i \neq j$.
Cette loi est la dérivation de deux lois ci-dessus.

PAR (U_1: (CHAN c: P_1), U_2: P_2)
-- <CHAN-PAR sym>
CHAN c: PAR(U_1*: P_1, U_2: P_2)

A condition que c ne soit pas libre dans U_2: P_2, où U_1* est l'union de U_1 et CHAN: c.

CHAN c: WHILE b P
----------------------------- <CHAN-WHILE distrib>
WHILE b (CHAN c: P)

VAR x: CHAN c: P
------------------------ <CHAN-VAR sym2>
CHAN c: VAR x: P

L'ordre de la déclaration de canaux et de variables n'est pas important.

Avec des lois de la portée des variables et des canaux, il est possible de normaliser leur utilisation en déplaçant leur déclaration jusqu'à l'endroit où ils seront utilisés.

Lois de ZERO

On identifie ZERO avec tout processus qui est possible de diverger.

ZERO = WHILE TRUE SKIP <ZERO>

* ALT(SKIP ZERO, G) = ZERO <ALT-SKIP zero>

Un processus qui peut choisir automatiquement de diverger est identifié avec ZERO.

* SEQ(ZERO, P) = ZERO <SEQ zero 1>

* SEQ(x := e, ZERO)
--------------------------- <SEQ zero 2>
ZERO

Ces deux lois expriment que la divergence du premier et deuxième processus argument peut causer la divergence de la construction entière de SEQ.

Lois de WHILE

WHILE b P
-- <WHILE expansion>
IF(b SEQ(P, WHILE b P), $\neg$b SKIP)

C'est la définition récursive WHILE.

WHILE b_1 (WHILE b_2 P)
-- <WHILE combine>
WHILE b_1 v b_2 IF (b_2 P, TRUE ZERO)

Cette loi exprime la fusion de deux boucles emboîtées.

WHILE b P
-- <infinite loop>
IF (b WHILE TRUE P, $\neg$b SKIP)

Si les variables de b ne sont jamais changées par le processus P, c'est une boucle infinie.

WHILE TRUE x := e
--------------------------- <divergent loop>
ZERO

Annexe A2 Lois d'ajustement du parallélisme

Les lois dans cette section sont l'application spécifique. Elles sont utiles dans les transformations d'applications. Elles peuvent être dérivées à partir des lois élémentaires.

```
          WHILE b SEQ(P, Q)
-----------------------------------------------        <WHILE-SEQ>
IF (b SEQ(P, WHILE b SEQ(Q, P), Q), ¬b SKIP)
```

Si les variables libres de P ne sont jamais changées par Q.

```
VAR ch1, ch2:
SEQ
    in?ch1
    WHILE TRUE
      SEQ (PAR (in ? ch2, out ! ch1),
           PAR (in ? ch1, out ! ch2))
-------------------------------------                  <infinite buffer>
CHAN c:
PAR
    WHILE TRUE
      VAR ch: SEQ ( in ? ch, c ! ch)
    WHILE TRUE
      VAR ch: SEQ (c ? ch, out ! ch)
```

Cette loi exprime deux version d'un tampon double infini.

```
PAR
    WHILE TRUE SEQ (in?x, r := f1(x), c!r),
    WHILE TRUE SEQ (c?y, r' := f2(y), c!r'))
------------------------------------------------------------     <WHILE-PAR>
SEQ
    in?x
    WHILE TRUE
        SEQ (PAR (in ?y, SEQ (r := f1(x), r' := f2(r), out!r'))
            in ?x
            SEQ (r := f1(y), r' := f2(r), out!r'))
```

Cette loi permet la fusion de deux boucles dans le cas où le pipeline de communications est la seule interaction entre ces deux boucles.

Une boucle séquentielle avec un nombre statique d'itérations est équivalente à un processus séquentiel.

```
SEQ( count := 0, WHILE ( count < N , P(count)) )
-----------------------------------------------
    SEQ count = 0 FOR N P(count)
-----------------------------------------------
        SEQ P(i), i = 1, ..., N                        <WHILE finite>
```

Les lois suivantes relient PAR et SEQ.

SEQ (x:=e, PAR(U_1: P_1, U_2: P_2[e/x]))

PAR(U_1: SEQ(c!e, P_1), U_2: SEQ(c?x, P_2))

<PAR-SEQ com1>

Cette loi exprime l'effet de communication. Elle est plus générale que <I/O>.

SEQ($\underline{x}$:= $\underline{e}$, PAR(U_1: P, U_2: Q))

PAR(U_1: SEQ($\underline{x}$:= $\underline{e}$, P), U_2: Q)

<PAR-SEQ assign assoc>

SEQ(PAR(P1, Q1), PAR(P2, Q2))

PAR(SEQ(P1, P2), SEQ(Q1, Q2))

<PAR-SEQ rel1>

à condition que :

librechan(P1) $\cap$ librechan(Q2) = $\varnothing$ et,

librechan(P2) $\cap$ librechan(Q1) = $\varnothing$.

Cette condition garantit l'absence de blocage dans PAR(P1, Q1) et PAR(P2, Q2).

ALT(SKIP SEQ(c?x, U*: PAR(P, SEQ(d?y, Q))),
SKIP SEQ(d?y, U*: PAR(Q, SEQ(c?x, P))))

PAR(U_1: SEQ(c?x, P), U_2: SEQ(d?y, Q))

<PAR-SEQ com2>

où U* est l'union de U_1 et U_2.
Si deux communications sur les canaux externes différents sont prêtes en même temps, alors chacune peut avoir lieu en premier.

ALT(SKIP SEQ(c!e, U*: PAR(P, SEQ(d!f, Q))),
SKIP SEQ(d!f, U*: PAR(Q, SEQ(c!e, P))))

PAR(U_1: SEQ(c!e, P), U_2: SEQ(d!f, Q))

<PAR-SEQ com3>

où U* est l'union de U_1 et U_2.
Cette loi est similaire que celle précédente.

ALT(SKIP SEQ(c?x, U*: PAR(P, SEQ(d!f, Q))),
SKIP SEQ(d!f, U*: PAR(Q, SEQ(c?x, P))))

PAR(U_1: SEQ(c?x, P), U_2: SEQ(d!f, Q))

<PAR-SEQ com4>

où U* est l'union de U_1 et U_2.

ALT(SKIP SEQ(c!e, U*: PAR(P, SEQ(d?y, Q))),
SKIP SEQ(d?y, U*: PAR(Q, SEQ(c!e, P))))

PAR(U_1: SEQ(c!e, P), U_2: SEQ(d?y, Q))

<PAR-SEQ com5>

où U* est l'union de U_1 et U_2.

A remarquer que les quatre lois ci-dessus sont des cas spéciaux de la loi <PAR-ALT expansion*>, puisque on a la loi : SEQ (c?x P) = ALT (c?x P).

```
PAR( U1: PAR(P, c?x), U2: PAR(Q, c!e))
------------------------------------------------------          <PAR-PAR com1>
      PAR( U*: PAR(P, Q), x := e)
```

à condition que c ne soit pas dans librechan(P) et librechan(Q). $U^* = U_1 \cup U_2$.

```
      SEQ(P,Q[e/x])
------------------------------------------                      <PAR-SEQ com6>
PAR(SEQ (P, c!e), SEQ(c?x,Q))
```

à condition que c ne soit pas libre dans output(P).
Cette loi exprime l'effet de pipeline de communication.

```
      SEQ(P,Q[e/x])
------------------------------------------                      <PAR-SEQ com6'>
PAR(PAR (P, c!e), SEQ(c?x,Q))
```

Cette loi exprime l'effet de pipeline de communication.

```
SEQ(PAR(P,Q), x := e, PAR (P', Q'[e/x]))
------------------------------------------------------          <PAR-SEQ com7>
PAR(SEQ (P, c!e, P'), SEQ(Q, c?x, Q'))
```

à condition que c ∉ output(P) et input(Q).
Cette loi est plus général que celle précédente, exprimant le même effet de pipeline.

```
PAR
    IF
        c1 SEQ(P1,c!e1)
        c2 SEQ(P2,c!e2)
    SEQ (c?x, Q)
----------------------------------------------------            <PAR-IF-SEQ pipe1>
PAR
    IF
        c1 SEQ(P1, x := e1, Q[e1/x])
        c2 SEQ(P2, x:=e2, Q[e2/x]))
```

à condition que c ∉ librechan(P1) et librechan(P2).
Cette loi peut être prouvée en transformant SEQ et IF en ALT, après utilisation de la loi PAR-ALT expansion.

```
IF
    c1 SEQ(P1, x := e1, PAR(R1, Q[e1/x])
    c2 SEQ(P2, x:= e2, PAR(R2, Q[e2/x]))
-------------------------------------------------------      <PAR-IF-SEQ pipe2>
```

```
PAR
      IF
            c1 SEQ(P1, c!e1, R1)
            c2 SEQ(P2, c!e2, R2)
      SEQ (c?x, Q))
```

à condition que c $\notin$ librechan(P1) et librechan(P2).

```
VAR loop, c:
SEQ (loop := 0, c := N,
      WHILE (loop < c) SEQ (P, loop := loop +1))
------------------------------------------------------------          <WHILE-SEQ conv>
      SEQ i = 0 FOR N
            P[i/loop]
```

A condition que la variable loop n'ait pas changé dans P.

Cette loi permet de simuler une construction séquentielle répliquée par une construction de WHILE; d'autre part, une construction de WHILE dont le nombre d'itération est connu statiquement peut être exprimée par une construction répliquée.

La preuve peut être faite en déroulant la construction N fois, ce qui remplace la boucle par une séquence de SEQ.

Il existe des lois similaires pour d'autres opérateurs.

```
VAR loop, c:
SEQ (loop := 0, c := N,
      WHILE (loop < c)
            IF ( c1 SEQ (P1, loop := loop +1),
                  c2 SEQ (P2, loop := loop +1)))
----------------------------------------------------------          <WHILE-IF conv>
      IF i = 0 FOR N
            c1 P1[i/loop]
            c2 P2[i/loop]
```

A condition que la variable loop n'ait pas changé dans Pi.

```
PAR (       SEQ i=0 FOR N SEQ (P1, c!e1),
            SEQ i=0 FOR N SEQ (P2, d!e2),
            SEQ i=0 FOR N SEQ (PAR(c?x, d?y), Q))
----------------------------------------------------------------          <PAR-SEQ pipe1>
SEQ i=0 FOR N
      SEQ ( PAR(P1, P2),
            x := e1, y := e2,
            Q[e1/x, e2/y])
```

à condition que c $\notin$ librechan(P1), d $\notin$ librechan(P2).

Cette loi peut être démontrée en appliquant des lois <SEQ-PAR> par induction sur i.

Il existe une loi plus générale que celle-ci, qui traite la structure de SEQ (P1, c ! e, R1) :

```
PAR (       SEQ i=0 FOR N SEQ (P1, c!e1, R1),
            SEQ i=0 FOR N SEQ (P2, d!e2, R2),
            SEQ i=0 FOR N SEQ (PAR(c?x, d?y), Q))
```

```
-----------------------------------------------------------------        <PAR-SEQ pipe2>
       SEQ i=0 FOR N
              SEQ ( PAR(P1, P2),
                     x := e1, y := e2,
                     PAR(R1, R2, Q[e1/x, e2/y]))
```

à condition que c ∉ librechan(P1), d ∉ librechan(P2).

Les lois suivantes expriment que l'ordre des communications est maintenu dans les constructions différentes de SEQ et PAR, et elles donnent le résultats x := e. A remarquer que ce sont les cas possibles de communiquer une liste de valeur e par une liste de canaux c.

```
PAR (PAR i=0 FOR n (c[i] ? x[i]),
       PAR i=0 FOR n (c[i] ! e[i]))
-----------------------------------------                <PAR-SEQ assign1>
PAR (SEQ i=0 FOR n (c[i] ? x[i]),
       PAR i=0 FOR n (c[i] ! e[i]))

PAR (PAR i=0 FOR n (c[i] ? x[i]),
       PAR i=0 FOR n (c[i] ! e[i]))
-----------------------------------------
PAR  (SEQ i=0 FOR n (c[i] ? x[i]),
       SEQ i=0 FOR n (c[i] ! e[i]))                      <PAR-SEQ assign2>

PAR (PAR i=0 FOR n (c[i] ? x[i]),
       SEQ i=0 FOR n (c[i] ! e[i]))
-----------------------------------------                <PAR-SEQ assign3>
PAR (SEQ i=0 FOR n (c[i] ? x[i]),
       PAR i=0 FOR n (c[i] ! e[i]))

PAR (PAR i=0 FOR n (c[i] ? x[i]),
       SEQ i=0 FOR n (c[i] ! e[i]))
-----------------------------------------                <PAR-SEQ assign4>
PAR  (SEQ i=0 FOR n (c[i] ? x[i]),
       SEQ i=0 FOR n (c[i] ! e[i]))


PAR ( PAR i=0 FOR n VAR x: SEQ ( c[i] ? x, x := fi(x), c[i+1] ! x),
       SEQ( c[0] ! e, c[n] ? z))
---------------------------------------------------------------------------   <PAR-SEQ pipe3>
PAR( VAR x: SEQ( c[0] ? x, SEQ i=0 FOR n (x := fi(x)), c[n]! x),
       SEQ( c[0] ! e, c[n] ? z))
```

Les processus donnent le résultat : z = f(e), où *f* est la composition fonctionnelle de *f1*, . . .*fn* . Il s'agit d'implémentation distribuée de la synchronisation globale.

```
PAR (PAR i= 0 FOR n SEQ (c[i] ? x[i], x[i] := fi(x[i]), d[i] ! x[i]),
       PAR( PAR i=0 FOR n (c[i] ! xin[i]),
              PAR i=0 FOR n (d[i] ? xout[i])))
---------------------------------------------------------------------------   <PAR-SEQ pipe4>
PAR (SEQ(SEQ j=i+1 FOR (n-i)+1 (c[i] ? xtmp[i], c[i+1] ! xtmp[i]),
              x[i] := fi(x[i])),
       SEQ j=0 FOR i (  c[i] ? xtmp[i], c[i+1] ! xtmp[i]),
                           c[i+1] ! x[i])))
```

La preuve de cette loi peut être faite par induction sur *n* en appliquant les lois <PAR-SEQ com*>, et ces processus donnent le même résultat :

SEQ i=0 FOR n (xout[i] := fi[xin[i])

```
PAR (SEQ i=0 FOR n (c[i] ? x[i]),
     SEQ i=0 FOR n (c[i] ! e[i]))
----------------------------------------
PAR (SEQ i=0 FOR n (c ? x[i]),                              <PAR-SEQ pipe5>
     SEQ i=0 FOR n (c ! e[i]))
----------------------------------------
     x := e
```

où x̲ et e̲ sont à dimension n.

Les quatre lois suivantes sont similaires aux celles ci-dessus, permettent la transformation des constructions séquentielles en pipeline en celles parallèles.

```
PAR (SEQ i=0 FOR n (PAR (c1 ? x, c2 ? y), d !  f(x,y)),
     SEQ i=0 FOR n (SEQ(P, PAR(c1! e1, c2 ! e2))) ,
     SEQ i=0 FOR n (VAR r: d ? r ))
-------------------------------------------------------------------------------------
PAR (PAR i=0 FOR n (VAR x, y: PAR (c1[i] ? x, c2[i] ? y), d[i] ! f(x,y)),
     PAR i=0 FOR n (VAR e1, e2 : SEQ(P, PAR(c1[i] ! e1, c2[i] ! e2))) ,
     PAR i=0 FOR n (VAR r: d[i] ? r ))
                                                                        <pipe 1>
```

```
PAR (SEQ i=0 FOR n (PAR (c1 ? x, c2 ? y), d ! f(x,y)),
     SEQ i=0 FOR n (SEQ(P, PAR(c! e1, d! e2))) ,
     SEQ i=0 FOR n (VAR r: d ? r ))
-------------------------------------------------------------------------------------
PAR (SEQ i=0 FOR n (PAR (c1? x, c2 ? y), d ! f(x,y)),
     PAR i=0 FOR n (VAR e1, e2 : SEQ (P, PAR(c1[i] ! e1, c2[i] ! e2))) ,
     PAR i=0 FOR n (VAR r: d[i] ? r ))
                                                                        <pipe 2>
```

```
PAR (SEQ i=0 FOR n (PAR (c1 ? x, c2 ? y), d! f(x,y)),
     SEQ i=0 FOR n (SEQ(P, PAR(c! e1, d! e2))) ,
     SEQ i=0 FOR n (VAR r: d ? r ))
--------------------------------------------------------------------------------
PAR (PAR i=0 FOR n (VAR x,y: PAR (c1[i] ? x, c2[i] ? y), d[i] ! f(x,y)),
     SEQ i=0 FOR n (SEQ(P, PAR(c! e1, d! e2))) ,
     PAR i=0 FOR n (VAR r: d[i] ? r ))
                                                                        <pipe 3>
```

```
PAR (SEQ i=0 FOR n (PAR (c1 ? x, c2 ? y), d ! f(x,y)),
     SEQ i=0 FOR n (SEQ(P, PAR(c ! e1, d! e2))) ,
     SEQ i=0 FOR n (VAR r: d ? r ))
-----------------------------------------------------------------
PAR (PAR i=0 FOR n (VAR x,y: PAR (c1[i] ? x, c2[i] ? y), d[i] ! f(x,y)),
     PAR i=0 FOR n (VAR e1, e2 : SEQ(P, PAR(c1[i] ! e1, c2[i] ! e2))) ,
     SEQ i=0 FOR n (VAR r: d ? r ))
                                                                        <pipe 4>
```

La loi suivante exprime l'implémentation distribuée de la synchronisation globale :

```
VAR maître[n], esclave[n]:
WHILE TRUE
      SEQ
         PAR i = 0 FOR n esclave[i] := f(maître[i])
         PAR (entrée ? maître[0],
               PAR i = 0 FOR n-1   maître [i+1] := esclave[i]
               sortie ! esclave[n] )
------------------------------------------------------------------------
         WHILE TRUE
            CHAN [n+1]:
            PAR i = 0 FOR n
               VAR d, r: SEQ ( c[i] ? d, r := f(d), c[i+1] ! r )
```

<SEQ-PAR distrib>

Les lois suivantes permettent le déplacement de PAR hors des constructions externes, donc utiles pour l'extraction du parallélisme.

```
SEQ (P,Q)
--------------
PAR (P, Q)
```

<PAR-SEQ paral1>

à condition que libre(P) $\cap$ libre(Q)=Ø.

Cette loi doit être utilisée à ce que P et Q communiquent avec l'environnement, l'ordre de communications doit être maintenu pour qu'un blocage ne se produise pas. A remarque cette loi représente le cas général d'absence de la dépendance de données.

```
SEQ (P,Q)
------------------------------------
CHAN c:
PAR (VAR x: SEQ (P, c!x),
      VAR y: SEQ (c?y, Q(y/x))
```

<PAR-SEQ paral2>

A condition que c ne soit pas libre dans P et Q, y ne soit pas libre dans Q;

x = final(P) $\cap$ init(Q).

Cette loi exprime la parallélisation des processus séquentiels avec la dépendance de données. A remarquer que cette loi représente le cas général de l'extraction de parallélisme.

```
SEQ (P, x := y, Q)
------------------------
CHAN c :
PAR (SEQ (P, c ! y)
      SEQ ( c ? x, Q)
```

<PAR-SEQ paral3>

à condition que libre(P) $\cap$ libre(Q) = Ø. C'est un cas particulier de la loi précédente.

```
SEQ (P, Q, R)
-------------------------------------------------
CHAN c,d:
PAR (SEQ (P, c ! init(Q), d? final(Q), R)),
      VAR librevar(Q):
```

<SEQ-PAR conv>

```
SEQ (c ? init(Q), Q, d ! final(Q))).
```

où Q n'a pas de canaux externes en commun avec P et R, cette condition garantit qu'il n'y aura pas de blocage possible du aux communications avec l'environnement pendant l'exécution parallèle de Q et P, R. c,d ∉ libre(P), libre(Q) et libre(R).

```
SEQ (P, PAR ( Q, R))
---------------------------
CHAN c:
PAR
      VAR librevar(P) ∪ librevar(Q):
      SEQ (P, Q)
      SEQ (P, R)
```

à condition que le processus P n'ait pas de communication avec le reste du programme.

```
          SEQ (P, PAR(Q, R), ...)
-----------------------------------------------------          <SEQ-PAR conv2>
CHAN c, d:
PAR (SEQ (c ? init(Q), Q, d ! final(Q)),
       SEQ(P, c ! init(Q), R, d ? final(Q)), ...)
```

où c,d ∉ libre(P) et libre(Q).

```
IF (b0 P0, ..., bi ( PAR(P,Q)), ...)
------------------------------------------                     <IF-PAR conv>
CHAN c, d:
PAR (SEQ (c ? init(P), P, d ! final(P)),
      IF (  b0 P0, ...,
            bi (SEQ(c ! init(P), Q, d ? final(P)),...)
```

où c,d ∉ libre(P) et libre(Q).

```
ALT(g0 G0, ..., gi ( PAR(P,Q)), ...)
---------------------------------------------------------      <ALT-PAR conv>
CHAN c, d:
PAR (SEQ (c ? init(P), P, d ! final(P)),
      ALT(g0 P0, ...,
            gi (SEQ(c ! init(P), Q, d ? final(P)),...)
```

où c,d ∉ libre(P) et libre(Q).

```
      WHILE e
         CHAN c:
         PAR ( P, Q)
--------------------------------------------                   <WHILE-PAR conv>
      CHAN c, init, data, bool:
      PAR
         VAR libre(P) ∩ A, loop:
         SEQ
            init ? init(P) ∩ A
```

```
    bool ? loop
    WHILE loop SEQ ( P, data ! final(P) ∩ A, bool ? loop)
  SEQ
    init ! init(P) ∩ A
    bool ! e
    WHILE e SEQ ( Q, data ? (P) ∩ A, bool ! e )
```

où A est l'ensemble des variables référencées dans la condition *e*, et les canaux *init*, *data, bool* ∉ libre(P) et libre(Q).

```
  WHILE e
    CHAN c:
    PAR ( P, Q)
----------------------------------------------------------
CHAN c, init, bool:
PAR
  SEQ
    init ? init(P) ∩ A
    WHILE TRUE SEQ (P, bool ? ANY)
  SEQ
    init ! init(P) ∩ A
    WHILE e SEQ (Q, bool ! ANY)              <WHILE-PAR conv2>
```

où final(P) ∩ A = Ø, et e = TRUE. C'est un cas simplifié de la loi <WHILE-PAR conv>.

Annexe B1 Programme systolique à deux dimension

C = A x B,
où A, B et C sont à dimension n x n.

```
1     VAR AIn[n, n], AOut[n, n],
2           BIn[n, n], BOut[n, n],
3           CIn[n, n], COut[n, n]:

4     CHAN Right[n, n+1],      -- canaux horizontaux
5     CHAN Up[n+1, n]:         -- canaux verticaux

6     SEQ

        -- PRE-TRAITEMENT

7       "Lecture des matrices d'entrée : A et B"
8       "Initialisation de la matrice de sortie : C"

9       PAR

          -- ENTREE DES CELLULES

          -- entrée vertical : injection du flot A

10        PAR col = 0 FOR n
11          SEQ row = 0 FOR n
              Up[col, 0] ! AIn[col, row]
12
          -- entrée horizontale : chargement du flot C et injection du flot B

13        PAR row = 0 FOR n
14          SEQ

15            SEQ col = 0 FOR n
16              Right[0, row] ! CIn[col, row]

17            SEQ col = 0 FOR n
18              Right[0, row] ! BIn[col, row]

          -- CELLULES DE CALCUL

19        PAR col = 0 FOR n
20          PAR row = 0 FOR n

21            VAR AElement, BElement, CElement:
22            SEQ

                -- entrée du flot C

23              Right[col, row] ? CElement

24              SEQ unused = 0 FOR (n-1)-col
25                VAR tmp:
```

```
26                      SEQ
27                          Right[col, row] ? tmp
28                          Right[col+1, row] ! tmp

                    -- calcul

29                  SEQ k = 0 FOR n
30                      SEQ

31                          PAR
32                              Up[col, row] ? AElement
33                              Right[col, row] ? BElement

                            -- calcul d'éléments de C
34                          CElement := CElement + AElement * BElement

35                          PAR
36                              Up[col, row+1] ! AElement
37                              Right[col+1, row] ! BElement

                    -- restituer le flot C

38                  SEQ k = 0 FOR col
39                      VAR tmp:
40                      SEQ
41                          Right[col, row] ? tmp
42                          Right[col+1, row] ! tmp

43                  Right[col+1, row] ! CElement

            -- SORTIE DES CELLULES

            -- sortie horizontale : extraction du flot B et restitution du flot C

44          PAR row = 0 FOR n
45              SEQ

46                  SEQ col = 0 FOR n
47                      Right[n, row] ? BOut[col, row]

48                  SEQ col = 0 FOR n
49                      Right[n, row] ? COut[col, row]

            -- sortie verticale : extraction du flot A

50          PAR col = 0 FOR n
51              SEQ row = 0 FOR n
52                  Up[col, n] ? AOut[col, row]

        -- POST-TRAITEMENT

53      "Lecture de la matrice de sortie : C"
```

Annexe B2 Programme systolique à une dimension après la projection

C = A x B,

où A, B et C sont à dimension n x n.

```
1     VAR AIn[n, n], AOut[n, n],
2           BIn[n, n], BOut[n, n],
3           CIn[n, n], COut[n, n]:

4     CHAN Right[n+1],       -- canaux horizontaux : convertis en une dimension

5     SEQ

        -- PRE-TRAITEMENT

6       "Lecture des matrices d'entrée : A et B"
7       "Initialisation de la matrice de sortie : C"

8       PAR

          -- ENTREE DES CELLULES

9         SEQ

          -- entrée du flot A

10        SEQ row = 0 FOR n                -- transformation de PAR en SEQ
11          SEQ col = 0 FOR n
12            Right[0] ! AIn[col, row]    -- projection de canaux: Up -> Right

          -- entrée du flot C et injection du flot B

13        SEQ row = 0 FOR n          -- transformation de PAR en SEQ
14          SEQ

15            SEQ col = 0 FOR n
16              Right[0] ! CIn[col, row]

17            SEQ col = 0 FOR n
18              Right[0] ! BIn[col, row]

          -- CELLULES DE CALCUL

19        PAR col = 0 FOR n

20          VAR AElement[n], BElement :

21          SEQ

22            SEQ row = 0 FOR n
23              SEQ
```

```
24              Right[col] ? AElement[row]

25              SEQ unused = 0 FOR (n-1)-col
26                VAR tmp:
27                SEQ
28                  Right[col] ? tmp
29                  Right[col+1] ! tmp

30          SEQ row = 0 FOR n
31            VAR CElement:
32            SEQ

              -- entrée du flot C

33              Right[col] ? CElement

34              SEQ unused = 0 FOR (n-1)-col
35                VAR tmp:
36                SEQ
37                  Right[col] ? tmp
38                  Right[col+1] ! tmp

              -- calcul des éléments de C

39              SEQ k = 0 FOR n
40                SEQ

41                  Right[col] ? BElement

                    -- calcul des éléments de C
42                  CElement := CElement + AElement[k] * BElement

43                  Right[col+1] ? BElement

              -- restitution du flot C

44              SEQ unused = 0 FOR col
45                VAR tmp:
46                SEQ
47                  Right[col] ? tmp
48                  Right[col+1] ! tmp

49              Right[col+1] ! CElement

          -- restitution du flot A

50          SEQ row = 0 FOR n
51            SEQ

52              SEQ unused = 0 FOR col
53                VAR tmp:
54                SEQ
55                  Right[col] ? tmp
56                  Right[col+1] ! tmp
```

```
57                    Right[col+1] ! AElement[row]

         -- SORTIE DE CELLULES

58       SEQ

           -- extraction du flot B et restitution du flot C

59         SEQ row = 0 FOR n
60           SEQ

61             SEQ col = 0 FOR n
62               Right[n] ? BOut[col, row]

63             SEQ col = 0 FOR n
64               Right[n, row] ? COut[col, row]

           -- restitution du flot A

65         SEQ row = 0 FOR n
66           SEQ col = 0 FOR n
67             Right[n] ? AOut[col, row]

       -- POST-TRAITEMENT

68     "Lecture de la matrice de sortie : C"
```

Annexe C1 Programme de flots de données

C = A B, où A, B et C sont de dimension n x n.

```
CHAN C0, C1, C2, C3, C4, C5, C6, C7, C8, C9, C10 :

VAR AElement[n,n], BElement[n,n], CElement[n,n]:
SEQ

   "initialisation des matrices A et B"

   PAR

      -- P1: générateur1 de boucle
      -- sortir une séquence d'entiers comme indexes de rang de la matrice A

      SEQ  i = 0 FOR n
         C0 ! i

      -- P2 : générateur2 de boucle
      -- sortir une séquence d'entiers comme indexes de colonne de la matrice B

      SEQ  i = 0 FOR n
         SEQ  j = 0 FOR n
            C2 ! j

      -- P3 :  générateur3 de boucle
      -- sortir une séquence d'entiers comme indexes de la matrice C

      SEQ  i = 0 FOR n
         SEQ
            C3 ! i
            SEQ j = 0 FOR n
               C4 ! j

      -- P4 : recevoir indexes et sortir un rang d'éléments de la matrice A

      VAR index:
      SEQ  i = 0 FOR n
         SEQ
            C0 ? index
            C1 ! AElement[index]    -- un rang d'éléments de A

      -- P5 : recevoir indexes et sortir une colonne d'éléments de la matrice B

      VAR index:
      SEQ  i = 0 FOR n
         SEQ j = 0 FOR n
               SEQ
                  C4 ? index
                  C5 ! BElement[index]
```

```
-- P6 : recevoir indexes et sortir un élément de la matrice A

VAR AElement[n], index:
SEQ  i = 0 FOR n
   SEQ
      PAR
         C1 ?AElement
         C3 ? index
      SEQ j = 0 FOR n
         C6 ! AElement[j]

-- P7 : recevoir indexes et sortir un élément de la matrice B

VAR BElement[n], index:
SEQ  i = 0 FOR n
   SEQ  j = 0 FOR n
      SEQ
         PAR
            C2 ? index
            C5 ? BElement
         SEQ  k = 0 FOR n
            C7 ! BElement[k]

-- P8 : Times : la  multiplication

VAR AElement, BElement:
SEQ  i = 0 FOR n
   SEQ j = 0 FOR n
      SEQ
         C6 ? AElement
         SEQ k = 0 FOR n
            SEQ
               C7 ? BElement
               C8 ! AElement * BElement

-- P9 : Reduce : faire la somme pour les éléments de  C

VAR CElement, tmp:
SEQ  i = 0 FOR n
   SEQ j = 0 FOR n
      SEQ
         SEQ k = 0 FOR n
            SEQ
               C8 ? CElement
               tmp := tmp + CElement
         C9 ! tmp
```

```
-- P10 : Gather1 : collectionner les éléments
-- pour former un rang d'éléments de C

SEQ  i = 0 FOR n
   SEQ
      SEQ j = 0 FOR n
         C9 ? CElement[j]
      C10 ! CElement

-- P11 : Gather2 : collectionner les rangs de C
SEQ i = 0 FOR n
   C10 ? CElement[i]
```

Annexe C2 Parallélisation complète

C = A x B,
où A, B et C sont de dimension n x n.

```
CHAN C0[n], C1[n, n], C2[n, n], C3[n, n], C4[n, n], C5[n, n], C6[n] :

PAR

  -- P1 : sortir un rang d'éléments de la matrice A

  VAR AElement[n,n]:
  "initialisation de la matrice A"
  PAR  i = 0 FOR n
    C0[i] ! AElement[i]          -- un rang d'éléments de A

  -- P2 : sortir  une colonne d'éléments de la matrice B

  "initialisation de la matrice B"
  PAR  i = 0 FOR n
    VAR B[n,n]:
    PAR j = 0 FOR n
      C1[i, j] ! BElement[j]

  -- P3 : sortir un élément de la matrice A

  PAR  i = 0 FOR n
    VAR AElement[n]:
    SEQ
      C0[i] ?AElement
      PAR j = 0 FOR n
        C2[i, j] ! AElement[j]

  -- P4 : sortir un élément de la matrice B

  PAR  i = 0 FOR n
    PAR  j = 0 FOR n
      VAR BElement[n]:
      SEQ
        C1[i, j] ? BElement
        SEQ  k = 0 FOR n
          C3[i, j] ! BElement[k]

  -- P5 : Times : la multiplication : a_ij * b_ij

  PAR  i = 0 FOR n
    PAR j = 0 FOR n
      VAR AElement:
      SEQ
        C2[i, j] ? AElement
        SEQ k = 0 FOR n
          VAR AElement:
          SEQ
```

```
                C3[i, j] ? BElement
                C4[i, j] ! AElement * BElement

-- P6 : Reduce : faire la somme pour les éléments de  C : cij' = cij' + cij

PAR  i = 0 FOR n
  PAR j = 0 FOR n
    VAR tmp:
    SEQ
      SEQ k = 0 FOR n
        VAR CElement:
        SEQ
          C4[i, j] ? CElement
          tmp := tmp + CElement
      C5[i, j] ! tmp

-- P7 : Gather1 : collectionner les éléments de C
-- pour former un rang d'éléments de C

PAR  i = 0 FOR n
  VAR CElement[n]:
  SEQ
    PAR j = 0 FOR n
      C5[i, j] ? CElement[j]
    C6[i] ! CElement

-- P8 : Gather2 : collectionner les rangs de C

VAR C[n,n]:
PAR i = 0 FOR n
  C6[i] ? CElement[i]
```

Annexe D Exemples de l'utilisation du système de transformation

Dans cette annexe, nous illustrons l'utilisation de notre système de transformation par quelques exemples (Cf. §2.2), sur l'interface utilisateur et les services offerts :

- Le premier exemple porte sur un programme simple, au niveau du processus et l'application individuelle, intuitive des lois élémentaires ;

- Le deuxième porte sur la transformation d'un programme fini en sa forme normale (Cf. §2.3), sur le niveau du contexte processus et l'application automatique des lois en fonction d'une stratégie spécifique de transformation ;

- Le troisième concerne l'adaptation du degré de parallélisme par transformation sur le programme de flot de données (§5 et annexe C), sur un niveau mixte de processus et de contexte, avec l'application individuelle et systématique des lois de transformation.

D.1 Lancement d'une session de transformation

Le système de transformation Occam est composé de types de données et de fonctions ML qui réalisent les lois algébriques [§2.2]. Ces types de données et fonctions seront chargés dans le système ML d'Edimburgh, qui est un environnement interactif pour le langage standard ML.

Il existe deux façons de lancer une session de transformation.

(1) Une façon de définir les lois Occam est de lancer d'abords l'interpréteur ML :

fam -h 3000 sml.exp

fam est l'interpréteur Edimburgh du langage ML, le fichier *sml.exp* étant le module système du langage ML, l'option '-h 3000' définit une espace de travail de 3000K Octets.

Et puis charger les programmes sources ML qui définissent les lois Occam comme suit :

```
use ["load.sml"];
```

Le programme*load.sml* contient les commandes qui chargent les programmes ML du système de transformation :

```
(* la syntaxe Occam *)
use ["ex.sml"];
use["UTIL.sml", "occam.sml", "aux1.sml", "print.sml", "eq.sml"] ;

(* module d'analyse syntaxique des programmes Occam *)
use["simpl.sml"];
use["containexp.sml","scexp.sml","subst.sml","contain.sml",
        "scnew.sml","make_using.sml"];
use["parse.sml", "aux2.sml", "aux3.sml"] ;

(* définir les lois élémentaires *)
use["iflaws.sml",
   "altlaws.sml",
   "aslaws.sml",
   "seqlaws.sml",
   "parlaws.sml"];
use["declaws.sml",
   "zerolaws.sml",
   "devlaws.sml",
   "whilelaws.sml"];

(* définit d'autres lois et la stratégie de forme normale *)
use["tact.sml","actnew.sml","aux4.sml","aux4a.sml","s1.sml","chanlaws.sml","strat.sml"];
use["devlaws2.sml","aux5.sml"];
use["makeguard.sml","substproc.sml","aux6.sml","get_proc.sml",
        "subst1s.sml","parassoc.sml","s1_1s.sml","act1s.sml",
        "move_var1.sml","hat.sml","auxsp.sml","ex_pd.sml","ifalt1s.sml"];
use["print_nf.sml", "normal.sml", "convertnf.sml"];

(* définit les nouvelles lois et stratégies *)
(* définit le niveau contexte, le déroulement du corps de boucles etc *)
use["strutil.sml", "depth.sml", "context.sml", "identtype.sml", "lineprint.sml",
        "move.sml", "actions.sml", "transutil.sml", "extras.sml",
        "as.sml", "seq.sml", "tidy.sml", "tactics.sml", "varwhile.sml",
        "newutil.sml", "chanextras.sml", "simpnf.sml", "unwind.sml"];
use["new_laws.sml", "new_strategies.sml"];
```

(2) La commande suivante permet de démarrer une session de transformation plus rapidement :

fam -h 3000 transform.exp

Le fichier*transform.exp* contient une image sauvegardé de l'espace de travail suite au chargement des programmes sources ML de transformation ci-dessus, donc combine le langage ML et les lois, stratégies de transformation Occam en un seul module.

Une image (qui représente l'état courant de l'interpréteur ML) peut être sauvegardée en lançant *fam* avec l'option *-r* :

fam -h 3000 sml.exp -r

et la commande : *w <nom du fichier>* après le chargement des programmes de transformation.

Après le lancement d'une session, nous avons le message suivant :

Occam Transformation System, Version 1.0
-

et les commandes interactives expliquées dans le chapitre 2, section 2.2 peuvent être utilisées sur différents niveaux de syntaxe, comme illustrées dans les exemples qui suivent.

D.2 Quelques exemples

Exemple 1 L'application simple d'une loi

Au niveau processus, nous pouvons appliquer une loi élémentaire sur un processus entier. Le programme segment suivant est une boucle simple, nous voulons dérouler une fois le corps de la boucle en utilisant la loi <WHILE expansion>.

```
- system("cat while1.occ");

WHILE compteur < NOMBRE.LIMITE
   SEQ
      x = produire(valeur)
      c ! x
      compteur := compteur + 1
   -
```

La commande *parse* permet de charger le programme Occam dans le système de transformation :

```
- parse("while1.occ");
```

Le processus obtenu est similaire que le processus original sauf probablement l'information de variables et de canaux a été ajoutée :

```
- view_process(it);

WHILE compteur < NOMBRE.LIMITE
   SEQ
      x = produire(valeur)
      c ! x
      compteur := compteur + 1
-
```

Nous appliquons la loi <WHILE expansion> sur ce processus. Notons que toute loi qui peut être appliquée dans les deux sens est réalisée par deux fonctions ML, correspondant à une utilisation de gauche à droite ou inverse. Ici, nous avons pour la loi <WHILE expansion> deux fonctions : WHILEexpf et WHILEexpb.

```
- apply_law(WHILEexpf, it);
```

Nous avons le programme résultat :

```
- file_process("while1_out.occ");   (* sauvegarder le processus résultat dans un fichier *)
- system("cat while1_out.occ")      (* et l'affichier. Ceci est équivalent à : view_process(it) *)

IF
  compteur < NOMBRE.LIMITE
    SEQ
      x = produire(valeur)
      c ! x
      WHILE compteur < NOMBRE.LIMITE
        SEQ
          x = produire(valeur)
          c ! x
  NOT compteur < NOMBRE.LIMITE
    SKIP

-
```

L'application des lois au niveau processus illustrée dans cet exemple ne permet pas d'appliquer une loi sur un processus composant spécifique à l'intérieur d'un processus. Ceci est possible dans le cas d'une strtégie de transformation ou la manipulation au niveau contexte de processus.

Exemple 2 Transformation d'un programme fini en sa forme normale

Nous avons un programme simple comme suit : il reçoit deux valeurs sur les canaux C1 et C2, et envoie le résultat calculé sur le canal C3,

```
- system("cat parseq.occ");

SEQ
   PAR
      C1 ? x
      C2 ? y
   C3 ! x * y

- parse("parseq.occ");
- view_process(it);

SEQ
   PAR
      -- USING (x as VAR, C1 as INPUT)
      C1 ? x
      -- USING (y as VAR, C2 as INPUT)
      C2 ? y
   C3 ! x * y

- normal_df(it);
- show_nf(it);

VAR x1, x2 :
ALT
   C1 ? x1
      ALT
         C2 ? x2
            ALT
               C3 ! x1 * x2
                  x, y := x1, x2
   C2 ? x2
      ALT
         C1 ? x1
            ALT
               C3 ! x1 * x2
                  x, y := x1, x2
```

normal_df transforme un programme fini complètement en sa forme normale, *show_nf* affiche un programme sous forme normale.

Exemple 3 Transformation d'un programme au niveau contexte processus

A ce niveau, il est possible de sélectionner un processus composant spécifique d'un programme, et sur lequel d'appliquer des lois de transformation. Nous prenons comme exemple le programme du flot de données dans le chapitre 5, section 5.3 (Cf. Annexe C1), pour illustrer l'application des lois sur des contextes processus.

```
- contemplate("dataflow1.occ");
- limitdepth 4;
- view(it);

CHAN C0, C1, C2, C3, C4, C5, C6, C7, C8, C9, C10 :
VAR AElement[n,n], BElement[n,n], CElement[n,n]:
SEQ
   "initialisation des matrices A et B"
   PAR
      -- P1: générateur1 de boucle
      SEQ  i = 0 FOR n
         C0 ! i

      -- P2 : générateur2 de boucle
      SEQ  i = 0 FOR n
         SEQ  j = 0 FOR n
            C2 ! j

      -- P3 :  générateur3 de boucle
      SEQ  i = 0 FOR n
         SEQ
            C3 ! i
            ... SEQ [2 clauses]

      -- P4 : recevoir indexes et sortir un rang d'éléments de la matrice A
      VAR index:
      SEQ  i = 0 FOR n
         SEQ
            C0 ? index
            C1 ! AElement[index]

      -- P5 : recevoir indexes et sortir une colonne d'éléments de la matrice B
      VAR index:
      SEQ  i = 0 FOR n
         SEQ j = 0 FOR n
            ... SEQ [3 clauses]

      -- P6 : recevoir indexes et sortir un élément de la matrice A
      VAR AElement[n], index:
      SEQ  i = 0 FOR n
         SEQ
            ... PAR [3 clauses]
            ... SEQ [2 clauses]
```

```
-- P7 : recevoir indexes et sortir un élément de la matrice B
VAR BElement[n], index:
SEQ  i = 0 FOR n
    SEQ  j = 0 FOR n
        ... PAR [3 clauses]
        ... SEQ [2 clauses]

-- P8 : Times : la  multiplication
VAR AElement, BElement:
SEQ  i = 0 FOR n
    SEQ j = 0 FOR n
        SEQ
            C6 ? AElement
            ... SEQ [4 clauses]

-- P9 : Reduce : faire la somme pour les éléments de  C
VAR CElement, tmp:
SEQ  i = 0 FOR n
    SEQ j = 0 FOR n
        SEQ
            ... SEQ [4 clauses]
            C9 ! tmp

-- P10 : Gather1 : collectionner les éléments
SEQ  i = 0 FOR n
    SEQ
        SEQ j = 0 FOR n
            C9 ? CElement[j]
        C10 ! CElement

-- P11 : Gather2 : collectionner les rangs de C
        SEQ i = 0 FOR n
                C10 ? CElement[i]
```

La commande *contemplate* convertit un programme en type <contexte>, *limitdepth* permet la commande *view* d'afficher le processus courant qu'au niveau syntaxique spécifié. Après le chargement du programme, nous suivons la transformation effectuée dans la section §5.3 :

- Normalisation des variables par la commande *pushVARall* sur le processus.

- Ajustement du parallélisme. Ceci est réalisé par
    (1) séléctionner le contexte spécifique ;
    (2) appliquer des lois appropriées.

Par exemple, pour le processus P9, nous le sélectionnons par la commande:

```
- zoom_in 1;  (* séléctionner le premier PAR comme le contexte courant *)
- zoom_in 8;  (* P9 comme le contexte courant. Compter à partir de zéro *)
```

Puis nous appliquons des lois <SEQ-PAR *> sur le contexte séléctionné par la commande suivante :

```
- apply_p_law(PARSEQcom3);
```

pour le transformer en P6 dans le programme résultat. Les autres lois nécessaires concernent celles du renommage de variables et de canaux.

```
- limitdepth 4;
- view (it);

CHAN C0[n], C1[n, n], C2[n, n], C3[n, n], C4[n, n], C5[n, n], C6[n] :
PAR

  -- P1 : sortir un rang d'éléments de la matrice A
  SEQ
    VAR AElement[n,n]:
    "initialisation de la matrice A"
    PAR  i = 0 FOR n
      C0[i] ! AElement[i]          -- un rang d'éléments de A

  -- P2 : sortir  une colonne d'éléments de la matrice B
  SEQ
    "initialisation de la matrice B"
    PAR  i = 0 FOR n
      VAR B[n,n]:
      PAR j = 0 FOR n
        C1[i, j] ! BElement[j]

  -- P3 : sortir un élément de la matrice A
  PAR  i = 0 FOR n
    VAR AElement[n]:
    SEQ
      C0[i] ?AElement
      PAR j = 0 FOR n
        C2[i, j] ! AElement[j]

  -- P4 : sortir un élément de la matrice B
  PAR  i = 0 FOR n
    PAR  j = 0 FOR n
      VAR BElement[n]:
      SEQ
        C1[i, j] ? BElement
        ... SEQ [2 clauses]

  -- P5 : Times : la multiplication : aij * bij
  PAR  i = 0 FOR n
    PAR j = 0 FOR n
      VAR AElement:
      SEQ
        C2[i, j] ? AElement
        ... SEQ [5 clauses]

  -- P6 : Reduce : faire la somme pour les éléments de  C : cij' = cij' + cij
  PAR  i = 0 FOR n
    PAR j = 0 FOR n
      VAR tmp:
      SEQ
        ... SEQ [5 clauses]
        C5[i, j] ! tmp
```

```
-- P7 : Gather1 : collectionner les éléments de C
PAR  i = 0 FOR n
    VAR CElement[n]:
    SEQ
        PAR j = 0 FOR n
            C5[i, j] ? CElement[j]
        C6[i] ! CElement

-- P8 : Gather2 : collectionner les rangs de C
VAR C[n,n]:
PAR i = 0 FOR n
    C6[i] ? CElement[i]
```

D.3 Performances

Nous donnons ici les performances du système de transformation, en ce qui concerne principalement sur l'aspect du temps de calcul (réponse) pour quelques commandes représentatives. Ces estimation et mesure sont relative à une implantation sur station SUN-3 à charge faible.

- Le temps de calcul de la forme normale d'un programme par les commandes *normal_df* et *normal_bf* dépend de la taille et de la profondeur des structures emboîtées (surtout les constructions PAR, qui doivent être éliminées par les lois <PAR expansion*>). Pour un programme de taille moyenne (à l'ordre de 1 page, comme l'exemple dans la section §2.3.3), le temps de réponse est de quelques à dizaine de secondes. La taille de programme résultat est presque exponentielle par rapport à sa taille originale. Donc des programmes qui ont été traités pour la vérification formelle sont encore simples et ont une taille modeste (Cf. 2.3.3).

- L'application d'une loi. Prenons l'exemple de vérification formelle (§2.4). Toute transformation par l'application d'une loi peut se faire dans notre système de transformation, sauf la transformation et la comparaison des expressions logiques, qui sont faites manuellement par des commandes de substitution d'expressions. L'application d'une loi sur un tel programme relativement simple et de taille modeste comme ceci prend un temps acceptable, de l'ordre de quelques secondes.

D.4 Un listage séléctif de programmes sources de transformation

occam.sml

```
datatype identifier = ident of string
and num = INT of int | HEX of string;

datatype assoc_arithop = plus | times;

datatype arithmetic_op = as_arith of assoc_arithop |
                di_minus|
                divis |
                rem
and comparison_op = lt | gt | le | ge | ne | eq
and logical_op = ∧ | ∨ | ><
and boolean_op = AND | OR
and shift_op = << | >>
and monadic_op = mon_minus | NOT;

datatype assoc_op =   arith_as of assoc_arithop |
              log_as of logical_op |
              bo_as of boolean_op;

datatype operator = ar_op of arithmetic_op |
             com_op of comparison_op |
             log_op of logical_op |
             bo_op of boolean_op |
             sh_op of shift_op;

datatype variable = var of identifier |
               var_sub of identifier * subscript |
               bv of channel * int

and channel  = chan of identifier | chan_exp of identifier * expression

and subscript = byte_sub of expression | subst of expression

and vector_constant = tab_vc of table | str_vc of string

and item = var_item of variable | vc_item of vector_constant * subscript

and table = byte_tab of expression * (expression list) |
        tab of expression * (expression list)

and element = num_el of num | item_el of item | TRUE | FALSE |
          char_const of string | ex_el of expression

and expression = sing_el of element |
             as_ex of assoc_op * (element list) |
             op_ex of element * operator * element |
             mon_ex of monadic_op * element;

datatype const_def = exp_const of (identifier * expression) |
               vc_const of (identifier * vector_constant);

datatype slice = byte_sl of identifier * expression * expression |
             sl of identifier * expression * expression;
```

```
datatype input = input_var of channel * (variable list) |
             input_vec of channel * slice |
             input_any of channel;

datatype output = output_exp of channel * (expression list) |
              output_vec of channel * slice |
              output_any of channel;

datatype replicator = rep of identifier * expression * expression;

datatype parm = sing_parm of identifier | mult_parm of identifier;

datatype form_parm = varfp of parm list |
                 chanfp of parm list |
             valfp of parm list;

datatype a_parm = ex of expression | ch of channel;

datatype a_parm_list = apl of a_parm list;

datatype claim =   ownchan of channel list |
           inchan of channel list |
           outchan of channel list |
           loc_inchan of channel list |
           loc_outchan of channel list |
           varc of variable list;

datatype par_dec = pd of claim list;

datatype    guard = exp_inputguard of expression * input |
             exp_outputguard of expression * output |
             exp_skipguard of expression |
             skip_guard |
             input_guard of input |
             output_guard of output;

datatype process = assign of (variable list) *  expression list |
               assign_vec of slice * slice |
               input_proc of input |
               output_proc of output |
               SKIP |
               STOP |
               ZERO |
              proc_call of identifier *
                      a_parm_list *
                      (variable list * variable list) *
                      (variable list * expression list) *
                      (channel list * channel list) |
             dec of declaration * process |
              seq_con of process list |
             par_con of (parproc list) * (variable list * variable list) |
             alt_con of guarded_process list |
              if_con of conditional list |
             seq_rep of replicator * process |
             par_rep of replicator * parproc |
             alt_rep of replicator * guarded_process |
             if_rep of replicator * conditional |
              while_con of expression * process |
              if_alt of int * (variable list) * process
```

```
and parproc = parp of par_dec * process

and declaration  = var_dec of variable list |
                   chan_dec of channel list |
                   const_dec of const_def list |
                   proc_dec of identifier *
                            (form_parm list) *
                             process *
                            par_dec

and guarded_process = sim_guard of guard * process |
                      alt_guard of guarded_process list |
                      rep_guard of replicator * guarded_process

and conditional     = sim_cond of expression * process |
                      if_cond of conditional list |
                      rep_cond of replicator * conditional;
```

seqlaws.sml

```
(* SEQ-SKIP unit *)

fun SEQSKIPf(seq_con[]) = SKIP |
   SEQSKIPf P = raise fail_law with ("SEQSKIPf",P);

fun SEQSKIPb SKIP = seq_con[] |
   SEQSKIPb P = raise fail_law with ("SEQSKIPb",P);


(* SEQ assoc *)

fun SEQassocf(seq_con(p::l)) = seq_con[p,seq_con l] |
   SEQassocf P = raise fail_law with ("SEQassocf",P);

fun SEQassocb(seq_con[p,seq_con l]) = seq_con(p::l) |
   SEQassocb P = raise fail_law with ("SEQassocb",P);


(* SEQ-IF distrib *)

fun SEQIFdist(P as seq_con[if_con l,q]) =
  let fun f(sim_cond(b,p)) = sim_cond(b,seq_con[p,q])
  in if_con(map f l)
  end handle ? => raise fail_law with ("SEQIFdist",P) |
   SEQIFdist P = raise fail_law with ("SEQIFdist",P);
```

parlaws.sml

```
(* PAR SKIP unit *)

fun PARskipf(par_con ([],_)) = SKIP |
   PARskipf P = raise fail_law with ("PARskipf",P);

fun PARskipb(SKIP) = par_con ([],([],[])) |
   PARskipb P = raise fail_law with ("PARskipb",P);


(* PAR assoc *)

fun PARassocf (par_con (h::t,hl)) =
  par_con([h,parp((claim_union t),par_con(t,hl))],hl) |
   PARassocf P = raise fail_law with ("PARassocf",P);

fun PARassocb (P as par_con([h,parp(u,par_con (l,_))],hl)) =
  let val u1 = claim_union l
      val (inl1,outl1,ownl1,vl1)=declare_pd1 u1
      val (inl2,outl2,ownl2,vl2)=declare_pd1 u
  in if eql(inl1,inl2,eqch) andalso eql(outl1,outl2,eqch) andalso
       eql(ownl1,ownl2,eqch) andalso eql(vl1,vl2,eqv) then
       loc_pd(par_con(h::l,hl))
     else raise fail_law with ("PARassocb",P)
  end |
   PARassocb P = raise fail_law with ("PARassocb",P);



(* PAR expansion1 *)

fun PARexp1(P as par_con([parp(u1,alt_con l),
                   (p2 as parp(u2,assign _))],h)) =
  let val s1 = set_diff(eqch,outs u1,ins u2)
      val s2 = set_diff(eqch,ins u1,outs u2)
      fun f(sim_guard(g,p),l) =
        (sim_guard(subst_guardv h (check_guard(g,s1,s2)), (* un-hat variables
                                                       *)
                par_con([parp(u1,p),p2],h)) :: l)
        handle check_guard => l
  in alt_con(red f l [])
  end handle ? => raise fail_law with ("PARexp1",P) |
   PARexp1 P = raise fail_law with ("PARexp1",P);
```

whilelaws.sml

```
(* WHILE expansion *)

fun WHILEexpf(while_con(b,p)) =
  let val b = simpl b
  in if_con[sim_cond(b,seq_con[p,while_con(b,p)]),
         sim_cond(simpl(neg b),SKIP)]
  end |
   WHILEexpf p = raise fail_law with ("WHILEexpf",p);

fun WHILEexpb(p as if_con[sim_cond(b1,seq_con[p1,while_con(b2,p2)]),
                  sim_cond(b3,SKIP)]) =
  if eq_exp(b1,b2) andalso eq_exp(b1,neg b3) andalso eqp(p1,p2) then
    while_con(simpl b1,p1)
  else
   raise fail_law with ("WHILEexpb",p) |
   WHILEexpb p = raise fail_law with ("WHILEexpb",p);


(* WHILE combine *)

fun WHILEcombf(while_con(b1,while_con(b2,p))) =
  while_con(simpl(disj(b1,b2)),if_con[sim_cond(simpl b2,p),
                                sim_cond(sing_el TRUE,ZERO)]) |
   WHILEcombf p = raise fail_law with ("WHILEcombf",p);

fun WHILEcombb(p as while_con(b1,if_con[sim_cond(b,p1),
                                sim_cond(sing_el TRUE,ZERO)])) =
  let val (b1,b2) = get_disj b1
  in if eq_exp(b,b2) then
      while_con(simpl b1,while_con(simpl b2,p1))
    else
      raise wrong_law
  end handle ? => raise fail_law with ("WHILEcombb",p) |
   WHILEcombb p = raise fail_law with ("WHILEcombb",p);
```

context.sml

```
(* context.sml : réalise le contexte processus et manipulation *)
 abstype 'a stack = stack of 'a list with
 ; val empty = stack []
 ; fun push x (stack s) = stack (x :: s)
 ; exception stack_empty
 ; fun pop (stack []) = raise stack_empty |
       pop (stack (x :: s)) = (x,stack s)
 ; fun topofstack s = fst (pop s)
 end;

 abstype ('control,'value) context =
    ctx of ( ('control -> 'value -> 'value)
          * (('control * 'value) -> 'value -> 'value)
          * (('value * (('control * 'value) stack)) ref) ) with
 ; exception outermost_context
 ; fun new_context (inf,outf) v = ctx (inf,outf,ref (v,empty))
 ; fun zoom_in x (context as ctx (inf,_,c as ref (v,s))) =
      let val v' = inf x v
        ; val s' = push (x,v) s
       in (c := (v',s'); context) end
 ; fun pan_out (context as ctx (_,outf,c as ref (v,s))) =
     let val (xv,s') = pop s handle stack_empty => raise outermost_context
        ; val v'      = outf xv v
       in (c := (v',s'); context) end
 ; fun re_take (context as ctx (inf,outf,c as ref (_,s))) =
     let val ((x,v),s') = pop s handle stack_empty => raise outermost_context
        ; val v' = inf x v
       in (c := (v',s); context) end
 ; fun modify_context modify (context as ctx (_,_,c as ref (v,s))) =
      let val v' = modify v
     in (c := (v',s); context) end
 ; fun last_control (ctx (_,_,ref(_,s))) =
     fst (pop s) handle stack_empty => raise outermost_context
 ; fun current_value (ctx (_,_,ref(v,_))) = v
 end;
```

Bibliographie

[AFR80] K.R. Apt, N. Francez et W.P. de Roever, *A proof system for communicating sequential processes,* Trans. Prog. Lang. Syst. 2, 1980.

[AHU74] A.V. Aho, J.E. Hopcroft, J.D. Ullman, *The Design and analysis of Computer algorithms*, Addison-Wesley, 1974.

[Apt83] K.R. Apt, *Formal justification of a proof system for communicating sequential processes*, JACM Vol. 30, No. 1, 1983.

[Arsac79] J. Arsac, *Syntactic source to source transformations and program manipulation*, Commun ACM 22, 1979.

[Arsac83] J. Arsac, *Les bases de la programmation*, DUNOD informatique, 1983.

[AS83] G.R.Andrews et F.B.Schneider, *Concepts and Notations for Concurrent Programming*, ACM Computing Surveys, Vol. 15, No.1, March 1983.

[ASU86] Aho, Sethi, Ullman, *Compilers : Principles, Techniques and Tools,* Addison Wesley, 1986.

[Back78] J. Backus, *Can Programming be liberated from the Von-Neumann style? A functional style and its algebra of programs*, CACM 21, 1978.

[Baia84] F.Baiardi et al, *Static Checking of Interprocess Communication in ECSP*, SIGPLAN Notice Vol.19, No.6, June 1984.

[Bail89] C.F. Baillie, *Comparing shared and distributed memory computers*, Parallel Computing 8, 1989.

[Bars79] D.R. Barstow, *The role of knowledge and deduction in program synthesis*, proceedings of the 6th international joint conf. on AI, 1979.

[Bau89] F.L.Bauer et al, *Formal Program Construction by Transformations Computer-Aided, Intuition-Guided Programming,* IEEE Transactions on software engineering, Vol.15,No.2,Feb. 1989.

[BD77] R.M. Burstall, J. Darlington, *A transformation system for developping recursive programs*, J.ACM 24, 1977.

[BFKK90] V. Balasundaram, G. Fox, K. Kennedy, U. Kremer, *An interactive environment for data partitionning and distribution.* Proc. 5th Distributed Memory Comput. Conf. 1990.

[BGW76] R. Balzer, N. Goldman, D. Wilde, *On the transformational implementation approach to programming*. Proceedings of 2nd International conference on software engineering, IEEE, N.Y. 1976.

[BK89] V.Balasundaram et K.Kenndy, *A Technique for Summarizing Data Access and Its Use in Parallelism Enhancing Transformations*, ACM 1989.

[BKPR87] K.C. Bowler, R.D. Kenway, G.S. Pawley, D.Roweth, *An introduction to*

OCCAM 2 Programming, Chartwell-Bratt Ltd, 1987.

[Boyle80] J.M. Boyle, *Software adaptability and program transformation*, Software Engineering, Academic Press, N.Y. 1980.

[BR84] S.D. Brookes, A.W. Roscoe, *An improved model for communicating processes*. LNCS, 1984.

[Broo83] S.D. Brookes, *A model for communicating sequential processes*. Thesis, Oxford University, 1983.

[BS83] G.N. Buckley et A. Silberschatz, *An effective Implementation for the Generalised Input-Output Construct of CSP*, ACM Trans. Prog. Lang. Syst. Vol 5, 1983.

[Burns89] Alan Burns, *Programming in occam 2*, University of Bradford, Addison-Wesley, 1989.

[Cami86] J. Camilleri, *An Operational Semantics for Occam*, PRG, Oxford Univ, 1986.

[Capon89] P.C. Capon, *Experiments in Algorithmic Parallelism*, Parallel Computing, 1989.

[CF89] R.Cartwright et M.Felleisen, *The Semantics of Program Dependence*, ACM 1989.

[Chea81] T.E. Cheatham, *Overview of the Harvard program development system*, Software Engineering Environments, Ed. H. Hünks, 1981.

[CLS89] R. Candlin, Q. Luo, N. Skilling, *The Investigation of Communications Patterns in Occam Programs*, Univ. Edinburgh, 1989.

[Cytr89] R.Cytron et al, *Automatic Generation of DAG Parallelism*, ACM 1989.

[DHD72] O.J. Dahl, C.A.R. Hoare, E.W. Dijkstra, *Structured Programming*, Academic Press, 1972.

[DHKL80] V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, *Programming Environments based on structured editors : The MENTOR Experience*. Report 26, INRIA, 1980.

[Dijk76] E.W. Dijkstra, *A Discipline of programming*, Prentice-Hall, Englewood Cliffs, N.J, 1976.

[Dong88] J.J.Dongarra et al, *Programming methodology and performance issues for advanced computer architectures*, Parallel Computing 8, 1988.

[Eudes88] J. Eudes et al, *PDS: Advanced Program Development System for Transputer Based Machines*, OUG Proceedings, IOS Amsterdam 1988.

[Eudes90] J. Eudes, *PDS : Un système du Développement de Programmes*, Thèse, IMAG, 1990.

[Feat82] M. Feather, *A system for assisting program transformation*,

ACM Trans. Prog. lang. syst. 4, 1982.

[Feaut89] P. Feautrier, *Asymptotically efficient algorithms for parallel architectures*, MASI, Univ Paris 6, June 1989.

[Gaud88] J.L.Gaudiot et al, *Program graph allocation in distributed multicomputers*, Parallel Computing 7, 1988.

[Gilo87] W. Giloi, *Interconnexion Networks for Parallel architectures*, LNCS, 1987.

[GJ89] M. Goldsmith, G. Jones, *Programming in Occam 2*, Prentice-Hall, 1989.

[GL89] J.L. Gaudiot et L.T. Lee, *Occamflow, A Methodology for Programming Multiprocessor Systems*, Journal of Parallel and Distributed Computing 7, 1989.

[Gold87] M. Goldsmith, *occam Transformations at Oxford*, OUG Proceedings, 1987.

[Gold88] M. Goldsmith, *The Oxford occam Transformation System*, PRG, Oxford University, January 1988.

[Green77] C. Green, *A Summary of the PSI program synthesis system.* Proc. 2nd Conf. on Software Engineering, SF. Ca. 1976.

[Gries81] David Gries, *The Science of Programming*, Springer Verlag, 1881.

[HBR84] C.A.R. Hoare, S.D. Brookes and A.W. Roscoe, *A theory of communicatingsequential processes*, Journal of the ACM, 1984.

[Hey89] A. J. G. Hey, *Experiments in MIMD parallelism*, Parallel computing, 1989.

[Hoare78] C. A. R. Hoare, *Communicating sequential processes.*, Communications ACM, 21, 8(Aug. 1978) 666-777

[Hoare80] C. A. R. Hoare, *A model for communicating sequential processes.*, PGR-22, Oxford Univ, 1980

[HR84] C.A.R. Hoare, A.W. Roscoe, *Programs as executable predicates*, Proceedings of the international conference on fifth generationcomputer systems, ICOT, 1984.

[HR85] C.A.R. Hoare, A.W. Roscoe, *The laws of occam programming*, PRG, Oxford University, 1985.

[Inmos84] Inmos, *The occam programming manual*, Prentice-Hall International, 1984.

[Inmos88] Inmos, *The occam 2 Reference Manual*, Prentice-Hall International, 1988.

[Inmos88] Inmos, *Transputer Reference Manual,*
Prentice-Hall International, 1988.

[Inmos88] Inmos, *Transputer instruction set: a compiler writer's guide,*
Prentice Hall, 1988.

[Inmos88] Inmos, *The Transputer programming developement system,*
1988.

[Jess88] C.Jesshope, *Transputers and switches as objects in OCCAM*,
Parallel Computing 8, 1988.

[Jones88] G. Jones, *Support for Occam Channels via Dynamic Switching in Multi-Transputer Machine*, PRG, Univ of Oxford 1988.

[Jones87] G. Jones, *Programming in Occam,*
Prentice Hall, 1987.

[Jord88] H.F.Jordan, *Programming language concepts for multiprocessors*,
Parallel Computing 8, 1988.

[KM89] O. Krämer, H. Müllenbein, *Mapping strategies in message-based multiprocessor systems*. Parallel Computing, 1989.

[Knuth71] Donald Knuth, *The Art of Computer Programming*, 1971.

[KoMe91] C. Koelbel, P. Mehrotra, *Compiling Global Name-Space Parallel Loops for Distributed Execution*, IEEE Trans. on Paral. and Distrib. Sys. 1991.

[KR89] A.Kaldewaij et M.Rem, *A derivation of a systolic rank order filter with constant response time*, Parallel Computing 1989.

[Kerr87] Jon Kerridge, *Occam Programming : a Practical Approach,*
Blackwell Scientific Publications, 1987.

[Krem88] U.Kremer et al, *Advanced tools and techniques for automatic parallelization* .
Parallel Computing 7, 1988.

[Leng88] C.Lengauer, *Towards Systolizing Compilation: an Overview,*
LNCS, 1988.

[LM87] G.J. Lispovski, M. Malek, *Parallel Computing : Theory and Comparisons,*
John Wiley & Sons, 1987.

[Love77] D.B. Loveman, *Program improvement by source-to-source transformation,*
JACM 24, 1977.

[LS88] F.C.M Lau, K.M. Shea, *Mapping a process Network onto a Processor Network*, OUG-9, IOS, 1988.

[LS89] C. Lengauer et J.W. Sanders, *The Projection of Systolic Programs,*
LNCS, 1989.

[May88] D. May et al, *Communicating Process Architecture: Transputer and Occam,*
LNCS, 1988.

[MC89] F.Mourlin et E.Cournarie, *A Graphical Environment for Occam Programming*, Rapport de Recherche n 481, Avril 1989.

[Miln80] R. Milner, *A calculus of communicating systems*, LNCS, n°92, 1980.

[MS89] D.May et D.Shepherd, *Towards totally verified systems*, LNCS, 1989.

[Muhl88] H. Mühlenbein et al, MUPPET: *A programming environment for message-based multiprocessors*, Parallel Computing 8 (1988).

[Munt85] T. Muntean, *Introduction à occam: un langage pour la programmation parallèle,* rapport de recherches, IMAG, 1985.

[Munt88] T. Muntean et al, *PARX: A Parallel Operating System for Transputer Based Machines*, OUG Proceedings, IOS Amsterdam 1988.

[MW77] Z. Manna et R. Waldinger, *The automatic synthesis of recursive programs*, Proceedings of Symposium on AI and Prog. Lang, SIGPLAN, ACM, 1977.

[MW79] Z. Manna et R. Waldinger, *Synthesis : Dreams => Programs*, IEEE Trans. Soft. Eng., Vol. SE-5, No 4, 1979.

[OI88] H. OHARA, H. IIZUKA, *A Preprocessor to augment the Description of Occam Processes for Multiprocessor Machines.* OUG-9, IOS, 1988.

[Part84] H. Partsch, *The CIP transformation system*, Prog. Trans. and Prog. Env. LNCS, 1984.

[Perr87] R.H. Perrot, *Parallel Programming*, Addison-Wesley, 1987.

[Pepp84] P. Pepper, *Program Transformation and Program Environments*, LNCS, 1984.

[PM87] D. Pountain, D. May, *A Tutorial Introduction to Occam Programming*, Blackwell Scientific Publications, Oxford, 1987.

[PS83] H.Partsch et R.Steinbrüggen, *Program Transformation Systems*, ACM Computing Surveys, Vol.15, No.3, September 1983.

[Quint86] P. Quinton, *An Introduction to Systolic Architectures.*, INRIA, 1986.

[Red88] U.S. Reddy, *Transformational Derivation of Programs Using the Focus system*, ACM SIGPLANS, 1988.

[RP89] A.Rogers et K.Pingali, *Process Decomposition Through Locality of Reference*, ACM 1989.

[Rosc84] A.W. Roscoe, *Denotational semantics for occam*, LNCS, n°197, 1984.

[Rosc82] A.W. Roscoe, *A methematical theory of communicating processes*, D.Phil. thesis, Oxford University, 1982.

[RS91] J. Ramanujam, P. Sadayappan, *Compile-time Techniques for Data Distribution in Distributed Memory Machines.* IEEE Trans. on Paral. and Distrib. Sys. 1991.

[SC91] J.P. Sheu, C.Y. Chang, *Synthesizing Loop Algorithms Using Nonlinear Transformation Method,* IEEE Trans. On Paral. Distrib. Sys. 1991.

[Schw88] K.Schwan et al, *A Language and System for the Construction and Tuning of Parallel Programs*, IEEE Transaction on Software Engineering, Vol. 14, No.4, Apr. 1988.

[Shar81] M. Sharir, *Formal integration: A program transformation technique*, Computer Languages 6, 1981.

[SKN76] T.A. Standish, D.F. Kibler, J.M. Neighbors, *Improving and Refining programs by program manipulation*, Proceedingss of Annal Conference, ACM, NY. 1976.

[Sorg87] P.Sorgaard, *Programming Environments and System Development Environments* , 1987.

[SP84] R. Steinbrüggen, H. Partsch, *Mathematical foundation of transformation systems, Tech Report,* Univ München, 1984.

[TD89] R.W.S Tregidgo, A.C. Downton, *Processor farm Analysis and Simulation for Embedded Parallel Processing Systems*, Univ of Essex, 1989.

[TM90] E-G. Talbi, T. Muntean, *Placement statique de processus sur une architecture parallèle*. Rapport de recherche, LGI-IMAG.

[Trel82] P.C.Treleaven et al, *Data-Driven and Demand-Driven Computer Architecture*, Computing Surveys, Vol.14, No.1, March 1982.

[Trel88] P.C. Treleaven, *Parallel architecture overview*, Parallel Computing 8, 1988.

[Wail91] Ph. Waille, *Architecture de Machines Parallèles*, Thèse, IMAG, 1991.

[Walk86] D.J. Walker, *An Operational Semantics for CSP*, MSc Thesis, PRG, 1986.

[WC89] A. West, P. Capon, *A High level software environment for Transputer based systems*, Univ Manchester, 1989.

[Wirt71] N. Wirth, *Program Development by stepwise refinement*, CACM 14, 1971.

[WL91] M.E. Wolf, M.S. Lam, *A Loop Transformation Theory and an Algorithm to Maximize Parallelism*, IEEE Trans. on Paral. and Distrib. Sys. 1991.

[Yeh77] R.T. Yeh, *Current trends in programming methodology*, Prentice-Hall, 1977.

Un système de transformation des programmes pour leur exécution sur machines parallèles

Xiaobo YU

Mots clés : programmation transformationnelle, programmation parallèle, système de transformation, lois algébriques de transformation, vérification formelle, placement de programmes sur machine parallèles à mémoire distribuée

Abstract

In parallel programming for massively parallel machines, the necessary software support for exploiting efficiently the available physical ressources has been a field of urgent research and development. In particular, interest is increasing in the transformational approach for supporting the parallel program development process.

We propose a program transformation system which provides formal methods and techniques for correct manipulation of parallel programs. The system consists of a collection of transformation rules represented by a rich and powerful set of algebraic laws of Occam and transformation strategies for achieving special goals. The transformation rules and strategies render the system an interactive environment for automatic program manipulation with controlled strategies.

Several important aspects of parallel programming in a distributed memory machine environment have been studied and investigated in our transformational approach, from program adaptation to a parallel architecture, program optimisation such as parallelism extraction and enhancement, distribution of global data structures, to the implementation of a process/processor mapping strategy.

Several examples of application are shown.