THESE

présentée à

L'UNIVERSITE JOSEPH FOURIER - GRENOBLE I

pour obtenir le grade de

DOCTEUR

spécialité :

INFORMATIQUE

par

Hubert GARAVEL

sujet de la thèse

COMPILATION ET VERIFICATION DE PROGRAMMES LOTOS

soutenue le 23 novembre 1989 devant le jury composé de :

MM.	**J.-P. VERJUS**	Président
	G. BERRY	Rapporteurs
	E. BRINKSMA	
	J. BARRE	Examinateurs
	J. SIFAKIS	
	J. VOIRON	

Thèse préaprée au sein du Laboratoire de Génie Informatique de Grenoble
en collaboration avec l'Institut National de Recherche en Informatique et en Automatique

Résumé : Lotos (*Language Of Temporal Ordering Specification*) est un langage de description de systèmes parallèles communicants, normalisé par l'ISO et le CCITT afin de permettre la définition formelle des protocoles et des services de télécommunications. Le langage utilise des types abstraits algébriques pour spécifier les données et un calcul de processus proche de CSP et CCS pour exprimer le contrôle.

Cette thèse propose une technique de compilation permettant de traduire un sous-ensemble significatif de Lotos vers un modèle réseau de Petri interprété (pouvant servir à produire du code exécutable) puis vers un modèle automate d'états finis (permettant la vérification formelle de programmes Lotos soit par réduction ou comparaison modulo des relations d'équivalence, soit par évaluation de formules de logiques temporelles).

La méthode employée diffère des approches usuelles basées sur la réécriture de termes, qui construisent directement le graphe d'états correspondant à un programme Lotos. Ici au contraire la traduction est effectuée en trois étapes successives (expansion, génération et simulation) s'appuyant sur des modèles sémantiques intermédiaires (le langage SubLotos et le modèle réseau). Elle met en œuvre une analyse statique globale du comportement des programmes. Elle prend en compte les données, celles-ci devant être compilées au moyen d'algorithmes déjà existants.

Ces principes de compilation ont été entièrement implémentés dans le logiciel Cæsar. Les performances obtenues confirment l'intérêt de la méthode.

Mots-clés : LOTOS, validation, protocole, réseau de Petri, compilation, automate, vérification, logique temporelle

Abstract: LOTOS (*Language Of Temporal Ordering Specification*) is a language for the description of concurrent and communicating systems, standardized by ISO and CCITT to allow formal definition of telecommunication protocols and services. LOTOS is based on algebraic abstract types to specify data structures and on a process calculus, close to CSP and CCS, to express control structures.

This thesis proposes a compiling technique which allows to translate a significant subset of LOTOS into both interpreted Petri nets (which may serve as a basis for executable code generation) and finite state automata (which allow formal validation of LOTOS programs, either by reduction and comparison according to equivalence relations, or by evaluation of temporal logics formulas).

The method is different from existing approaches, based on term rewriting, that directly build the state graph corresponding to a given Lotos program. Conversely, translation is achieved by three successive steps (expansion, generation and simulation) dealing with intermediate semantic models (namely SubLotos language and networks). It involves a static and global analysis of program behaviours. It can handle data structures, that have to be compiled by already known algorithms.

These compiling principles are fully implemented in the software tool Cæsar. Its demonstrated performances confirm the interest of the approach.

Keywords: LOTOS, validation, protocol, Petri net, compilation, automaton, verification, temporal logic

ISBN - 2 - 7261 - 0621 - 8

Remerciements

Je tiens à remercier :

Jean-Pierre Verjus, *Directeur de recherche au C.N.R.S., Directeur de l'Institut I.M.A.G., pour son soutien et pour l'honneur qu'il m'accorde en présidant le jury de cette thèse*

Gérard Berry, *Maître de recherches à l'Ecole Nationale Supérieure des Mines de Paris, pour avoir accepté de juger ce travail. Il faut convenir que ses remarques sur le mécanisme des* ε*-transitions ont eu d'heureuses conséquences sur les performances de* CÆSAR

Ed Brinksma, *Professeur associé à l'Université de Twente (Pays-Bas), que son activité déterminante au sein de l'ISO autorise à considérer comme le père spirituel du langage* LOTOS. *C'est assez dire combien je suis sensible à sa présence dans ce jury, ainsi qu'à l'attention remarquable qu'il a portée à la lecture de ce document*

Jacky Barré, *Directeur de recherche à l'I.N.R.I.A., qui m'a accueilli chaleureusement dans son équipe pendant toute la première partie de ce travail, et pour la confiance et les encouragements qu'il m'a témoignés par la suite*

Joseph Sifakis, *Directeur de recherche au C.N.R.S., pour m'avoir accordé les moyens de mener à bien ce travail, au sein du projet SPECTRE de l'I.M.A.G. Il a su donner à mes efforts une impulsion et une orientation convenables, et ce n'est jamais en vain que j'ai fait appel à lui pour surmonter les difficultés rencontrées. Qu'il trouve ici un nouvel aboutissement — parfois inattendu — des idées qu'il a défendu et continue d'illustrer avec persévérance*

Jacques Voiron, *Maître de conférences à l'Université Joseph-Fourier, qui a assumé la direction de cette thèse, qui a su m'aider à dégager les points essentiels de ce travail, et qui est parvenu à me faire renoncer au style pamphlétaire dans la rédaction du manuscrit*

*De mon séjour à l'I.N.R.I.A, je n'oublierai pas la gentillesse d'*Edmonde Duteurtre, *d'*Ahmed Serhrouchni *et de* José Queiroz, *ni la bienveillance de* Pierre Boullier *et de* Laure Reinhart.

Ma reconnaissance va ensuite à Ahmed Bouajjani, Jean-Claude Fernandez, Suzanne Graf, Xavier Nicollin *et* Carlos Rodríguez *du Laboratoire de Génie Informatique de l'I.M.A.G., dont les critiques et les suggestions ont exercé une influence que je me plais à reconnaître.*

J'ai gardé d'excellents souvenirs de la collaboration avec Pascal Bouchon, Jean-Michel Houdoin *et* Christian Bard *qui, dans le cadre de leurs études, ont participé au projet* CÆSAR.

Je remercie également ceux qui, à un titre ou à un autre, m'ont apporté leur aide, notamment Paul Amblard, Saddek Bensalem, Pierre Berlioux, Philippe Bizard, Nicolas Halbwachs, Jean-Michel Hufflen, Pierre Laforgue, Fabienne Lagnier, Philippe Leblanc, Michel Lévy, Christophe Mauras, Daniel Pilaud, John Plaice, Anne Rasse *et* Philippe Schnoebelen.

Je serais heureux si mes parents pouvaient voir dans cette étude un peu abstraite le résultat de leur affection et de leur dévouement.

A celles et ceux qui m'appellent Béru, j'adresse enfin mes salutations les plus réglementaires.

Ce document a été rédigé en TEX/LATEX en utilisant le préprocesseur français CorTEX développé par l'auteur. Les figures ont été décrites en PIC, traduites en PostScript grâce au logiciel DevPs, puis intégrées au restant du texte en utilisant l'outil Dvi2Ps inclus dans la distribution TPS.

Table des matières

Introduction **1**

Notations **11**

I Représentations de programmes 15

1 Graphe **17**

1.1 Présentation du modèle graphe . . . 17
1.2 Syntaxe du graphe . . . 20
1.2.1 Graphe — Automate . . . 20
1.2.2 Etats . . . 21
1.2.3 Arcs . . . 21
1.2.4 Portes . . . 21
1.2.5 Sortes . . . 21
1.2.6 Opérations . . . 21
1.2.7 Valeurs . . . 22
1.2.8 Labels . . . 22
1.2.9 Syntaxe graphique . . . 22
1.3 Relations d'équivalence . . . 22
1.3.1 Equivalence forte . . . 23
1.3.2 Equivalence de trace . . . 23
1.3.3 Equivalence observationnelle . . . 24

2 LOTOS **25**

2.1 Syntaxe abstraite de LOTOS . . . 25
2.1.1 Symboles non-terminaux . . . 25
2.1.2 Expressions de valeur . . . 26
2.1.3 Définitions d'opérateurs . . . 26
2.1.4 Equations algébriques . . . 27
2.1.5 Remplacements . . . 27
2.1.6 Définitions de types . . . 27
2.1.7 Offres . . . 27
2.1.8 Opérateurs parallèles . . . 28
2.1.9 Résultats . . . 28
2.1.10 Expressions de comportement . . . 28
2.1.11 Blocs . . . 28
2.1.12 Fonctionnalités . . . 29

2.1.13 Définitions de processus 29
2.1.14 Spécification LOTOS 29
2.1.15 Syntaxe concrète et syntaxe abstraite 29
2.1.16 Attributs locaux 31
2.2 Sémantique dynamique de LOTOS 32
2.2.1 Expressions de comportement étendues 32
2.2.2 Relation de transition 32
2.2.3 Règles de dérivation et de substitution 33
2.2.4 Notation "assign" 33
2.2.5 Notation "rename" 34
2.2.6 Opérateur "stop" 34
2.2.7 Opérateur ";" 34
2.2.8 Opérateur "[]" 35
2.2.9 Opérateur "choice" sur les portes 36
2.2.10 Opérateurs "||", "|||" et "|[...]|" 36
2.2.11 Opérateur "par" 37
2.2.12 Opérateur "hide" 37
2.2.13 Opérateur "->" 38
2.2.14 Opérateur "let" 38
2.2.15 Opérateur "choice" sur les valeurs 38
2.2.16 Opérateur "exit" 39
2.2.17 Opérateur ">>" 39
2.2.18 Opérateur "[>" 40
2.2.19 Processus et instanciation 40
2.2.20 Construction du graphe 40

3 SUBLOTOS **43**
3.1 Restrictions sur le contrôle 43
3.1.1 Notations préliminaires 43
3.1.2 Récursion à travers les opérateurs "||", "|||" et "|[...]|" 44
3.1.3 Récursion à travers l'opérateur ">>" 45
3.1.4 Récursion à travers l'opérateur "[>" 46
3.1.5 Propriété du contrôle statique 47
3.1.6 Algorithme du contrôle statique 49
3.2 Restrictions sur les données 50
3.3 Présentation de SUBLOTOS 52
3.4 Syntaxe abstraite de SUBLOTOS 53
3.4.1 Symboles non-terminaux 53
3.4.2 Types, sortes et opérations 53
3.4.3 Objets et duplications 53
3.4.4 Portes 54
3.4.5 Variables 55
3.4.6 Processus 55
3.4.7 Expressions de valeur 55
3.4.8 Offres 55
3.4.9 Opérateurs parallèles 55
3.4.10 Expressions de comportement 55
3.4.11 Processus 56
3.4.12 Spécification SUBLOTOS 57
3.4.13 Attributs locaux 57

3.5 Sémantique dynamique de SUBLOTOS 58
3.5.1 Contextes 58
3.5.2 Relation de transition 59
3.5.3 Règles de dérivation et de substitution 59
3.5.4 Opérateur “stop” 59
3.5.5 Opérateur “;” 59
3.5.6 Opérateur “[]” 60
3.5.7 Opérateur “||” 60
3.5.8 Opérateur “|[...]|” 61
3.5.9 Opérateur “hide” 61
3.5.10 Opérateur “->” 61
3.5.11 Opérateur “let” 61
3.5.12 Opérateur “choice” 62
3.5.13 Opérateur “[...>” 62
3.5.14 Processus et instanciation 62
3.5.15 Construction du graphe 62

4 Réseau **65**
4.1 Critères de conception du réseau 65
4.2 Avantages du modèle réseau 67
4.3 Présentation du modèle réseau 68
4.4 Syntaxe du réseau 75
4.4.1 Réseau 75
4.4.2 Places 75
4.4.3 Unités 76
4.4.4 Transitions 76
4.4.5 Offres 77
4.4.6 Actions 77
4.4.7 Syntaxe graphique 77
4.5 Sémantique opérationnelle du réseau 78
4.5.1 Marquages 78
4.5.2 Contextes 79
4.5.3 Positions 80
4.5.4 Spontanéité des ε-transitions 82
4.5.5 Atomicité des ε-transitions 84
4.5.6 Relation de transition 89
4.5.7 Construction du graphe 89
4.6 Propriétés invariantes du réseau 89
4.6.1 Marquage sauf 89
4.6.2 Séquentialité des unités 91

II Phases de traduction **93**

5 Expansion **95**
5.1 Principes de l’expansion 95
5.2 Obtention de noms uniques 96
5.3 Expansion des opérateurs “choice” et “par” 97
5.4 Expansion de la porte de terminaison 98
5.4.1 Expansion de l’opérateur “exit” 98

5.4.2 Expansion de l'opérateur ">>" . . . 99
5.4.3 Expansion de l'opérateur "[>" . . . 100
5.4.4 Expansion des opérateurs "||", "|||" et "|[...]|" . . . 100
5.4.5 Expansion des listes de paramètres portes . . . 100
5.5 Expansion des processus . . . 100
5.5.1 Développement des processus non récursifs . . . 100
5.5.2 Duplication des processus récursifs . . . 103
5.6 Algorithme d'expansion . . . 108
5.6.1 Renommage des portes . . . 108
5.6.2 Renommage des variables . . . 109
5.6.3 Renommage des processus . . . 109
5.6.4 Expressions de valeur . . . 111
5.6.5 Résultats . . . 111
5.6.6 Offres . . . 112
5.6.7 Opérateurs parallèles . . . 112
5.6.8 Expressions de comportement . . . 112
5.6.9 Construction du programme SUBLOTOS . . . 115
5.7 Implémentation . . . 115

6 Génération 117
6.1 Principes de la génération . . . 117
6.1.1 Génération de l'opérateur "stop" . . . 118
6.1.2 Génération de l'opérateur ";" . . . 118
6.1.3 Génération de l'opérateur "[]" . . . 119
6.1.4 Génération des opérateurs "||" et "|[...]|" . . . 120
6.1.5 Génération de l'opérateur "hide" . . . 123
6.1.6 Génération de l'opérateur "->" . . . 123
6.1.7 Génération de l'opérateur "let" . . . 123
6.1.8 Génération de l'opérateur "choice" . . . 124
6.1.9 Génération de l'opérateur "[...>" . . . 124
6.1.10 Génération de l'instanciation . . . 126
6.1.11 Clôture des rendez-vous . . . 128
6.2 Algorithme de génération . . . 129
6.2.1 Attributs . . . 129
6.2.2 Notations préliminaires . . . 130
6.2.3 Expressions de comportement . . . 132
6.2.4 Construction du réseau . . . 135
6.3 Implémentation . . . 136

7 Optimisation 137
7.1 Principes de l'optimisation . . . 137
7.1.1 Notations préliminaires . . . 138
7.2 Optimisation E0 : places et transitions improductives . . . 139
7.3 Optimisation E1 : places et transitions inaccessibles . . . 139
7.4 Optimisation E2 : places parallèles . . . 140
7.5 Optimisation E3 : pré-compactage . . . 140
7.6 Optimisation E4 : post-compactage $1 \to n$ . . . 142
7.7 Optimisation E5 : post-compactage $n \to 1$ . . . 144
7.8 Optimisation E6 : unités vides . . . 145
7.9 Optimisation V0 : affectations redondantes . . . 145

7.10 Optimisation V1 : variables inutilisées 146
7.11 Optimisation V2 : variables référencées 146
7.12 Ordonnancement des optimisations 147
7.13 Résultats de l'optimisation 148
7.14 Autres optimisations 148

8 Simulation **151**
8.1 Traitement des données 151
8.2 Algorithme de simulation 155
8.2.1 Programme simulateur 155
8.2.2 Exploration et construction du graphe 156
8.2.3 Calcul des états successeurs 157
8.3 Implémentation des marquages 161
8.3.1 Représentation des marquages en mémoire 161
8.3.2 Calcul des transitions franchissables 162
8.3.3 Calcul des marquages successeurs 165
8.4 Implémentation des contextes 166
8.4.1 Représentation des contextes en mémoire 166
8.4.2 Linéarisation des actions d'affectation 169
8.4.3 Linéarisation des actions collatérales 171
8.4.4 Calcul des contextes successeurs 172
8.5 Implémentation de la table des états 175
8.6 Implémentation de la table des positions 177
8.7 Implémentation de la table des arcs 178

9 Vérification **179**
9.1 Valeurs étendues 179
9.2 Aspects logiques 180
9.2.1 Syntaxe des formules logiques 180
9.2.2 Sémantique des formules logiques 182
9.2.3 Applications 186
9.3 Aspects temporels 187
9.3.1 Syntaxe des expressions temporelles 188
9.3.2 Sémantique des expressions temporelles 188
9.3.3 Lien entre formules logiques et expressions temporelles 190
9.3.4 Applications 190
9.4 Comparaison avec les logiques temporelles existantes 192
9.5 Implémentation 195

Conclusion **199**

A Présentation du langage LOTOS : les structures de données **203**
A.1 Présentation des types abstraits 203
A.2 Eléments lexicographiques et syntaxiques 204
A.3 Types importés 206
A.4 Types élémentaires 206
A.5 Types combinés 207
A.6 Types paramétrés 210
A.7 Types instanciés 212
A.8 Types renommés 213

B Présentation du langage LOTOS : les structures de contrôle **215**
B.1 Eléments lexicographiques et syntaxiques . . . 215
B.1.1 Expressions de comportement . . . 215
B.1.2 Expressions de valeur . . . 215
B.1.3 Variables . . . 216
B.1.4 Portes . . . 216
B.1.5 Identificateurs . . . 216
B.2 Opérateur "stop" . . . 217
B.3 Opérateur "`;`" . . . 217
B.4 Opérateur "`[]`" . . . 218
B.5 Opérateur "choice" sur les portes . . . 220
B.6 Opérateurs "`||`", "`|||`" et "`|[...]|`" . . . 221
B.7 Opérateur "par" . . . 225
B.8 Opérateur "hide" . . . 226
B.9 Opérateur "`->`" . . . 227
B.10 Opérateur "let" . . . 228
B.11 Opérateur "choice" sur les valeurs . . . 229
B.12 Opérateur "exit" . . . 230
B.13 Opérateur "`>>`" . . . 231
B.14 Opérateur "`[>`" . . . 232
B.15 Processus et instanciation . . . 233
B.16 Spécification LOTOS . . . 235
B.17 Styles de programmation . . . 235

C Application 1 : protocole du bit alterné **239**
C.1 Description du service . . . 239
C.2 Description du protocole . . . 240
C.3 Architecture du protocole . . . 240
C.4 Spécification du medium des messages . . . 242
C.5 Spécification du medium des acquittements . . . 242
C.6 Spécification de l'émetteur . . . 242
C.7 Spécification du récepteur . . . 243
C.8 Validation . . . 244
C.9 Notes bibliographiques . . . 246

D Application 2 : convolution systolique **247**
D.1 Produit de convolution . . . 247
D.2 Réseau systolique asynchrone . . . 247
D.3 Environnement . . . 248
D.4 Architecture B1 . . . 250
D.5 Architecture F . . . 252
D.6 Architecture W1 . . . 254
D.7 Architecture W2 . . . 256
D.8 Validation . . . 258
D.9 Notes bibliographiques . . . 258

Bibliographie **259**

Introduction

"Formal approaches to system construction and analysis are directly motivated by the concern to construct trustworthy systems — systems whose developers will be willing to stand before the public and take responsibility for the consequences of their deployment.
As yet, this goal is unrealized, but work in formal verification is a serious contribution toward that goal, and is both socially and scientifically responsible." [ABG[+]88]

Architecture OSI

Pour permettre à des systèmes informatiques hétérogènes ainsi qu'à des systèmes de commutation téléphonique de communiquer entre eux, deux organismes internationaux, l'ISO[1] et le CCITT[2], ont normalisé le concept d'*architecture ouverte* OSI[3] qui fait désormais l'objet d'une large acceptation.

L'architecture OSI est basée sur un *modèle de référence* [ISO84] qui établit une décomposition en sept niveaux d'abstraction (*couches*) pour les aspects communication et application des systèmes ouverts. Les fonctionnalités de chaque couche (*services*) sont définies par un ensemble de règles et de formats de dialogue (*protocoles*).

Le travail des comités de normalisation consiste à éditer des descriptions (*normes* ou *standards*) des services et des protocoles, qui constituent la référence à laquelle toute implémentation doit se conformer pour mériter le label OSI. De par leur objet même, ces descriptions sont complexes ; ce fait est parfois aggravé par l'existence de nombreuses classes et options.

Jusqu'à présent les descriptions étaient données en langage naturel et complétées par des diagrammes de transitions (*tables d'états*). Il va sans dire que ce mélange de texte informel et de schémas semi-formels conduisait à des définitions volumineuses, imprécises, ambiguës [LS88, § 2] [SAC88, § 2] et entachées d'erreurs [PG88].

C'est pourquoi de meilleurs formalismes de description ont été recherchés. Ce travail, qui a duré près de huit années, a abouti à la conception de trois langages, désignés sous le nom générique de FDT[4]. Les définitions de ces langages sont elles-mêmes normalisées, soit par l'ISO, soit par le CCITT .

[1] Organisation Internationale de Normalisation
[2] Comité Consultatif International pour la Téléphonie et la Télégraphie
[3] Open Systems Interconnection
[4] Formal Description Techniques

Langages FDT

Ils sont principalement destinés à spécifier des propriétés *fonctionnelles*, c'est-à-dire la description qualitative du comportement des systèmes vis-à-vis de leurs utilisateurs, par opposition aux propriétés *opérationnelles*, qui expriment des caractéristiques quantitatives comme le temps de réponse, les performances, ...

De par leur origine, les langages FDT ont été conçus pour permettre la *spécification* des systèmes distribués. C'est pourquoi il s'agit de langages déclaratifs plutôt qu'impératifs. Les langages FDT sont néanmoins *exécutables* — contrairement à d'autres formalismes tels que les logiques temporelles — et peuvent donc servir également à la *programmation*, sinon des systèmes, du moins de prototypes.

Les langages FDT permettent de décrire la structure interne des systèmes : chaque système est décomposé en sous-systèmes qui sont exécutés en parallèle, se synchronisent et communiquent par envois de messages. Pour les présenter, on distingue le *contrôle*, qui spécifie la façon dont un système réagit aux stimuli qu'il reçoit de son environnement, et les *données* qui concernent les informations mémorisées par le système et les messages qu'il émet ou reçoit.

Les trois langages FDT diffèrent par les concepts qu'ils mettent en œuvre :

ESTELLE[5], développé de 1980 à 1988, fait l'objet de la norme ISO 9074 (définition formelle : [ISO88a], présentation générale : [BD88]). En ESTELLE un système est composé d'instances de modules, organisées hiérarchiquement et exécutées simultanément de manière asynchrone. Le contrôle de chaque module est décrit par un automate d'états finis ; les données sont décrites en PASCAL. Les modules communiquent par des points d'interaction, connectés entre eux par des canaux qui autorisent des échanges bi-directionnels de messages. Les messages sont rangés dans des files d'attente non bornées. La création et la destruction dynamiques de canaux et d'instances de modules sont possibles. Enfin, ESTELLE permet d'exprimer des contraintes temporelles de type délai

SDL[6], développé dès 1974, a connu plusieurs versions successives dont la plus récente, datant de 1988, est standardisée par la recommandation Z.100 du CCITT (définition formelle : [CCI88], présentation générale : [ST87]). Comme ESTELLE, SDL est basé sur une extension du modèle des automates d'états finis. SDL permet de spécifier des automates, structurés de manière arborescente, qui fonctionnent de manière asynchrone et qui communiquent par échanges de signaux. Chaque automate possède des variables locales et une file non bornée pour stocker les messages reçus. Les données sont décrites par des types abstraits algébriques : un type comprend des domaines de valeurs, des opérations sur ces valeurs et des équations qui définissent la sémantique de ces opérations. De nouveaux types peuvent être créés à partir de types existants par enrichissement, renommage, paramétrisation, ... Enfin, SDL possède deux syntaxes : la forme texte, analogue à un langage de programmation classique, et la forme graphique, dont les éléments de base sont des boîtes connectées par des flèches

LOTOS[7], conçu entre 1980 et 1988, est défini par la norme ISO 8807 (définition formelle : [ISO88b], présentation générale : [BB88]). Il existe en LOTOS une nette séparation entre contrôle et données. Comme en SDL, les données sont décrites au moyen de types abstraits algébriques [EFH83] [EM85]. Contrairement à ESTELLE et SDL, le contrôle en LOTOS s'inspire, non pas des automates communicants, mais des algèbres de processus dont les plus connues sont CCS [Mil80] et CSP [BHR84]. LOTOS permet de décrire l'ordonnancement temporel d'un ensemble d'actions, tel qu'il serait perçu par un observateur extérieur. Il est possible de spécifier des comportements

[5] Extended Finite State Machine Language
[6] Specification and Description Language
[7] Language Of Temporal Ordering Specification

complexes en combinant des actions élémentaires au moyen d'opérateurs tels que la composition séquentielle, le choix non-déterministe, la composition parallèle avec différents degrés de liberté allant du synchronisme jusqu'à l'asynchronisme . . . Les communications s'effectuent uniquement au moyen de rendez-vous symétriques, qui peuvent être binaires ou n-aires et à partir desquels d'autres mécanismes de synchronisation peuvent être construits

Les avantages des langages FDT sont désormais reconnus :

- ils permettent de définir les normes de manière concise, complète et non ambiguë ; en particulier les relations entre les divers composants d'un système sont complètement formalisées
- ils permettent de décrire les fonctionnalités d'un système sans faire état de détails de réalisation qui imposeraient des contraintes inutiles aux implémenteurs
- les descriptions formelles offrent aux implémenteurs un point de départ à partir duquel ceux-ci peuvent — par raffinements successifs — produire une implémentation conforme, avec l'avantage que les choix de réalisation sont clairement distingués des caractéristiques des algorithmes

D'ores et déjà, les langages FDT ont été utilisés pour la spécification de nombreux services et protocoles OSI (Transport [LS88], Session [SAC88], OPD, FTAM, ACSE, X-25, ISDN, ROSE [FA88]). Leur introduction graduelle dans les standards ISO et CCITT est officiellement prévue[8]. La Communauté Economique Européenne a largement encouragé la définition des langages FDT et le développement d'outils correspondants au travers des programmes RACE (projets BEST et SPECS) et ESPRIT (projets PANGLOSS, SEDOS et SEDOS-ESTELLE Demonstrator).

Au delà des comités de normalisation de l'ISO et du CCITT, les langages FDT sont susceptibles d'intéresser d'autres partenaires du monde de l'informatique et des télécommunications :

- en milieu industriel, l'importance croissante des systèmes OSI contribue déjà à répandre l'usage des langages FDT. Les experts [Vis88] prédisent que leur emploi va se déplacer du secteur public vers le domaine privé : en effet, les constructeurs, familiarisés avec les langages FDT, devraient les utiliser pour leurs propres besoins, afin de réduire leurs coûts de développement.

  Utilisés dès la phase de conception, les langages FDT permettent une modélisation rigoureuse des fonctionnalités d'un système ; c'est ainsi que bon nombre de difficultés, d'ambiguïtés et d'erreurs peuvent être détectées. Une fois seulement que la conception a été validée, on peut engager la phase de réalisation et les investissements correspondants

- pour la communauté scientifique, les langages FDT coïncident avec un intérêt croissant pour les systèmes répartis qui se manifeste, entre autres, par l'évolution des méthodologies de programmation (d'abord structurée, puis modulaire, orientée-objet et parallèle).

  Cet intérêt général a eu comme effet la création d'un grand nombre de langages parallèles, dont la plupart sont morts-nés par absence de moyens. Face à cette moderne tour de Babel, les langages FDT possèdent des atouts incontestables : une définition rigoureuse et stable ayant fait l'objet d'un consensus international, une évolution contrôlée qui garantit une longue durée de vie, un domaine d'application bien réel, une large communauté d'utilisateurs et des appuis tant politiques qu'économiques.

  Les langages FDT constituent un progrès pour l'ingénierie des protocoles, mais leur portée dépasse de beaucoup ce cadre ; ils peuvent notamment servir à véhiculer les idées et les techniques de l'algorithmique parallèle asynchrone

[8] *cf.* ISO-CCITT/JTC1/N145

Outils pour les langages FDT

Toutefois l'apprentissage des langages FDT est difficile, tant par leur complexité intrinsèque qui apparaît comme l'inévitable contrepartie de leur précision et de leur richesse d'expression, que par la méconnaissance quasi-générale des concepts de programmation parallèle. Il faut donc assister l'utilisateur dans sa tâche par des logiciels appropriés et il semble que l'obstacle principal à la diffusion des langages FDT soit le manque d'outils de qualité industrielle. Idéalement on devrait disposer, pour chaque langage FDT, des outils de développement suivants :

analyseurs : ils comprennent notamment des éditeurs syntaxiques, des éditeurs graphiques, des paragrapheurs, des analyseurs de syntaxe et de sémantique statique, des analyseurs de complexité, ...

interpréteurs : souvent appelés simulateurs, ils permettent d'exécuter symboliquement une spécification en mode pas-à-pas, en assurant la correspondance avec le texte source, souvent au moyen d'un système de multi-fenêtrage. L'utilisateur peut franchir des séquences d'actions, revenir en arrière, inspecter les valeurs des variables, l'état des files d'attente, ...

générateurs de code : à partir d'une spécification, ils produisent un programme exécutable. L'utilisateur peut donc obtenir automatiquement un prototype correspondant à sa spécification. A l'heure actuelle, on ne peut toutefois espérer que les performances d'un tel prototype puissent rivaliser avec celles d'une implémentation directe

vérificateurs : ils servent à prouver qu'une spécification possède de "bonnes propriétés" et, plus fréquemment, à détecter des erreurs de conception, comme le fait, par exemple, qu'un protocole n'implémente pas le service pour lequel il est défini [PG88]. Il y a plusieurs approches au problème de la vérification :

démonstrateurs automatiques : il s'agit d'utiliser des systèmes de preuve opérant sur une axiomatisation semblable à celle de Hoare [Hoa69]. Cette technique ne permet pas d'établir ou de réfuter toutes les propriétés souhaitées — puisque, par contraposition du théorème de Gödel, dans tout système ayant la puissance de l'arithmétique il existe des théorèmes valides qui ne peuvent être démontrés. Surtout, les systèmes de preuve généraux sont extrêmement lents ; n'ayant pas produit de résultats convaincants pour les programmes séquentiels, il est douteux qu'ils le puissent pour les programmes parallèles dont la complexité inhérente est encore plus grande

générateurs de modèles : une autre technique de vérification (*model checking*) consiste à traduire, quand c'est possible, la spécification à valider vers un modèle (graphe, automate d'états finis, ...) qui représente toutes les évolutions possibles. Lorsque ce modèle est fini toute propriété est décidable. Cette approche, bien que limitée par la taille des modèles, est celle qui donne les meilleurs résultats

générateurs de tests : étant donné une spécification en langage FDT, ils produisent automatiquement des séquences de tests qui permettant de vérifier, au moins partiellement, qu'une implémentation existante est conforme à la spécification

Approche dynamique de la traduction de LOTOS

Cette étude est consacrée au langage LOTOS dont les choix de conception nous semblent préférables à ceux d'ESTELLE et de SDL. Actuellement LOTOS a l'inconvénient d'être en avance sur l'état de l'art : alors que les outils consacrés à ESTELLE et SDL utilisent au mieux des techniques éprouvées de compilation et d'analyse des programmes, les méthodes adaptées à LOTOS sont un domaine mal connu.

C'est pourquoi, en ce qui concerne les outils de développement, LOTOS accuse un net retard par rapport à ses concurrents pour lequel des outils sont dès à présent disponibles. Au contraire beaucoup d'outils dévolus à LOTOS sont encore au stade de prototypes de recherche.

Parmi eux on peut citer un analyseur syntaxique (LOTTE [Hul88]), des interpréteurs (HIPPO [Tre87] [Eij88], Ottawa LOTOS Interpreter [GHHL88], SPIDER [Joh88]) et un générateur de code qui traduit LOTOS vers le langage C [ndM88].

Beaucoup de tentatives ont été faites en direction de la vérification, mais peu d'outils existent :

- en ce qui concerne l'utilisation de systèmes de preuve, on peut mentionner l'application à LOTOS du système de démonstration automatique de Boyer-Moore [AF88]
- la technique de génération de modèles est mise en œuvre par le système SQUIGGLES [BC88] qui, à partir d'une spécification LOTOS ne comportant pas de données, construit un automate qui modélise tous les comportements possibles. La vérification s'effectue ensuite en réduisant cet automate selon diverses relations (équivalence forte, équivalence observationnelle [Mil80], équivalence de test [BS86], . . .)

En pratique on constate que ces outils, à l'exception peut-être du générateur de code, ne peuvent être utilisés que sur des spécifications très simples et non sur la description de systèmes complexes. L'analyse de ces outils révèle qu'ils possèdent plusieurs caractéristiques communes, que nous qualifions ici par le terme *approche dynamique* et qui contribuent de manière significative à la dégradation des performances :

- pour la partie contrôle de LOTOS, tous ces outils sont basés sur un modèle théorique commun, l'automate d'états finis ou système de transitions étiquetées (*labelled transition system*), appelé ici *graphe*. Ils comportent tous une phase de traduction, plus ou moins explicite et plus ou moins complète, de LOTOS vers ce modèle graphe.

  Cette traduction est effectuée par application des règles de sémantique dynamique de LOTOS [ISO88b]. Ces règles définissent l'évolution d'un système par des transformations successives de programmes. Comme leur application n'est pas déterministe, un moteur de réécriture est utilisé. En particulier, dans tous les interpréteurs, l'état courant est un fragment de programme LOTOS
- pour la partie données, l'évaluation des types abstraits algébriques est faite par réécriture à partir des équations présentes dans la spécification LOTOS, auxquelles une orientation conventionnelle est imposée. Même le générateur de code C implémente les types en produisant un moteur de réécriture, qui n'est toutefois pas générique mais optimisé en tirant parti des propriétés des équations. Les outils de vérification évacuent le problème en ne traitant pas les données
- enfin l'inefficacité est souvent due à l'utilisation de langages de programmation fonctionnelle ou logique, totalement inadéquats dans un contexte qui exige que les ressources de la machine — notamment la mémoire — soient gérées au mieux

Approche statique de la traduction de LOTOS

Pour surmonter ces difficultés nous préconisons l'adoption de principes radicalement différents, que nous qualifierons d'*approche statique*. Cette étude propose une méthode originale de traduction des programmes LOTOS en graphes, conçue selon ces principes et applicable à un large sous-ensemble de programmes LOTOS.

Pour traiter efficacement les données, nous avons choisi de ne pas utiliser de solution basée sur la réécriture de termes algébriques et d'implémenter, au contraire, les types abstraits, de la même façon que peuvent l'être les types d'un langage impératif. Il existe un algorithme [Sch88a] [Gar89b] qui répond à ce besoin ; c'est pourquoi cette étude ne s'attarde pas sur la compilation des types abstraits.

Seuls les aspects concernant la sémantique du parallélisme sont donc abordés ici. Notre méthode s'applique à un large sous-ensemble du langage LOTOS en tenant compte, bien entendu, de la présence des données. L'objectif initial de ce travail était de produire, à partir d'un programme LOTOS donné, le graphe correspondant, en vue de la vérification formelle. Toutefois les résultats obtenus sont suffisamment généraux pour servir dans le domaine plus vaste de la compilation de LOTOS.

L'approche statique est fondée sur l'idée qu'il ne faut pas traduire directement une spécification LOTOS en graphe. En effet il n'est pas souhaitable d'opérer au niveau du programme source, même s'il est implémenté sous forme d'arbre abstrait syntaxique : les représentations textuelles sont lentes à manipuler et coûteuses en mémoire. Il faut, au contraire, effectuer une traduction par étapes successives, en produisant à chaque fois une forme intermédiaire compacte, capable de modéliser un système composé de processus parallèles communicants et dotée d'une sémantique opérationnelle susceptible d'être implémentée de façon performante.

En adoptant ces formes intermédiaires, il ne suffit plus uniquement de "paraphraser" la sémantique de LOTOS en faisant de la réécriture, comme c'est le cas dans l'approche dynamique ; il faut au contraire s'en démarquer résolument et définir de nouvelles méthodes de traduction.

La première partie de ce document, intitulée *Représentations de programmes*, introduit les formalismes qui sont tour à tour utilisés au cours de la traduction.

Le chapitre 1 définit le concept de graphe, qui est le modèle sémantique "naturel" des programmes LOTOS dont il sert à exprimer le comportement. L'objectif final de la compilation d'une spécification LOTOS est la production du graphe correspondant. Il s'agit d'un modèle bien connu dont la définition est brièvement donnée et suivie du rappel de diverses relations d'équivalences entre graphes

Le chapitre 2 décrit le langage LOTOS. La définition de LOTOS adoptée ici est celle contenue dans la proposition de norme (*Draft International Standard* [ISO87]). Cette version a été récemment remplacée par la norme définitive (*International Standard* [ISO88b]), qui présente quelques différences mineures en ce qui concerne les définitions de types. On peut aussi consulter un ensemble de propositions intéressantes pour étendre LOTOS (*Extended* LOTOS, [Bri88]).

Dans cette étude, le langage LOTOS est considéré comme une donnée de base. Nous avons accepté la définition telle qu'elle figure dans la norme, d'autant plus volontiers que les choix de conception de LOTOS sont sains et que les progrès scientifiques dans le domaine du parallélisme passent par le regroupement des efforts. C'est pourquoi cette étude ne contient ni critiques ni suggestions à l'égard de LOTOS.

Par rapport à la norme, dont la compréhension n'est pas toujours facile, un effort de synthèse, de simplification et d'abstraction a été fait dans ce chapitre, tout en préservant la conformité. Seules les notions indispensables au traitement sémantique du parallélisme ont été conservées. Nous proposons une syntaxe abstraite pour LOTOS, sur laquelle la sémantique dynamique des programmes est définie en fonction du modèle graphe

Le chapitre 3 définit un sous-ensemble "raisonnable" des programmes LOTOS que l'on est capable de traiter efficacement. Cette restriction porte sur l'emploi de la récursion à travers certains opérateurs, ce qui revient intuitivement à interdire la création et la destruction dynamique de processus, ainsi que les modifications dynamiques des canaux de communication. En effet l'utilisation de telles possibilité rend très difficile l'analyse statique des programmes parallèles et constitue une source majeure d'inefficacité lorsque l'on construit le modèle graphe. Ces contraintes, bien qu'elles limitent l'expressivité, ne sont toutefois pas exagérées et l'utilisateur a toujours le recours de décrire les évolutions dynamiques à l'aide des données.

Plus généralement, le rejet des structures de contrôle dynamiques est une réalité constante en informatique (interdiction des programmes non réentrants, disparition des branchements calculés, abandon des variables de type procédure, ...). De tels choix améliorent à la fois la fiabilité et les performances sans entraîner un appauvrissement excessif des moyens d'expression.

Nous introduisons ensuite une première forme intermédiaire utilisée au cours de la compilation de LOTOS. Il s'agit d'un langage, appelé SUBLOTOS, qui constitue une version simplifiée de LOTOS dans laquelle certaines constructions ont été remplacées par des formes plus simples. De plus, contrairement à LOTOS, SUBLOTOS est un langage impératif et non plus fonctionnel. Comme LOTOS, SUBLOTOS est défini par une syntaxe abstraite et sa sémantique dynamique est définie en fonction du modèle graphe

Le chapitre 4 propose une seconde forme intermédiaire, appelée *réseau.* Il s'agit d'une extension des réseaux de Petri auxquels sont ajoutées des variables, des actions et des transitions atomiques. Comme dans le cas de LOTOS et de SUBLOTOS, la sémantique du modèle réseau est définie par référence au modèle graphe. Hormis la présence des transitions atomiques, elle est très proche de celle des réseaux de Petri interprétés, car nous avons soigneusement évité d'introduire de nouvelles constructions lorsque cela n'était pas indispensable. Cette économie de concepts débouche sur un modèle dont la généralité dépasse le cadre de LOTOS.

La manière dont est décrit le parallélisme en LOTOS fait que la taille du modèle graphe peut croître très rapidement dès que l'on traite des spécifications complexes. L'objectif principal du modèle réseau est d'éviter cette explosion combinatoire en fournissant une représentation compacte du contrôle et des données.

Ainsi le réseau constitue le modèle d'exécution fondamental pour le langage LOTOS, alors que le graphe n'est que le résultat de cette exécution. Il existe cependant une complémentarité entre ces deux formalismes. Le graphe est un modèle théorique, qui permet de définir la sémantique de LOTOS et sert essentiellement à la validation formelle, puisque les propriétés que l'on souhaite vérifier sont décidables sur le graphe. Au contraire, le réseau est un modèle exécutable, doté d'une sémantique opérationnelle à caractère impératif, susceptible d'être implémentée pour des applications autres que la validation (génération de code, simulation, ...)

La seconde partie de ce document, intitulée *Phases de traduction*, décrit les différentes étapes du processus de traduction, présentées successivement selon leur ordre d'enchaînement. Pour chacune la méthode de traduction employée est d'abord expliquée, puis définie formellement.

Le chapitre 5 présente la phase d'*expansion* qui traduit un programme LOTOS en programme SUBLOTOS

Le chapitre 6 décrit la phase de *génération* qui construit le réseau correspondant à un programme SUBLOTOS

Le chapitre 7 est consacré à la phase d'*optimisation* dont le rôle est de réduire la taille du réseau produit par la phase précédente. Il ne s'agit pas d'un simple "nettoyage" du réseau car, d'ores et déjà, des transformations non triviales sont effectuées, notamment sur les données. A plus long terme, le développement de nouvelles optimisations opérant au niveau du réseau constitue probablement une réponse aux problèmes soulevés par l'explosion combinatoire de la taille des graphes engendrés

Le chapitre 8 présente la phase de *simulation* qui produit le graphe correspondant à un réseau. Il s'agit donc d'une implémentation de la sémantique opérationnelle du réseau, destinée ici à produire exhaustivement le graphe, mais qui peut aisément être adaptée à d'autres objectifs (génération de code, génération de séquences de test, ...).

Lorsque le graphe correspondant à un programme LOTOS a été construit, la validation peut être faite de deux manières, soit en comparant ce graphe à un autre graphe au moyen de relations d'équivalences semblables à celles définies au chapitre 1 (typiquement on cherche à vérifier qu'un protocole implémente un service), soit en vérifiant que le graphe satisfait des propriétés exprimées dans un formalisme autre que LOTOS et le modèle graphe, par exemple une logique temporelle

Le chapitre 9 propose une logique temporelle originale adaptée à LOTOS, la logique R$\star$ICO (*Regular Information Chronological Ordering*). En effet l'étude des logiques existantes a montré qu'elles convenaient mal au modèle sémantique de LOTOS. La logique R$\star$ICO est construite à partir d'opérateurs logiques et d'expressions régulières, dont la combinaison lui donne un grand pouvoir expressif et permet de retrouver, comme cas particuliers, certains opérateurs temporels utilisés dans les logiques classiques

Les annexes A et B constituent une introduction informelle, en français, à LOTOS, qui s'ajoute au *tutorial* du langage [BB88].

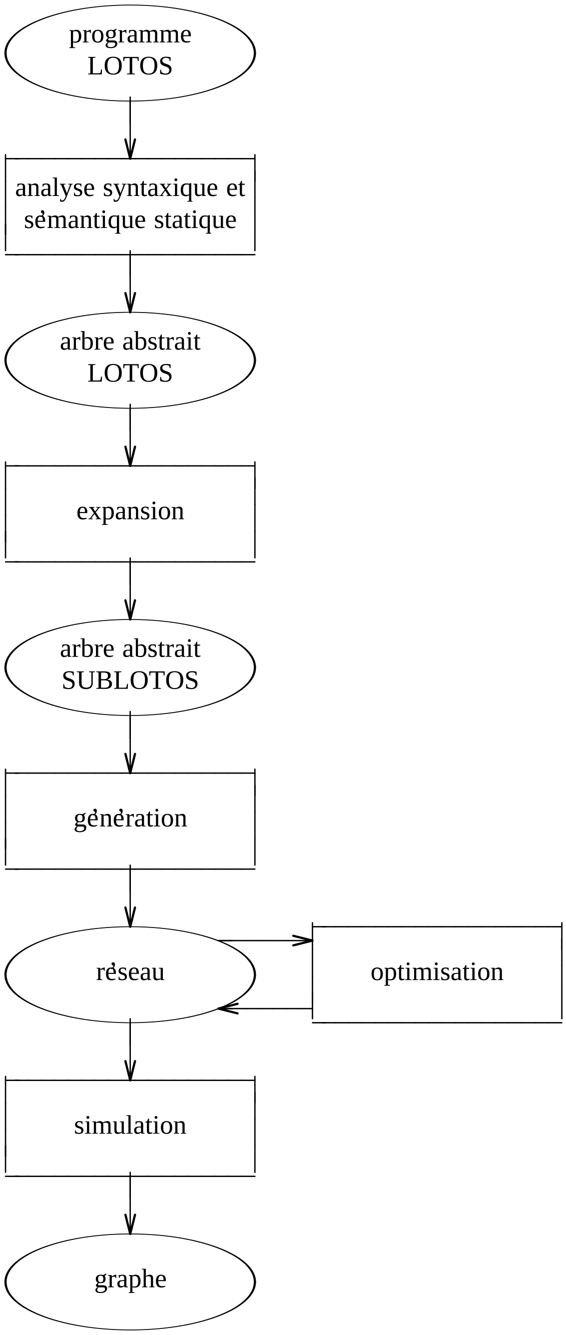
Les annexes C à D sont consacrées aux applications : elles illustrent la démarche de spécification et de validation, pour différents types d'algorithmes répartis.

Application de l'approche statique : le système CÆSAR

Pour expérimenter en grandeur réelle nos propositions, nous avons entrepris la réalisation d'un outil de vérification par génération de modèle : comme SQUIGGLES ce système engendre le graphe complet correspondant à une spécification LOTOS.

Ce traducteur est appelé CÆSAR parce qu'il fait partie de la famille CESAR d'outils de vérification pour les systèmes distribués, famille qui comprend également QUASAR [Que82] [Sch83] [FSS83] [FRV85] et XESAR [RRSV87a] [Rod88] [GRRV89]. Ces outils possèdent en commun deux principes : ils traduisent un langage parallèle de haut niveau (une variante de CSP pour QUASAR, une variante d'ESTELLE pour XESAR, LOTOS pour CÆSAR) en un graphe ; ils permettent ensuite d'exprimer des propriétés au moyen de formules de logique temporelle et de vérifier que ces formules admettent le graphe pour modèle.

Le système CÆSAR comporte plusieurs phases qui implémentent fidèlement notre méthode de traduction et emploient le langage SUBLOTOS et le modèle réseau comme formes intermédiaires. Le fonctionnement simplifié de CÆSAR est représenté par le schéma suivant :



Il faut souligner à quel point les contraintes d'efficacité qui pèsent sur un tel système de vérification sont beaucoup plus fortes que pour un interpréteur, puisqu'il faut construire et conserver en mémoire *tous* les états du graphe (et non seulement l'état courant ou une liste d'états).

Les performances de CÆSAR constituent donc le critère d'appréciation le plus sévère et le plus impartial sur l'intérêt pratique de l'approche statique proposée ici. Alors que les outils conçus selon l'approche dynamique ne permettent guère de dépasser le millier d'états, on vise pour CÆSAR des performances bien supérieures, à savoir la possibilité d'engendrer, sur une station de travail, des graphes ayant plusieurs centaines de milliers d'états, dans un laps de temps variant entre quelques minutes et quelques heures.

Les graphes produits par CÆSAR peuvent être vérifiés à l'aide d'outils existants, notamment des réducteurs d'automates comme ALDEBARAN [Fer88] et AUTO [LMV87].

Notations

La définition des notations utilisées dans ce document est donnée ici.

Symboles arithmétiques, logiques et ensemblistes

Les symboles arithmétiques usuels sont employés, ainsi que les opérateurs logiques suivants :

notation	*signification*
false	proposition toujours fausse
true	proposition toujours vraie
$\neg P_0$	négation de la proposition P_0
$P_1 \wedge P_2$	conjonction des propositions P_1 et P_2
$P_1 \vee P_2$	disjonction des propositions P_1 et P_2
$P_1 \Longrightarrow P_2$	implication de la proposition P_2 par P_1
$P_1 \Longleftrightarrow P_2$	équivalence des propositions P_1 et P_2

On utilise également les opérateurs ensemblistes classiques :

notation	*signification*
$\emptyset$	ensemble vide
$\{x_m, \ldots\ x_n\}$	ensemble ($\emptyset$ si $m > n$)
$\mathrm{card}(E)$	nombre d'éléments dans l'ensemble E
$\mathit{oneof}(E)$	élément quelconque de l'ensemble E
$E_1 \cap E_2$	intersection des ensembles E_1 et E_2
$E_1 \cup E_2$	union des ensembles E_1 et E_2
$E_1 - E_2$	différence des ensembles E_1 et E_2
$E_1 \times E_2$	produit cartésien des ensembles E_1 et E_2
$\langle\rangle$	liste[9] vide
$()$	*idem*
$\langle x_m, \ldots\ x_n\rangle$	liste ($\langle\rangle$ si $m > n$)
$(x_m, \ldots\ x_n)$	*idem*

Enfin on emploie fréquemment des notations abrégées de la forme "$N_m + \ldots\ N_n$", "$P_m \wedge \ldots\ P_n$", "$E_m \cup \ldots\ E_n$", ... dont la signification est définie formellement comme suit ; si "$\star$" représente une opération binaire associative admettant un élément neutre "ε", on note "$x_m \star \ldots\ x_n$" l'expression

[9] ou n-uplet, ou séquence

$f(m,n)$ définie par :

$$f(i,j) = \begin{cases} \textbf{si } i > j \textbf{ alors } \varepsilon \\ \textbf{sinon } x_i \star f(i+1,j) \end{cases}$$

Symboles fonctionnels

On appelle *application partielle* d'un ensemble X vers un ensemble Y toute application de X vers $(Y \cup \{\textit{indéfini}\})$. Par contraste, une application de X vers Y sera appelée *application totale*, parce qu'elle ne renvoie jamais la valeur *indéfini*.

Les applications partielles servent essentiellement ici à décrire la notion d'*environnement* [Ten76], c'est-à-dire une fonction qui, à chaque élément de X (considéré comme un identificateur) associe un élément de Y (correspondant à la définition ou la valeur de cet identificateur). Pour manipuler les environnements en tenant compte de la visibilité des identificateurs et des portées imbriquées, on utilise les notations suivantes :

- on note "$\perp$" l'application nulle part définie
- si f_1 et f_2 sont deux applications partielles de X vers Y, on note "$f_1 \oslash f_2$" l'application partielle de X vers Y formée par la *somme* de f_1 et f_2 avec éventuellement *masquage* de f_1 par f_2 :

$$(f_1 \oslash f_2)(x) = \begin{cases} \textbf{si } f_2(x) = \textit{indéfini} \textbf{ alors } f_1(x) \\ \textbf{si } f_2(x) \neq \textit{indéfini} \textbf{ alors } f_2(x) \end{cases}$$

- si f_1 et f_2 sont deux applications partielles de X vers Y vérifiant :

$$(\forall x \in X)\ \ (f_1(x) = \textit{indéfini})\ \vee\ (f_2(x) = \textit{indéfini})$$

  on note "$f_1 \oplus f_2$" l'application partielle de X vers Y formée par la *somme directe* de f_1 et de f_2 :

$$(f_1 \oplus f_2)(x) = \begin{cases} \textbf{si } f_1(x) \neq \textit{indéfini} \textbf{ alors } f_1(x) \\ \textbf{si } f_2(x) \neq \textit{indéfini} \textbf{ alors } f_2(x) \\ \textbf{si } f_1(x) = f_2(x) = \textit{indéfini} \textbf{ alors } \textit{indéfini} \end{cases}$$

  Si $f_m, \ldots\ f_n$ sont des applications partielles de X vers Y dont les domaines de définition sont deux à deux disjoints, on note souvent "$\bigoplus_{i \in \{m, \ldots\ n\}} f_i$" l'application partielle $f_m \oplus \ldots\ f_n$

- si x_0 appartient à un ensemble X et si y_0 appartient à un ensemble Y, on note "$x_0 \rightsquigarrow y_0$" l'application partielle f de X vers Y définie seulement en x_0 à qui elle fait correspondre y_0 :

$$f(x) = \begin{cases} \textbf{si } x = x_0 \textbf{ alors } y_0 \\ \textbf{si } x \neq x_0 \textbf{ alors } \textit{indéfini} \end{cases}$$

- si $\widehat{x}$ désigne une liste $x_0, \ldots\ x_n$ d'éléments de X et si f est une application totale (*resp.* partielle) de X vers Y, on note "$f(\widehat{x})$" la liste $\widehat{y}$ d'éléments de Y (*resp.* $Y \cup \{\textit{indéfini}\}$) définie par :

$$\widehat{y} = f(x_0), \ldots\ f(x_n)$$

- si $\widehat{x}$ désigne une liste $x_0, \ldots\ x_n$ d'éléments de X et si y_0 appartient à Y, on note "$\widehat{x} \rightsquigarrow y_0$" l'application partielle f de X vers Y définie seulement en $x_0, \ldots\ x_n$ à qui elle fait correspondre y_0 :

$$f = (x_0 \rightsquigarrow y_0) \oplus \ldots\ (x_n \rightsquigarrow y_0)$$

- si $\widehat{x}$ désigne une liste $x_0, \dots\ x_n$ d'éléments de X et si $\widehat{y}$ désigne une liste $y_0, \dots\ y_n$ d'éléments de Y, on note "$\widehat{x} \rightsquigarrow \widehat{y}$" l'application partielle f de X vers Y définie seulement en $x_0, \dots\ x_n$ à qui elle fait correspondre respectivement $y_0, \dots\ y_n$:

$$f = (x_0 \rightsquigarrow y_0) \oplus \dots\ (x_n \rightsquigarrow y_n)$$

Méta-langage de description syntaxique

Pour les définitions syntaxiques on utilise le méta-langage défini comme suit :

- les symboles terminaux sont notés en caractères gras
- les symboles non-terminaux sont notés en caractères italiques
- le méta-symbole "≡" introduit une règle syntaxique, qui associe au non-terminal situé en partie gauche sa définition figurant en partie droite
- la concaténation est notée par juxtaposition
- le méta-symbole "|" dénote le choix entre ses deux opérandes
- le méta-symbole "[ ]" dénote zéro ou une occurrence de l'opérande qu'il encadre[10]

Par commodité et pour différencier les différentes occurrences d'un même symbole non-terminal dans une partie droite de règle, on utilise des notations étendues[11] qui tiennent compte du contexte :

- si A est un non-terminal alors A', A'', A_i, A'_i et A''_i dénotent des non-terminaux ayant la même définition syntaxique que A
- la notation "$A_1, \dots\ A_n$" désigne la concaténation de n éléments A. La valeur de n $(n \geq 0)$ est déterminée par l'analyse syntaxique et A_i dénote le i$^{\text{ème}}$ élément de cette séquence
- la notation "$A_0, \dots\ A_n$" désigne la concaténation de $n+1$ éléments A. La valeur de n $(n \geq 0)$ est déterminée par l'analyse syntaxique et A_i dénote le i$^{\text{ème}}$ élément de cette séquence
- la notation "$\widehat{A}$" désigne la concaténation d'un nombre non nul d'éléments de A

Méta-langage de description sémantique

Pour définir la sémantique, on utilise des *grammaires attribuées* [Knu68], formalisme qui nous semble préférable aux systèmes de règles [Plo81] parce qu'il requiert que les définitions soient constructives, donc susceptibles d'être implémentées efficacement.

A chaque unité syntaxique on peut associer trois sortes de paramètres :

attributs hérités : ce sont des informations reçues de l'environnement. Leur valeur est calculée de manière *descendante* car, dans un arbre abstrait, les attributs hérités sont transmis par un nœud à ses fils. Dans la grammaire attribuée ces attributs sont précédés du méta-symbole "↓"

attributs synthétisés : ce sont des informations fournis à l'environnement. Leur valeur est calculée de manière *ascendante* puisque, dans un arbre abstrait, les attributs synthétisés sont transmis par un nœud à son père. Ces attributs sont précédés du méta-symbole "↑"

attributs locaux : ce sont des informations attachées aux unités syntaxiques et dont il est possible de consulter et de modifier la valeur

[10] ne pas confondre les méta-symbole "|", "[" et "]" avec les symboles terminaux "|", "[" et "]"
[11] variables syntaxiques

Partie I

Représentations de programmes

Chapitre 1

Graphe

La définition de LOTOS [ISO87] exprime la sémantique dynamique du langage en fonction d'un modèle de base (*graphe d'états*, *système de transitions*, *labelled transition system*), appelé ici *graphe* ou *automate*. Dans ce document ces deux termes, lorsqu'ils se réfèrent à ce modèle de base, sont rigoureusement synonymes.

L'objectif principal du système CÆSAR est de produire le graphe correspondant à une spécification LOTOS. Cette traduction se fait par étapes successives, le graphe étant obtenu en dernier. Le modèle graphe est cependant présenté ici en premier, parce qu'il constitue le support formel de référence à partir de laquelle la sémantique de LOTOS, celle de SUBLOTOS et celle du réseau sont définies.

On introduit le modèle graphe, d'abord de manière intuitive, en indiquant les implications sémantiques liées au choix de cette représentation. Puis on le définit formellement en donnant sa syntaxe. On donne enfin plusieurs relations d'équivalence qui permettent de comparer entre eux des graphes afin de décider s'ils possèdent des propriétés communes.

1.1 Présentation du modèle graphe

Systèmes asynchrones

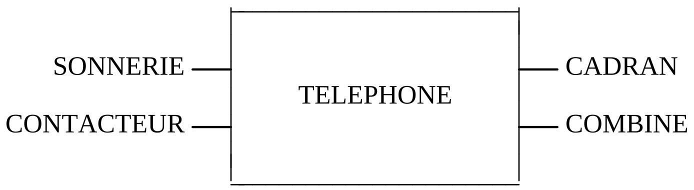
Le modèle graphe — tout comme le langage LOTOS ou le modèle réseau — vise à représenter les *systèmes asynchrones*[12]. De tels systèmes peuvent être intuitivement conçus comme des "boîtes noires" placées dans un environnement constitué d'un observateur humain et/ou d'autres systèmes asynchrones.

Pour communiquer avec son environnement, un système asynchrone possède des points d'interaction appelés *portes* (*gates*). Les portes autorisent la synchronisation ainsi que les échanges d'informations par rendez-vous symétriques, binaires ou n-aires. Les communications du système avec son environnement sont appelées ici *interactions*, *rendez-vous*, *événements* ou encore *signaux*.

Exemple 1-1
Le fonctionnement d'un téléphone, tel qu'il est perçu par l'usager, constitue un système asynchrone. Si l'on veut prendre en compte toutes ses caractéristiques, il s'agit d'un système relativement complexe. On peut, en première approximation, le modéliser par une "boîte noire" comportant quatre portes

[12] par opposition aux *systèmes synchrones* [BCG87] [CHPP87] dont il n'est pas question ici

appelées respectivement `SONNERIE`, `CONTACTEUR`, `CADRAN` et `COMBINÉ`.



Chacune de ces portes peut transmettre, éventuellement, divers *messages* :

- la porte `SONNERIE` n'est accompagnée d'aucun message : la sonnerie du téléphone est, en soi, une information suffisante

- la porte `CONTACTEUR` accepte deux sortes de messages : `DÉCROCHAGE` (l'usager décroche le téléphone) ou `RACCROCHAGE` (l'usager raccroche)

- la porte `CADRAN` n'est accompagnée d'aucun message, afin de conserver à cet exemple sa simplicité. Dans une modélisation plus fine on pourrait prendre en compte les numéros de téléphone composés par l'usager

- la porte `COMBINÉ` accepte cinq sortes de messages : `TONALITÉ` (l'usager entend la tonalité), `SONNERIE_DISTANTE` (l'usager entend la sonnerie du téléphone de son correspondant), `DÉCROCHAGE_DISTANT` (l'usager entend son correspondant décrocher), `RACCROCHAGE_DISTANT` (l'usager entend la sonnerie "occupé") et `DIALOGUE` (l'usager et son correspondant communiquent)

■

Par hypothèse, chaque événement est *atomique*, c'est-à-dire indivisible dans le temps. En outre on suppose qu'il n'est pas possible que deux événements aient lieu simultanément. Enfin le système peut également effectuer des interactions *internes* ou *invisibles* qui ne peuvent pas être perçues par l'environnement.

Etats, arcs et labels

Pour représenter les systèmes asynchrones, le modèle graphe sous-jacent à LOTOS s'apparente à la fois aux graphes finis orientés et aux automates d'états finis. C'est pourquoi on emprunte des termes appartenant aux vocabulaires de ces deux théories.

Un graphe est formé d'un ensemble non vide et fini d'*états*, parmi lesquels on distingue un *état initial*, reliés entre eux par des *arcs*. Les états du graphe correspondent aux situations dans lesquelles le système asynchrone peut se trouver. Les arcs du graphe représentent les interactions atomiques effectuées par le système.

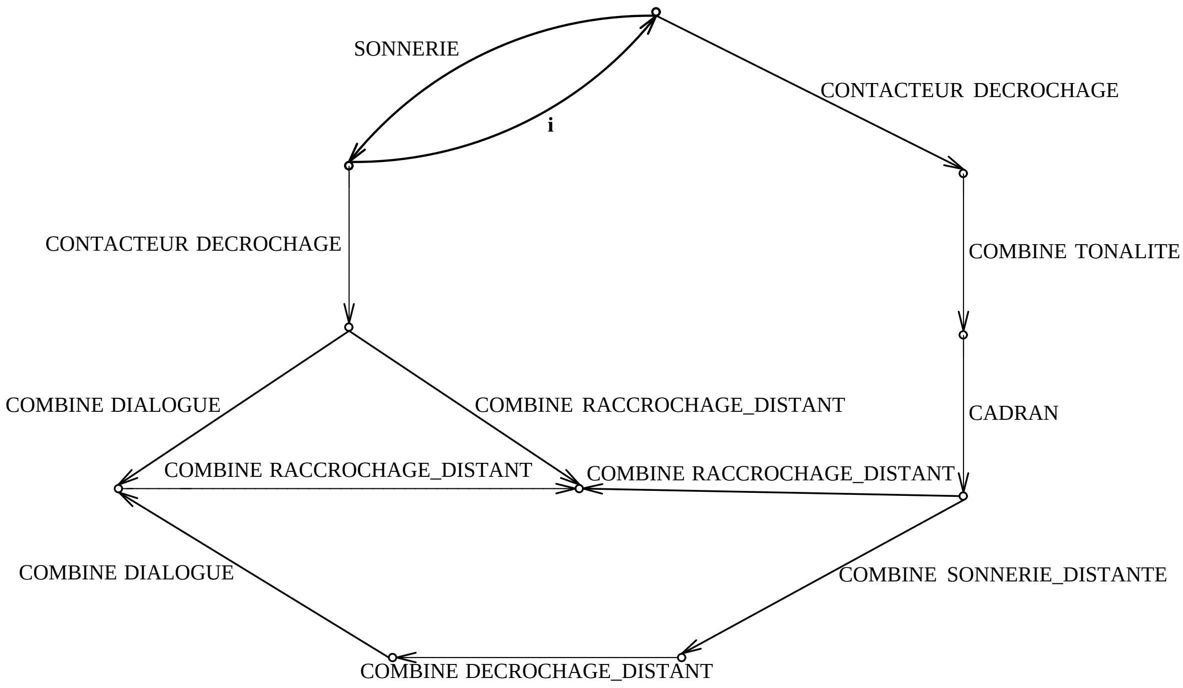
Chaque arc est étiqueté par un *label*, constitué d'une *porte* (qui est un nom symbolique) et d'une liste de *valeurs*, qui sont des expressions typées. Ces valeurs représentent les messages échangés par le système et son environnement, via le point de communication que constitue la porte.

Dans le modèle graphe, les évolutions internes du système s'expriment de la même manière que les interactions observables. Elles sont représentées par des arcs dont le label possède une porte spéciale, notée "**i**". Au-delà de cette signification intuitive, les arcs étiquetés "**i**" sont susceptibles de diverses interprétations formelles (§ 1.3, p. 22).

Les chemins issus de l'état initial du graphe représentent des séquences de fonctionnement possibles, c'est-à-dire des suites d'interactions pouvant se produire par suite de l'évolution du système. La suite des labels qui étiquettent les arcs d'un chemin issu de l'état initial du graphe, souvent appelée *trace*, définit donc une séquence d'événements ordonnés chronologiquement, des plus anciens aux plus récents.

Exemple 1-2
Le fonctionnement du téléphone présenté dans l'exemple 1-1 (p. 17) peut être représenté graphiquement. L'état initial du graphe est celui qui se trouve au-dessus de tous les autres. Pour plus de lisibilité on a omis de représenter les arcs étiquetés `CONTACTEUR RACCROCHAGE` qui, partant de tous les états autres que l'état initial, convergent vers l'état initial.

Ce graphe comporte un arc étiqueté "**i**" qui exprime le fait que la sonnerie, après avoir retenti, peut cesser. Il s'agit d'une évolution interne du système sur laquelle l'usager n'a aucune influence, à moins qu'il ne décroche le combiné.



■

Implications sémantiques

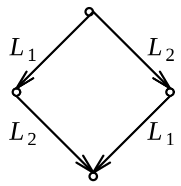
L'étude du modèle graphe appelle plusieurs remarques :

non-déterminisme : l'atomicité des événements fait qu'il n'est pas possible de franchir deux arcs simultanément. Lorsque plusieurs arcs sont issus d'un même état, cela implique donc que le système qui se trouve dans cet état possède plusieurs évolutions possibles. On parle alors de *choix non-déterministe* :

– si les labels des arcs sont distincts, c'est l'environnement qui détermine l'évolution en acceptant l'une des interactions et en refusant l'autre

– si les labels sont identiques, c'est le système qui prend une décision sans que l'environnement puisse intervenir (ni LOTOS ni le modèle graphe ne permettent de contrôler l'évolution, par exemple en associant des priorités relatives aux différentes éventualités)

séquentialité : bien qu'utilisé pour représenter des systèmes parallèles décrits en LOTOS, le modèle graphe est purement séquentiel. En effet lorsqu'un programme LOTOS est traduit sous forme d'automate, la concurrence est modélisée par l'*entrelacement* des événements ; lorsque deux interactions distinctes L_1 et L_2 sont simultanément possibles dans le système parallèle, la traduction produit le graphe suivant (qui justifie le terme *sémantique du losange* souvent employé à propos de LOTOS) :



abstraction : on ne retrouve plus dans le graphe la structure du programme LOTOS initial, constitué de processus parallèles communicants. Il n'est généralement pas possible de déterminer quel processus est à l'origine d'un événement. En outre, les communications étant totalement symétriques, les labels attachés aux arcs du graphe ne permettent pas de savoir s'il s'agit de messages émis ou reçus par le système.

Enfin il existe une différence essentielle entre le modèle sémantique de LOTOS et ceux d'autres langages comme LUSTRE [CHPP87] et ESTELLE [ISO88a] : aucune information n'est attachée aux états LOTOS ; au contraire les états LUSTRE et ESTELLE sont définis en fonction des valeurs d'un ensemble de variables appartenant au programme source (la connaissance de ces valeurs est une donnée indispensable à la compréhension du graphe)

1.2 Syntaxe du graphe

Pour simplifier la présentation, on aura recours, dans cette section, à des définitions en avant, signalées par le symbole "†".

1.2.1 Graphe — Automate

On appelle *graphe* ou *automate* un sextuplet $(\Sigma, \sigma_0, \mathcal{E}, \mathcal{G}, \mathcal{S}, \mathcal{F})$ où :

- Σ est un ensemble non vide et fini d'états†
- σ_0 est un élément de Σ, appelé *état initial*
- $\mathcal{E}$ est un ensemble fini d'arcs†, construits sur Σ, $\mathcal{G}$, $\mathcal{S}$ et $\mathcal{F}$
- $\mathcal{G}$ est un ensemble fini de portes†
- $\mathcal{S}$ est un ensemble fini de sortes†
- $\mathcal{F}$ est un ensemble fini d'opérations†

1.2.2 Etats

On appelle *état* un élément d'un ensemble Σ (sur lequel on n'impose aucune contrainte particulière).

1.2.3 Arcs

Si Σ est un ensemble d'états, $\mathcal{G}$ un ensemble de portes, $\mathcal{S}$ un ensemble de sortes et $\mathcal{F}$ un ensemble d'opérations, on appelle *arc* construit sur Σ, $\mathcal{G}$, $\mathcal{S}$ et $\mathcal{F}$, un triplet (σ_1, L, σ_2) où :

- σ_1 est un élément de Σ, appelé *état de départ*
- σ_2 est un élément de Σ, appelé *état d'arrivée*
- L est un label† construit sur $\mathcal{G}$, $\mathcal{S}$ et $\mathcal{F}$

Il est souvent commode d'exprimer l'ensemble de triplets $\mathcal{E}$ sous deux autres formes :

relation de transition entre états : il s'agit d'une relation entre deux états σ_1, σ_2 et un label L, notée "$\sigma_1 \xrightarrow{L} \sigma_2$" et définie par :

$$(\sigma_1 \xrightarrow{L} \sigma_2) \Longleftrightarrow (\sigma_1, L, \sigma_2) \in \mathcal{E}$$

fonction de succession entre états : si σ_1 est un état et L un label, on note "$succ(\sigma_1, L)$" l'ensemble défini par :

$$succ(\sigma_1, L) = \{\sigma_2 \mid (\sigma_1, L, \sigma_2) \in \mathcal{E}\}$$

Dans la suite, on considère uniquement les graphes dont tous les états sont accessibles à partir de l'état initial σ_0 :

$$(\forall \sigma) \ \ \sigma \neq \sigma_0 \Longrightarrow (\exists \sigma_1, \ldots \ \sigma_n) \ \ (\exists L_0, \ldots \ L_n) \ \ \sigma_0 \xrightarrow{L_0} \sigma_1 \xrightarrow{L_1} \ldots \ \sigma_n \xrightarrow{L_n} \sigma$$

1.2.4 Portes

On appelle *porte* un nom appartenant à un ensemble $\mathcal{G}$ d'identificateurs. Cet ensemble contient toujours une porte spéciale, notée "**i**".

1.2.5 Sortes

On appelle *sorte* un nom appartenant à un ensemble $\mathcal{S}$ d'identificateurs. Cet ensemble contient toujours une sorte spéciale, appelée *sorte booléenne*.

1.2.6 Opérations

On appelle *opération* un nom appartenant à un ensemble $\mathcal{F}$ d'identificateurs. A chaque opération F sont associés les attributs suivants :

- un entier n ($n \geq 0$), appelé *arité* de F
- une liste de sortes $S_1, \ldots \ S_n$ de $\mathcal{S}$, appelées *sortes des arguments* de F
- une sorte S de $\mathcal{S}$, appelé *sorte du résultat* de F

1.2.7 Valeurs

Si $\mathcal{S}$ est un ensemble de sortes et $\mathcal{F}$ un ensemble d'opérations, on appelle *expression de valeur*, ou plus simplement *valeur*, construite sur $\mathcal{S}$ et $\mathcal{F}$ tout terme syntaxique obtenu par application des règles suivantes :

- si F est une opération dont l'arité est 0 et dont le résultat a pour sorte S, alors le terme "F" est une valeur de sorte S
- si F est une opération dont l'arité est n ($n > 0$), dont les arguments ont pour sortes $S_1, \ldots S_n$ et dont le résultat a pour sorte S, si $V_1, \ldots V_n$ sont des valeurs de sortes respectives $S_1, \ldots S_n$ alors le terme "F ($V_1, \ldots V_n$)" est une valeur de sorte S
- si V_1 et V_2 sont deux valeurs de sortes respectives S_1 et S_2 alors le terme "V_1 = V_2" est une valeur de sorte booléenne

1.2.8 Labels

Si $\mathcal{G}$ est un ensemble de portes, $\mathcal{S}$ est un ensemble de sortes et $\mathcal{F}$ est un ensemble d'opérations, on appelle *label* construit sur $\mathcal{G}$, $\mathcal{S}$ et $\mathcal{F}$, un $(n+1)$-uplet de la forme "$G\ V_1, \ldots V_n$", avec $n \geq 0$, où :

- G est une porte de $\mathcal{G}$; on note "$gate(L)$" la porte du label L
- $V_1, \ldots V_n$ sont des valeurs construites sur $\mathcal{S}$ et $\mathcal{F}$

Le nombre de valeurs d'un label peut être quelconque mais si $G =$"**i**", il doit être nul.

1.2.9 Syntaxe graphique

Pour représenter un graphe, on peut employer une représentation graphique comme celle qui a été utilisée dans l'exemple 1-2 (p. 19) :

- chaque état est figuré par un petit cercle
- l'état initial est situé au-dessus de tous les autres
- chaque arc est figuré par une flèche reliant deux états, étiquetée par le label de l'arc

1.3 Relations d'équivalence

Pour comparer les graphes entre eux, un grand nombre de relations ont été proposées : équivalence de sûreté [Rod88], équivalences de test [NH84] [BS86], équivalences de refus [BR85], équivalences d'acceptation [Hen85] [GS86b]. En particulier les *bisimulations* [Par81] constituent une famille de relations pour lesquelles il existe des algorithmes efficaces [Fer88]. On rappelle ici les définitions de trois relations de comparaison qui diffèrent essentiellement par la signification qu'elles attribuent aux arcs étiquetés par le label "**i**" :

- l'équivalence forte : il s'agit de la bisimulation la plus fine. Les arcs étiquetés "**i**" sont traités exactement de la même manière que les autres arcs.

  Le fonctionnement de Cæsar est basé sur l'équivalence forte : la traduction d'un programme Lotos sera considérée comme correcte si le graphe obtenu est fortement équivalent au graphe défini par la sémantique opérationnelle de Lotos [ISO87, § 7.5]

- l'équivalence de trace : cette relation [Hoa78] [BHR84] — qui n'est pas une bisimulation — correspond au concept d'*équivalence des langages* que l'on rencontre en théorie des automates [ASU86, p. 118–121]. Les arcs étiquetés "**i**" sont traités comme les ε-transitions des automates non-déterministes

- l'équivalence observationnelle : cette bisimulation, introduite par [Mil80], est bien adaptée aux systèmes parallèles. Elle joue un rôle intermédiaire par rapport aux relations précédentes : elle est moins fine que l'équivalence forte mais plus fine que l'équivalence de trace

Ces trois relations sont présentées ici sous une forme qui met en évidence les analogies profondes qui existent entre elles. Bien entendu, les définitions données ici sont équivalentes aux définitions usuelles.

1.3.1 Equivalence forte

Deux graphes $(\Sigma, \sigma_0, \mathcal{E}, \mathcal{G}, \mathcal{S}, \mathcal{F})$ et $(\Sigma', \sigma'_0, \mathcal{E}', \mathcal{G}', \mathcal{S}', \mathcal{F}')$ sont dits *fortement équivalents* si et seulement si les conditions suivantes sont satisfaites :

- $\mathcal{G} = \mathcal{G}'$
- $\mathcal{S} = \mathcal{S}'$
- $\mathcal{F} = \mathcal{F}'$
- il existe une relation notée "$\sigma \sim \sigma'$", où σ est un état de Σ et σ' est un état de Σ', vérifiant les deux propriétés ci-dessous :
  - $\sigma_0 \sim \sigma'_0$
  - $(\forall \sigma_1)\ (\forall \sigma'_1)\ (\forall L)\ \sigma_1 \sim \sigma'_1 \Longrightarrow \left\{ \begin{array}{l} (\forall \sigma_2 \in \mathit{succ}(\sigma_1, L))\ (\exists \sigma'_2 \in \mathit{succ}(\sigma'_1, L))\ \sigma_2 \sim \sigma'_2 \\ (\forall \sigma'_2 \in \mathit{succ}(\sigma'_1, L))\ (\exists \sigma_2 \in \mathit{succ}(\sigma_1, L))\ \sigma_2 \sim \sigma'_2 \end{array} \right.$

1.3.2 Equivalence de trace

Soit un graphe $(\Sigma, \sigma_0, \mathcal{E}, \mathcal{G}, \mathcal{S}, \mathcal{F})$; si σ_1 est un état de Σ on appelle *fermeture* de σ_1 et on note $\mathit{closure}(\sigma_1)$ la partie de Σ définie par l'équation récursive :

$$\mathit{closure}(\sigma_1) = \{\sigma_1\} \cup \left(\bigcup_{\sigma_2 \in \mathit{succ}(\sigma_1, \mathbf{i})} \mathit{closure}(\sigma_2) \right)$$

Deux graphes $(\Sigma, \sigma_0, \mathcal{E}, \mathcal{G}, \mathcal{S}, \mathcal{F})$ et $(\Sigma', \sigma'_0, \mathcal{E}', \mathcal{G}', \mathcal{S}', \mathcal{F}')$ sont dits *équivalents pour la trace* si et seulement si les conditions suivantes sont satisfaites :

- $\mathcal{G} = \mathcal{G}'$
- $\mathcal{S} = \mathcal{S}'$
- $\mathcal{F} = \mathcal{F}'$
- il existe une relation notée "$\xi \sim \xi'$", où ξ est un élément de l'ensemble des parties de Σ et ξ' est un élément de l'ensemble des parties de Σ', vérifiant les deux propriétés ci-dessous :
  - $\mathit{closure}(\sigma_0) \sim \mathit{closure}(\sigma'_0)$

– $(\forall \xi_1)\ \ (\forall \xi'_1)\ \ (\forall L \neq \mathbf{i})\ \ \xi_1 \sim \xi'_1 \Longrightarrow \left\{\begin{array}{l} (\xi_2 = \emptyset)\ \wedge\ (\xi'_2 = \emptyset) \\ \vee \\ (\xi_2 \neq \emptyset)\ \wedge\ (\xi'_2 \neq \emptyset)\ \wedge\ (\xi_2 \sim \xi'_2) \end{array}\right.$

$$\text{avec}\ \ \xi_2 = \bigcup_{\sigma_1 \in \xi_1} \left(\bigcup_{\sigma_2 \in succ(\sigma_1, L)} closure(\sigma_2) \right)\ \ \text{et}\ \ \xi'_2 = \bigcup_{\sigma'_1 \in \xi'_1} \left(\bigcup_{\sigma'_2 \in succ(\sigma'_1, L)} closure(\sigma'_2) \right)$$

1.3.3 Equivalence observationnelle

Soit un graphe $(\Sigma, \sigma_0, \mathcal{E}, \mathcal{G}, \mathcal{S}, \mathcal{F})$; si σ_1 est un état de Σ et L un label on note $succ_{\mathbf{i}}(\sigma_1, L)$ la partie de Σ définie par l'équation récursive :

$$succ_{\mathbf{i}}(\sigma_1, L) = \bigcup_{\sigma \in closure(\sigma_1)} \left(\bigcup_{\sigma_2 \in succ(\sigma, L)} closure(\sigma_2) \right)$$

Deux graphes $(\Sigma, \sigma_0, \mathcal{E}, \mathcal{G}, \mathcal{S}, \mathcal{F})$ et $(\Sigma', \sigma'_0, \mathcal{E}', \mathcal{G}', \mathcal{S}', \mathcal{F}')$ sont dits *observationnellement équivalents* [Mil80] si et seulement si les conditions suivantes sont satisfaites :

- $\mathcal{G} = \mathcal{G}'$
- $\mathcal{S} = \mathcal{S}'$
- $\mathcal{F} = \mathcal{F}'$
- il existe une relation notée "$\sigma \sim \sigma'$", où σ est un état de Σ et σ' est un état de Σ', vérifiant les deux propriétés ci-dessous :
  - $\sigma_0 \sim \sigma'_0$
  - $(\forall \sigma_1)\ \ (\forall \sigma'_1)\ \ (\forall L)\ \ \sigma_1 \sim \sigma'_1 \Longrightarrow \left\{\begin{array}{l} (\forall \sigma_2 \in succ_{\mathbf{i}}(\sigma_1, L))\ \ (\exists \sigma'_2 \in succ_{\mathbf{i}}(\sigma'_1, L))\ \ \sigma_2 \sim \sigma'_2 \\ (\forall \sigma'_2 \in succ_{\mathbf{i}}(\sigma'_1, L))\ \ (\exists \sigma_2 \in succ_{\mathbf{i}}(\sigma_1, L))\ \ \sigma_2 \sim \sigma'_2 \end{array}\right.$

Chapitre 2

LOTOS

Ce chapitre présente les notions syntaxiques et sémantiques du langage LOTOS nécessaires à la compréhension de la suite de ce document. Ces définitions sont suffisantes, de sorte que la connaissance de la norme LOTOS [ISO87] ne constitue pas un pré-requis obligatoire. Il ne s'agit pourtant pas d'une introduction à LOTOS : la définition proposée ici peut sembler ardue. Une présentation plus accessible de LOTOS est fournie par les annexes A (p. 203) et B (p. 215).

De la définition formelle du langage LOTOS [ISO87] ne sont repris que les aspects strictement indispensables à l'étude de la sémantique dynamique du parallélisme. C'est pourquoi les notions de sémantique statique [ISO87, § 7.3] (liaison des identificateurs, typage des expressions, ...) ne sont pas traitées ici[13], de même que la définition sémantique des types abstraits algébriques.

On propose tout d'abord une syntaxe abstraite pour le langage LOTOS, qui diffère de la syntaxe concrète donnée dans [ISO87]. Elle correspond fidèlement à la structure de l'arbre abstrait utilisé par le système CÆSAR. Les différences qui existent entre cette syntaxe abstraite et la syntaxe concrète de LOTOS sont précisées, ainsi que les règles qui permettent de passer de l'une à l'autre.

Ensuite on décrit de façon formelle la sémantique dynamique de LOTOS. Il s'agit d'une sémantique opérationnelle basée sur une relation de transition. Cette relation est définie par un système de dérivation dont les règles sont accompagnées d'explications en langage naturel. Elle spécifie comment construire le graphe correspondant à une spécification LOTOS.

2.1 Syntaxe abstraite de LOTOS

La grammaire abstraite décrite ci-dessous a été conçue à partir de la syntaxe concrète de LOTOS [ISO87, § 6.1] et des diagrammes syntaxiques fournis en annexe [ISO87, § D].

On fait l'hypothèse que tous les programmes LOTOS considérés sont corrects, c'est-à-dire qu'ils vérifient les règles imposées par la syntaxe et la sémantique statique.

2.1.1 Symboles non-terminaux

Le tableau suivant présente tous les symboles non-terminaux de la syntaxe abstraite. La syntaxe abstraite proposée couvre la totalité du langage LOTOS. Les constructions syntaxiques concernant

[13] *cf.* [BH88]

les types abstraits algébriques, définies ici, ne seront pas exploitées par la suite ; les non-terminaux qui leur correspondent sont suivis d'un tiret alors que les non-terminaux "utiles" sont marqués d'une croix :

non-terminal	usage	signification
B	×	expression de comportement
block	–	définition de type ou de processus
ceq	–	équation algébrique quantifiée
eqns	–	liste d'équations algébriques
F	×	identificateur d'opération
func	×	fonctionnalité d'un processus
G	×	identificateur de porte
λ	×	identificateur de la spécification
meq	–	équation algébrique non quantifiée
O	×	offre de synchronisation
op	×	opérateur de composition parallèle
opns	–	liste de définitions d'opérations
P	×	identificateur de processus
process	×	définition de processus
program	×	axiome de la grammaire syntaxique
R	×	résultat de terminaison
repl	–	renommage de sortes et d'opérations
S	×	identificateur de sorte
T	×	identificateur de type
type	–	définition de type
V	×	expression de valeur
X	×	identificateur de variable

L'axiome de la grammaire syntaxique est le non-terminal *program*.

La partie essentielle de la grammaire est constituée par la définition des *expressions de comportement*, appelées aussi *comportements*.

2.1.2 Expressions de valeur

$$
\begin{array}{lcl}
V & \equiv & X \\
 & | & F\ \texttt{(}V_1, \ldots\ V_n\texttt{)} \\
 & | & V_1\ \texttt{=}\ V_2
\end{array}
$$

2.1.3 Définitions d'opérateurs

$$
opns \quad \equiv \quad F_0, \ldots\ F_m\ \texttt{:}\ S_1, \ldots\ S_n\ \texttt{->}\ S
$$

2.1.4 Equations algébriques

$$\begin{array}{rcl} \mathit{meq} & \equiv & V_1, \ldots\ V_n \ \texttt{=>}\ V \\ \mathit{ceq} & \equiv & \textbf{ofsort}\ S\ \textbf{forall}\ \widehat{X_1}\!:\!S_1, \ldots\ \widehat{X_m}\!:\!S_m\ \mathit{meq}_0, \ldots\ \mathit{meq}_n \\ \mathit{eqns} & \equiv & \textbf{forall}\ \widehat{X_1}\!:\!S_1, \ldots\ \widehat{X_m}\!:\!S_m\ \mathit{ceq}_0, \ldots\ \mathit{ceq}_n \end{array}$$

2.1.5 Remplacements

$$\begin{array}{rcl} \mathit{repl} & \equiv & \textbf{sortnames}\ S_1\ \textbf{for}\ S'_1, \ldots\ S_p\ \textbf{for}\ S'_p \\ & & \textbf{opnnames}\ F_1\ \textbf{for}\ F'_1, \ldots\ F_q\ \textbf{for}\ F'_q \end{array}$$

2.1.6 Définitions de types

$$\begin{array}{rcl} \mathit{type} & \equiv & \textbf{type}\ T\ \textbf{is}\ T_1, \ldots\ T_n \\ & & \quad \textbf{formalsorts}\ S_1, \ldots\ S_p \\ & & \quad \textbf{formalopns}\ \mathit{opns}_1, \ldots\ \mathit{opns}_q \\ & & \quad \textbf{formaleqns}\ [\mathit{eqns}] \\ & & \quad \textbf{sorts}\ S'_1, \ldots\ S'_{p'} \\ & & \quad \textbf{opns}\ \mathit{opns}'_1, \ldots\ \mathit{opns}'_{q'} \\ & & \quad \textbf{eqns}\ [\mathit{eqns}'] \\ & & \textbf{endtype} \\ & \mid & \textbf{type}\ T\ \textbf{is}\ T' \\ & & \quad \textbf{actualizedby}\ T_0, \ldots\ T_n \\ & & \quad \textbf{using}\ \mathit{repl} \\ & & \textbf{endtype} \\ & \mid & \textbf{type}\ T\ \textbf{is}\ T' \\ & & \quad \textbf{renamedby}\ \mathit{repl} \\ & & \textbf{endtype} \\ & \mid & \textbf{library}\ T_0, \ldots\ T_n \\ & & \textbf{endlib} \end{array}$$

2.1.7 Offres

$$\begin{array}{rcl} O & \equiv & !V \\ & \mid & ?X_0, \ldots\ X_n\!:\!S \end{array}$$

2.1.8 Opérateurs parallèles

$$
\begin{array}{rcl}
op & \equiv & \texttt{||} \\
 & | & \texttt{|||} \\
 & | & \texttt{|[}G_0, \ldots\ G_n\texttt{]|}
\end{array}
$$

2.1.9 Résultats

$$
\begin{array}{rcl}
R & \equiv & V \\
 & | & \textbf{any}\ S
\end{array}
$$

2.1.10 Expressions de comportement

$$
\begin{array}{rcl}
B & \equiv & \textbf{stop} \\
 & | & \textbf{i}\ \texttt{;}\ B_0 \\
 & | & G\ O_1, \ldots\ O_n\ [\texttt{[}V_0\texttt{]}]\ \texttt{;}\ B_0 \\
 & | & B_1\ \texttt{[]}\ B_2 \\
 & | & \textbf{choice}\ \widehat{G_0}\ \textbf{in}\ \texttt{[}\widehat{G'_0}\texttt{]}, \ldots\ \widehat{G_n}\ \textbf{in}\ \texttt{[}\widehat{G'_n}\texttt{]}\ \texttt{[]}\ B_0 \\
 & | & B_1\ op\ B_2 \\
 & | & \textbf{par}\ \widehat{G_0}\ \textbf{in}\ \texttt{[}\widehat{G'_0}\texttt{]}, \ldots\ \widehat{G_n}\ \textbf{in}\ \texttt{[}\widehat{G'_n}\texttt{]}\ op\ B_0 \\
 & | & \textbf{hide}\ G_0, \ldots\ G_n\ \textbf{in}\ B_0 \\
 & | & \texttt{[}V_0\texttt{]}\ \texttt{->}\ B_0 \\
 & | & \textbf{let}\ \widehat{X_0}\texttt{:}S_0\texttt{=}V_0, \ldots\ \widehat{X_n}\texttt{:}S_n\texttt{=}V_n\ \textbf{in}\ B_0 \\
 & | & \textbf{choice}\ \widehat{X_0}\texttt{:}S_0, \ldots\ \widehat{X_n}\texttt{:}S_n\ \texttt{[]}\ B_0 \\
 & | & \textbf{exit}\ \texttt{(}R_1, \ldots\ R_n\texttt{)} \\
 & | & B_1\ \texttt{>>}\ \textbf{accept}\ \widehat{X_1}\texttt{:}S_1, \ldots\ \widehat{X_n}\texttt{:}S_n\ \textbf{in}\ B_2 \\
 & | & B_1\ \texttt{[>}\ B_2 \\
 & | & P\ \texttt{[}G_1, \ldots\ G_n\texttt{]}\ \texttt{(}V_1, \ldots\ V_m\texttt{)}
\end{array}
$$

2.1.11 Blocs

$$
\begin{array}{rcl}
block & \equiv & process \\
 & | & type
\end{array}
$$

2.1.12 Fonctionnalités

$$
\begin{array}{rcl}
\mathit{func} & \equiv & \textbf{noexit} \\
 & | & \textbf{exit}\ \texttt{(}S_1, \ldots\ S_n\texttt{)}
\end{array}
$$

2.1.13 Définitions de processus

$$
\begin{array}{rcl}
\mathit{process} & \equiv & \textbf{process}\ P\ \texttt{[}G_1, \ldots\ G_m\texttt{]}\ \texttt{(}\widehat{X_1}\texttt{:}S_1, \ldots\ \widehat{X_n}\texttt{:}S_n\texttt{)}\ \texttt{:}\ \mathit{func}\ \texttt{:=} \\
 & & \quad B \\
 & & \textbf{where}\ \mathit{block}_1, \ldots\ \mathit{block}_p \\
 & & \textbf{endproc}
\end{array}
$$

2.1.14 Spécification LOTOS

$$
\begin{array}{rcl}
\mathit{program} & \equiv & \textbf{specification}\ \lambda\ \texttt{[}G_1, \ldots\ G_m\texttt{]}\ \texttt{(}\widehat{X_1}\texttt{:}S_1, \ldots\ \widehat{X_n}\texttt{:}S_n\texttt{)}\ \texttt{:}\ \mathit{func} \\
 & & \mathit{type}_1, \ldots\ \mathit{type}_p \\
 & & \textbf{behaviour} \\
 & & \quad B \\
 & & \textbf{where}\ \mathit{block}_1, \ldots\ \mathit{block}_q \\
 & & \textbf{endspec}
\end{array}
$$

2.1.15 Syntaxe concrète et syntaxe abstraite

Cette section — qui peut être évitée en première lecture — explique et justifie les différences qui existent entre la syntaxe abstraite donnée ci-dessus et la syntaxe concrète de LOTOS, telle qu'elle est définie par [ISO87].

Le tableau ci-dessous établit le lien entre les symboles non-terminaux de la syntaxe abstraite de LOTOS et les symboles non-terminaux correspondants de la syntaxe concrète [ISO87, § 6.1] :

non-terminal abstrait	non-terminal concret
B	<behaviour-expression>
block	<local-definition>
ceq	<equation-list>
eqns	<equation-lists>
F	<operation-identifier>
func	<functionality-list>
G	<gate-identifier>
$\widehat{G}$	<gate-identifier-list>
λ	<specification-identifier>
meq	<equation>
O	<experiment-offer>
op	<parallel-operator>
opns	<operation>
P	<process-identifier>
process	<process-definition>
program	<specification>
R	<exit-parameter>
repl	<replacement>
S	<sort-identifier>
T	<type-identifier>
type	<data-type-definition>
V	<value-expression> ou <simple-equation>
X	<value-identifier>
$\widehat{X}$	<value-identifier-list>

Le passage de la syntaxe concrète à la syntaxe abstraite permet de simplifier la définition des traitements sémantiques effectués sur les programmes Lotos ; il vise en particulier à :

- éliminer les symboles terminaux qui ne servent pas directement à l'expression de la sémantique ; par exemple, les parenthèses utilisées pour factoriser les expressions ne figurent plus dans la syntaxe abstraite

- réduire au strict minimum le nombre de symboles non-terminaux ; la syntaxe concrète de Lotos en comporte beaucoup qui peuvent être simplement remplacés par leur définition

- unifier des constructions qui, séparées dans la syntaxe concrète, font pourtant partie du même concept. On remédie ainsi à certains défauts d'orthogonalité du langage, ce qui permet d'obtenir une représentation uniforme des constructions Lotos

- éviter autant que possible l'emploi de la notation "[...]" qui dénote la présence optionnelle d'une construction syntaxique ; en effet l'existence de clauses facultatives complique les définitions sémantiques, notamment celles des attributs.

  Pour cela on effectue une substitution syntaxique : par exemple la clause "[**sorts** $S_0, \dots S_n$]" figurant dans la syntaxe concrète est remplacée par "**sorts** $S_1, \dots S_n$" dans la syntaxe abstraite. On remarque que, dans le second cas, le mot-clé "**sorts**" est présent même lorsque la liste $S_1, \dots S_n$ est vide (i.e. $n = 0$) ; cette différence est sans conséquence ici.

  Cette transformation est largement employée, en particulier pour les mots-clés "**forall**", "**sorts**", "**opns**", "**eqns**", "**formalsorts**", "**formalopns**", "**formaleqns**", "**sortnames**", "**opnnames**", "**accept**", "**where**" et "`=>`" ainsi que pour les symboles "`(...)`" et "`[...]`" lorsqu'ils délimitent des listes éventuellement vides

La syntaxe abstraite des valeurs présente les différences suivantes par rapport à la syntaxe concrète :

- les opérations binaires infixées sont représentées de façon préfixée ; par exemple "$X_1 + X_2$" est mis sous la forme "$+(X_1, X_2)$"
- la clause "**of**" ne figure pas dans la syntaxe abstraite car elle sert uniquement à lever les ambiguïtés au moment de l'analyse des sortes et de la résolution des surcharges d'opérateurs
- dans la syntaxe abstraite, il a été décidé d'unifier les notions de <simple-equation> et de <value-expression> qui étaient distinctes dans la syntaxe concrète. C'est pourquoi les valeurs de la forme "V_1 `=` V_2" ont été introduites : elles correspondent aux équations simples de la syntaxe concrète (le symbole "`=`" dénote la relation d'identité structurelle entre termes et non une quelconque opération d'égalité définie par l'utilisateur).

  Cette transformation simplifie l'expression de la sémantique, statique et dynamique. En revanche elle conduit à la définition d'un *sur-langage* ; la syntaxe abstraite de LOTOS permet certaines constructions qui sont interdites par la syntaxe concrète, comme par exemple "F `(`V_1 `=` V_2`)`". Mais cette différence est sans conséquence
- en LOTOS une prémisse peut être soit une équation simple "V_1 `=` V_2" soit une valeur "V_0". Dans la syntaxe abstraite les prémisses de la forme "V_0" sont converties en valeurs "V_0 `= true`" où "`true`" est une opération définie par l'utilisateur qui a le profil suivant :

  `true : ->` $sort(V_0)$

  Bien entendu cette transformation est compatible avec la sémantique des prémisses [ISO87, § 7.3.4.5.v]

Pour les expressions de comportement, les différences sont moindres :

- dans la syntaxe abstraite, l'opérateur de composition parallèle général "`|[`$G_0, \ldots G_n$`]|`" contient toujours au moins une porte. Tout opérateur "`|[]|`" apparaissant dans le texte source doit être remplacé par "`|||`"
- quand un opérateur de préfixage "`;`" possède une liste d'offres $O_1, \ldots O_n$ vide, la présence d'une garde "`[`V_0`]`" n'est pas autorisée

2.1.16 Attributs locaux

On fait l'hypothèse que tous les identificateurs présents dans la représentation abstraite d'un programme LOTOS sont deux à deux distincts. En effet la compilation attribue à chaque objet un *nom unique*, même si dans le texte source les identificateurs étaient lexicalement identiques. En particulier les opérations ne sont plus surchargées.

On suppose qu'on dispose d'un mécanisme d'*évaluation* des valeurs, c'est-à-dire que toute expression de valeur ne comportant aucune variable peut être réécrite sous une forme normale unique. Si V_1 et V_2 sont deux valeurs de même sorte, le prédicat "$V_1 = V_2$" signifie que V_1 et V_2 ont la même forme normale.

Aux éléments de la syntaxe abstraite sont attachées diverses informations, qui sont des attributs locaux dont la valeur est déterminée par les règles de sémantique statique de LOTOS :

- LOTOS est un langage fortement typé dans lequel toute expression de valeur possède une sorte et une seule. Si V est une expression de valeur, on note "$sort(V)$" la sorte de V, calculée suivant des règles semblables à celles définies précédemment (§ 1.2.7, p. 22)

- en particulier si X est une variable, on note "$sort(X)$" la sorte qui est associée à X par sa déclaration

- si S est une sorte, on appelle *domaine* (*data-carrier*) de S l'ensemble, noté "$domain(S)$", de toutes les valeurs de sorte S. On fait l'hypothèse qu'il n'existe aucune sorte dont le domaine soit vide

- si P est, soit un processus, soit l'identificateur λ de la spécification, P est complètement caractérisé par la donnée des attributs suivants :
  - on appelle *portes formelles* de P, notées "$gate_1(P), \ldots\ gate_m(P)$" les paramètres portes $G_1, \ldots\ G_m$ associés à P par sa définition
  - on appelle *variables formelles* de P, notées "$var_1(P), \ldots\ var_n(P)$" les paramètres variables $X_1, \ldots\ X_n$ associés à P par sa définition
  - on appelle *corps* de P, noté "$behaviour(P)$", le comportement B associé à P par sa définition. Le corps de P est paramétré par les sortes et les variables formelles de P

On note Γ l'ensemble de tous les identificateurs de portes définis par l'utilisateur et qui figurent dans la spécification LOTOS. Deux portes spéciales sont prédéfinies, qui n'appartiennent pas à Γ :

- la porte invisible, notée "**i**"

- la porte de terminaison, notée "δ"

2.2 Sémantique dynamique de LOTOS

On définit la sémantique dynamique des comportements LOTOS de manière compatible — quoiqu'un peu différente dans sa forme — avec la définition du langage [ISO87, § 7.3 et § 7.5].

2.2.1 Expressions de comportement étendues

On appelle *expression de comportement étendue* une expression de comportement définie par les règles syntaxiques de LOTOS (§ 2.1.10, p. 28) auxquelles on ajoute la règle suivante :

$$B \quad \equiv \quad \textbf{rename } G_0 := G'_0, \ldots\ G_n := G'_n \textbf{ in } B_0$$

Cette construction permet le renommage de portes ; sa signification sera donnée plus loin (§ 2.2.5, p. 34).

2.2.2 Relation de transition

La sémantique dynamique de LOTOS est une sémantique de type opérationnel qui spécifie les règles d'évolution des programmes. Elle est complètement définie par la donnée d'une *relation de transition*, notée "$B_1 \xrightarrow{L} B_2$" où :

- B_1 et B_2 sont des expressions de comportement étendues

- L est un label (§ 1.2.8, p. 22) construit sur l'ensemble des portes $\Gamma \cup \{\mathbf{i}, \delta\}$, l'ensemble des sortes et l'ensemble des opérations définies dans la spécification LOTOS. Ce label a donc la forme "$G\ V_1, \ldots\ V_n$", G désignant une porte et $V_1, \ldots\ V_n$ des expressions de valeur ne contenant aucune variable

La signification intuitive de cette relation est "on peut passer du comportement B_1 au comportement B_2 en effectuant l'action définie par le label L".

2.2.3 Règles de dérivation et de substitution

Cette relation de transition est définie par récurrence sur la structure syntaxique des comportements LOTOS. Pour cela on utilise un système de dérivation dont les règles ont la forme usuelle :

$$\frac{P}{Q}$$

qui signifie que le prédicat P implique le prédicat Q.

Afin d'alléger les notations, on emploie aussi des règles de substitution, qui s'écrivent :

$$\underbrace{B_1}_{B_2}$$

et qui signifient que le comportement B_1 doit être remplacé par le comportement B_2. Cette abréviation est équivalente à la règle :

$$\frac{B_2 \xrightarrow{L} B}{B_1 \xrightarrow{L} B}$$

Les règles de substitution permettent d'exprimer des changements d'état qui ne donnent pas lieu à une action observable (ni même à une action étiquetée "**i**").

L'axiome du système de dérivation est "*behaviour*(λ)" si la spécification λ n'a aucune variable formelle, et "**choice** $X_1 : S_1, \ldots\ X_n : S_n$ `[]` *behaviour*(λ)" si elle possède n variables formelles $X_1, \ldots\ X_n$ de sortes respectives $S_1, \ldots\ S_n$. Cette approche permet d'éliminer les variables $X_1, \ldots\ X_n$, de telle sorte que le résultat de l'évaluation des expressions de valeur ne contienne plus aucune variable ; elle diffère légèrement de la définition donnée dans [ISO87] qui manipule $X_1, \ldots\ X_n$ sous forme symbolique. Quant aux portes formelles de la spécification, il faut les conserver car elles correspondent aux actions visibles du système.

2.2.4 Notation "assign"

Dans la définition de la sémantique dynamique de LOTOS, on emploie une notation "**assign**" dont le rôle est d'exprimer la liaison qui existe entre un ensemble de variables et les valeurs qu'elles dénotent. Si $X_0, \ldots\ X_n$ sont des variables, si $V_0, \ldots\ V_n$ sont des expressions de valeur et si B_0 est un comportement contenant des occurrences d'utilisation des variables $X_0, \ldots\ X_n$ alors la notation :

$$\textbf{assign}\ X_0 \texttt{:=} V_0, \ldots\ X_n \texttt{:=} V_n\ \textbf{in}\ B_0$$

dénote le comportement obtenu à partir de B_0 en remplaçant simultanément, pour i décrivant $\{0, \ldots\ n\}$, toutes les occurrences de la variable X_i par la valeur correspondante V_i. Comme toutes les variables ont un nom unique (§ 2.1.16, p. 31), cette substitution est sans problème.

De la même manière on définit la notation :

$$\textbf{assign } X_0\texttt{:=}V_0,\ldots\ X_n\texttt{:=}V_n \textbf{ in } V'_0$$

qui dénote la valeur obtenue en remplaçant simultanément dans V'_0 les variables $X_0,\ldots\ X_n$ par leurs valeurs respectives $V_0,\ldots\ V_n$.

2.2.5 Notation "rename"

En LOTOS une porte peut servir à en repérer une autre ; c'est pourquoi les expressions de comportement étendues possèdent une notation "**rename**" qui permet d'exprimer la liaison entre un ensemble de portes et les portes qu'elles désignent. Si $G_0,\ \ldots\ G_n$ et $G'_0,\ \ldots\ G'_n$ sont des portes et si B_0 est un comportement contenant des occurrences d'utilisation des portes $G_0,\ \ldots\ G_n$ alors la notation suivante :

$$B \quad\equiv\quad \textbf{rename } G_0\texttt{:=}G'_0,\ldots\ G_n\texttt{:=}G'_n \textbf{ in } B_0$$

dénote le comportement obtenu en modifiant les portes des labels des actions effectuées par B_0 de la manière suivante : pour i décrivant $\{0,\ldots\ n\}$ la porte G_i est renommée en G'_i, les autres portes restant inchangées. Comme toutes les portes ont un nom unique (§ 2.1.16, p. 31), ce renommage ne pose aucune difficulté.

Contrairement à "**assign**", la notation "**rename**" ne peut pas être éliminée par simple renommage des portes. Il ne s'agit pas de remplacer syntaxiquement les portes G_i par les portes G'_i dans B_0. Ceci correspondrait à un pré-renommage, avant évolution de B_0 ; au contraire, la signification de "**rename**" est celle d'un post-renommage. C'est pour cette raison qu'il faut ajouter "**rename**" à la syntaxe des expressions de comportement et qu'il faut définir la relation de transition pour "**rename**" :

$$\frac{(B_0 \xrightarrow{G\ V_1,\ldots\ V_m} B'_0)\ \wedge\ (G \notin \{G_0,\ldots\ G_n\})}{\begin{array}{c}(\textbf{rename } G_0\texttt{:=}G'_0,\ldots\ G_n\texttt{:=}G'_n \textbf{ in } B_0) \xrightarrow{G\ V_1,\ldots\ V_m} \\ (\textbf{rename } G_0\texttt{:=}G'_0,\ldots\ G_n\texttt{:=}G'_n \textbf{ in } B'_0)\end{array}}$$

$$\frac{(\exists i \in \{0,\ldots\ n\})\ \ (B_0 \xrightarrow{G_i\ V_1,\ldots\ V_m} B'_0)}{\begin{array}{c}(\textbf{rename } G_0\texttt{:=}G'_0,\ldots\ G_n\texttt{:=}G'_n \textbf{ in } B_0) \xrightarrow{G'_i\ V_1,\ldots\ V_m} \\ (\textbf{rename } G_0\texttt{:=}G'_0,\ldots\ G_n\texttt{:=}G'_n \textbf{ in } B'_0)\end{array}}$$

2.2.6 Opérateur "stop"

Cet opérateur exprime l'*inaction*. Aucune règle d'inférence n'est associée à "**stop**", ce qui signifie qu'aucune action ne peut être issue du comportement "**stop**".

2.2.7 Opérateur ";"

Cet opérateur exprime le *préfixage* d'un comportement B_0 par une synchronisation (*rendez-vous*) sur une porte G avec une liste d'offres $O_1,\ldots\ O_n$. Avant d'exécuter B_0 il faut avoir effectué une action étiquetée par un label L qui dépend de G et de $O_1,\ldots\ O_n$.

Si G est la porte cachée "**i**", il n'y a aucune offre et le franchissement de l'action est toujours possible (on dit que l'action est *invisible* ou encore *spontanée*) :

$$\frac{\mathit{true}}{(\mathbf{i}\ ;\ B_0)\ \xrightarrow{\mathbf{i}}\ B_0}$$

Pour les portes G autres que "**i**", il faut tenir compte des offres $O_1, \ldots\ O_n$ éventuelles. On *linéarise* les offres de réception afin que chaque "?" ne soit suivi que d'une seule variable : "$?X_0, \ldots\ X_n : S$" est remplacé par "$?X_0 : S, \ldots\ ?X_n : S$".

On associe ensuite à chaque offre O_i une valeur V_i' compatible avec cette offre. Intuitivement, V_i' appartient à l'ensemble des valeurs qui peuvent être émises ou reçues par l'offre O_i au moment du rendez-vous sur la porte G :

$$(\forall i \in \{1, \ldots\ n\})\ \ V_i' = \begin{cases} \textbf{si } O_i \equiv\ !V_i \textbf{ alors } V_i \\ \textbf{si } O_i \equiv\ ?X_i : S_i \textbf{ alors } \mathit{oneof}(\mathit{domain}(S_i)) \end{cases}$$

Pour chaque offre O_i et pour chaque valeur V_i' ainsi calculée on définit ensuite l'action A_i réalisée par O_i. Selon la nature de O_i cette action A_i dénote soit l'action vide, soit une affectation. L'exécution du comportement B_0 se fait en tenant compte des actions $A_1, \ldots\ A_n$:

$$(\forall i \in \{1, \ldots\ n\})\ \ A_i = \begin{cases} \textbf{si } O_i \equiv\ !V_i \textbf{ alors } \text{``''} \\ \textbf{si } O_i \equiv\ ?X_i : S_i \textbf{ alors } \text{``}X_i := V_i'\text{''} \end{cases}$$

Si l'opérateur de préfixage ne comporte pas de garde, c'est-à-dire si la clause "$[V_0]$" est absente, il y a autant d'actions possibles que de n-uplets $(V_1', \ldots\ V_n')$. L'une d'entre elles est choisie de manière non-déterministe :

$$\frac{\mathit{true}}{(G\ O_1, \ldots\ O_n\ ;\ B_0)\ \xrightarrow{G\ V_1', \ldots\ V_n'}\ (\textbf{assign } A_1, \ldots\ A_n \textbf{ in } B_0)}$$

Si la garde "$[V_0]$" existe, il y a autant d'actions possibles que de n-uplets $(V_1', \ldots\ V_n')$ pour qui la condition V_0 est vérifiée. L'une d'entre elles est choisie de manière non-déterministe :

$$\frac{(\textbf{assign } A_1, \ldots\ A_n \textbf{ in } V_0) = \mathit{true}}{(G\ O_1, \ldots\ O_n\ [V_0]\ ;\ B_0)\ \xrightarrow{G\ V_1', \ldots\ V_n'}\ (\textbf{assign } A_1, \ldots\ A_n \textbf{ in } B_0)}$$

où $V_1', \ldots\ V_n'$ et $A_1, \ldots\ A_n$ sont définis comme précédemment.

2.2.8 Opérateur "[]"

Cet opérateur exprime le *choix non-déterministe* entre les actions permises par deux comportements B_1 et B_2. Sa sémantique est décrite par deux règles duales :

$$\frac{B_1\ \xrightarrow{L}\ B_1'}{(B_1\ [\,]\ B_2)\ \xrightarrow{L}\ B_1'}$$

$$\frac{B_2\ \xrightarrow{L}\ B_2'}{(B_1\ [\,]\ B_2)\ \xrightarrow{L}\ B_2'}$$

2.2.9 Opérateur "choice" sur les portes

Cet opérateur exprime le *choix non-déterministe généralisé* entre une famille de comportements B_0 paramétrés par des portes $\widehat{G_0}, \ldots \widehat{G_n}$, lorsque ces portes varient respectivement parmi des ensembles de portes $\widehat{G'_0}, \ldots \widehat{G'_n}$.

L'opérateur "**choice**" sur les portes n'est pas primitif car il peut toujours être remplacé par l'opérateur "`[]`". Trois règles de substitution décrivent les transformations nécessaires :

$$\underbrace{\textbf{choice } \widehat{G_0} \textbf{ in } [\widehat{G'_0}], \ldots \widehat{G_n} \textbf{ in } [\widehat{G'_n}] \texttt{ [] } B_0}_{\textbf{choice } \widehat{G_0} \textbf{ in } [\widehat{G'_0}] \texttt{ [] } \ldots \textbf{ choice } \widehat{G_n} \textbf{ in } [\widehat{G'_n}] \texttt{ [] } B_0}$$

$$\underbrace{\textbf{choice } G_0, \ldots G_n \textbf{ in } [\widehat{G'}] \texttt{ [] } B_0}_{\textbf{choice } G_0 \textbf{ in } [\widehat{G'}] \texttt{ [] } \ldots \textbf{ choice } G_n \textbf{ in } [\widehat{G'}] \texttt{ [] } B_0}$$

$$\underbrace{\textbf{choice } G \textbf{ in } [G'_0, \ldots G'_n] \texttt{ [] } B_0}_{(\textbf{rename } G \texttt{:=} G'_0 \textbf{ in } B_0) \texttt{ [] } \ldots (\textbf{rename } G \texttt{:=} G'_n \textbf{ in } B_0)}$$

2.2.10 Opérateurs "`||`", "`|||`" et "`|[...]|`"

Ces trois opérateurs expriment la *composition parallèle* de deux comportements B_1 et B_2. L'opérateur "$\texttt{|[}G_0, \ldots G_n\texttt{]|}$" est le plus général : les portes $G_0, \ldots G_n$ sont appelées *portes synchronisées.*

Si B_1 ou B_2 peut effectuer une action étiquetée par un label L dont la porte n'appartient pas à l'ensemble des portes synchronisées (à laquelle on ajoute "δ"), alors cette action peut être faite de manière asynchrone. Deux règles duales décrivent cette évolution :

$$\frac{(B_1 \xrightarrow{L} B'_1) \wedge (gate(L) \notin \{G_0, \ldots G_n, \delta\})}{(B_1 \texttt{ |[}G_0, \ldots G_n\texttt{]| } B_2) \xrightarrow{L} (B'_1 \texttt{ |[}G_0, \ldots G_n\texttt{]| } B_2)}$$

$$\frac{(B_2 \xrightarrow{L} B'_2) \wedge (gate(L) \notin \{G_0, \ldots G_n, \delta\})}{(B_1 \texttt{ |[}G_0, \ldots G_n\texttt{]| } B_2) \xrightarrow{L} (B_1 \texttt{ |[}G_0, \ldots G_n\texttt{]| } B'_2)}$$

Si B_1 et B_2 peuvent effectuer chacun une action étiquetée par le même label L dont la porte fait partie de l'ensemble des portes synchronisées, alors cette action doit être faite de manière synchrone :

$$\frac{(B_1 \xrightarrow{L} B'_1) \wedge (B_2 \xrightarrow{L} B'_2) \wedge (gate(L) \in \{G_0, \ldots G_n, \delta\})}{(B_1 \texttt{ |[}G_0, \ldots G_n\texttt{]| } B_2) \xrightarrow{L} (B'_1 \texttt{ |[}G_0, \ldots G_n\texttt{]| } B'_2)}$$

L'opérateur "`||`" exprime la composition parallèle totalement synchrone : B_1 et B_2 doivent se synchroniser sur toutes les portes de Γ (mais pas sur la porte "**i**"). L'opérateur "`|||`" exprime la composition parallèle totalement asynchrone : B_1 et B_2 ne doivent se synchroniser sur aucune porte, sauf "δ". Deux règles de substitution permettent de se ramener au cas de l'opérateur de composition parallèle général :

$$\underbrace{B_1 \;\texttt{||}\; B_2}_{B_1 \;\texttt{|[}\Gamma\texttt{]|}\; B_2}$$

$$\underbrace{B_1 \;\texttt{|||}\; B_2}_{B_1 \;\texttt{|[]|}\; B_2}$$

2.2.11 Opérateur "par"

Cet opérateur exprime la *composition parallèle généralisée* entre une famille de comportements B_0 paramétrés par des portes $\widehat{G_0}, \ldots\ \widehat{G_n}$, lorsque ces portes varient respectivement parmi des ensembles de portes $\widehat{G'_0}, \ldots\ \widehat{G'_n}$.

L'opérateur "**par**... *op*" sur les portes n'est pas primitif car il peut toujours être remplacé par l'opérateur "*op*". Trois règles de substitution décrivent les transformations nécessaires :

$$\underbrace{\textbf{par}\ \widehat{G_0}\ \textbf{in}\ \texttt{[}\widehat{G'_0}\texttt{]}, \ldots\ \widehat{G_n}\ \textbf{in}\ \texttt{[}\widehat{G'_n}\texttt{]}\ \ op\ B_0}_{\textbf{par}\ \widehat{G_0}\ \textbf{in}\ \texttt{[}\widehat{G'_0}\texttt{]}\ \ op\ \ldots\ \textbf{par}\ \widehat{G_n}\ \textbf{in}\ \texttt{[}\widehat{G'_n}\texttt{]}\ \ op\ B_0}$$

$$\underbrace{\textbf{par}\ G_0, \ldots\ G_n\ \textbf{in}\ \texttt{[}\widehat{G'}\texttt{]}\ \ op\ B_0}_{\textbf{par}\ G_0\ \textbf{in}\ \texttt{[}\widehat{G'}\texttt{]}\ \ op\ \ldots\ \textbf{par}\ G_n\ \textbf{in}\ \texttt{[}\widehat{G'}\texttt{]}\ \ op\ B_0}$$

$$\underbrace{\textbf{par}\ G\ \textbf{in}\ \texttt{[}G'_0, \ldots\ G'_n\texttt{]}\ \ op\ B_0}_{(\textbf{rename}\ G\texttt{:=}G'_0\ \textbf{in}\ B_0)\ \ op\ \ldots\ (\textbf{rename}\ G\texttt{:=}G'_n\ \textbf{in}\ B_0)}$$

2.2.12 Opérateur "hide"

Cet opérateur exprime l'*abstraction* : il permet de rendre "invisibles" certaines actions effectuées par un comportement B_0. Cette abstraction est déterminée par un ensemble de portes $G_0, \ldots\ G_n$, appelées *portes cachées*.

Si B_0 peut effectuer une action étiquetée par un label L dont la porte ne fait pas partie de l'ensemble des portes cachées, alors cette action reste observable :

$$\frac{(B_0 \xrightarrow{L} B'_0) \wedge (gate(L) \notin \{G_0, \ldots\ G_n\})}{(\textbf{hide}\ G_0, \ldots\ G_n\ \textbf{in}\ B_0) \xrightarrow{L} (\textbf{hide}\ G_0, \ldots\ G_n\ \textbf{in}\ B'_0)}$$

Si, au contraire, la porte de L appartient à l'ensemble des portes cachées, alors L est remplacé par "**i**" :

$$\frac{(B_0 \xrightarrow{L} B'_0) \wedge (gate(L) \in \{G_0, \ldots\ G_n\})}{(\textbf{hide}\ G_0, \ldots\ G_n\ \textbf{in}\ B_0) \xrightarrow{\textbf{i}} (\textbf{hide}\ G_0, \ldots\ G_n\ \textbf{in}\ B'_0)}$$

2.2.13 Opérateur "->"

Cet opérateur exprime la *garde* d'un comportement B_0 par une valeur V_0 : l'exécution de B_0 n'est possible que si la condition V_0 est vérifiée :

$$\frac{(B_0 \xrightarrow{L} B_0') \wedge (V_0 = \mathit{true})}{(\texttt{[}V_0\texttt{] ->} B_0) \xrightarrow{L} B_0'}$$

2.2.14 Opérateur "let"

Cet opérateur permet de définir dans un comportement B_0 des variables $\widehat{X_0}, \ldots \widehat{X_n}$, de sortes respectives $S_0, \ldots S_n$, et de leur donner respectivement des valeurs $V_0, \ldots V_n$.

Deux règles de substitution remplacent la forme générale de l'opérateur "**let**" par une forme simple ne comprenant qu'une seule définition de variable :

$$\underbrace{\textbf{let } \widehat{X_0}\!:\!S_0\texttt{=}V_0, \ldots \widehat{X_n}\!:\!S_n\texttt{=}V_n \textbf{ in } B_0}_{\textbf{let } \widehat{X_0}:S_0\texttt{=}V_0 \textbf{ in } \ldots \textbf{ let } \widehat{X_n}:S_n\texttt{=}V_n \textbf{ in } B_0}$$

$$\underbrace{\textbf{let } X_0, \ldots X_n\!:\!S\texttt{=}V \textbf{ in } B_0}_{\textbf{let } X_0:S\texttt{=}V \textbf{ in } \ldots \textbf{ let } X_n:S\texttt{=}V \textbf{ in } B_0}$$

Une troisième règle exprime que les occurrences dans B_0 de la variable X doivent être remplacées par sa valeur V :

$$\underbrace{\textbf{let } X\!:\!S\texttt{=}V \textbf{ in } B_0}_{\textbf{assign } X\texttt{:=}V \textbf{ in } B_0}$$

2.2.15 Opérateur "choice" sur les valeurs

Cet opérateur exprime le *choix non-déterministe généralisé* entre une famille de comportements B_0 paramétrés par des variables $\widehat{X_0}, \ldots \widehat{X_n}$, de sortes respectives $S_0, \ldots S_n$, lorsque ces variables décrivent respectivement toutes les valeurs appartenant aux domaines de leurs sortes.

Deux règles de substitution ramènent la forme générale de l'opérateur "**choice**"sur les valeurs à une forme simple ne comprenant qu'une seule définition de variable :

$$\underbrace{\textbf{choice } \widehat{X_0}\!:\!S_0, \ldots \widehat{X_n}\!:\!S_n \texttt{ [] } B_0}_{\textbf{choice } \widehat{X_0}:S_0 \texttt{ [] } \ldots \textbf{ choice } \widehat{X_n}:S_n \texttt{ [] } B_0}$$

$$\underbrace{\textbf{choice } X_0, \ldots X_n\!:\!S \texttt{ [] } B_0}_{\textbf{choice } X_0:S \texttt{ [] } \ldots \textbf{ choice } X_n:S \texttt{ [] } B_0}$$

Une troisième règle exprime que les occurrences dans B_0 de la variable X doivent être remplacées par une valeur de sorte S sans qu'il soit précisé comment cette valeur est choisie :

$$\underbrace{\textbf{choice } X\!:\!S \texttt{ [] } B_0}_{\textbf{assign } X\texttt{:=}\mathit{oneof}(\mathit{domain}(S)) \textbf{ in } B_0}$$

2.2.16 Opérateur "exit"

Cet opérateur exprime la *terminaison* avec éventuellement émission d'une liste de résultats $R_1, \ldots R_n$.

La terminaison est modélisée par un rendez-vous sur la porte "δ". On associe à chaque résultat R_i une valeur V_i' compatible avec ce résultat. Intuitivement, V_i' appartient à l'ensemble des valeurs qui peuvent être envoyées comme résultat :

$$(\forall i \in \{1, \ldots n\})\ V_i' = \begin{cases} \textbf{si } R_i \equiv V_i \textbf{ alors } V_i \\ \textbf{si } R_i \equiv \textbf{any } S_i \textbf{ alors } \mathit{oneof}(\mathit{domain}(S_i)) \end{cases}$$

Il y a autant d'actions possibles que de n-uplets $(V_1', \ldots V_n')$. L'une d'entre elles est choisie de manière non-déterministe :

$$\frac{\mathit{true}}{\textbf{exit } (R_1, \ldots R_n) \xrightarrow{\delta\ V_1', \ldots V_n'} \textbf{stop}}$$

2.2.17 Opérateur ">>"

Cet opérateur exprime la *composition séquentielle* entre un comportement B_1 qui s'achève et un comportement B_2 qui commence.

On commence par linéariser les déclarations de variables qui suivent le mot "**accept**": chaque déclaration "$X_0, \ldots X_n\!:\!S$" est remplacée par "$X_0\!:\!S, \ldots X_n\!:\!S$".

Tant que B_1 n'effectue aucune action dont la porte est "δ", il est le seul à pouvoir évoluer :

$$\frac{(B_1 \xrightarrow{L} B_1') \wedge (\mathit{gate}(L) \neq \delta)}{\begin{array}{c}(B_1 \texttt{ >> } \textbf{accept } X_1\!:\!S_1, \ldots X_n\!:\!S_n \textbf{ in } B_2) \xrightarrow{L} \\ (B_1' \texttt{ >> } \textbf{accept } X_1\!:\!S_1, \ldots X_n\!:\!S_n \textbf{ in } B_2)\end{array}}$$

Dès que B_1 effectue une action étiquetée "$\delta\ V_1, \ldots V_n$" l'exécution de B_1 est terminée et celle de B_2 commence. Cette passation du contrôle entre B_1 et B_2 est modélisée par un rendez-vous sur la porte "δ" qui, après abstraction, produit une action "invisible".On exécute ensuite B_2 en donnant respectivement les valeurs $V_1, \ldots V_n$ aux variables $X_1, \ldots X_n$ de B_2 définies dans la clause "**accept**" :

$$\frac{(B_1 \xrightarrow{\delta\ V_1, \ldots V_n} B_1')}{\begin{array}{c}(B_1 \texttt{ >> } \textbf{accept } X_1\!:\!S_1, \ldots X_n\!:\!S_n \textbf{ in } B_2) \xrightarrow{\mathbf{i}} \\ (\textbf{assign } X_1\texttt{:=}V_1, \ldots X_n\texttt{:=}V_n \textbf{ in } B_2)\end{array}}$$

2.2.18 Opérateur "[>"

Cet opérateur exprime l'*interruption* d'un comportement B_1 par un comportement B_2. Si B_2 peut effectuer une action, l'exécution de B_1 peut être abandonnée au profit de celle de B_2 :

$$\frac{B_2 \xrightarrow{L} B_2'}{(B_1 \texttt{ [> } B_2) \xrightarrow{L} B_2'}$$

Tant que B_1 n'effectue pas d'action ayant "δ" pour porte, il peut évoluer, tout en restant interruptible par B_2 :

$$\frac{(B_1 \xrightarrow{L} B_1') \wedge (gate(L) \neq \delta)}{(B_1 \texttt{ [> } B_2) \xrightarrow{L} (B_1' \texttt{ [> } B_2)}$$

Dès que B_1 a effectué une action ayant "δ" pour porte, il ne peut plus être interrompu par B_2 :

$$\frac{(B_1 \xrightarrow{L} B_1') \wedge (gate(L) = \delta)}{(B_1 \texttt{ [> } B_2) \xrightarrow{L} B_1'}$$

2.2.19 Processus et instanciation

Une définition de processus permet de donner un nom à un comportement paramétré par des portes formelles et des variables formelles. Inversement, une *instanciation de processus* permet de retrouver le comportement associé à un processus P, en substituant aux portes formelles de P des portes effectives $G_1, \ldots\ G_m$ et aux variables formelles de P des valeurs effectives $V_1, \ldots\ V_n$:

$$\underbrace{P \texttt{ [} G_1, \ldots\ G_m \texttt{] (} V_1, \ldots\ V_n \texttt{)}}_{\substack{\textbf{rename } gate_1(P)\texttt{:=}G_1, \ldots\ gate_m(P)\texttt{:=}G_m \textbf{ in} \\ \textbf{assign } var_1(P)\texttt{:=}V_1, \ldots\ var_n(P)\texttt{:=}V_n \textbf{ in} \\ behaviour(P)}}$$

2.2.20 Construction du graphe

A partir de la spécification LOTOS λ on construit un automate $(\Sigma, \sigma_0, \mathcal{E}, \mathcal{G}, \mathcal{S}, \mathcal{F})$ où :

- les états de Σ sont des expressions de comportement étendues. Plus précisément, Σ est défini par l'équation suivante :

$$\Sigma = \{behaviour(\lambda)\} \cup \{B_2 \mid (\exists B_1 \in \Sigma)\ (\exists L)\ (B_1 \xrightarrow{L} B_2)\}$$

- en notant $X_1, \ldots\ X_n$ les variables formelles $var_1(\lambda), \ldots\ var_n(\lambda)$ de la spécification et $S_1, \ldots\ S_n$ leurs sortes respectives, l'état initial σ_0 est formé par l'expression de comportement étendue B_0 telle que :

$$B_0 = \begin{cases} \textbf{si } n = 0 \textbf{ alors } behaviour(\lambda) \\ \textbf{sinon (choice } X_1 \texttt{:} S_1, \ldots\ X_n \texttt{:} S_n \texttt{ [] } behaviour(\lambda)\texttt{)} \end{cases}$$

- les arcs de $\mathcal{E}$ sont déterminés par la relation de transition :

$$\mathcal{E} = \{(B_1, L, B_2) \mid (B_1 \in \Sigma) \ \wedge \ (B_2 \in \Sigma) \ \wedge \ (B_1 \ \xrightarrow{L} \ B_2)\}$$

- $\mathcal{G}$ est l'ensemble des portes LOTOS visibles de λ, c'est-à-dire $gate_1(\lambda), \ldots \ gate_n(\lambda)$ auxquelles on ajoute la porte "**i**" et la porte "δ". En effet, les règles de sémantique statique et dynamique font que, dans chaque label L, les portes de Γ autres que $gate_1(\lambda), \ldots \ gate_n(\lambda)$ et "δ" ont été renommées en "**i**"

- $\mathcal{S}$ est l'ensemble des sortes LOTOS présentes dans λ

- $\mathcal{F}$ est l'ensemble des opérations LOTOS présentes dans λ

Chapitre 3

SUBLOTOS

Le but de CÆSAR est de produire, à partir d'une spécification LOTOS, le graphe correspondant. Pour cela, il faut que le comportement décrit par la spécification soit représentable par un automate *fini*. On dit alors que le programme LOTOS est *régulier*.

Toutefois cette condition n'est pas suffisante car CÆSAR impose des contraintes supplémentaires aux programmes LOTOS pour que la génération d'une forme intermédiaire à contrôle statique (réseau) soit possible.

On énonce et on justifie ces restrictions, qui portent sur le contrôle. Un algorithme permettant de les vérifier efficacement est donné. On discute ensuite le bien-fondé de restrictions semblables pour les données.

La traduction d'un programme LOTOS en réseau est une opération trop complexe pour être effectuée directement. C'est pourquoi il a été jugé préférable de la diviser en deux phases successives, appelées respectivement *expansion* et *génération*. Cette décomposition améliore la modularité du système et réduit sa complexité.

Une forme intermédiaire a été conçue pour servir d'interface entre l'expansion et la génération: il s'agit du langage SUBLOTOS qui peut être considéré comme un sous-ensemble simplifié de LOTOS. On donne tout d'abord un aperçu de SUBLOTOS, notamment en le comparant à LOTOS. On définit ensuite formellement ce langage en donnant sa syntaxe et sa sémantique dynamique.

3.1 Restrictions sur le contrôle

On cherche à mettre en évidence et à caractériser, de manière syntaxique, un certain nombre de situations qui, lorsqu'elles sont présentes dans une spécification LOTOS, sont susceptibles de produire un graphe infini et d'empêcher la génération d'un modèle à contrôle statique.

3.1.1 Notations préliminaires

On dit qu'une instanciation d'un processus P est *récursive à travers un opérateur unaire de comportement* — noté ici "$\diamond$" — si et seulement si :

- elle est récursive, directement ou transitivement, c'est-à-dire qu'elle est située dans le corps de P ou dans le corps d'un processus pouvant être instancié par P. Attention, ce n'est pas parce que P est récursif que toute instanciation de P est récursive !

- elle figure dans un comportement auquel s'applique l'opérateur "$\diamond$"

On dit qu'une instanciation d'un processus P est *récursive à gauche (*resp. *à droite) d'un opérateur binaire de comportement* — noté ici "$\diamond$" — si et seulement si :

- elle est récursive, directement ou transitivement
- elle figure dans un comportement qui apparaît comme opérande gauche (*resp.* droit) de l'opérateur "$\diamond$"

La sémantique opérationnelle de LOTOS conduit à développer les instanciations, c'est-à-dire à remplacer chaque instanciation d'un processus P par le corps du processus P. C'est pourquoi, si P est un processus récursif, on définit une suite de comportements dont le $n^{\text{ième}}$ terme, noté $B(P, n)$, est obtenu en développant n fois les instanciations récursives du processus P. Cette suite est définie par récurrence :

- $B(P, 0)$ est égal à "**stop**"
- $B(P, n + 1)$ est égal au corps du processus (c'est-à-dire "*behaviour*(P)") dans lequel les instanciations de P sont remplacées par $B(P, n)$

3.1.2 Récursion à travers les opérateurs "`||`", "`|||`" et "`|[...]|`"

On constate que, très souvent, un programme LOTOS n'est pas régulier quand il contient une instanciation récursive à gauche et/ou à droite d'un opérateur parallèle ("`||`", "`|[...]|`" et "`|||`").

Exemple 3-1
Le comportement "P [G_1, G_2]" dans lequel P est le processus défini comme suit :

process P [G_1, G_2] : **noexit** :=
  G_1 ; **stop** ||| G_2 ; P [G_1, G_2]
endproc

n'est pas régulier. Pour le prouver, il suffit de montrer que l'ensemble $\mathcal{L}$ des séquences d'exécution finies dérivées de ce comportement ne constitue pas un langage régulier[14] (ce qui constitue la contraposée du fait que la relation d'équivalence forte est plus fine que la relation d'équivalence de trace). Dans le cas présent $\mathcal{L}$ comprend tous les mots construits sur le vocabulaire $\{G_1, G_2\}$ tels que tous les préfixes de ces mots vérifient la propriété suivante : si n_1 et n_2 sont les nombres respectifs de symboles G_1 et G_2 présents dans un préfixe donné, on a toujours $n_1 \leq n_2 + 1$. L'intersection de deux langages régulier est toujours un langage régulier. Or l'intersection de $\mathcal{L}$ avec le langage défini par l'expression régulière $G_1^*.G_2^*$ est égale au langage :

$$\{G_1^{n_1}.G_2^{n_2} \mid n_1 \leq n_2 + 1\}$$

qui n'est pas régulier ; par conséquent $\mathcal{L}$ n'est pas régulier. ■

Toutefois, dans certains cas particuliers, la récursion à gauche (*resp.* à droite) d'un opérateur n'empêche pas la régularité.

Exemple 3-2
Le comportement "P [G_1, G_2]" dans lequel P est le processus défini comme suit :

process P [G_1, G_2] : **noexit** :=

[14] on dit aussi rationnel

$$G_1 \ ; \ G_2 \ ; \ \textbf{stop} \ || \ G_1 \ ; \ G_2 \ ; \ P \ [G_1, G_2]$$
endproc

est régulier, puisqu'il est équivalent à "G_1 ; G_2 ; **stop**". Il ne le serait probablement pas si toutes les portes de l'opérande gauche n'étaient pas synchronisées par l'opérateur parallèle, en particulier si ces portes étaient cachées. ■

La récursion à travers l'opérateur "**par**" a les mêmes conséquences que la récursion à gauche ou à droite d'un opérateur parallèle, sauf dans le cas où chaque porte variable définie dans l'opérateur "**par**" n'a qu'une valeur possible.

Exemple 3-3
Le comportement "P [G_1, G_2]" dans lequel P est le processus défini comme suit :

process P [G_1, G_2] : **noexit** :=
  par H_1, H_2 **in** [G_1], H_3 **in** [G_2] ||| H_1 ; P [H_2, H_3]
endproc

est régulier, puisqu'il est équivalent à "G_1 ; G_1 ; G_1 ; ...". ■

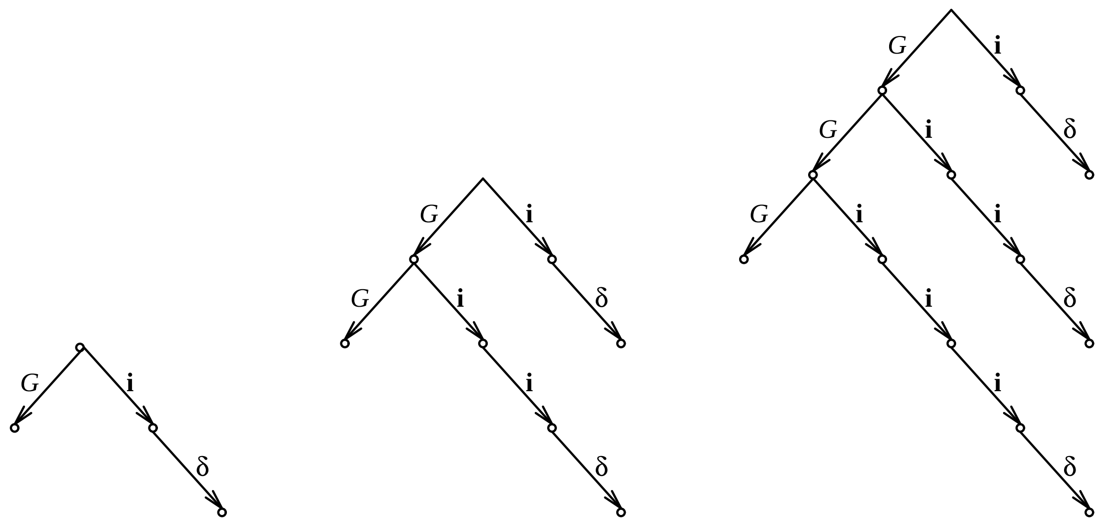
3.1.3 Récursion à travers l'opérateur ">>"

Très souvent, un programme LOTOS n'est pas régulier quand il contient une instanciation récursive à gauche de l'opérateur ">>". En revanche la récursion à droite de l'opérateur ">>" ne soulève pas de difficulté.

Exemple 3-4
Le comportement "P [G]" dans lequel P est le processus défini comme suit :

process P [G] : **exit** :=
  (G ; P [G] [] **exit**) >> **exit**
endproc

n'est pas régulier. La représentation graphique des premiers termes de la suite $B(P, n)$ le montre clairement :



graphe de B(P, 1) *graphe de B(P, 2)* *graphe de B(P, 3)*

Plus formellement l'ensemble $\mathcal{L}$ des séquences d'exécution dérivées de ce comportement est égal au langage :

$$\{G^n \mid n \geq 0\} \cup \{G^n.\mathbf{i}^m \mid (n \geq 0) \ \wedge \ (m \leq n+1)\} \cup \{G^n.\mathbf{i}^{n+1}.\delta \mid n \geq 0\}$$

qui n'est pas évidemment pas régulier ; le comportement "P [G]" n'est donc pas régulier. ■

Bien entendu, il existe des cas où la récursion à gauche de l'opérateur ">>" n'empêche pas la régularité.

Exemple 3-5
Le comportement "P [G_1, G_2]" dans lequel P est le processus défini comme suit :

process P [G_1, G_2] : **exit** :=
G_1 ; P [G_1, G_2] >> G_2 ; **exit**
endproc

est régulier, puisqu'il est équivalent à "G_1 ; G_1 ; G_1 ; ...". ■

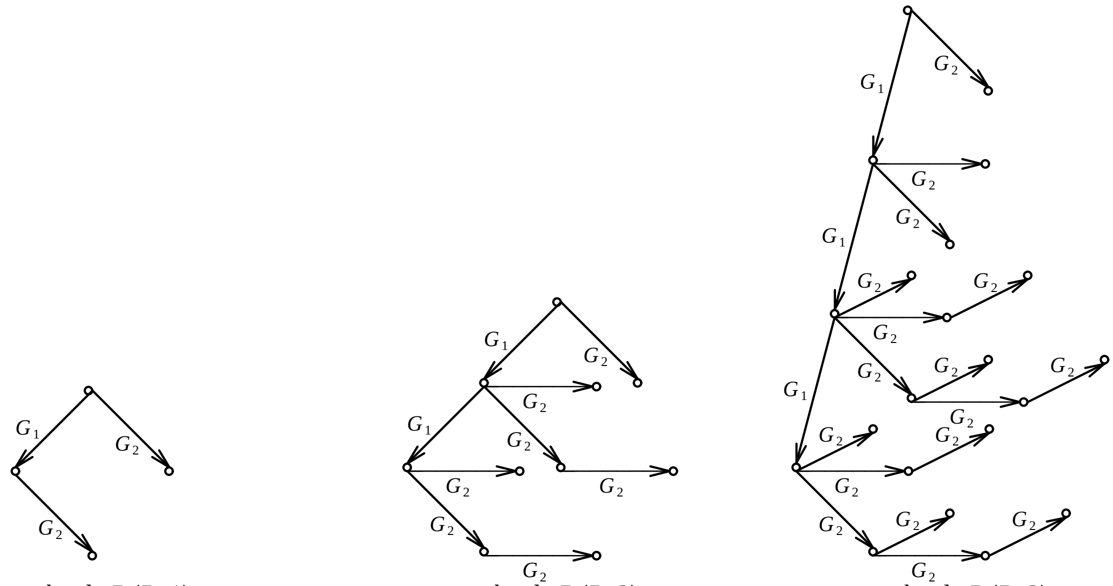
3.1.4 Récursion à travers l'opérateur "[>"

Un programme Lotos n'est généralement pas régulier quand il contient une instanciation récursive à gauche de l'opérateur "[>". En revanche la récursion à droite de l'opérateur "[>" ne pose pas de problème.

Exemple 3-6
Le comportement "P [G_1, G_2]" dans lequel P est le processus défini comme suit :

process P [G_1, G_2] : **noexit** :=
(G_1 ; P [G_1, G_2]) [> G_2 ; **stop**
endproc

n'est pas régulier, comme le suggère la représentation graphique des premiers termes de la suite B(P,n) :



graphe de $B(P,1)$ *graphe de $B(P,2)$* *graphe de $B(P,3)$*

L'ensemble $\mathcal{L}$ des séquences d'exécution dérivées de ce comportement est égal au langage :

$$\{G_1{}^n \mid n \geq 0\} \cup \{G_1{}^n.G_2{}^m \mid (n \geq 0) \ \wedge \ (m \leq n+1)\}$$

qui n'est pas évidemment pas régulier ; le comportement "P [G_1, G_2]" n'est donc pas régulier. ■

Il existe cependant des cas où la récursion à gauche de l'opérateur "[>" ne nuit pas à la régularité.

Exemple 3-7
Le comportement "P [G]" dans lequel P est le processus défini comme suit :

```
process P [G] : exit :=
   (G ; P [G]) [> exit
endproc
```

est régulier. En notant "$B_1 \longrightarrow B_2$" le fait qu'il est possible de dériver le comportement B_2 à partir du comportement B_1 en appliquant les règles de sémantique dynamique de LOTOS, on a en effet :

$$\begin{array}{lll} P \ [G] & \longrightarrow & (G \ ; \ P \ [G]) \ [> \ \mathbf{exit} \\ & \longrightarrow & \mathbf{exit} \ [\,] \ G \ ; \ (P \ [G] \ [> \ \mathbf{exit}) \\ & \longrightarrow & \mathbf{exit} \ [\,] \ G \ ; \ (((G \ ; \ P \ [G]) \ [> \ \mathbf{exit}) \ [> \ \mathbf{exit}) \end{array}$$

En utilisant l'associativité de l'opérateur "[>", il vient :

$$\begin{array}{lll} P \ [G] & \longrightarrow & \mathbf{exit} \ [\,] \ G \ ; \ ((G \ ; \ P \ [G]) \ [> \ (\mathbf{exit} \ [> \ \mathbf{exit})) \\ & \longrightarrow & \mathbf{exit} \ [\,] \ G \ ; \ ((G \ ; \ P \ [G]) \ [> \ \mathbf{exit}) \\ & \longrightarrow & \mathbf{exit} \ [\,] \ G \ ; \ P \ [G] \end{array}$$

ce qui prouve que le comportement P [G] est régulier. ■

3.1.5 Propriété du contrôle statique

On dit qu'un programme LOTOS vérifie la *propriété du contrôle statique* si et seulement si il ne comporte pas de récursion à travers les opérateurs parallèles, "**par**", "[>" et ">>". La négation de contrôle statique est *contrôle dynamique*.

Tous les programmes qui ne vérifient pas la propriété du contrôle statique sont systématiquement rejetés par CÆSAR. Plusieurs raisons motivent cette interdiction :

- selon un résultat établi par [Ail86], tout programme LOTOS dont le contrôle est statique et qui ne contient ni valeurs ni sortes est régulier. On peut donc espérer que cette contrainte, après avoir été complétée par des restrictions appropriées portant sur les valeurs et les sortes, constituera une condition suffisante de régularité

- la propriété du contrôle statique est décidable : un algorithme simple et performant est donné plus loin (§ 3.1.6, p. 49)

- pour l'utilisateur cette contrainte quasi-syntaxique est simple à décrire et sa compréhension ne nécessite pas la connaissance du fonctionnement interne de CÆSAR

- cette restriction ne réduit pas exagérément le sous-ensemble des programmes LOTOS acceptés par CÆSAR. Bien entendu il est possible d'exprimer tout graphe fini par un programme LOTOS dont le contrôle est statique (en utilisant uniquement les opérateurs "`;`", "`[]`" et l'instanciation de processus).

La réciproque n'est pas vraie : les exemples 3-2 (p. 44), 3-3 (p. 45), 3-5 (p. 46) et 3-7 (p. 47) montrent plusieurs situations dans lesquelles un programme à contrôle dynamique peut néanmoins s'avérer régulier. Ces comportements peuvent être décrits beaucoup plus simplement sans utiliser de récursion interdite ; il appartient à l'utilisateur de les exprimer autrement.

En pratique, on constate que les possibilités offertes par le contrôle dynamique ont deux utilisations principales :

– l'emploi de la récursion à travers un opérateur parallèle permet de décrire des structures de données dynamiques (files d'attente, piles, ...). Il s'agit sans doute d'une utilisation abusive des structures de contrôle ; [Led87] souligne l'aspect néfaste de ce style de programmation, alors que les possibilités offertes par les types abstraits répondent parfaitement à ces besoins

– l'utilisation de la récursion *limitée* à travers un opérateur parallèle constitue un moyen élégant pour lancer simultanément N comportements en parallèle.

Exemple 3-8
Le comportement "P `[`G`]` `(`N`)`" dans lequel P est le processus défini comme suit :

$$\begin{array}{l}\textbf{process}\ P\ \texttt{[}G\texttt{]}\ \texttt{(}X\texttt{:}NAT\texttt{)}\ \texttt{:}\ \textbf{noexit}\ \texttt{:=}\\ \quad \texttt{[}X \geq 1\texttt{]}\ \texttt{->}\ \texttt{(}P_0\ \texttt{[}G\texttt{]}\ \texttt{(}X\texttt{)}\ \texttt{|||}\ P\ \texttt{[}G\texttt{]}\ \texttt{(}X-1\texttt{))}\\ \textbf{endproc}\end{array}$$

est régulier si le comportement du processus P_0 est régulier ; en effet il est équivalent à :

$$P_0\ \texttt{[}G\texttt{]}\ \texttt{(1)}\ \texttt{|||}\ \ldots\ P_0\ \texttt{[}G\texttt{]}\ \texttt{(}N\texttt{)}$$

■

Si la valeur de N peut être déterminée statiquement, l'utilisateur doit utiliser la seconde forme, c'est-à-dire développer lui-même la récursion. Dans le cas contraire, l'approche adoptée pour CÆSAR ne permet pas de traiter ce genre de situations

- cet inconvénient est le prix à payer pour l'efficacité de la méthode de traduction. En effet on montrera plus loin (§ 4.6.1, p. 90) que les programmes à contrôle dynamique ne peuvent être traités qu'au détriment des performances : pour les prendre en compte il faudrait renoncer aux bénéfices de l'approche statique. En exigeant que le contrôle soit statique on choisit de ne pas pénaliser la grande majorité des programmes "utiles"

Remarque 3-1
On dit qu'une instanciation récursive est *non gardée* lorsqu'il n'est pas possible de franchir un arc entre deux appels récursifs. L'exemple le plus simple est le suivant :

$$\begin{array}{l}\textbf{process}\ P\ \texttt{:}\ \textbf{noexit}\ \texttt{:=}\\ \quad P\\ \textbf{endproc}\end{array}$$

mais on peut imaginer des exemples plus complexes :

$$\begin{array}{l}\textbf{process}\ P\ \texttt{(}X\texttt{:}S\texttt{)}\ \texttt{:}\ \textbf{noexit}\ \texttt{:=}\\ \quad \textbf{let}\ X'\texttt{:}S\texttt{=}F\ \texttt{(}X\texttt{)}\ \textbf{in}\\ \qquad P\ \texttt{(}X'\texttt{)}\\ \textbf{endproc}\end{array}$$

La récursion non gardée n'est pas interdite, sauf si elle enfreint la propriété du contrôle statique. Elle est équivalente à "**stop**" puisqu'on ne peut en dériver aucun arc. ■

3.1.6 Algorithme du contrôle statique

On rappelle que chaque processus est identifié par un nom unique. Dans cette section on considèrera que l'identificateur λ de la spécification fait partie des processus. A chaque processus P on associe un attribut local noté $\mathcal{C}(P)$ qui est un ensemble de sextuplets (P', L, R, M, E, D), où P' est un processus et L, R, M, E, D sont cinq booléens. Le sextuplet (P', L, R, M, E, D) appartient à $\mathcal{C}(P)$ si et seulement si :

- le corps de P contient une instanciation du processus P'
- $L = \mathit{true}$ si et seulement si cette instanciation apparaît dans l'opérande gauche d'un opérateur "`||`", "`|[...]|`" ou "`|||`" (L pour *left*)
- $R = \mathit{true}$ si et seulement si cette instanciation apparaît dans l'opérande droit d'un opérateur "`||`", "`|[...]|`" ou "`|||`" (R pour *right*)
- $M = \mathit{true}$ si et seulement si cette instanciation apparaît dans l'opérande d'un opérateur "**par**" (M pour *multiple*)
- $E = \mathit{true}$ si et seulement si cette instanciation apparaît dans l'opérande gauche d'un opérateur "`>>`" (E pour *enable*)
- $D = \mathit{true}$ si et seulement si cette instanciation apparaît dans l'opérande gauche d'un opérateur "`[>`" (D pour *disable*)

Un programme LOTOS ne possède pas la propriété du contrôle statique si et seulement si il comporte n processus mutuellement récursifs et une instanciation interdite, c'est-à-dire s'il existe un entier n et $(n+1)$ sextuplets $(P_i, L_i, R_i, M_i, E_i, D_i)$, avec $i \in \{0, \ldots n\}$, tels que :

$$(\forall i \in \{0, \ldots n\}) \;\; (P_i, L_i, R_i, M_i, E_i, D_i) \in \mathcal{C}(P_{(i-1) \bmod (n+1)})$$

et :

$$(\exists i \in \{0, \ldots n\}) \;\; (L_i \vee R_i \vee M_i \vee E_i \vee D_i)$$

L'algorithme permettant de savoir si un programme LOTOS possède la propriété du contrôle statique fonctionne en deux phases successives.

La première phase détermine $\mathcal{C}(P)$, pour tout processus P. Ce calcul est fait par récurrence sur l'expression de comportement constituant le corps de P, notée $\mathit{behaviour}(P)$. Il est spécifié au moyen d'une grammaire attribuée utilisant six attributs hérités : un processus $\mathcal{P}$ et cinq booléens $\mathcal{L}$, $\mathcal{R}$, $\mathcal{M}$, $\mathcal{E}$ et $\mathcal{D}$.

Pour optimiser l'algorithme et réduire le nombre de sextuplets dans $\mathcal{C}(P)$ on utilise une opération d'*union compactée* entre ensembles de sextuplets, notée $\uplus$, définie de la façon suivante (l'opérande droit est toujours un singleton) :

$$\begin{array}{l}
\mathcal{S} \uplus \{(P, L, R, M, E, D)\} = \\
\quad \textbf{si } \exists (P, L', R', M', E', D') \in \mathcal{S} \textbf{ alors} \\
\quad\quad \mathcal{S} - \{(P, L', R', M', E', D')\} \cup \{(P, L \vee L', R \vee R', M \vee M', E \vee E', D \vee D')\} \\
\quad \textbf{sinon} \\
\quad\quad \mathcal{S} \cup \{(P, L, R, M, E, D)\}
\end{array}$$

Si $\widehat{G}$ désigne une liste de portes, on note *size* $(\widehat{G})$ le nombre de portes qui figurent dans $\widehat{G}$

Pour chaque processus P, l'évaluation des attributs s'effectue sur le comportement *behaviour*(P) en initialisant $\mathcal{C}(P)$ à $\emptyset$ et en donnant aux attributs $\mathcal{P}$, $\mathcal{L}$, $\mathcal{R}$, $\mathcal{M}$, $\mathcal{E}$ et $\mathcal{D}$ les valeurs respectives P, *false*, *false*, *false*, *false* et *false*.

$B \downarrow \mathcal{P} \downarrow \mathcal{L} \downarrow \mathcal{R} \downarrow \mathcal{M} \downarrow \mathcal{E} \downarrow \mathcal{D} \equiv$
stop
| **i** ; $B_0 \downarrow \mathcal{P} \downarrow \mathcal{L} \downarrow \mathcal{R} \downarrow \mathcal{M} \downarrow \mathcal{E} \downarrow \mathcal{D}$
| $G\ O_1, \ldots\ O_n$ [$[V]$] ; $B_0 \downarrow \mathcal{P} \downarrow \mathcal{L} \downarrow \mathcal{R} \downarrow \mathcal{M} \downarrow \mathcal{E} \downarrow \mathcal{D}$
| $B_1 \downarrow \mathcal{P} \downarrow \mathcal{L} \downarrow \mathcal{R} \downarrow \mathcal{M} \downarrow \mathcal{E} \downarrow \mathcal{D}$ [] $B_2 \downarrow \mathcal{P} \downarrow \mathcal{L} \downarrow \mathcal{R} \downarrow \mathcal{M} \downarrow \mathcal{E} \downarrow \mathcal{D}$
| **choice** $\widehat{G_0}$ **in** [$\widehat{G'_0}$], ... $\widehat{G_n}$ **in** [$\widehat{G'_n}$] [] $B_0 \downarrow \mathcal{P} \downarrow \mathcal{L} \downarrow \mathcal{R} \downarrow \mathcal{M} \downarrow \mathcal{E} \downarrow \mathcal{D}$
| $B_1 \downarrow \mathcal{P} \downarrow \mathit{true} \downarrow \mathcal{R} \downarrow \mathcal{M} \downarrow \mathcal{E} \downarrow \mathcal{D}$ *op* $B_2 \downarrow \mathcal{P} \downarrow \mathcal{L} \downarrow \mathit{true} \downarrow \mathcal{M} \downarrow \mathcal{E} \downarrow \mathcal{D}$
| **par** $\widehat{G_0}$ **in** [$\widehat{G'_0}$], ... $\widehat{G_n}$ **in** [$\widehat{G'_n}$] *op* $B_0 \downarrow \mathcal{P} \downarrow \mathcal{L} \downarrow \mathcal{R} \downarrow \mathcal{M}_o \downarrow \mathcal{E} \downarrow \mathcal{D}$
$\{\ \mathcal{M}_0 = \mathcal{M} \vee (\mathit{size}\ (G'_0) > 1) \vee \ldots\ (\mathit{size}\ (G'_n) > 1)$
| **hide** $G_0, \ldots\ G_n$ **in** $B_0 \downarrow \mathcal{P} \downarrow \mathcal{L} \downarrow \mathcal{R} \downarrow \mathcal{M} \downarrow \mathcal{E} \downarrow \mathcal{D}$
| [V_0] -> $B_0 \downarrow \mathcal{P} \downarrow \mathcal{L} \downarrow \mathcal{R} \downarrow \mathcal{M} \downarrow \mathcal{E} \downarrow \mathcal{D}$
| **let** $\widehat{X_0}$:S_0=V_0, ... $\widehat{X_n}$:S_n=V_n **in** $B_0 \downarrow \mathcal{P} \downarrow \mathcal{L} \downarrow \mathcal{R} \downarrow \mathcal{M} \downarrow \mathcal{E} \downarrow \mathcal{D}$
| **choice** $\widehat{X_0}$:S_0, ... $\widehat{X_n}$:S_n [] $B_0 \downarrow \mathcal{P} \downarrow \mathcal{L} \downarrow \mathcal{R} \downarrow \mathcal{M} \downarrow \mathcal{E} \downarrow \mathcal{D}$
| **exit** ($R_1, \ldots\ R_n$)
| $B_1 \downarrow \mathcal{P} \downarrow \mathcal{L} \downarrow \mathcal{R} \downarrow \mathcal{M} \downarrow \mathit{true} \downarrow \mathcal{D}$ >> **accept** $\widehat{X_1}$:S_1, ... $\widehat{X_n}$:S_n **in** $B_2 \downarrow \mathcal{P} \downarrow \mathcal{L} \downarrow \mathcal{R} \downarrow \mathcal{M} \downarrow \mathcal{E} \downarrow \mathcal{D}$
| $B_1 \downarrow \mathcal{P} \downarrow \mathcal{L} \downarrow \mathcal{R} \downarrow \mathcal{M} \downarrow \mathcal{E} \downarrow \mathit{true}$ [> $B_2 \downarrow \mathcal{P} \downarrow \mathcal{L} \downarrow \mathcal{R} \downarrow \mathcal{M} \downarrow \mathcal{E} \downarrow \mathcal{D}$
| P' [$G_1, \ldots\ G_n$] ($V_1, \ldots\ V_m$)
$\{\ \mathcal{C}(\mathcal{P}) := \mathcal{C}(\mathcal{P}) \uplus \{(P', \mathcal{L}, \mathcal{R}, \mathcal{M}, \mathcal{E}, \mathcal{D})\}$

La seconde phase opère sur le *graphe des appels*, qui est ainsi défini :

- à chaque processus P correspond un sommet, noté π_P
- il y a un arc de π_P vers $\pi_{P'}$ si et seulement si $\mathcal{C}(P)$ contient un sextuplet de la forme (P', L', R', M', E', D')
- la racine de ce graphe est π_λ

En appliquant l'algorithme de Tarjan [Tar72] [Xuo84, p. 139] on détermine les composantes fortement connexes de ce graphe. Le contrôle n'est pas statique s'il existe un processus P tel que $\mathcal{C}(P)$ contienne un sextuplet (P', L, R, M, E, D) tel que :

$$(\pi_P \textit{ et } \pi_{P'} \textit{ sont dans la même composante}) \wedge (L \vee R \vee M \vee E \vee D)$$

Dans le cas d'une récursion directe on a $P = P'$.

Remarque 3-2
Cette seconde phase permet aussi de détecter et de signaler à l'utilisateur de CÆSAR les processus qui ne sont jamais utilisés ; ce sont les processus P tels que π_P n'est pas accessible à partir de la racine du graphe des appels. ■

3.2 Restrictions sur les données

On suppose que tous les programmes LOTOS dont le contrôle est dynamique ont été préalablement rejetés. Mais ce n'est pas parce qu'un programme possède la propriété du contrôle statique qu'il est forcément régulier.

Exemple 3-9
Le comportement "P [G] (0)" dans lequel P est le processus défini comme suit :

$$\begin{array}{l}\textbf{process } P \texttt{ [}G\texttt{] (}N\texttt{:}NAT\texttt{) : } \textbf{noexit} \texttt{ :=} \\ \quad G \texttt{ !}N \texttt{ ; } P \texttt{ [}G\texttt{] (}N+1\texttt{)} \\ \textbf{endproc}\end{array}$$

n'est pas régulier. ■

Pour renforcer la propriété du contrôle statique, on recherche une contrainte, portant sur les sortes et les valeurs employées dans les programmes LOTOS. Etant donné la diversité des situations, la seule contrainte possible semble être l'interdiction des sortes dont le domaine est infini.

Tout programme LOTOS dont le contrôle est statique et dont les domaines des sortes sont finis est régulier. En effet le contrôle est régulier — d'après [Ail86] — et la partie données peut être modélisée par un nombre fini de variables d'état dont le domaine est fini.

Cette interdiction appelle toutefois plusieurs remarques :

- dans la théorie des types abstraits algébriques, l'analyse des opérations et des équations associées à cette sorte ne permet pas toujours de déterminer si le domaine d'une sorte est fini. Comment interdire ce qu'on ne sait pas caractériser ?
- cette interdiction semble trop contraignante. En effet on manipule souvent des sortes dont le domaine est infini sans utiliser toutes les valeurs de ce domaine. Par exemple l'offre "?X:S" ne nécessite pas l'énumération de tous les éléments de S lorsqu'elle est synchronisée avec une offre "!V" : on sait que $X = V$

Pour ces raisons et contrairement à ce qui a été fait pour le contrôle, il semble donc impossible d'imposer des restrictions sur l'usage des valeurs et des sortes. Leur emploi doit être fait sous la responsabilité de l'utilisateur. Pratiquement on n'a pas de règle absolue, tout au plus une liste non exhaustive de recommandations destinées à l'utilisateur :

- ne pas écrire "**choice** X:S" lorsque le domaine de la sorte S est infini
- ne pas écrire "?X:S" lorsqu'il n'y a pas synchronisation avec "!V" et que le domaine de la sorte S est infini
- ne pas écrire "**exit** (**any** S)" lorsqu'il n'y a pas synchronisation avec "**exit** (V)" et que le domaine de la sorte S est infini
- ne pas écrire de récursion telle que chaque instanciation récursive engendre une nouvelle valeur. Toutefois cette contrainte doit être nuancée, car CÆSAR est capable de traiter correctement certaines situations.

  Exemple 3-10
  La récursion illimitée :

  $$\begin{array}{l}\textbf{process } P \texttt{ (}X\texttt{:}NAT\texttt{) : } \textbf{noexit} \texttt{ :=} \\ \quad P \texttt{ (}X+1\texttt{)} \\ \textbf{endproc}\end{array}$$

  est éliminée[15] et le corps du processus P réduit à "**stop**". ■

[15] pendant la phase d'optimisation

Il n'est pas possible de vérifier automatiquement si ces recommandations sont appliquées. Leur non-respect par l'utilisateur peut provoquer l'abandon de la traduction, au cours de la phase de simulation, par un débordement du nombre d'états (*state explosion*) qui excède la capacité mémoire de CÆSAR.

Hormis l'interdiction du contrôle statique, on ne peut pas caractériser exactement l'ensemble des programmes LOTOS traités par CÆSAR : en effet, les interdictions qu'on devrait formuler ne portent pas directement sur la syntaxe du programme LOTOS source, mais sur des critères sémantiques complexes ; la corrélation entre l'acceptation — ou le refus — d'un programme par CÆSAR et son expression en LOTOS n'est nullement évidente à établir.

En pratique, cette impossibilité de déterminer *à l'avance* les programmes acceptés ne pose guère de problèmes aux utilisateurs, qui ont recours au schéma *tentative-erreur-correction*. Les difficultés découlent davantage des performances du système (notamment le nombre d'états des automates engendrés) que des limitations théoriques sur le pouvoir d'expression du sous-ensemble de LOTOS accepté.

3.3 Présentation de SUBLOTOS

SUBLOTOS est un langage intermédiaire entre LOTOS et le modèle réseau. De fait SUBLOTOS ressemble assez à LOTOS mais s'en distingue sur plusieurs points :

- SUBLOTOS a une puissance d'expression moindre que celle de LOTOS. En effet tous les programmes SUBLOTOS doivent vérifier la propriété du contrôle statique. En revanche tous les programmes LOTOS qui possèdent cette propriété peuvent être traduits en SUBLOTOS, même s'ils définissent des comportements non réguliers

- certains opérateurs de comportement et certaines constructions LOTOS n'apparaissent pas en SUBLOTOS. Ils sont remplacés par des formes sémantiquement équivalentes, quoique plus primitives

- toutes les occurrences de la porte de terminaison "δ" apparaissent explicitement en SUBLOTOS ; cette porte est traitée comme une porte ordinaire

- la sémantique de SUBLOTOS ne possède pas la notion de renommage de portes, car toutes les portes qui figurent dans les programmes SUBLOTOS sont constantes. En particulier, pour toute instanciation de processus SUBLOTOS, les portes paramètres effectifs sont égales aux portes paramètres formels. Pour réaliser cette élimination la phase d'expansion doit développer (de manière finie) la récursion et créer de nouveaux processus

- tous les processus SUBLOTOS sont récursifs. Tous les processus LOTOS non récursifs ont été éliminés et chaque intanciation de ces processus a été développée (i.e. remplacée par le corps du processus correspondant)

- LOTOS est un langage fonctionnel : chaque variable LOTOS est introduite par une déclaration d'identité qui lui associe une valeur constante. En revanche les variables peuvent être créées et détruites dynamiquement par une gestion d'environnements. Au contraire SUBLOTOS n'est pas un langage fonctionnel : l'expansion construit un ensemble fini de variables d'état correspondant aux variables LOTOS qui figurent dans les programmes LOTOS. Ces variables, dont la valeur initiale est indéfinie, peuvent être affectées plusieurs fois. En outre leur existence est globale

- aucune variable SUBLOTOS n'est partagée entre plusieurs processus concurrents : deux processus exécutés en parallèle modifient des variables distinctes

- la sémantique dynamique de LOTOS est entièrement basée sur une réécriture d'expressions de comportement : par exemple l'évaluation d'une variable s'exprime par le remplacement textuel de cette variable par sa valeur. En revanche la sémantique dynamique de SUBLOTOS comporte une séparation claire entre contrôle et données : comme en LOTOS la partie contrôle est définie par des règles de dérivation ; en revanche la partie données s'appuie sur la notion de contexte, qui mémorise l'ensemble des valeurs des variables d'état à un instant donné

3.4 Syntaxe abstraite de SUBLOTOS

3.4.1 Symboles non-terminaux

Le tableau suivant présente tous les symboles non-terminaux de la syntaxe abstraite :

non-terminal	signification
B	expression de comportement
F	identificateur d'opération
G	identificateur de porte
Λ	identificateur de la spécification
O	offre de synchronisation
op	opérateur de composition parallèle
P	identificateur de processus
process	définition de processus
program	axiome de la grammaire syntaxique
S	identificateur de sorte
T	identificateur de type
V	expression de valeur
X	identificateur de variable

L'axiome de la grammaire syntaxique est le non-terminal *program*.

3.4.2 Types, sortes et opérations

Les types, sortes et opérations apparaissant dans un programme SUBLOTOS sont exactement les mêmes que ceux qui figurent dans le programme LOTOS correspondant. Pour cette raison les définitions de types, sortes et opérations LOTOS ne seront pas reprises en SUBLOTOS :

- l'ensemble des types d'une spécification SUBLOTOS, dénoté par le non-terminal T, est identique à l'ensemble des types de la spécification LOTOS correspondante
- l'ensemble des sortes d'une spécification SUBLOTOS, dénoté par le non-terminal S, est identique à l'ensemble des sortes de la spécification LOTOS correspondante
- l'ensemble des opérations d'une spécification SUBLOTOS, dénoté par le non-terminal F, est identique à l'ensemble des opérations de la spécification LOTOS correspondante

3.4.3 Objets et duplications

On appelle *objets* LOTOS l'ensemble des portes, des variables et des processus des programmes LOTOS. De même on appelle *objets* SUBLOTOS l'ensemble des portes, des variables et des processus

SubLotos.

Au cours de la traduction d'un programme Lotos en SubLotos, on est amené à reproduire plusieurs fois certains fragments du programme Lotos. C'est ainsi que les occurrences de définition de certaines portes, variables et processus Lotos peuvent être dupliquées en plusieurs exemplaires. Il faut faire en sorte que, malgré ces transformations, les objets du programme SubLotos obtenu conservent des noms uniques.

C'est pourquoi on appelle *duplication* d'un objet Lotos noté Y, un objet SubLotos obtenu en concaténant à Y un suffixe numérique noté "$\bullet i$" : par convention les objets SubLotos créés lorsque l'on duplique un objet Lotos Y ont pour noms $Y\bullet 1$, ... $Y\bullet n$. Cette notation permet de savoir que l'objet SubLotos $Y\bullet i$ est la $i^{\text{ème}}$ duplication de l'objet Lotos Y.

Réciproquement on définit aussi une notation qui, à tout objet SubLotos permet d'associer son *origine*, c'est-à-dire l'objet Lotos dont il provient :

$$origin(Y\bullet i) = Y$$

Remarque 3-3
En fait, les noms de portes, de variables et de processus qui sont communiqués à l'utilisateur de Cæsar ont la forme "$Y\bullet m\bullet n$" où :

- Y est la chaîne de caractère qui dénote cet objet dans le programme Lotos
- m est un suffixe numérique attribué à Y au cours de l'analyse sémantique statique afin de distinguer les différents objets qui portent le même nom lexical Y. Ainsi $Y\bullet m$ peut être considéré comme un nom unique dans la syntaxe abstraite Lotos
- n est un suffixe numérique attribué à $Y\bullet m$ au cours de l'expansion afin de distinguer les différentes duplications de $Y\bullet m$. Ainsi $Y\bullet m\bullet n$ peut être considéré comme un nom unique pour la syntaxe abstraite SubLotos

■

3.4.4 Portes

On ajoute à l'ensemble des portes qui figurent dans le programme Lotos une porte spéciale, notée "δ", qui correspond à la porte de terminaison (les duplications de la porte "δ" sont introduites par l'expansion des opérateurs "**exit**" et ">>").

L'ensemble des portes d'une spécification SubLotos, dénoté par le non-terminal G, est égal à l'union de deux ensembles :

- l'ensemble des duplications des portes (y compris "δ" et "**i**") de la spécification Lotos correspondante, sauf celles qui sont paramètres formels de la spécification. Il peut exister plusieurs duplications de la porte "δ". Il n'existe qu'une seule duplication de la porte "**i**", notée "$\mathbf{i}\bullet 1$"
- l'ensemble des portes Lotos qui sont paramètres formels de la spécification Lotos, auxquelles on ajoute "δ". Pour assurer l'homogénéité des notations avec les autres portes SubLotos, ces portes G sont représentées comme des duplications ayant la forme $G\bullet 0$. On généralise la notation "*origin*" à ces duplications en posant :

$$origin(G\bullet 0) = G$$

3.4.5 Variables

On ajoute à l'ensemble des variables qui figurent dans le programme LOTOS une variable spéciale notée "ξ" (les duplications de la variable "ξ" sont introduites par l'expansion des clauses "**any**" de "**exit**").

L'ensemble des variables d'une spécification SUBLOTOS, dénoté par le non-terminal X, est identique à l'ensemble des duplications des variables (y compris "ξ") de la spécification LOTOS correspondante.

3.4.6 Processus

L'ensemble des processus d'une spécification SUBLOTOS, dénoté par le non-terminal P, est égal à l'ensemble des duplications des processus de la spécification LOTOS correspondante.

3.4.7 Expressions de valeur

$$
\begin{array}{rcl}
V & \equiv & X \\
 & | & F\ \texttt{(}V_1, \ldots\ V_n\texttt{)} \\
 & | & V_1\ \texttt{=}\ V_2
\end{array}
$$

3.4.8 Offres

$$
\begin{array}{rcl}
O & \equiv & \texttt{!}V \\
 & | & \texttt{?}X\texttt{:}S
\end{array}
$$

Remarque 3-4
L'expansion linéarise les offres multiples "$\texttt{?}X_0, \ldots\ X_n\texttt{:}S$" qui sont transformées en "$\texttt{?}X_0\texttt{:}S, \ldots\ \texttt{?}X_n\texttt{:}S$". ■

3.4.9 Opérateurs parallèles

$$
\begin{array}{rcl}
op & \equiv & \texttt{||} \\
 & | & \texttt{|[}G_0, \ldots\ G_n\texttt{]|}
\end{array}
$$

Remarque 3-5
L'expansion élimine l'opérateur "|||", qui n'existe plus en SUBLOTOS. ■

3.4.10 Expressions de comportement

Les *expressions de comportement* sont dénotées par le non-terminal B. Les non-terminaux *from_exit* et *from_enable* qui apparaissent dans la définition syntaxique de B dénotent des valeurs booléennes (*true* ou *false*) qui sont des indications transmises par la phase d'expansion à la phase de génération afin d'optimiser la construction du réseau.

$$
\begin{array}{rcl}
B & \equiv & \textbf{stop} \\
& | & G\ O_1, \ldots\ O_n\ \texttt{[}[V_0]\texttt{]}\ \texttt{;}\ B_0\ \textit{(from\_exit, from\_enable)} \\
& | & B_1\ \texttt{[]}\ B_2 \\
& | & B_1\ \textit{op}\ B_2\ \textit{(from\_enable)} \\
& | & \textbf{hide}\ G_0, \ldots\ G_n\ \textbf{in}\ B_0 \\
& | & \texttt{[}V_0\texttt{]}\ \texttt{->}\ B_0 \\
& | & \textbf{let}\ \widehat{X_0}:S_0\texttt{=}V_0, \ldots\ \widehat{X_n}:S_n\texttt{=}V_n\ \textbf{in}\ B_0 \\
& | & \textbf{choice}\ \widehat{X_0}:S_0, \ldots\ \widehat{X_n}:S_n\ \texttt{[]}\ B_0 \\
& | & B_1\ \texttt{[}G\texttt{>}\ B_2 \\
& | & P\ \texttt{[}G_0, \ldots\ G_m\texttt{]}\ \texttt{(}V_1, \ldots\ V_n\texttt{)}
\end{array}
$$

Remarque 3-6

- pour l'opérateur de préfixage "**;**", la porte G peut être égale à "**i**"
- les opérateurs LOTOS d'itération sur les portes ("**par**" et "**choice**") ne figurent plus en SUBLOTOS
- par suite de l'expansion de la porte "δ" les opérateurs LOTOS "**exit**" et "**>>**" ont été éliminés
- pour la même raison, on ajoute une porte G à l'opérateur "`[>`" de SUBLOTOS, qui est la porte de terminaison après laquelle l'interruption n'est plus possible
- pour la même raison, on ajoute un paramètre porte supplémentaire à tous les processus SUBLOTOS, qui est la porte sur laquelle ils se terminent. Un processus SUBLOTOS a donc toujours au moins un paramètre porte

■

3.4.11 Processus

$$
\begin{array}{rcl}
\textit{process} & \equiv & \textbf{process}\ P\ \texttt{[}G_0, \ldots\ G_m\texttt{]}\ \texttt{(}X_1:S_1, \ldots\ X_n:S_n\texttt{)}\ \texttt{:=} \\
& & \quad B \\
& & \textbf{endproc}
\end{array}
$$

Remarque 3-7

- l'expansion linéarise les listes de variables paramètres formels : "$X_1, \ldots\ X_p:S$" est remplacé par "$X_1:S, \ldots\ X_p:S$"
- contrairement aux processus LOTOS, il est inutile de préciser la fonctionnalité[16] des processus SUBLOTOS ; en effet cette information n'est utile que pour la sémantique statique des programmes LOTOS

[16] *cf.* non-terminal *func* (§ 2.1.12, p. 29)

■

3.4.12 Spécification SUBLOTOS

$$
\begin{array}{lll}
\textit{program} & \equiv & \textbf{specification}\ \Lambda\ [G_0, \ldots\ G_m]\ \textbf{behaviour} \\
& & \quad B \\
& & \textbf{where}\ \textit{process}_1, \ldots\ \textit{process}_n \\
& & \textbf{endspec}
\end{array}
$$

Remarque 3-8

- la spécification SubLotos ne comporte plus de variables paramètres formels ; après l'expansion celles-ci sont déclarées avec l'opérateur "**choice**". On dit alors que la spécification SubLotos est *fermée*
- comme pour les processus il est inutile d'indiquer la fonctionnalité de la spécification
- par commodité on omet de faire figurer les déclarations de types

■

3.4.13 Attributs locaux

Grâce aux duplications, la propriété du *nom unique* est vérifiée ; tous les identificateurs utilisés dans un programme SubLotos sont deux à deux distincts.

Comme en Lotos, on suppose qu'on dispose d'un mécanisme d'*évaluation* des valeurs, c'est-à-dire que toute expression de valeur SubLotos ne comportant aucune variable peut être réécrite sous une forme normale unique. Si V_1 et V_2 sont deux valeurs de même sorte, le prédicat "$V_1 = V_2$" signifie que V_1 et V_2 ont la même forme normale.

Aux éléments de la syntaxe abstraite sont attachées diverses informations sémantiques[17] analogues à celles définies pour Lotos (§ 2.1.16, p. 31) :

- comme Lotos, SubLotos est un langage fortement typé dans lequel toute expression de valeur possède une sorte et une seule. Si V est une expression de valeur, on note "$sort(V)$" la sorte de V
- en particulier si X est une variable, on note "$sort(X)$" la sorte de X. Chaque variable SubLotos a la même sorte que la variable Lotos qu'elle duplique. Toutefois cette règle ne s'applique pas à la pseudo-variable "ξ" dont la sorte est indéfinie et dont les diverses duplications peuvent avoir des sortes différentes
- si P est, soit un processus, soit l'identificateur Λ de la spécification, P est complètement caractérisé par la donnée des attributs suivants :
  - on appelle *portes formelles* de P, notées "$gate_0(P), \ldots\ gate_m(P)$" les paramètres portes $G_0, \ldots\ G_m$ associés à P par sa définition. La première de ces portes, $gate_0(P)$, dénote toujours la porte de terminaison de P ; c'est une duplication de la porte "δ".

[17] attributs locaux

Dans toute instanciation de P (si $P \neq \Lambda$) les portes fournies comme paramètres effectifs de l'instanciation sont deux à deux identiques aux portes formelles de P.

En outre les portes formelles de Λ sont des duplications des portes formelles de λ :

$$(gate_0(\Lambda) = \delta{\bullet}0) \;\wedge\; (\forall i \in \{1, \dots\; m\}) \;\; (gate_i(\Lambda) = gate_i(\lambda){\bullet}0)$$

– on appelle *variables formelles* de P, notées "$var_1(P), \dots\; var_n(P)$" les paramètres variables $X_1, \dots\; X_n$ (après linéarisation des listes de variables) associés à P par sa définition
– on appelle *corps* de P, noté "$behaviour(P)$", le comportement B associé à P par sa définition. Le corps de P est paramétré par les portes et les variables formelles de P

3.5 Sémantique dynamique de SUBLOTOS

Après avoir donné la syntaxe abstraite du langage SubLotos il convient d'en définir la sémantique. Celle-ci est proche de celle de Lotos ; toutefois le problème de leur compatibilité ne se pose pas ici, mais au niveau de l'algorithme d'expansion qui traduit Lotos en SubLotos.

3.5.1 Contextes

On appelle *contexte* une application partielle de l'ensemble des variables SubLotos de la spécification vers l'ensemble des valeurs SubLotos qui, à chaque variable, fait correspondre sa valeur[18]. Lorsqu'une variable n'a pas été initialisée, la valeur du contexte pour cette variable est indéfinie. En particulier l'application nulle part définie "$\perp$" dénote le *contexte vide* dans lequel aucune variable n'a reçu de valeur.

Pour manipuler les contextes on utilise les notations relatives aux applications partielles que l'on complète par les définitions suivantes :

- on note "$eval(V\;,\;C)$", où V est une valeur et C un contexte, la valeur résultant de l'évaluation de V dans le contexte C. Cette fonction est définie par récurrence sur la complexité de V :

$$eval(V\;,\;C) = \begin{cases} \textbf{si } (V \equiv X) \textbf{ alors } C(X) \\ \textbf{sinon si } (V \equiv F\;(V_1, \dots\; V_n)) \textbf{ alors } F\;(eval(V_1\;,\;C), \dots\; eval(V_n\;,\;C)) \\ \textbf{sinon si } (V \equiv V_1 = V_2) \textbf{ alors } eval(V_1\;,\;C) = eval(V_2\;,\;C) \end{cases}$$

- on note "$merge\;(C_0, C_1, C_2)$", où C_0, C_1 et C_2 sont trois contextes qui vérifient l'invariant :

$$(C_0(x) = C_1(x)) \;\vee\; (C_0(x) = C_2(x))$$

le contexte défini par :

$$merge\;(C_0, C_1, C_2)(x) = \begin{cases} \textbf{si } C_1(x) = C_2(x) \textbf{ alors } C_1(x) \\ \textbf{si } (C_1(x) \neq C_2(x)) \;\wedge\; (C_1(x) \neq C_0(x)) \textbf{ alors } C_1(x) \\ \textbf{si } (C_1(x) \neq C_2(x)) \;\wedge\; (C_2(x) \neq C_0(x)) \textbf{ alors } C_2(x) \end{cases}$$

[18] ce qui correspond à la notion classique d'*environnement* [Ten76]

3.5.2 Relation de transition

La sémantique dynamique de SubLotos est une sémantique de type opérationnel qui spécifie les règles d'évolution des programmes. La sémantique dynamique de SubLotos est complètement définie par la donnée d'une *relation de transition*, notée "$\langle B \ , \ C \rangle \ \xrightarrow{L} \ \langle B' \ , \ C' \rangle$" où :

- $\langle B \ , \ C \rangle$ et $\langle B' \ , \ C' \rangle$ sont des couples dont le premier élément (qui décrit l'état du contrôle) est une expression de comportement SubLotos et dont le second élément (qui représente les variables d'état du système) est un contexte
- L est un label (§ 1.2.8, p. 22) construit sur l'ensemble des portes SubLotos, l'ensemble des sortes et l'ensemble des opérations définies dans la spécification Lotos. Ce label a donc la forme "$G \ V_1, \ldots \ V_n$", G désignant une porte et $V_1, \ldots \ V_n$ des expressions de valeur ne contenant aucune variable

La signification intuitive de cette relation est "on peut passer de $\langle B \ , \ C \rangle$ à $\langle B' \ , \ C' \rangle$ en effectuant l'action définie par le label L".

3.5.3 Règles de dérivation et de substitution

Cette relation de transition est définie par récurrence sur la structure syntaxique des comportements SubLotos. Pour cela on utilise un système de dérivation dont les règles ont la forme usuelle :

$$\frac{P}{Q}$$

qui signifie que le prédicat P implique le prédicat Q.

Afin d'alléger les notations, on emploie aussi des règles de substitution, qui s'écrivent :

$$\underbrace{\langle B_1 \ , \ C_1 \rangle}_{\langle B_2 \ , \ C_2 \rangle}$$

et qui signifient que $\langle B_1 \ , \ C_1 \rangle$ doit être remplacé par $\langle B_2 \ , \ C_2 \rangle$. Cette abréviation est équivalente à la règle :

$$\frac{\langle B_2 \ , \ C_2 \rangle \ \xrightarrow{L} \ \langle B \ , \ C \rangle}{\langle B_1 \ , \ C_1 \rangle \ \xrightarrow{L} \ \langle B \ , \ C \rangle}$$

Les règles de substitution permettent d'exprimer des changements d'état qui ne donnent pas lieu à une action observable (ni même à une action étiquetée "**i**") : il s'agit essentiellement de modifications de la valeur des variables du contexte.

L'axiome du système de dérivation est $\langle \mathit{behaviour}(\Lambda) \ , \ \perp \rangle$.

3.5.4 Opérateur "stop"

Aucune règle d'inférence n'est associée à "**stop**".

3.5.5 Opérateur ";"

$$\frac{\mathit{true}}{\langle G \ O_1, \ldots \ O_n \ \ ; \ B_0 \ , \ C \rangle \ \xrightarrow{G \ V'_1, \ldots \ V'_n} \ \langle B_0 \ , \ C' \rangle}$$

où :

$$(\forall i \in \{1, \dots\ n\})\ \ V'_i = \begin{cases} \textbf{si } O_i = \texttt{!}V_i \textbf{ alors } \mathit{eval}(V_i\ ,\ C) \\ \textbf{si } O_i = \texttt{?}X_i\texttt{:}S_i \textbf{ alors } \mathit{oneof}(\mathit{domain}(S_i)) \end{cases}$$

$$C' = C \oslash \bigoplus_{i \in \{1, \dots\ n\}} C'_i$$

$$(\forall i \in \{1, \dots\ n\})\ \ C'_i = \begin{cases} \textbf{si } O_i = \texttt{!}V_i \textbf{ alors } \perp \\ \textbf{si } O_i = \texttt{?}X_i\texttt{:}S_i \textbf{ alors } X_i \rightsquigarrow V'_i \end{cases}$$

$$\frac{\mathit{eval}(V_0\ ,\ C') = \mathit{true}}{\langle G\ O_1, \dots\ O_n\,\texttt{[}V_0\texttt{]}\ \ \texttt{;}\ B_0\ ,\ C\rangle \ \xrightarrow{G\ V'_1, \dots\ V'_n}\ \langle B_0\ ,\ C'\rangle}$$

où $V'_1, \dots\ V'_n$, $C'_1, \dots\ C'_n$ et C' sont définis comme précédemment.

Remarque 3-9
Les booléens *from_exit* et *from_enable*, quelle que soit leur valeur, ne modifient pas la sémantique de l'opérateur "`;`". ■

3.5.6 Opérateur "`[]`"

$$\frac{\langle B_1\ ,\ C\rangle \ \xrightarrow{L}\ \langle B'_1\ ,\ C'\rangle}{\langle B_1\ \texttt{[]}\ B_2\ ,\ C\rangle \ \xrightarrow{L}\ \langle B'_1\ ,\ C'\rangle}$$

$$\frac{\langle B_2\ ,\ C\rangle \ \xrightarrow{L}\ \langle B'_2\ ,\ C'\rangle}{\langle B_1\ \texttt{[]}\ B_2\ ,\ C\rangle \ \xrightarrow{L}\ \langle B'_2\ ,\ C'\rangle}$$

3.5.7 Opérateur "`||`"

$$\frac{\langle B_1\ ,\ C\rangle \ \xrightarrow{\mathbf{i}_{\bullet 1}}\ \langle B'_1\ ,\ C'\rangle}{\langle B_1\ \texttt{||}\ B_2\ ,\ C\rangle \ \xrightarrow{\mathbf{i}_{\bullet 1}}\ \langle B'_1\ \texttt{||}\ B_2\ ,\ C'\rangle}$$

$$\frac{\langle B_2\ ,\ C\rangle \ \xrightarrow{\mathbf{i}_{\bullet 1}}\ \langle B'_2\ ,\ C'\rangle}{\langle B_1\ \texttt{||}\ B_2\ ,\ C\rangle \ \xrightarrow{\mathbf{i}_{\bullet 1}}\ \langle B_1\ \texttt{||}\ B'_2\ ,\ C'\rangle}$$

$$\frac{(\langle B_1\ ,\ C\rangle \ \xrightarrow{L}\ \langle B'_1\ ,\ C'_1\rangle)\ \wedge\ (\langle B_2\ ,\ C\rangle \ \xrightarrow{L}\ \langle B'_2\ ,\ C'_2\rangle)\ \wedge\ (\mathit{gate}(L) \neq \mathbf{i}_{\bullet 1})}{\langle B_1\ \texttt{||}\ B_2\ ,\ C\rangle \ \xrightarrow{L}\ \langle B'_1\ \texttt{||}\ B'_2\ ,\ \mathit{merge}\ (C, C'_1, C'_2)\rangle}$$

Remarque 3-10
Le booléen *from_enable*, quelle que soit sa valeur, ne modifie pas la sémantique de l'opérateur "`||`".
■

3.5.8 Opérateur "|[...]|"

$$\frac{(\langle B_1 \;,\; C\rangle \xrightarrow{L} \langle B_1' \;,\; C'\rangle) \;\wedge\; (gate(L) \notin \{G_0,\ldots\; G_n\})}{\langle B_1 \;\texttt{|[}G_0,\ldots\; G_n\texttt{]|}\; B_2 \;,\; C\rangle \xrightarrow{L} \langle B_1' \;\texttt{|[}G_0,\ldots\; G_n\texttt{]|}\; B_2 \;,\; C'\rangle}$$

$$\frac{(\langle B_2 \;,\; C\rangle \xrightarrow{L} \langle B_2' \;,\; C'\rangle) \;\wedge\; (gate(L) \notin \{G_0,\ldots\; G_n\})}{\langle B_1 \;\texttt{|[}G_0,\ldots\; G_n\texttt{]|}\; B_2 \;,\; C\rangle \xrightarrow{L} \langle B_1 \;\texttt{|[}G_0,\ldots\; G_n\texttt{]|}\; B_2' \;,\; C'\rangle}$$

$$\frac{(\langle B_1 \;,\; C\rangle \xrightarrow{L} \langle B_1' \;,\; C_1'\rangle) \;\wedge\; (\langle B_2 \;,\; C\rangle \xrightarrow{L} \langle B_2' \;,\; C_2'\rangle) \;\wedge\; (gate(L) \in \{G_0,\ldots\; G_n\})}{\langle B_1 \;\texttt{|[}G_0,\ldots\; G_n\texttt{]|}\; B_2 \;,\; C\rangle \xrightarrow{L} \langle B_1' \;\texttt{|[}G_0,\ldots\; G_n\texttt{]|}\; B_2' \;,\; merge\;(C, C_1', C_2')\rangle}$$

Remarque 3-11
Le booléen *from_enable*, quelle que soit sa valeur, ne modifie pas la sémantique de l'opérateur "|[...]|". ■

3.5.9 Opérateur "hide"

$$\frac{(\langle B_0 \;,\; C\rangle \xrightarrow{L} \langle B_0' \;,\; C'\rangle) \;\wedge\; (gate(L) \in \{G_0,\ldots\; G_n\})}{\langle \textbf{hide}\; G_0,\ldots\; G_n \;\textbf{in}\; B_0 \;,\; C\rangle \xrightarrow{\mathbf{i}\bullet 1} \langle \textbf{hide}\; G_0,\ldots\; G_n \;\textbf{in}\; B_0' \;,\; C'\rangle}$$

$$\frac{(\langle B_0 \;,\; C\rangle \xrightarrow{L} \langle B_0' \;,\; C'\rangle) \;\wedge\; (gate(L) \notin \{G_0,\ldots\; G_n\})}{\langle \textbf{hide}\; G_0,\ldots\; G_n \;\textbf{in}\; B_0 \;,\; C\rangle \xrightarrow{L} \langle \textbf{hide}\; G_0,\ldots\; G_n \;\textbf{in}\; B_0' \;,\; C'\rangle}$$

3.5.10 Opérateur "->"

$$\frac{(\langle B_0 \;,\; C\rangle \xrightarrow{L} \langle B_0' \;,\; C'\rangle) \;\wedge\; (eval(V_0 \;,\; C) = true)}{(\langle \texttt{[}V_0\texttt{]} \;\texttt{->}\; B_0 \;,\; C\rangle) \xrightarrow{L} \langle B_0' \;,\; C'\rangle}$$

3.5.11 Opérateur "let"

$$\underbrace{\langle \textbf{let}\; \widehat{X_0} : S_0 \texttt{=} V_0, \ldots\; \widehat{X_n} : S_n \texttt{=} V_n \;\textbf{in}\; B_0 \;,\; C\rangle}_{\langle B_0 \;,\; C \oslash \bigoplus_{i \in \{0,\ldots\; n\}} (\widehat{X_i} \rightsquigarrow eval(V_i \;,\; C))\rangle}$$

3.5.12 Opérateur “choice”

$$\underbrace{\langle \textbf{choice}\ \widehat{X_0}\,{:}\,S_0, \ldots\ \widehat{X_n}\,{:}\,S_n\ \texttt{[]}\ B_0\ ,\ C\rangle}_{\langle B_0\ ,\ C \oslash \bigoplus_{i\in\{0,\ldots\ n\}} C_i\rangle}$$

où :

$$\textbf{si}\ \widehat{X_i} \equiv X_0, \ldots\ X_p\ \textbf{alors}\ C_i = \bigoplus_{j\in\{0,\ldots\ p\}} (X_j \rightsquigarrow \mathit{oneof}(\mathit{domain}(S_i)))$$

3.5.13 Opérateur “[...>”

$$\frac{\langle B_1\ ,\ C\rangle \xrightarrow{L} \langle B_1'\ ,\ C'\rangle \wedge (\mathit{gate}(L) \neq G_0)}{\langle B_1\ \texttt{[}G_0\texttt{>}\ B_2\ ,\ C\rangle \xrightarrow{L} \langle B_1'\ \texttt{[}G_0\texttt{>}\ B_2\ ,\ C'\rangle}$$

$$\frac{\langle B_1\ ,\ C\rangle \xrightarrow{L} \langle B_1'\ ,\ C'\rangle \wedge (\mathit{gate}(L) = G_0)}{\langle B_1\ \texttt{[}G_0\texttt{>}\ B_2\ ,\ C\rangle \xrightarrow{L} \langle B_1'\ ,\ C'\rangle}$$

$$\frac{\langle B_2\ ,\ C\rangle \xrightarrow{L} \langle B_2'\ ,\ C'\rangle}{\langle B_1\ \texttt{[}G_0\texttt{>}\ B_2\ ,\ C\rangle \xrightarrow{L} \langle B_2'\ ,\ C'\rangle}$$

3.5.14 Processus et instanciation

$$\underbrace{\langle P\ \texttt{[}G_0, \ldots\ G_m\texttt{]}\ \texttt{(}V_1, \ldots\ V_n\texttt{)}\ ,\ C\rangle}_{\langle \mathit{behaviour}(P)\ ,\ C \oslash \bigoplus_{i\in\{1,\ldots\ n\}} (\mathit{var}_i(P) \rightsquigarrow \mathit{eval}(V_i\ ,\ C))\rangle}$$

3.5.15 Construction du graphe

A partir de la spécification SUBLOTOS Λ, correspondant à une spécification LOTOS λ, on construit un automate $(\Sigma, \sigma_0, \mathcal{E}, \mathcal{G}, \mathcal{S}, \mathcal{F})$ où :

- les états de Σ sont des couples $\langle B\ ,\ C\rangle$. Plus précisément Σ est défini par l'équation suivante :

  $$\Sigma = \{\langle \mathit{behaviour}(\Lambda)\ ,\ \bot\rangle\} \cup \{\langle B_2\ ,\ C_2\rangle \mid (\exists \langle B_1\ ,\ C_1\rangle \in \Sigma)\ (\exists L)\ (\langle B_1\ ,\ C_1\rangle \xrightarrow{L} \langle B_2\ ,\ C_2\rangle)\}$$

- l'état initial σ_0 est formé par le couple $\langle \mathit{behaviour}(\Lambda)\ ,\ \bot\rangle$
- les arcs de $\mathcal{E}$ sont déterminés par la relation de transition :

  $$\mathcal{E} = \left\{ (\langle B_1\ ,\ C_1\rangle, \mathit{origin}(L), \langle B_2\ ,\ C_2\rangle) \mid \begin{array}{l} (\langle B_1\ ,\ C_1\rangle \in \Sigma) \wedge (\langle B_2\ ,\ C_2\rangle \in \Sigma) \wedge \\ (\langle B_1\ ,\ C_1\rangle \xrightarrow{L} \langle B_2\ ,\ C_2\rangle) \end{array} \right\}$$

  où “$\mathit{origin}(L)$” dénote le label obtenu à partir de L en remplaçant la porte SUBLOTOS $\mathit{gate}(L)$ par la porte LOTOS qui constitue son origine :

  $$(L = G\ V_1, \ldots\ V_n) \Longrightarrow (\mathit{origin}(L) = \mathit{origin}(G)\ V_1, \ldots\ V_n)$$

- $\mathcal{G}$ est l'ensemble des portes LOTOS visibles de λ, c'est-à-dire $gate_1(\lambda), \dots\ gate_n(\lambda)$ auxquelles on ajoute la porte "**i**" et la porte "δ". En effet les règles de sémantique statique et dynamique font que, dans chaque label L, toutes les portes SUBLOTOS autres que $gate_0(\Lambda), \dots\ gate_n(\Lambda)$ ont été renommées en "$\mathbf{i}_{\bullet 1}$"
- $\mathcal{S}$ est l'ensemble des sortes LOTOS présentes dans λ
- $\mathcal{F}$ est l'ensemble des opérations LOTOS présentes dans λ

Chapitre 4

Réseau

La traduction des spécifications SubLotos en automates se fait en deux étapes, appelées *génération* et *simulation*, entre lesquelles une forme intermédiaire, appelée *réseau*, joue le rôle d'interface.

On justifie tout d'abord l'existence de cette forme intermédiaire, alors qu'une traduction directe du langage SubLotos vers le modèle graphe aurait été envisageable ; on indique également les critères et les choix qui ont présidé à sa conception.

Le modèle réseau est présenté, d'abord de manière intuitive, en mettant l'accent sur ses caractéristiques originales, puis formellement en donnant sa syntaxe et sa sémantique opérationnelle.

On énonce ensuite un certain nombre de propriétés invariantes qui caractérisent l'évolution du réseau ; elles sont d'une extrême importance à cause de l'utilisation constante qui en est faite pendant la phase de simulation.

4.1 Critères de conception du réseau

Les logiciels de la famille CESAR ont en commun le fait de produire une forme intermédiaire :

- pour Quasar [Que82] [Sch83] il s'agit de *réseaux de Petri* [Bra83]
- pour Xesar [RRSV87b] [Rod88] il s'agit d'*automates communicants*

Ces deux modèles seront comparés plus loin, afin d'évaluer leur application à Lotos, mais sans préjuger du modèle retenu pour Cæsar, on peut affirmer que le réseau doit posséder les propriétés suivantes :

- il doit avoir une puissance d'expression égale à celle de SubLotos : tout programme SubLotos peut être traduit en un réseau équivalent
- il doit permettre une description complètement statique de la structure de contrôle, sans création dynamique ni de processus ni de canaux de communication ; toutes les possibilités de rendez-vous sont prévues et explicitées dans le modèle
- il doit autoriser une *modélisation concise* du parallélisme sans développer les opérateurs de composition parallèle, c'est-à-dire sans distribuer l'opérateur "`|[...]|`" sur les opérateurs "`;`" et "`[]`", ce qui provoquerait une explosion combinatoire de la taille du réseau. Pour donner une idée de cette propriété, on peut montrer comment — a contrario — un automate ne la satisfait

pas. Si B est un comportement, on note $C(B)$ la complexité de l'automate correspondant, mesurée en nombre d'états et/ou d'arcs. Si "`|||`" dénote l'opérateur de composition parallèle totalement asynchrone, on a :

$$C(B_1 \;\texttt{|||}\; B_2) \;\simeq\; C(B_1).C(B_2)$$

ce qui exprime que la complexité des automates croît exponentiellement quand le degré de parallélisme augmente. Au contraire, si on note C' la complexité d'une forme intermédiaire qui modélise le parallélisme de façon concise, on doit avoir :

$$C'(B_1 \;\texttt{|||}\; B_2) \;\simeq\; C'(B_1) + C'(B_2)$$

ce qui revient à dire que la complexité croît linéairement quand le degré de parallélisme augmente. C'est pourquoi la taille du réseau est presque toujours nettement inférieure à celle du graphe

- l'existence d'une forme intermédiaire de complexité moindre que celle des automates fournit à l'utilisateur des informations synthétiques qui lui permettent de mieux comprendre la structure de contrôle statique (*architecture*) de ses spécifications. En particulier cette forme intermédiaire se prête bien à une visualisation graphique, sous l'aspect de graphes de contrôle contenus dans des boîtes interconnectées par des canaux de communication. Un outil de ce type a été développé pour Xesar [BLM88]

- en principe la forme intermédiaire constitue un point d'entrée du système de vérification qui en autorise l'adaptation à de nouveaux langages moyennant l'écriture d'une partie avant de compilateur. Pour Quasar et Xesar une telle expérience n'a pu être tentée car il semble que la forme intermédiaire dépende trop étroitement du langage d'entrée. Toutefois le modèle utilisé par Cæsar est plus général et rend cette idée plausible

Il y a deux grandes classes de modèles qui permettent de décrire des systèmes distribués sans développer le parallélisme :

automates communicants : aussi appelés *machines abstraites*, ils modélisent un nombre fini de processus séquentiels (décrits par des automates finis), qui sont exécutés en parallèle et se synchronisent.

Utilisés dans des systèmes de vérification tels que SCAN [BGLN85] et Xesar, ils conviennent bien aux langages qui présentent une nette séparation entre les aspects séquentiels et parallèles (CSP, SDL, Estelle, ...).

En revanche, les automates communicants ne permettent pas de traiter les langages dérivés des algèbres de processus qui, comme CCS et LOTOS, permettent de combiner arbitrairement les opérateurs de composition séquentielle et parallèle. Par exemple, les comportements Lotos de la forme :

$$B_1 \;\texttt{>>}\; \texttt{(}B_2 \;\texttt{|||}\; B_3\texttt{)} \;\texttt{>>}\; B_4$$

ou :

$$B_1 \;\texttt{|||}\; \texttt{(}G \;\texttt{;}\; \texttt{(}B_2 \;\texttt{|||}\; B_3\texttt{))}$$

ne peuvent pas être décrits en termes d'automates communicants parce qu'un opérateur parallèle apparaît dans l'opérande d'un opérateur séquentiel

réseaux de Petri : plus puissants que le modèle précédent[19], les réseaux de Petri constituent un modèle bien adapté aux algèbres de processus. En effet, ce modèle et ses multiples extensions

[19] un ensemble d'automates communicants est un cas particulier de réseau de Petri

sont suffisamment expressifs pour décrire tous les systèmes parallèles dont le contrôle est statique. En outre la possibilité de construire un réseau à l'aide de sous-réseaux que l'on combine entre eux est un atout essentiel, utilisé par divers algorithmes de traduction (pour CCS sans les données [GM84], pour LOTOS sans les données [ML88])

C'est pourquoi le modèle réseau utilisé dans CÆSAR est basé sur les réseaux de Petri.

Remarque 4-1
Il n'est pas souhaitable d'utiliser les réseaux de Petri comme langage de spécification car leur trop grande richesse d'expression conduit à des descriptions peu structurées. En revanche, lorsqu'ils sont engendrés automatiquement, ils constituent une forme intermédiaire valable. Sous cet angle il y a un progrès semblable entre l'abandon des "**goto**" au profit de la programmation structurée et le passage des réseaux de Petri aux langages évolués comme LOTOS. ■

4.2 Avantages du modèle réseau

Le fait d'utiliser un tel modèle intermédiaire constitue la différence essentielle entre les deux approches pour la traduction de LOTOS :

- dans l'approche dynamique, le graphe est construit directement à partir du programme LOTOS : chaque état du graphe est une expression de comportement LOTOS. La situation serait analogue si l'on utilisait SUBLOTOS au lieu de LOTOS : chaque état serait un couple ayant pour premier élément une expression de comportement SUBLOTOS (§ 3.5.15, p. 62)

- au contraire dans l'approche statique, l'état courant du réseau peut être complètement caractérisé par un couple dont le premier élément est un ensemble de places du réseau de Petri et dont le second contient les valeurs des variables d'état

Il en résulte une amélioration considérable des performances de la phase de simulation :

gain en mémoire : avec l'approche statique, chaque état est une expression algébrique dont la complexité peut atteindre ou dépasser celle du programme LOTOS traité. Par comparaison, un état d'un réseau est beaucoup plus compact (pour les exemples qui l'on peut vérifier exhaustivement, la taille d'un état varie généralement entre quelques dizaines et quelques centaines d'octets)

gain en temps : étant donné un état, il faut calculer ses successeurs. Lorsque les états sont représentés par des expressions de comportement, le problème est alors de trouver quelles règles de réécriture sont applicables à ce terme. A l'heure actuelle on ne connaît pas d'algorithme efficace pour identifier les redex d'un comportement LOTOS. Au contraire, les règles d'évolution des réseau de Petri permettent de déterminer quasi-immédiatement quelles transitions sont franchissables à partir de l'état courant.

En outre lorsque l'on effectue une simulation exhaustive, chaque nouvel état doit être comparé aux états existants afin de détecter les circuits dans le graphe et de construire un graphe fini. Dans le cas de l'approche statique, il faut comparer deux expressions de comportement LOTOS pour déterminer si elles sont "équivalentes" : il s'agit d'un problème difficile, tant au plan théorique que pratique. En revanche, dans l'approche statique, chaque état du réseau est, par construction, sous forme normale et la comparaison de deux états se ramène au test d'égalité des chaînes de bits qui les représentent

4.3 Présentation du modèle réseau

Les différents aspects du réseau sont introduits ici progressivement et de manière informelle.

Places, transitions et marques

Comme le modèle réseau de Petri dont il s'inspire, le réseau comporte un nombre fini de *places*[20] et de *transitions*[21]. Chaque place est désignée par un numéro ; aucun autre attribut n'est attaché aux places. Chaque transition possède un ensemble de *places d'entrée* et un ensemble de *places de sortie*.

Chaque place peut recevoir au plus une *marque*. Les marques évoluent selon les règles usuelles : une transition ne peut être franchie que si toutes les places d'entrée sont marquées. Après franchissement, les places d'entrée perdent leurs marques alors que chaque place de sortie en reçoit une.

Par rapport aux réseaux de Petri ordinaires, le modèle adopté pour LOTOS présente des différences notables que nous allons énumérer.

Unités

Les réseaux produits par CÆSAR sont structurés : ils sont composées de "boîtes", appelées *unités*. Une unité modélise un comportement SUBLOTOS, exécuté en parallèle avec d'autres comportements représentés par d'autres unités.

Chaque unité regroupe un ensemble de places entre lesquelles le contrôle est de nature strictement séquentielle : à tout instant, parmi toutes les places d'une unité, une seule au plus est marquée.

Une unité peut toutefois contenir des sous-unités qui définissent des sous-comportements concurrents. Cette relation d'imbrication entre unités définit une arborescence qui structure les unités de façon hiérarchique.

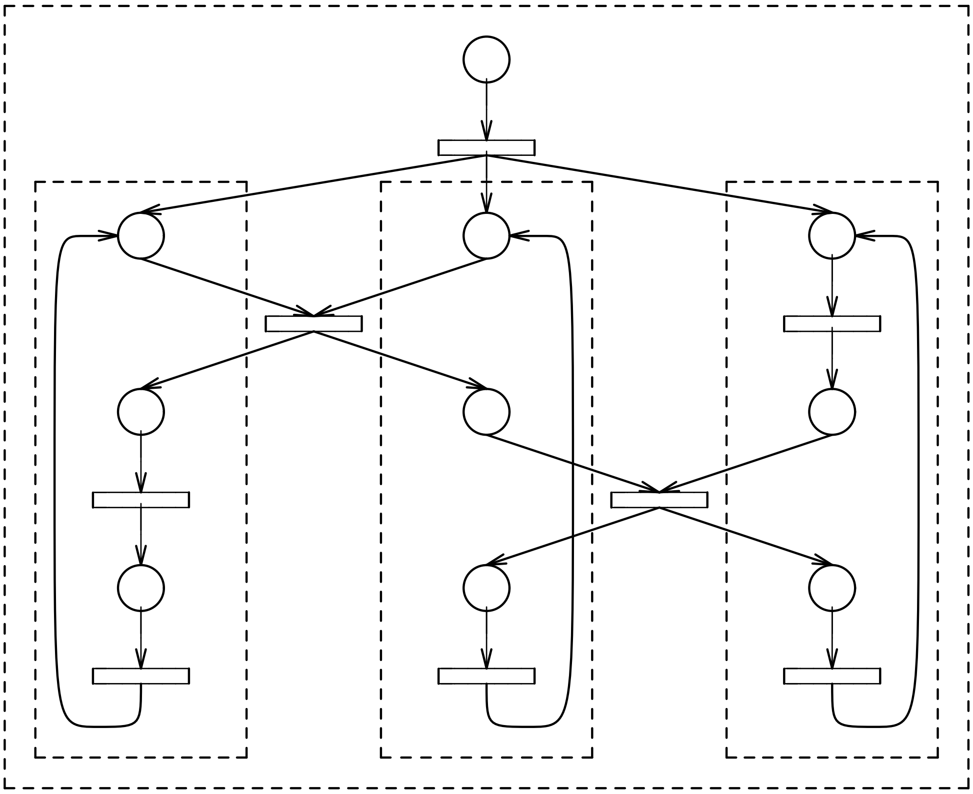
Les transitions qui relient des places appartenant à des unités disjointes traduisent la synchronisation et les communications entre les comportements associés aux unités.

Cette structuration du réseau répond à plusieurs objectifs :

- améliorer la lisibilité du réseau, considéré comme outil de documentation graphique (§ 4.1, p. 66)
- augmenter l'efficacité de la simulation, qui tire parti du caractère séquentiel des unités (§ 4.6.2, p. 91)
- permettre une simulation progressive dans laquelle on réduit le graphe par application de critères d'équivalence au fur et à mesure qu'il est produit. Cette approche est encore un thème de recherche

Exemple 4-1

[20] ne pas confondre avec les "états ", terme réservé au modèle graphe
[21] ne pas confondre avec les "arcs" du modèle graphe



■

Variables d'états

Les places et les transitions ne constituent qu'une partie du réseau, servant à décrire le contrôle. Pour modéliser la totalité du langage LOTOS il faut compléter le modèle par une partie données. Ce modèle étendu est connu sous le nom de *réseau de Petri interprété*.

La partie données est identique à celle des spécifications SUBLOTOS (§ 3.5.1, p. 58) : elle est constituée d'un ensemble fini de variables d'état, ayant une portée globale, dont la valeur, initialement indéfinie, peut être modifiée plusieurs fois.

Remarque 4-2
En ce qui concerne les données, la sémantique du modèle réseau est donc celle d'un langage impératif, ce qui lui confère une grande efficacité. Mais, comme les réseaux sont construits à partir de spécifications LOTOS, ils héritent ainsi des propriétés liées aux langages fonctionnels ; en particulier, les problèmes liés aux variables non initialisées et aux variables partagées entre des processus concurrents ne se posent pas. ■

Actions

Pour manipuler les variables d'état, les transitions du réseau sont étiquetées par des *actions* qui peuvent prendre diverses formes :

- l'*action vide*, notée "**none**", est sans effet

- une *condition*, notée "**when** V", empêche le franchissement de la transition si l'expression booléenne V portant sur les valeurs des variables d'état est évaluée à *false*
- une *affectation*, notée "$\widehat{X_0}, \ldots \widehat{X_n}$ **:=** $V_0, \ldots V_n$", permet d'affecter aux variables $\widehat{X_0}, \ldots \widehat{X_n}$ les valeurs respectives des expressions $V_0, \ldots V_n$, évaluées au moment où l'on franchit la transition
- une *itération*, notée "**for** $\widehat{X_0}, \ldots \widehat{X_n}$ **among** $S_0, \ldots S_n$" permet de faire prendre successivement aux variables $\widehat{X_0}, \ldots \widehat{X_n}$ toutes les valeurs appartenant aux domaines respectifs des sortes $S_0, \ldots S_n$. La même transition est donc franchie plusieurs fois
- enfin il est possible de fabriquer des actions plus complexes en combinant les actions au moyen d'opérateurs de composition séquentielle et collatérale, notés respectivement "`;`" et "`&`"

Remarque 4-3
Ce modèle est plus général que celui de QUASAR pour lequel une transition peut être étiquetée uniquement par une condition et une liste d'affectations vectorielles. ■

Portes et offres

Comme pour LOTOS et SUBLOTOS, la sémantique du réseau est définie à partir du modèle graphe. C'est ainsi que le franchissement d'une transition du réseau produit un ou plusieurs arcs au niveau du graphe correspondant. Les labels de ces arcs sont déterminés à partir de deux attributs qui étiquettent les transitions du réseau :

- une porte SUBLOTOS
- une liste d'offres de rendez-vous SUBLOTOS

ε-transitions

Finalement, on complète le modèle réseau par des transitions spéciales, appelées *ε-transitions*, qui jouent à rôle analogue à celui des ε-transitions dans la théorie des automates non-déterministes [ASU86, p. 114].

Ces transitions sont étiquetées par une porte fictive, notée "ε" et par une liste d'offres vide. Le franchissement des ε-transitions est *spontané* : il ne correspond pas à une évolution observable et ne produit donc aucun arc dans le graphe. Il est aussi *atomique*, ce qui signifie qu'il ne peut pas être interrompu par une action concurrente.

Les ε-transitions, qui apportent une nouvelle forme de non-déterminisme dans le fonctionnement des réseaux de Petri, ont été introduites pour que le réseau puisse être construit simplement et progressivement, par compositions successives de sous-réseaux. Elles peuvent être étiquetées par des actions, ce qui permet d'exprimer les modifications du contexte définies par les règles de substitution de la sémantique de SUBLOTOS (§ 3.5.3, p. 59). Dans CÆSAR les ε-transitions sont employées notamment :

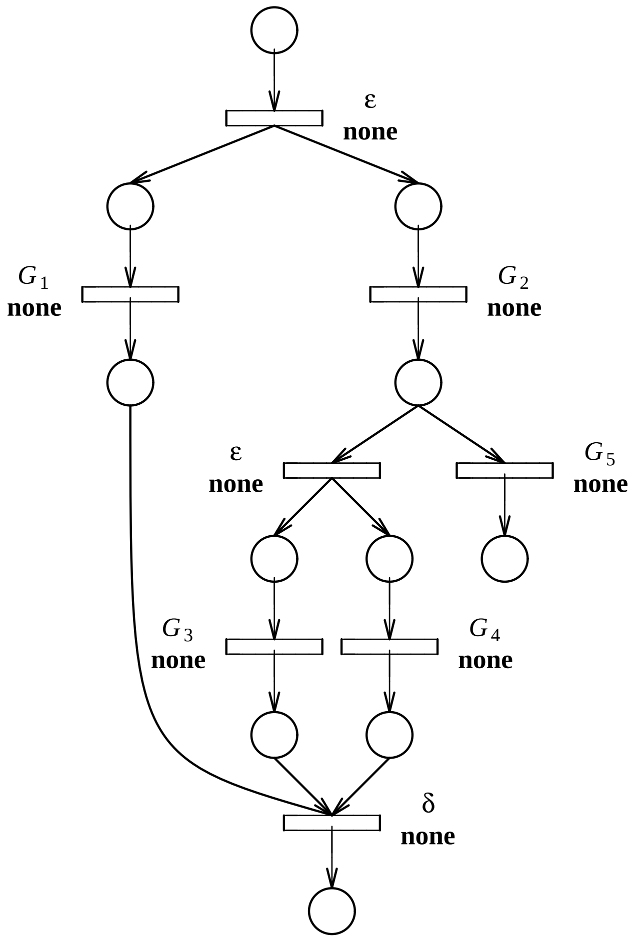
- pour traduire les opérateurs "`||`", "`|||`" et "`|[`$G_0, \ldots G_n$`]|`" : en LOTOS le lancement synchrone (*fork*) de deux processus n'est pas un événement observable[22]

  Exemple 4-2
  Le comportement :

$$G_1 \ ; \ \textbf{exit} \ ||| \ G_2 \ ; \ ((G_3 \ ; \ \textbf{exit} \ ||| \ G_4 \ ; \ \textbf{exit}) \ [] \ G_5 \ ; \ \textbf{stop})$$

[22] alors que la terminaison synchrone (*join*) est modélisée par un rendez-vous sur la porte "δ"

produit le réseau suivant :



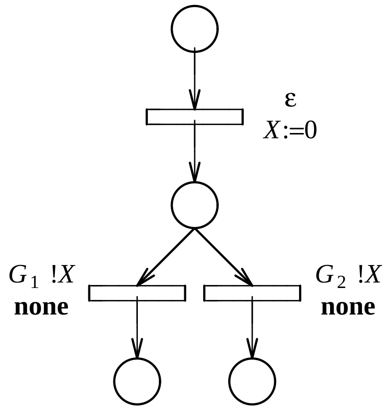
■

- pour traduire les opérateurs "->", "**let**", "**choice**" qui donnent lieu à une action (condition, affectation, itération) servant à modifier ou consulter la valeur des variables d'état sans que cela occasionne un rendez-vous :

  Exemple 4-3
  Le comportement :

  let X:NAT=0 **in** (G_1 !X ; **stop** [] G_2 !X ; **stop**)

  produit le réseau suivant :



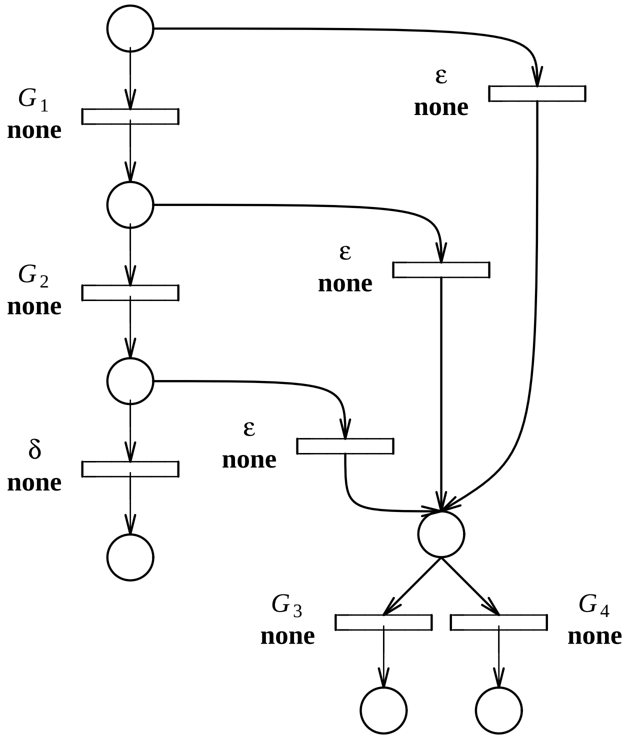
■

- pour traduire l'opérateur "[>" : on utilise des ε-transitions afin d'exprimer l'interruption d'un comportement par un autre

  Exemple 4-4
  Le comportement :

$$G_1 \ ; \ G_2 \ ; \ \textbf{exit} \ \texttt{[>} \ (G_3 \ ; \ \textbf{stop} \ \texttt{[]} \ G_4 \ ; \ \textbf{stop})$$

  produit le réseau suivant :



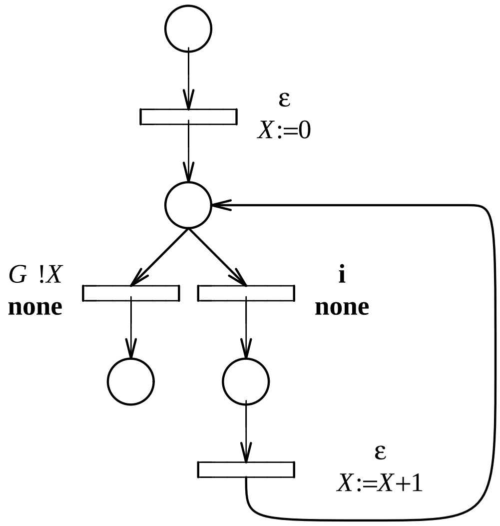
■

- pour traduire les instanciations récursives : on utilise des ε-transitions qui permettent de reboucler vers l'état initial et d'affecter aux paramètres formels la valeur des paramètres effectifs

Exemple 4-5
Le comportement "P [G] (0)" dans lequel P est le processus défini comme suit :

process P [G] (X : NAT) :=
  G !X ; **stop** [] **i** ; P[G] ($X+1$)
endproc

produit le réseau suivant :



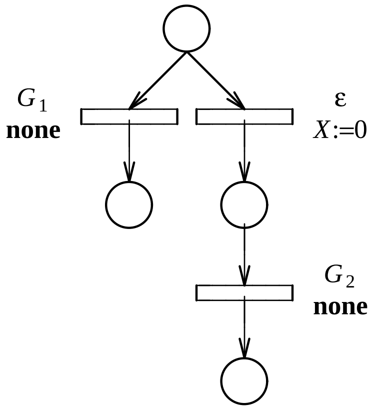
■

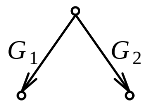
Il ne faut pas confondre les ε-transitions avec les transitions dont la porte est masquée par une instruction "**hide**", qui produisent dans le graphe des arcs étiquetés "**i**". L'exemple suivant illustre la différence ; le problème de la sémantique des ε sera détaillé plus loin (§ 4.5.6, p. 89).

Exemple 4-6
Le comportement :

G_1 ; **stop** [] **let** X : NAT=0 **in** G_2 ; **stop**

produit comme réseau et comme graphe :

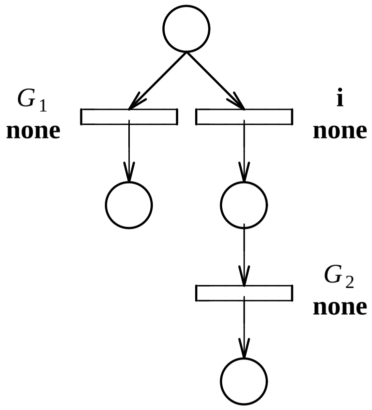


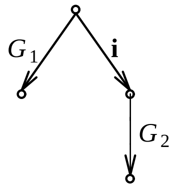


alors que le comportement :

$$G_1 \ ; \ \mathbf{stop} \ [] \ \mathbf{i} \ ; \ G_2 \ ; \ \mathbf{stop}$$

produit :





■

Remarque 4-4
Il aurait été possible de ne pas utiliser la notion d'ε-transitions dans le modèle réseau et de s'inspirer des algorithmes de traduction existants [GM84] [ML88] qui possèdent plusieurs avantages :

- la sémantique d'un modèle réseau sans ε-transition est plus simple
- l'algorithme de simulation, qui construit un graphe à partir d'un réseau, est plus efficace lorsqu'il n'a pas à traiter les ε-transitions

Mais plusieurs arguments plaident en faveur de l'introduction des ε-transitions dans le cas de la compilation de LOTOS :

- la notion d'ε-transition, qui permet d'effectuer des actions spontanées et atomiques dans les réseaux de Petri, est intéressante en soi
- la meilleure méthode pour engendrer directement un réseau sans ε-transition est basée sur l'algorithme de la résiduelle [BS87]. Or, dans certains cas, cet algorithme peut nécessiter une place mémoire exponentielle. Au contraire, l'algorithme de génération utilisé par CÆSAR fonctionne de manière "hors-contexte" et ne présente donc pas cet inconvénient
- puisque la présence d'ε-transitions amoindrit les performances de la phase de simulation, il est toujours possible d'éliminer les ε-transitions[23]. Cette idée est mise en œuvre par la phase d'optimisation de CÆSAR.

  L'élimination d'ε-transitions est très facile lorsque le réseau a été complètement construit, car on dispose alors de toutes les informations nécessaires. Il s'agit de transformations simples, dont le coût en temps et en espace mémoire est faible.

  Conceptuellement, il peut sembler choquant d'avoir introduit les ε-transitions pour les supprimer ensuite. Mais la décomposition de la traduction en deux étapes, génération puis optimisation, permet de fonctionner avec un espace mémoire plus restreint. En pratique, les performances en temps de CÆSAR pour la génération et l'optimisation du réseau sont excellentes (quelques secondes pour des réseaux de Petri ayant environ un millier de places) et c'est

[23] sauf éventuellement l'ε-transition *fork* initiale dont le rôle est de lancer l'exécution des unités qui constituent le réseau

essentiellement à la phase de simulation que doivent être consacrés les efforts pour améliorer les performances

- à plus long terme, l'utilisation d'ε-transitions pourrait constituer un avantage décisif. Il semble en effet probable que l'avenir des techniques de validation pour LOTOS passe par une analyse approfondie des propriétés statiques globales du réseau. On ne pourra pas indéfiniment faire l'économie de techniques d'analyse du flux de contrôle et du flux des variables, semblables à celles que mettent en œuvre les compilateurs optimiseurs.

  Or le coût de ces algorithmes est généralement polynômial en fonction de la taille et de la complexité du modèle analysé. Dans cet esprit, le fait d'avoir un réseau compact, dans lequel les transferts de contrôle et les opérations sur les données sont factorisées grâce aux ε-transitions, est un point positif.

  En effet la taille d'un réseau construit à l'aide d'ε-transitions est souvent moindre que celle d'un réseau équivalent sans ε-transitions ; dans certains cas la différence est considérable[24]. En outre les réseaux construits sans ε-transitions ne possèdent plus les "bonnes propriétés" de structuration : ils ne permettraient plus de visualiser graphiquement les comportements LOTOS auxquels ils correspondent

■

4.4 Syntaxe du réseau

Pour simplifier la présentation, on aura recours, dans cette section, à des définitions en avant, signalées par le symbole "†".

4.4.1 Réseau

On appelle *réseau* construit à partir d'une spécification SUBLOTOS Λ un heptuplet $(\mathcal{Q},\mathcal{U},\mathcal{T},\mathcal{G},\mathcal{X},\mathcal{S},\mathcal{F})$ où :

- $\mathcal{Q}$ est un ensemble non vide et fini de places†
- $\mathcal{U}$ est une unité† construite sur $\mathcal{Q}$ et appelée *unité racine*
- $\mathcal{T}$ est un ensemble fini de transitions† construites sur $\mathcal{Q}$, $\mathcal{G}$, $\mathcal{X}$, $\mathcal{S}$ et $\mathcal{F}$
- $\mathcal{G}$ est l'ensemble des portes SUBLOTOS présentes dans Λ
- $\mathcal{X}$ est l'ensemble des variables SUBLOTOS présentes dans Λ
- $\mathcal{S}$ est l'ensemble des sortes SUBLOTOS présentes dans Λ
- $\mathcal{F}$ est l'ensemble des opérations SUBLOTOS présentes dans Λ

4.4.2 Places

On appelle *place* un élément d'un ensemble $\mathcal{Q}$ (sur lequel on n'impose aucune contrainte particulière).

[24]soit R l'expression régulière $(a|b)^* a \underbrace{(a|b) \ldots (a|b)}_{(N-1)\ fois}$; tout automate déterministe reconnaissant le langage de R possède au moins 2^N états alors qu'il existe un automate non-déterministe équivalent, utilisant une transition ε et n'ayant que $(N+1)$ états

4.4.3 Unités

Si $\mathcal{Q}$ est un ensemble de places, on appelle *unité* construite sur $\mathcal{Q}$ un triplet $(\widetilde{Q}, Q_0, \widetilde{U})$ où :

- $\widetilde{U}$ est un ensemble d'unités construites sur $\mathcal{Q}$, appelées *sous-unités.* On note "*units*(U)" l'ensemble des sous-unités de l'unité U et *units*$^\star(U)$ l'ensemble des sous-unités transitivement incluses dans U

- $\widetilde{Q}$ est un ensemble non vide de places de $\mathcal{Q}$, appelées *places propres* de U : ce sont des places qui appartiennent à l'unité U sans appartenir à aucune sous-unité de U. On note "*places*(U)" l'ensemble des places propres de l'unité U et *places*$^\star(U)$ l'ensemble des places contenues dans U et dans les unités transitivement incluses dans U

- Q_0 est une place appartenant à $\widetilde{Q}$, appelée *place initiale.* On note "*first*(U)" la place initiale de l'unité U

Remarque 4-5
Dans le logiciel CÆSAR, on associe aussi à chaque unité le comportement SUBLOTOS auquel elle correspond afin de permettre éventuellement à l'utilisateur de mieux comprendre la structure du réseau. ■

L'*unité racine* $\mathcal{U}$ du réseau est l'unité qui englobe transitivement toutes les places et toutes les autres unités du réseau. La structuration arborescente du réseau en unités est telle que les ensembles *places*(U), pour U décrivant *units*$^\star(\mathcal{U})$, constituent une partition de l'ensemble $\mathcal{Q}$ des places du réseau.

La place initiale de l'unité racine, c'est-à-dire *first*$(\mathcal{U})$, est appelée *place initiale du réseau.*

4.4.4 Transitions

Si $\mathcal{Q}$ est un ensemble de places, $\mathcal{G}$ un ensemble de portes, $\mathcal{X}$ un ensemble de variables, $\mathcal{S}$ un ensemble de sortes et $\mathcal{F}$ un ensemble d'opérations, on appelle *transition* construite $\mathcal{Q}$, $\mathcal{G}$, $\mathcal{X}$, $\mathcal{S}$ et $\mathcal{F}$ un quintuplet $(\widetilde{Q_i}, \widetilde{Q_o}, G, \widehat{O}, A)$ où :

- $\widetilde{Q_i}$ est un ensemble de places de $\mathcal{Q}$, appelées *places d'entrée.* On note "*in*(T)" l'ensemble des places d'entrée de la transition T

- $\widetilde{Q_o}$ est un ensemble de places de $\mathcal{Q}$, appelées *places de sortie.* On note "*out*(T)" l'ensemble des places de sortie de la transition T

- G est une porte de $\mathcal{G} \cup \{\varepsilon\}$. On note "*gate*$(T)$" la porte qui étiquette la transition T

- $\widehat{O}$ est une liste d'offres† construites sur $\mathcal{X}$, $\mathcal{S}$ et $\mathcal{F}$. On note "*offer*(T)" la liste d'offres qui étiquette la transition T ; par abus de langage on appelle *offer*(T) l'*offre de* T, bien qu'il s'agisse en réalité d'une liste d'offres

- A est une action† construite sur $\mathcal{X}$, $\mathcal{S}$ et $\mathcal{F}$. On note "*action*(T)" l'action qui étiquette la transition T

Il est commode de partitionner l'ensemble $\mathcal{T}$ des transitions du réseau en trois classes :

transitions visibles : il s'agit des transitions T dont la porte est de la forme $G_{\bullet 0}$, c'est-à-dire que *gate*(T) fait partie des paramètres portes $gate_0(\Lambda), \ldots\ gate_m(\Lambda)$ de la spécification Λ

transitions cachées : il s'agit des transitions T dont la porte est de la forme $G_{\bullet j}$ avec $j > 0$, c'est-à-dire que *gate*(T) est, soit égale à "$\mathbf{i}_{\bullet 1}$", soit déclarée par une instruction "**hide**"

ε**-transitions :** il s'agit des transitions T dont la porte est égale à ε

Dans la suite de ce chapitre, la lettre T désigne une transition et non pas un type.

4.4.5 Offres

Si $\mathcal{X}$ est un ensemble de variables, $\mathcal{S}$ un ensemble de sortes et $\mathcal{F}$ un ensemble d'opérations, on appelle *offre* construite sur $\mathcal{X}$, $\mathcal{S}$ et $\mathcal{F}$ une offre de rendez-vous O_i définie selon les règles de la syntaxe SubLotos (§ 3.4.8, p. 55). L'offre d'une transition T est donc un n-uplet $\langle O_1, \ldots\ O_n \rangle$.

Toute transition T dont la porte est "ε" a comme offre le n-uplet vide ($n = 0$), noté "$\langle\rangle$".

Remarque 4-6
Une fois la phase de génération terminée, toutes les offres attachées aux transitions du réseau ont la forme $\langle O_1, \ldots\ O_n \rangle$ où chaque O_i est de la forme "$!\,V_i$" exclusivement. En effet, à la fin de la génération (§ 6.1.11, p. 128), toutes les offres "$?X:S$" figurant dans le réseau sont remplacées par "$!\,X$" en même temps que des actions d'itération "**for** X **among** S" sont introduites. En revanche pendant la génération, les deux types d'offres ("$!\,V$" et "$?X:S$") coexistent. ■

4.4.6 Actions

Si $\mathcal{X}$ est un ensemble de variables, $\mathcal{S}$ un ensemble de sortes et $\mathcal{F}$ un ensemble d'opérations, on appelle *action* construite sur $\mathcal{X}$, $\mathcal{S}$ et $\mathcal{F}$ une expression A définie par la syntaxe suivante :

$$
\begin{array}{rcl}
A & \equiv & \textbf{none} \\
 & | & \textbf{when } V \\
 & | & \widehat{X_0}, \ldots\ \widehat{X_n} \texttt{ := } V_0, \ldots\ V_n \ \textit{(from\_sync)} \\
 & | & \textbf{for } \widehat{X_0}, \ldots\ \widehat{X_n} \textbf{ among } S_0, \ldots\ S_n \\
 & | & A_1 \texttt{ ; } A_2 \\
 & | & A_1 \texttt{ \& } A_2
\end{array}
$$

où les non-terminaux V, $\widehat{X}$, S dénotent respectivement une valeur, une liste de variables et une sorte définies selon les règles de la syntaxe SubLotos. Les variables qui figurent dans les listes $\widehat{X_0}, \ldots\ \widehat{X_n}$ sont deux à deux distinctes.

Le non-terminal *from_sync* qui apparaît dans la définition syntaxique de A dénote une valeur booléenne (*true* ou *false*) qui indique si l'affectation est due ou non à un rendez-vous ; il s'agit d'une indication transmise par la phase de génération à la phase d'optimisation.

4.4.7 Syntaxe graphique

Il est possible de représenter les réseaux sous forme graphique. Cette notation qui, dans les exemples précédents, a été utilisée informellement, possède néanmoins une signification précise :

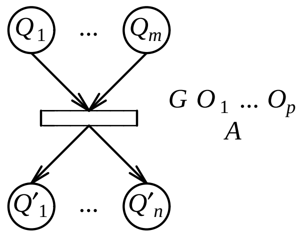
- une place est figurée par un cercle
- la place initiale d'une unité est située au-dessus de tous les autres places propres de cette unité

- une unité est figurée par un contour en pointillés englobant des places et d'autres unités : *cf.* exemple 4-1 (p. 68). En général on ne représentera pas les unités, sauf lorsque cela est indispensable

- une transition est figurée par un rectangle auquel sont attachés ses attributs. C'est ainsi qu'une transition T ayant pour attributs :

$$\begin{cases} in(T) = \{Q_1, \ldots\ Q_m\} \\ out(T) = \{Q'_1, \ldots\ Q'_n\} \\ gate(T) = G \\ offer(T) = \langle O_1, \ldots\ O_p \rangle \\ action(T) = A \end{cases}$$

  sera représentée par le schéma suivant :



4.5 Sémantique opérationnelle du réseau

Définir la sémantique du réseau, c'est spécifier comment construire un automate qui modélise toutes les évolutions possibles du réseau. Le modèle réseau possède une sémantique opérationnelle. Elle est exprimée par un état initial et une relation de transition qui spécifie les règles de passage d'un état à un autre.

Cette relation est définie par étapes, d'abord pour la partie contrôle, puis pour la partie données, ensuite en superposant contrôle et données et finalement en prenant en compte les ε-transitions. On note $(\mathcal{Q}, \mathcal{U}, \mathcal{T}, \mathcal{G}, \mathcal{X}, \mathcal{S}, \mathcal{F})$ le réseau dont on définit la sémantique.

4.5.1 Marquages

Dans un premier temps on considère uniquement la partie "contrôle" du réseau, sans tenir compte des offres ni des actions qui étiquettent les transitions. On ne prend pas non plus en compte la différence qui existe entre transitions visibles, transitions cachées et ε-transitions.

Sous ces hypothèses, les règles d'évolution du réseau sont identiques à celles d'un réseau de Petri ordinaire [Bra83, p. 13] dans lequel, par construction, chaque place peut recevoir au plus une marque.

- on appelle *marquage* un sous-ensemble des places de $\mathcal{Q}$; un marquage caractérise les places marquées à un instant donné

- on appelle *marquage initial* et on note "M_0" le marquage dans lequel seule la place initiale de l'unité racine $\mathcal{U}$ est marquée :

$$M_0 = \{first(\mathcal{U})\}$$

- on appelle *relation de transition entre marquages* la relation, notée "$[M_1\ ,\ M'\ ,\ M''] \xrightarrow{m} M_2$", qui exprime que l'on peut passer du marquage M_1 au marquage M_2 en franchissant une transition T telle que $in(T) = M'$ et $out(T) = M''$ (la lettre m située au-dessus de la flèche indique qu'il s'agit de marquages). D'après les règles d'évolution des réseaux de Petri :
  - pour qu'une transition T puisse être franchie à partir d'un marquage M_1, il faut et il suffit que toutes les places d'entrée de T soient marquées dans M_1
  - après le franchissement de la transition T, on obtient un marquage M_2 obtenu à partir de M_1 dans lequel les places d'entrée de T perdent leur marque alors que les places de sortie de T en reçoivent une

  Cette relation est donc définie par :

$$\frac{(M_1 \supseteq M_{in})\ \wedge\ (M_2 = (M_1 - M_{in}) \cup M_{out})}{[M_1\ ,\ M_{in}\ ,\ M_{out}] \xrightarrow{m} M_2}$$

Remarque 4-7
Lorsque M_1, M_{in} et M_{out} sont fixés, il existe au plus un marquage M_2 tel que $[M_1\ ,\ M_{in}\ ,\ M_{out}] \xrightarrow{m} M_2$. ■

4.5.2 Contextes

A présent on considère seulement la partie "données" du réseau, c'est-à-dire les variables d'état et les actions qui étiquettent les transitions. Cet aspect de la sémantique du réseau prolonge, en la complétant, la notion de "contexte" telle qu'elle a été définie pour les spécifications SubLotos.

- on appelle *contexte* une application partielle de l'ensemble $\mathcal{X}$ des variables SubLotos vers l'ensemble des valeurs SubLotos. Pour manipuler les contextes on emploie les notations définies précédemment (§ 3.5.1, p. 58)
- on appelle *contexte initial* et on note "C_0" le contexte dans lequel aucune variable n'a été affectée :

$$C_0 = \perp$$

- on appelle *relation de transition entre contextes* la relation, notée "$[C_1\ ,\ A] \xrightarrow{c} C_2$", qui signifie que l'exécution de l'action A fait passer du contexte C_1 au contexte C_2 (la lettre c située au-dessus de la flèche indique qu'il s'agit de contextes). Cette relation est définie par récurrence sur la complexité de A :
  - l'action "**none**" n'a aucun effet sur le contexte courant :

$$\frac{C_1 = C_2}{[C_1\ ,\ \mathbf{none}] \xrightarrow{c} C_2}$$

  - l'action "**when** V" joue un rôle de garde : le franchissement n'est possible que si la condition V est vraie. Cette action ne modifie pas le contexte courant :

$$\frac{(eval(V\ ,\ C_1) = true)\ \wedge\ (C_1 = C_2)}{[C_1\ ,\ \mathbf{when}\ V] \xrightarrow{c} C_2}$$

– l'action "$\widehat{X_0}, \ldots\ \widehat{X_n}$:= $V_0, \ldots\ V_n$" modifie le contexte courant en donnant, pour tout i de $\{0, \ldots\ n\}$, la valeur V_i aux variables de la liste $\widehat{X_i}$. L'affectation est *vectorielle*, ce qui signifie que toutes les valeurs V_i sont évaluées dans le contexte courant et que toutes les variables sont affectées simultanément :

$$\frac{C_2 = C_1 \oslash \bigoplus_{i \in \{0, \ldots\ n\}} (\widehat{X_i} \rightsquigarrow eval(V_i\ ,\ C_1))}{[C_1\ ,\ \widehat{X_0}, \ldots\ \widehat{X_n}\ \texttt{:=}\ V_0, \ldots\ V_n] \xrightarrow{c} C_2}$$

Remarque 4-8
Le booléen *from_sync*, quelle que soit sa valeur, ne modifie pas la sémantique de l'affectation. ■

– l'action "**for** $\widehat{X_0}, \ldots\ \widehat{X_n}$ **among** $S_0, \ldots\ S_n$" modifie le contexte courant en affectant, pour tout i de $\{0, \ldots\ n\}$, des valeurs quelconques prises dans $domain(S_i)$ aux variables de la liste $\widehat{X_i}$. Les variables X_j faisant partie d'une même liste $\widehat{X_i}$ ne reçoivent pas nécessairement la même valeur :

$$\frac{C_2 = C_1 \oslash \bigoplus_{i \in \{0, \ldots\ n\}} (\bigoplus_{X_j \in \widehat{X_i}} (X_j \rightsquigarrow oneof(S_i)))}{[C_1\ ,\ \textbf{for}\ \widehat{X_0}, \ldots\ \widehat{X_n}\ \textbf{among}\ S_0, \ldots\ S_n] \xrightarrow{c} C_2}$$

– l'action "A_1 ; A_2" correspond à l'exécution séquentielle des actions A_1 puis A_2

$$\frac{([C_1\ ,\ A_1] \xrightarrow{c} C) \wedge ([C\ ,\ A_2] \xrightarrow{c} C_2)}{[C_1\ ,\ A_1\ \texttt{;}\ A_2] \xrightarrow{c} C_2}$$

– l'action "A_1 & A_2" correspond à l'exécution collatérale des actions A_1 et A_2, le résultat ne devant pas dépendre de l'ordre d'évaluation de A_1 et A_2 :

$$\frac{([C_1\ ,\ A_1\ \texttt{;}\ A_2] \xrightarrow{c} C_2) \wedge ([C_1\ ,\ A_2\ \texttt{;}\ A_1] \xrightarrow{c} C_2)}{[C_1\ ,\ A_1\ \texttt{\&}\ A_2] \xrightarrow{c} C_2}$$

On n'a donc pas le droit d'écrire "X := $X + 1$ & Y := X". Il faut soit imposer un ordre entre les deux affectations, en utilisant la composition séquentielle, soit employer une affectation vectorielle. L'opérateur "&" a été introduit afin de préserver le non-déterminisme (qui s'avère utile pour optimiser la simulation)

Remarque 4-9
Lorsque C_1 et A sont fixés, le nombre de contextes C_2 tels que $[C_1\ ,\ A] \xrightarrow{c} C_2$ peut être égal à 0 (si une garde "**when**" apparaissant dans A n'est pas franchissable), égal à 1 ou supérieur à 1 (à cause des actions "**for** ... **among**"). ■

4.5.3 Positions

A présent on prend en compte simultanément les marquages et les contextes, mais sans différencier encore les ε-transitions des autres transitions.

- on appelle *position* un couple $\langle M, C\rangle$ formé d'un marquage M et d'un contexte C
- si π est une position $\langle M, C\rangle$ on note "$marking(\pi)$" le marquage M de π
- si π est une position $\langle M, C\rangle$ on note "$context(\pi)$" le contexte C de π

- on appelle *position initiale* et on note π_0 la position formée du marquage initial et du contexte initial :

$$\pi_0 = \langle M_0, C_0 \rangle$$

- on note L les labels (§ 1.2.8, p. 22) construits sur l'ensemble des portes formelles de la spécification LOTOS $gate_1(\lambda), \ldots\ gate_n(\lambda)$ auxquelles on ajoute "**i**", "δ" et "ε", l'ensemble des sortes et l'ensemble des opérations définies dans la spécification LOTOS. Ces labels ont donc la forme "$G\ V_1, \ldots\ V_n$", G désignant une porte et $V_1, \ldots\ V_n$ des expressions de valeur ne contenant aucune variable

- il est nécessaire de convertir la porte et l'offre d'une transition en un label. D'après la remarque 4-6 (p. 77), toutes les offres sont de la forme $\langle !V_1, \ldots\ !V_n \rangle$. Trois cas sont à distinguer, selon que la porte est visible, cachée ou égale à "ε" :

$$eval\_label(G, \langle !V_1, \ldots\ !V_n \rangle, C) =$$
$$\begin{cases} \textbf{si } G \textbf{ est de la forme } G_{0\bullet 0} \textbf{ alors } G_0\ eval(V_1\ ,\ C), \ldots\ eval(V_n\ ,\ C) \\ \textbf{sinon si } G \textbf{ est de la forme } G_{0\bullet j} \textbf{ avec } j > 0 \textbf{ alors i} \\ \textbf{sinon si } G = \varepsilon \textbf{ alors } \varepsilon \end{cases}$$

- on appelle *relation de transition entre positions* la relation, notée "$[\pi_1\ ,\ T] \xrightarrow{p} [\pi_2\ ,\ L]$", qui exprime que l'on peut passer de la position π_1 à la position π_2 en franchissant la transition T et en produisant une action dont le label est L (la lettre p située au-dessus de la flèche indique qu'il s'agit de positions). Cette relation est définie par :

$$\frac{\begin{array}{c}([M_1\ ,\ in(T)\ ,\ out(T)] \xrightarrow{m} M_2) \wedge \\ ([C_1\ ,\ action(T)] \xrightarrow{c} C_2) \wedge \\ (L = eval\_label(gate(T), offer(T), C_2))\end{array}}{[\langle M_1, C_1 \rangle\ ,\ T] \xrightarrow{p} [\langle M_2, C_2 \rangle\ ,\ L]}$$

Remarque 4-10
Les expressions de valeur présentes dans $offer(T)$ sont évaluées dans le contexte C_2, c'est-à-dire *après* l'exécution de $action(T)$; elles ne sont pas évaluées dans C_1, *avant* l'exécution de $action(T)$. Cette décision est justifiée par la nécessité de traiter les rendez-vous de la forme "$G\ ?X\!:\!S$" qui, dans le réseau, apparaissent sous la forme d'une transition T telle que :

$$\begin{cases} gate(T) = G \\ offer(T) = \langle !X \rangle \\ action(T) = \textbf{for } X \textbf{ among } S \end{cases}$$

■

Remarque 4-11
Lorsque π_1 et T sont fixés, le nombre de positions π_2 et de labels L tels que $[\pi_1\ ,\ T] \xrightarrow{p} [\pi_2\ ,\ L]$ peut être quelconque. ■

- on appelle *ε-chaîne allant de la position π_1 à la position π_2* tout n-uplet de positions $\pi'_1, \ldots\ \pi'_n$, avec $n \geq 1$, vérifiant les propriétés suivantes :

  - π'_1 est égal à π_1
  - π'_n est égal à π_2

– les positions π'_i sont deux à deux distinctes :

$$(\forall i, j \in \{1, \dots\ n\})\ \ i \neq j \Longrightarrow \pi'_i \neq \pi'_j$$

– on passe de π'_i à π'_{i+1} en franchissant une ε-transition :

$$(\forall i \in \{1, \dots\ n-1\})\ \ (\exists T)\ \ [\pi'_i\ ,\ T] \xrightarrow{p} [\pi'_{i+1}\ ,\ \varepsilon]$$

4.5.4 Spontanéité des ε-transitions

La relation de transition entre positions traite de la même manière les ε-transitions que les autres transitions. Il s'agit pourtant de notions sémantiquement distinctes. Il faut donc compléter la définition de la sémantique du réseau pour prendre en compte les ε-transitions.

C'est pourquoi, sur la base de la relation de transition entre positions, on cherche à définir une relation plus abstraite, appelée *relation de transition entre états*. C'est cette relation qui détermine complètement, pour un réseau donné, le graphe correspondant. Si le réseau ne comportait aucune ε-transition, la relation de transition entre états serait identique à la relation de transition entre positions.

Le franchissement d'une ε-transition est "spontané" : il modifie l'état du réseau sans que cette évolution donne lieu à un événement observable, ni même à un événement invisible "**i**"[25].

La signification des ε-transitions du réseau est donc très voisine de celle des ε-transitions utilisées dans la théorie des automates non-déterministes. L'analogie est simple : la relation de transition entre positions définit un automate non-déterministe A comportant des ε-transitions. La relation de transition entre états correspond au contraire à un automate A' sans ε-transition et équivalent à A.

C'est pourquoi la définition de la relation de transition entre états s'apparente au problème d'éliminer les ε-transitions d'un automate non-déterministe A et de produire un automate déterministe équivalent. Le problème n'est toutefois pas identique puisque l'automate A' n'est pas forcément déterministe : d'un état de A' peuvent être issus plusieurs arcs étiquetés par le même label.

On utilise l'algorithme "$\varepsilon^* L$", ainsi appelé parce que son principe consiste à franchir un nombre quelconque d'ε-transitions jusqu'à atteindre une transition étiquetée par un label L différent de "ε".

En première approximation on appelle *relation de transition entre états* la relation notée "$\pi_1 \xrightarrow{L} \pi_2$" et définie par :

$$\frac{(\exists T)\ \ (\exists \pi)\ \ (gate(T) \neq \varepsilon)\ \wedge\ (\textit{il existe une } \varepsilon\textit{-chaîne allant de } \pi_1 \textit{ à } \pi)\ \wedge\ ([\pi\ ,\ T] \xrightarrow{p} [\pi_2\ ,\ L])}{\pi_1 \xrightarrow{L} \pi_2}$$

Remarque 4-12
La méthode "$\varepsilon^* L$" a été choisie pour sa grande simplicité. Il existe d'autres techniques pour rendre déterministe un automate non-déterministe, notamment l'algorithme de *subset construction* [ASU86, p. 117–121] qui a été utilisé dans la version 1.0 de CÆSAR.

Cet algorithme consiste à définir la relation de transition entre états comme quotient de la relation de transition entre positions par une opération de *fermeture* (ε-*closure*) sur les ε-transitions. La fermeture d'un ensemble de positions Π est définie par :

$$closure(\Pi) = \Pi \cup \{\pi_2 \mid (\exists \pi_1 \in \Pi)\ \ (\textit{il existe une } \varepsilon\textit{-chaîne allant de } \pi_1 \textit{ à } \pi_2)\}$$

[25] idée que l'on résume par cette formule lapidaire : "les ε ne sont pas des τ", "τ" dénotant l'action invisible en CCS

On obtient ainsi une relation sur des ensembles de positions Π_1 et Π_2, notée "$\Pi_1 \xrightarrow{L} \Pi_2$" et définie par :

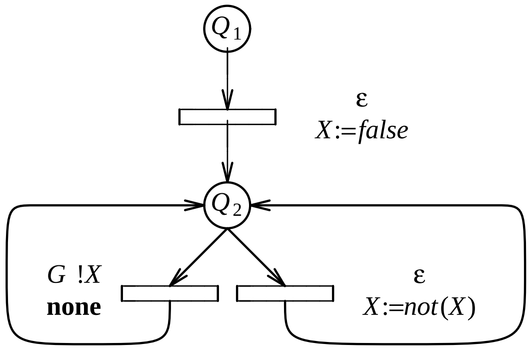
$$\frac{(\exists T)\ (\exists \pi_1, \pi_2)\ (gate(T) \neq \varepsilon)\ \wedge\ (\pi_1 \in \Pi_1)\ \wedge\ ([\pi_1\ ,\ T] \xrightarrow{p} [\pi_2\ ,\ L])\ \wedge\ (\Pi_2 = closure(\pi_2))}{\Pi_1 \xrightarrow{L} \Pi_2}$$

Dans certains cas, cette relation de transition est "meilleure" que celle obtenue avec la méthode "$\varepsilon^* L$", parce qu'elle permet d'engendrer des graphes comportant moins d'états et moins d'arcs.

Exemple 4-7
Le comportement "P [G] ($false$)" dans lequel P est le processus défini comme suit :

process P [G] (X : $BOOL$) :=
  G !X ; P [G] (X) [] P [G] (not (X))
endproc

produit le réseau suivant :



A partir de la position initiale π_0, la technique "$\varepsilon^* L$" permet d'atteindre deux nouvelles positions π_1 et π_2, avec :

$$\begin{array}{lcl} \pi_0 & = & \langle \{Q_1\}, \perp \rangle \\ \pi_1 & = & \langle \{Q_2\}, X \rightsquigarrow false \rangle \\ \pi_2 & = & \langle \{Q_2\}, X \rightsquigarrow true \rangle \end{array}$$

en franchissant six transitions entre états :

$$\begin{array}{ll} \pi_0 \xrightarrow{G\ !false} \pi_1 & \pi_0 \xrightarrow{G\ !true} \pi_2 \\ \pi_1 \xrightarrow{G\ !false} \pi_1 & \pi_1 \xrightarrow{G\ !true} \pi_2 \\ \pi_2 \xrightarrow{G\ !false} \pi_1 & \pi_2 \xrightarrow{G\ !true} \pi_2 \end{array}$$

Au contraire, la technique de *subset construction* ne produit que deux transitions entre états, à partir de l'ensemble de positions Π_0 obtenu par fermeture de la position initiale π_0 :

$$\Pi_0 = closure(\{\pi_0\}) = \{\pi_0, \pi_1, \pi_2\}$$

Ces deux transitions sont :

$$\Pi_0 \xrightarrow{G\ !false} \Pi_0 \qquad \Pi_0 \xrightarrow{G\ !true} \Pi_0$$

■

On peut montrer que, pour que de telles différences existent, il faut nécessairement que le réseau contienne des cycles d'ε-transitions, ce qui traduit la présence de récursion non gardée (*cf.* remarque 3-1 (p. 48)) dans le programme LOTOS source. Pour tous les programmes qui ne comportent pas de récursion non gardée, les deux méthodes conduisent aux mêmes résultats ; on peut donc adopter la méthode "$\varepsilon^* L$" sans dégradation significative des performances pour la quasi-totalité des spécifications LOTOS.

C'est pourquoi l'approche *subset construction*, mise en œuvre dans la version 1.0 de CÆSAR, a été abandonnée par la suite au profit de la technique "$\varepsilon^* L$". En effet le coût du calcul de la fermeture était prohibitif : il fallait mémoriser, trier et comparer des milliers d'ensembles de positions (ensembles dont le cardinal n'était pas majoré a priori). ■

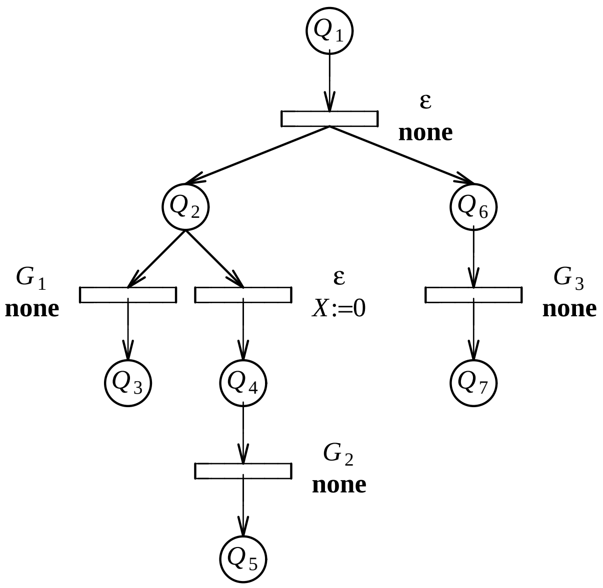
4.5.5 Atomicité des ε-transitions

L'expérience a montré que la relation de transition entre états construite selon la technique "$\varepsilon^* L$" ne correspond pas à la sémantique de LOTOS et de SUBLOTOS. Elle produit en effet des graphes souvent trop gros (comportant jusqu'à 10 fois plus d'états redondants et 20 fois plus d'arcs redondants que l'automate minimal fortement équivalent) et parfois incorrects, comme le montre l'exemple suivant.

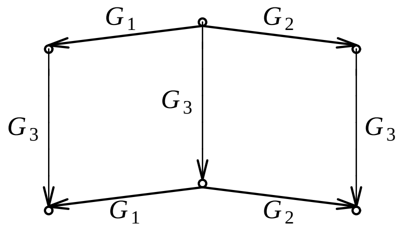
Exemple 4-8
Le comportement :

$$(G_1\ \textbf{; stop}\ \texttt{[]}\ (\textbf{let}\ X\!:\!NAT\!=\!0\ \textbf{in}\ G_2\ \textbf{; stop}))\ \texttt{|||}\ G_3\ \textbf{; stop}$$

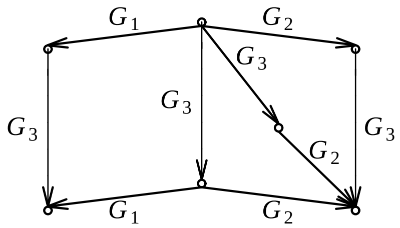
produit le réseau suivant :



Le graphe engendré par l'algorithme "$\varepsilon^* L$" ne correspond pas au graphe attendu pour le comportement LOTOS :



graphe attendu



graphe obtenu

L'évolution incriminée du réseau est :

$$\langle\{Q_1\},\bot\rangle \xrightarrow{\varepsilon} \langle\{Q_2,Q_6\},\bot\rangle \xrightarrow{\varepsilon} \langle\{Q_4,Q_6\},X\leadsto 0\rangle \xrightarrow{G_3} \underline{\langle\{Q_4,Q_7\},X\leadsto 0\rangle}$$

Lorsqu'on se trouve dans la position soulignée, on n'a plus la possibilité de franchir la transition étiquetée G_1, ce qui contredit la sémantique de LOTOS. ■

Ce problème peut s'expliquer de plusieurs façons. On doit d'abord se rappeler que l'algorithme "ε^*L" a été conçu pour préserver l'équivalence de trace (§ 1.3.2, p. 23) mais non l'équivalence forte des automates ; or c'est celle-ci qui doit être prise compte pour l'analyse des systèmes parallèles.

Exemple 4-9
Les deux graphes de l'exemple 4-8 (p. 84) sont équivalents pour la trace (c'est-à-dire qu'ils reconnaissent le même langage) mais pas fortement équivalents : en effet dans le second cas il est possible d'effectuer G_3 et de refuser ensuite G_1. ■

L'utilisation qui est faite des ε-transitions pour la construction du réseau suppose qu'elles ont une signification "atomique" : le franchissement d'une chaîne d'ε-transitions n'a de sens que s'il est suivi de celui d'une transition dont la porte est visible ou cachée ; en outre il ne doit pas être interrompu par une transition évoluant de manière asynchrone dans une autre composante du système. Ce concept d'atomicité est résumé par une pseudo-équation qui doit être interprétée comme un théorème d'expansion suivant :

$$\varepsilon^* . L \;\texttt{||}\; L' \;=\; \varepsilon^* . L . L' \;\texttt{[]}\; L' . \varepsilon^* . L$$

En revanche, il n'est pas permis d'interrompre l'exécution de la séquence "ε^*L" par l'action L' : les séquences de la forme "$\varepsilon^* . L' . L$" ou encore "$\varepsilon^* . L' . \varepsilon^* L$" ne sont pas autorisées.

Exemple 4-10
Dans l'exemple 4-8 (p. 84), le franchissement "normal" de la séquence de transitions :

$$\langle\{Q_1\},\bot\rangle \xrightarrow{\varepsilon} \langle\{Q_2,Q_6\},\bot\rangle \xrightarrow{\varepsilon} \langle\{Q_4,Q_6\},X\leadsto 0\rangle \xrightarrow{G_2} \ldots$$

a été interrompu par la transition étiquetée G_3 :

$$\langle\{Q_1\},\bot\rangle \xrightarrow{\varepsilon} \langle\{Q_2,Q_6\},\bot\rangle \xrightarrow{\varepsilon} \langle\{Q_4,Q_6\},X\leadsto 0\rangle \xrightarrow{G_3} \underline{\langle\{Q_4,Q_7\},X\leadsto 0\rangle} \xrightarrow{G_2} \ldots$$

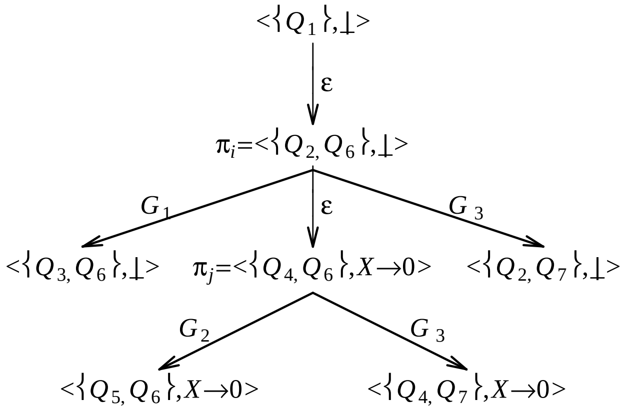
ce qui a conduit à la position soulignée à partir de laquelle il n'est plus possible de franchir la transition étiquetée G_1. ■

On peut donner à l'atomicité des ε-transitions une définition plus utile en pratique. Soit π_1 une position dont on cherche à calculer les successeurs par la relation de transition entre états. Supposons qu'il existe une ε-chaîne issue de π_1, constituée des positions $\pi_1, \ldots\ \pi_n$, ainsi qu'une transition T dont la porte n'est pas ε et qui peut être franchie à partir de deux positions π_i et π_j avec $i < j$. Deux cas doivent être distingués.

Premier cas :

Si les marquages de π_i et π_j sont distincts, il ne faut franchir T qu'une seule fois et le faire à partir de π_i et non π_j.

Exemple 4-11
Dans le cas signalé par l'exemple 4-8 (p. 84), il existe une ε-chaîne et une transition étiquetée G_3 qui peut être franchie, d'abord à partir de la position $\pi_i = \langle\{Q_2, Q_6\}, \perp\rangle$ puis à partir de $\pi_j = \langle\{Q_4, Q_6\}, X \rightsquigarrow 0\rangle$:

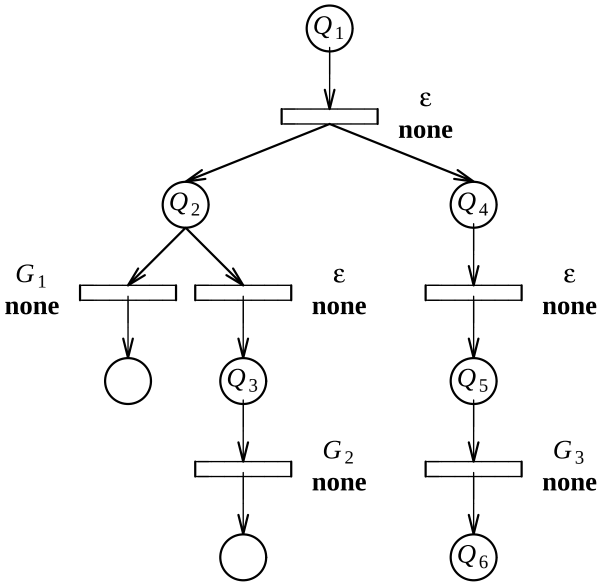


Le franchissement de cette transition G_3 à partir de π_j est incorrect puisqu'il conduit à la position $\langle\{Q_4, Q_7\}, X \rightsquigarrow 0\rangle$ à partir de laquelle la transition étiquetée G_1 ne peut plus être franchie. ■

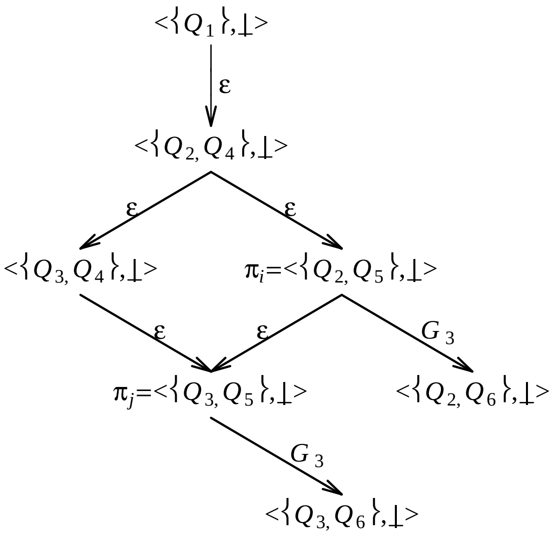
Cette modification de l'algorithme "ε^*L" sera désignée par le sigle "$\mu\varepsilon^*L$", le symbole "μ" signifiant que l'action L doit être franchie "au plus tôt" immédiatement après "la plus courte" des ε-chaînes possibles.

La condition imposée par l'algorithme "$\mu\varepsilon^*L$" pour le franchissement de transition T à partir de la position π_i doit être vérifiée sur *toutes* les ε-chaînes allant de π_1 à π_i.

Exemple 4-12
Pour le réseau suivant :



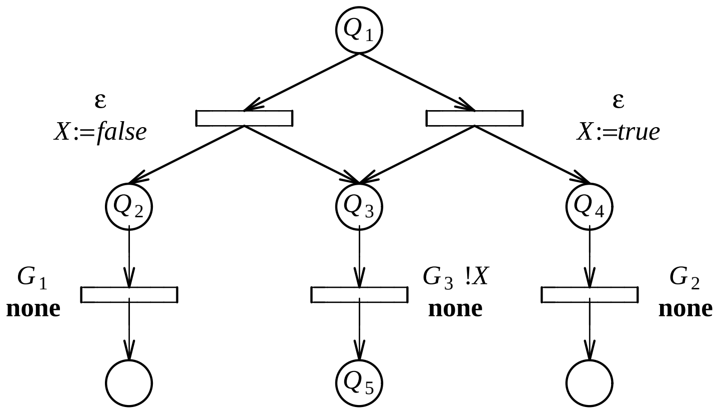
on aboutit à la situation ci-dessous :



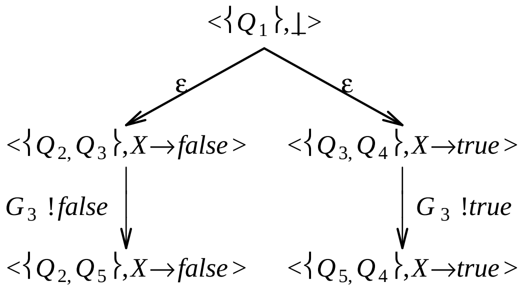
On ne doit pas franchir le transition étiquetée G_3 à partir de la position $\pi_j = \langle\{Q_3, Q_5\}, \perp\rangle$, bien qu'il existe un chemin allant de la position initiale à π_j tel que, sur ce chemin, la transition G_3 ne puisse être franchie qu'à partir de π_j. En effet il existe un autre chemin, passant par la position $\pi_i = \langle\{Q_2, Q_5\}, \perp\rangle$ à partir de laquelle on peut franchir la transition G_3. Franchir la transition G_3 à partir de π_j conduit à la position $\langle\{Q_3, Q_6\}, \perp\rangle$ à partir de laquelle il n'est plus possible de franchir la transition étiquetée G_1. ■

Ces contraintes n'interdisent pourtant pas qu'une transition T soit franchie plusieurs fois, notamment à partir de deux positions π_i et π_j ayant des marquages distincts et pouvant être atteintes à partir de π_1 en suivant des ε-chaînes, à condition qu'il n'existe aucune ε-chaîne allant de π_i à π_j ou de π_j à π_i.

Exemple 4-13
Pour le réseau suivant :



on aboutit à la situation ci-dessous :



On constate donc que la transition étiquetée G_3 peut être franchie deux fois ■

Deuxième cas :

Si les marquages de π_i et π_j sont identiques, il faut franchir T deux fois, aussi bien à partir de π_i que de π_j. En effet le problème de l'atomicité est lié au contrôle (marquages) et non aux données (contextes).

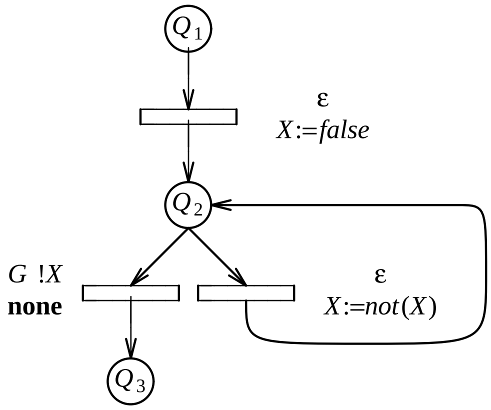
Exemple 4-14
Le comportement "*P* [*G*] (*false*)" dans lequel *P* est le processus défini comme suit :

process *P* [*G*] (*X* : *BOOL*) :=
G !*X* ; **stop** [] *P* [*G*] (*not* (*X*))
endproc

qui est équivalent à :

G !*false* ; **stop** [] *G* !*true* ; **stop**

produit le réseau suivant :



A cause du cycle constitué par l'ε-transition, il faut franchir la transition étiquetée G à partir des deux positions $\pi_i = \langle\{Q_2\}, X \leadsto \mathit{false}\rangle$ et $\pi_j = \langle\{Q_2\}, X \leadsto \mathit{true}\rangle$ ■

4.5.6 Relation de transition

On appelle *relation de transition entre états* la relation notée “$\pi_1 \xrightarrow{L} \pi_2$” et définie par :

$$\frac{\begin{array}{c}(\exists T)\ \ (\exists \pi)\ \ (gate(T) \neq \varepsilon)\ \wedge\ ([\pi\ ,\ T] \xrightarrow{p} [\pi_2\ ,\ L]) \wedge \\ (\textit{il existe une } \varepsilon\textit{-chaîne allant de } \pi_1 \textit{ à } \pi) \wedge \\ (\textit{pour toute } \varepsilon\textit{-chaîne } \pi'_1, \ldots\ \pi'_n \textit{ allant de } \pi_1 \textit{ à } \pi)\ \ (\forall i \in \{1, \ldots\ n-1\}) \\ (in(T) \not\subseteq marking(\pi'_i))\ \vee\ (marking(\pi'_i) = marking(\pi))\end{array}}{\pi_1 \xrightarrow{L} \pi_2}$$

4.5.7 Construction du graphe

A partir d’un réseau $(\mathcal{Q}, \mathcal{U}, \mathcal{T}, \mathcal{G}, \mathcal{X}, \mathcal{S}, \mathcal{F})$, on construit un automate $(\Sigma, \sigma_0, \mathcal{E}, \mathcal{G}, \mathcal{S}, \mathcal{F})$ où :

- les états de Σ sont des positions. Plus précisément Σ est défini par l’équation suivante :

$$\Sigma = \{\pi_0\} \cup \{\pi_2 \mid (\exists \pi_1 \in \Sigma)\ \ (\exists L)\ \ (\pi_1 \xrightarrow{L} \pi_2)\}$$

- l’état initial σ_0 est égal à la position initiale π_0
- les arcs de $\mathcal{E}$ sont déterminés par la relation de transition :

$$\mathcal{E} = \{(\pi_1, L, \pi_2) \mid (\pi_1 \in \Sigma)\ \wedge\ (\pi_2 \in \Sigma)\ \wedge\ (\pi_1 \xrightarrow{L} \pi_2)\}$$

Remarque 4-13
Par définition tout état est une position. La réciproque est fausse : toute position — même accessible à partir de la position initiale selon la relation de transition entre positions — ne correspond pas forcément à un état du graphe. ■

4.6 Propriétés invariantes du réseau

Les réseaux obtenus par traduction de spécifications SubLotos, pendant la phase de génération, possèdent des propriétés invariantes qui sont satisfaites par toute évolution. Au contraire tous les réseaux que l’on peut construire par application des règles de syntaxe (§ 4.4, p. 75) ne satisfont pas forcément ces “bonnes propriétés” : ils n’ont alors aucune signification puisque les règles de sémantique opérationnelle ne peuvent pas leur être appliquées.

On appelle *marquage accessible* tout marquage qui peut être dérivé à partir du marquage initial M_0 en appliquant la relation de transitions entre marquages. L’ensemble $\mathcal{M}$ des marquages accessibles est défini par l’équation suivante :

$$\mathcal{M} = \{M_0\} \cup \{M_2 \mid (\exists M_1 \in \mathcal{M})\ \ (\exists T)\ \ [M_1\ ,\ in(T)\ ,\ out(T)] \xrightarrow{m} M_2\}$$

4.6.1 Marquage sauf

A tout instant, chaque place contient au plus une seule marque : on dit que le marquage est *sauf.* Autrement dit pour toute transition T et pour tous marquages M_1 et M_2, si M_1 est accessible et $[M_1\ ,\ in(T)\ ,\ out(T)] \xrightarrow{m} M_2$ alors toutes les places marquées de $out(T)$ ne doivent pas figurer dans

M_1, à moins qu'elles ne soient également dans $in(T)$, sinon certaines places de M_2 seraient marquées deux fois :

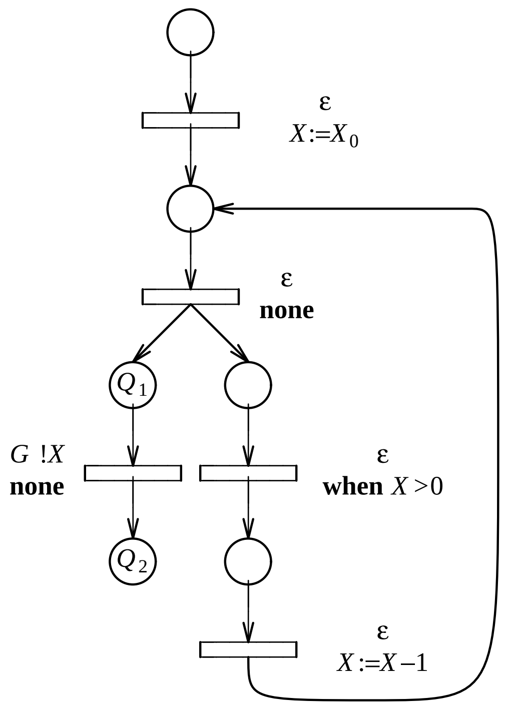
$$(M_1 - in(T)) \cap out(T) = \emptyset$$

L'hypothèse du marquage sauf, cruciale pour l'efficacité de la phase de simulation, ne pourrait pas être satisfaite si l'on acceptait les programmes LOTOS dont le contrôle n'est pas statique (§ 3.1.5, p. 47). Si la récursion à travers le parallélisme était autorisée, il faudrait plusieurs marques par place.

Exemple 4-15
Considérons le comportement LOTOS suivant "P [G] (X_0)" dans lequel P est le processus défini comme suit :

process P [G] (X:NAT) : **noexit** :=
  G !X ; **stop** ||| [$X > 0$] -> P [G] ($X - 1$)
endproc

Le réseau correspondant à ce comportement serait :



On constate que, pour $X_0 > 0$, les marques s'accumulent dans les places Q_1 et Q_2. ■

D'autres modifications de la sémantique du réseau seraient nécessaires si l'on autorisait le contrôle dynamique. On a vu que chaque état est un couple <marquage, contexte> ; autrement dit, toutes les marques d'un marquage partagent le même contexte. Avec le contrôle dynamique, il faudrait associer à *chacune* des marques d'un même marquage un contexte différent.

Exemple 4-16
Dans le cas de figure décrit dans l'exemple 4-15 (p. 90) pour que l'offre "!X" soit correctement évaluée, il faudrait que la première marque qui arrive dans la place Q_1 porte l'information $X = X_0$, la seconde $X = X_0 - 1, \ldots$ ■

Dans la sémantique du réseau, toutes les variables ont une portée globale : à tout instant, chaque variable n'existe qu'à un seul exemplaire et dénote une valeur unique. Si l'on autorisait la récursion à gauche de l'opérateur "`>>`" les variables deviendraient locales et il faudrait effectuer une gestion en pile de leurs valeurs.

Exemple 4-17
En effet, considérons le comportement LOTOS suivant "P `[`G_1, G_2`]` `(`X_0`)`" dans lequel P est le processus défini comme suit :

process P `[`G_1, G_2`]` `(`X`:`NAT`)` `:` **noexit** `:=`
  G_1 `?`X'`:`NAT `;` `([`$X > 0$`]` `->` P `[`G_1, G_2`]` `(`$X - 1$`)` `>>` G_2 `!`X' `;` **exit**`)`
endproc

La variable X' est utilisée de manière "récursive auto-imbriquée" : sa valeur avant l'appel récursif, reçue sur la porte G_1, doit être restituée au retour, pour être émise sur la porte G_2. ■

4.6.2 Séquentialité des unités

A tout instant, pour chaque unité U, au plus une place propre de U est marquée. Autrement dit le contrôle est séquentiel entre les places propres d'une même unité. Pour tout marquage M accessible, pour toute unité U on a :

$$\mathrm{card}(M \cap \mathit{places}(U)) \leq 1$$

Remarque 4-14
Cette propriété n'est vraie que pour les places *propres* d'une unité. La formule précédente serait en général fausse si l'on remplaçait $\mathit{places}(U)$ par $\mathit{places}^\star(U)$. ■

Il est possible qu'aucune des places propres de U ne soit marquée. Cette situation peut arriver quand le seul rôle d'une unité est de lancer en parallèle l'exécution de sous-unités : *cf.* l'unité racine dans l'exemple 4-1 (p. 68).

Cette propriété est utilisée dans la phase de simulation car elle permet de coder les marquages de manière compacte.

Partie II

Phases de traduction

Chapitre 5

Expansion

Le langage SubLotos est une forme intermédiaire entre Lotos et le modèle réseau : en effet la génération du réseau correspondant à une spécification Lotos est une opération suffisamment complexe pour qu'il soit nécessaire de la décomposer en deux phases successives, expansion et génération. Le rôle de la phase d'expansion est de traduire un programme Lotos — qui vérifie la propriété du contrôle statique — en un programme SubLotos équivalent.

On introduit d'abord informellement les concepts fondamentaux de l'expansion, tâche relativement complexe car plusieurs problèmes s'enchevêtrent ; pour faciliter la compréhension ces problèmes sont abordés séparément, ce qui constitue une simplification parfois abusive. Puis l'algorithme d'expansion est complètement et formellement défini au moyen d'une grammaire attribuée.

Cet algorithme met en œuvre des techniques de compilation très classiques (traduction dirigée par la syntaxe, liaison des identificateurs et gestion de noms uniques, ...) mais il comporte également des aspects originaux. C'est le cas, notamment, du traitement des définitions récursives de processus Lotos ; des approches similaires [QPF88] font, à l'heure actuelle, l'objet de recherches[26].

5.1 Principes de l'expansion

La phase d'expansion remplit plusieurs objectifs :

- elle vise à simplifier les expressions de comportement en remplaçant les constructions complexes de Lotos par des constructions équivalentes, mais plus primitives. C'est ainsi que les opérateurs de comportement "**choice**" et "**par**" sont éliminés
- c'est également le cas des opérateurs "**exit**", "`>>`" et "`|||`". Plus généralement la composition séquentielle est définie en Lotos au moyen d'une porte de terminaison "δ" qui intervient de manière implicite. Grâce à l'expansion, cette porte "δ" perd son statut spécial et devient semblable à toutes les autres portes
- la sémantique de SubLotos est exempte de la notion de renommage de portes (*gate relabelling*) qui est présente en Lotos. Cette caractéristique permet aux programmes SubLotos d'avoir une structure de contrôle entièrement statique dans laquelle les canaux de communication sont constants.

[26] la méthode décrite ici possède le privilège de l'antériorité (la version 1.0 de CÆSAR date d'octobre 1987)

C'est pourquoi l'expansion doit éliminer toutes les portes LOTOS qui dénotent d'autres portes. Ces portes "du 1er ordre" seront appelées ici *portes génériques*. Les portes génériques sont nuisibles parce qu'elles définissent des canaux de communication variables. Elles apparaissent dans trois constructions LOTOS différentes :

– les opérateurs "**choice**" et "**par**" introduisent des portes génériques qui permettent d'itérer sur un ensemble de portes. Ces portes génériques sont éliminées lors de l'expansion de ces opérateurs

– la porte de terminaison "δ" peut-être considérée comme une porte générique. L'expansion des opérateurs "**exit**" et ">>" résoud ce problème

– les portes paramètres formels des processus sont génériques puisqu'elles doivent être instanciées par des portes paramètres effectifs.

  Pour les éliminer on a conçu une transformation dont le principe consiste à remplacer, autant que faire se peut, chaque instanciation par le corps du processus instancié. L'existence de processus LOTOS récursifs ne permettant pas de supprimer de cette manière toutes les instanciations, on utilise alors une technique plus élaborée qui aboutit, en définitive, à ce que, dans toutes les instanciations qui subsistent, les paramètres portes effectifs soient identiques aux portes formelles correspondantes

• en ce qui concerne les données manipulées dans les expressions de comportement, l'expansion assure le passage de LOTOS, langage fonctionnel dont la sémantique est définie par réécriture de termes syntaxiques, à SUBLOTOS, langage impératif dans lequel les valeurs sont décrites par un ensemble fini de variables d'état.

  Mais en aucun cas l'expansion n'évalue les expressions de valeur présentes dans la spécification LOTOS ; toutes les valeurs sont manipulées de manière symbolique. Ce choix est dicté par un souci d'efficacité dans le traitement des valeurs ainsi que par la volonté d'avoir des représentations intermédiaires[27] compactes

Enfin l'expansion doit produire un programme SUBLOTOS conforme aux règles de sémantique statique et vérifiant, notamment, la propriété du nom unique.

5.2 Obtention de noms uniques

L'expansion prend en entrée un programme LOTOS et produit en sortie un programme SUBLOTOS : dans les deux cas, la traduction opère au niveau des arbres syntaxiques décrits par la syntaxe abstraite de LOTOS et de SUBLOTOS et non au niveau textuel.

Le programme LOTOS reçu en entrée possède la propriété du nom unique : chaque objet (§ 3.4.3, p. 53) est désigné par un nom qui lui est propre et deux objets distincts ont des noms distincts. On doit retrouver cette propriété dans le programme SUBLOTOS fourni en sortie.

Or l'algorithme d'expansion provoque la recopie en plusieurs exemplaires de certaines parties du programme LOTOS ; c'est le cas des comportements qui servent d'opérandes à "**choice**" et "**par**", ainsi qu'aux corps des processus lorsque ceux-ci sont remplacés par leur définition.

Si une partie de programme ainsi reproduite contient des occurrences de définition d'objets, la propriété du nom unique risque d'être perdue pour ces objets. C'est pourquoi on doit, pour chaque partie de programme LOTOS, dupliquer (§ 3.4.3, p. 53) les objets qu'elle définit en leur donnant des noms uniques. Les règles pour cela sont les suivantes :

[27] les programmes SUBLOTOS et les réseaux

- chaque fois que l'on rencontre, dans le programme LOTOS, une occurrence de définition d'un objet Y, on crée un nouvel objet SUBLOTOS correspondant $Y_{\bullet i}$
- chaque fois que l'on rencontre, dans le programme LOTOS, une occurrence d'utilisation d'un objet Y figurant dans la portée d'une occurrence de définition pour laquelle on a créé un objet SUBLOTOS $Y_{\bullet i}$, on la traduit dans le programme SUBLOTOS par $Y_{\bullet i}$

Exemple 5-1
En LOTOS l'opérateur “**hide**” constitue une occurrence de définition de portes. Si le comportement LOTOS suivant :

$$\textbf{hide } G_1, G_2 \textbf{ in } G_1 \;;\; G_2 \;;\; \ldots$$

doit être dupliqué plusieurs fois, on donnera à chaque fois aux portes G_1 et G_2 déclarées par “**hide**” des indices différents. Dans cet exemple et dans ceux qui suivront, les indices auront des valeurs initiales remarquables, de la forme $55, 66, 77, \ldots$:

$$\textbf{hide } G_{1\bullet 55}, G_{2\bullet 66} \textbf{ in } G_{1\bullet 55} \;;\; G_{2\bullet 66} \;;\; \ldots$$

puis :

$$\textbf{hide } G_{1\bullet 56}, G_{2\bullet 67} \textbf{ in } G_{1\bullet 56} \;;\; G_{2\bullet 67} \;;\; \ldots$$

et ainsi de suite. ■

5.3 Expansion des opérateurs “choice” et “par”

L'expansion remplace l'opérateur LOTOS “**choice**” — uniquement la forme qui permet d'itérer sur un ensemble de portes — par l'opérateur de choix non-déterministe “`[]`”. De même l'opérateur “**par**” disparaît au profit des opérateurs de composition parallèle.

Exemple 5-2
Les exemples suivants indiquent comment les comportements LOTOS sont développés afin d'éliminer l'opérateur “**choice**”. La transformation est identique pour l'opérateur “**par**”.

Au-dessus de la barre de fraction figure un comportement LOTOS et, en-dessous, le comportement SUBLOTOS correspondant. On suppose que les portes LOTOS G_1, G_2, G_3 et G_4 figurent dans la portée d'une occurrence de définition qui leur associe, respectivement, les portes SUBLOTOS $G_{1\bullet 55}, G_{2\bullet 66}, G_{3\bullet 77}$ et $G_{4\bullet 88}$; par la suite ces correspondances entre objets LOTOS et SUBLOTOS ne seront plus détaillées.

$$\frac{\textbf{choice } G \textbf{ in } [G_1, G_2, G_3] \;[]\; (G \;;\; G \;;\; \textbf{stop})}{\begin{array}{l} G_{1\bullet 55} \;;\; G_{1\bullet 55} \;;\; \textbf{stop} \;[] \\ G_{2\bullet 66} \;;\; G_{2\bullet 66} \;;\; \textbf{stop} \;[] \\ G_{3\bullet 77} \;;\; G_{3\bullet 77} \;;\; \textbf{stop} \end{array}}$$

$$\frac{\textbf{choice } G, G' \textbf{ in } [G_1, G_2] \;[]\; (G \;;\; G' \;;\; \textbf{stop})}{\begin{array}{l} G_{1\bullet 55} \;;\; G_{1\bullet 55} \;;\; \textbf{stop} \;[] \\ G_{1\bullet 55} \;;\; G_{2\bullet 66} \;;\; \textbf{stop} \;[] \\ G_{2\bullet 66} \;;\; G_{1\bullet 55} \;;\; \textbf{stop} \;[] \\ G_{2\bullet 66} \;;\; G_{2\bullet 66} \;;\; \textbf{stop} \end{array}}$$

$$\frac{\textbf{choice}\ G\ \textbf{in}\ [G_1, G_2],\ G'\ \textbf{in}\ [G_3, G_4]\ \texttt{[]}\ (G\ ;\ G'\ ;\ \textbf{stop})}{\begin{array}{l} G_1{\bullet}{\scriptstyle 55}\ ;\ G_3{\bullet}{\scriptstyle 77}\ ;\ \textbf{stop}\ \texttt{[]} \\ G_1{\bullet}{\scriptstyle 55}\ ;\ G_4{\bullet}{\scriptstyle 88}\ ;\ \textbf{stop}\ \texttt{[]} \\ G_2{\bullet}{\scriptstyle 66}\ ;\ G_3{\bullet}{\scriptstyle 77}\ ;\ \textbf{stop}\ \texttt{[]} \\ G_2{\bullet}{\scriptstyle 66}\ ;\ G_4{\bullet}{\scriptstyle 88}\ ;\ \textbf{stop} \end{array}}$$

■

Dans le cas où le comportement dupliqué contient des occurrences de définition d'objets, il faut donner un nom unique aux objets.

Exemple 5-3
Dans cet exemple le comportement dupliqué contient une déclaration de la variable X.

$$\frac{\textbf{par}\ G\ \textbf{in}\ [G_1, G_2]\ \texttt{|||}\ (G\ \texttt{?}X\!:\!S\ ;\ G\ \texttt{!}X\ ;\ \textbf{stop})}{\begin{array}{l} G_1{\bullet}{\scriptstyle 55}\ \texttt{?}X{\bullet}{\scriptstyle 77}\!:\!S\ ;\ G_1{\bullet}{\scriptstyle 55}\ \texttt{!}X{\bullet}{\scriptstyle 77}\ ;\ \textbf{stop}\ \texttt{[]} \\ G_2{\bullet}{\scriptstyle 66}\ \texttt{?}X{\bullet}{\scriptstyle 78}\!:\!S\ ;\ G_2{\bullet}{\scriptstyle 66}\ \texttt{!}X{\bullet}{\scriptstyle 78}\ ;\ \textbf{stop} \end{array}}$$

■

Exemple 5-4
Dans cet exemple le comportement dupliqué contient une déclaration de la porte G'.

$$\frac{\begin{array}{l} \textbf{choice}\ G\ \textbf{in}\ [G_1, G_2]\ \texttt{[]} \\ \quad (\textbf{hide}\ G'\ \textbf{in}\ (G\ ;\ \textbf{stop}\ \texttt{||}\ G'\ ;\ \textbf{stop})) \end{array}}{\begin{array}{l} (\textbf{hide}\ G'{\bullet}{\scriptstyle 77}\ \textbf{in}\ (G_1{\bullet}{\scriptstyle 55}\ ;\ \textbf{stop}\ \texttt{||}\ G'{\bullet}{\scriptstyle 77}\ ;\ \textbf{stop}))\ \texttt{[]} \\ (\textbf{hide}\ G'{\bullet}{\scriptstyle 77}\ \textbf{in}\ (G_2{\bullet}{\scriptstyle 66}\ ;\ \textbf{stop}\ \texttt{||}\ G'{\bullet}{\scriptstyle 77}\ ;\ \textbf{stop})) \end{array}}$$

■

5.4 Expansion de la porte de terminaison

La porte de terminaison "δ" intervient dans la sémantique dynamique de LOTOS de manière implicite : elle n'est pas déclarée par l'utilisateur et elle n'apparaît jamais dans le texte des programmes LOTOS. L'expansion permet de normaliser l'emploi de cette porte "δ" afin qu'elle puisse par la suite être traitée de la même manière que les autres portes.

5.4.1 Expansion de l'opérateur "exit"

L'opérateur de comportement LOTOS "**exit**" est traduit en SUBLOTOS par un rendez-vous sur la porte "δ". En effet, dans la composition séquentielle de LOTOS, un comportement qui s'achève par "**exit**" doit le signaler aux comportements qui attendent pour commencer, après quoi il devient inactif.

Exemple 5-5
On suppose que la porte LOTOS "δ" figure dans le contexte d'un opérateur "`>>`" qui lui fait correspondre la porte SUBLOTOS "$\delta{\bullet}{\scriptstyle 55}$" :

$$\frac{\textbf{exit}}{\delta{\bullet}{\scriptstyle 55}\ ;\ \textbf{stop}}$$

■

Si le comportement "**exit**" possède des résultats, l'opérateur de préfixage obtenu après expansion comporte des offres correspondantes. Pour traduire la clause "**any**" on crée des duplications de la pseudo-variable LOTOS "ξ" ; on obtient ainsi des variables SUBLOTOS dont la portée est restreinte au comportement "**stop**".

Exemple 5-6

$$\frac{\textbf{exit}\ (V_1, V_2)}{\delta_{\bullet 55}\ \ !V_1\ \ !V_2\ \ ;\ \textbf{stop}}$$

$$\frac{\textbf{exit}\ (\textbf{any}\ S_1, \textbf{any}\ S_2)}{\delta_{\bullet 55}\ \ ?\xi_{\bullet 66}\!:\!S_1\ \ ?\xi_{\bullet 67}\!:\!S_2\ \ ;\ \textbf{stop}}$$

■

5.4.2 Expansion de l'opérateur ">>"

Pour traduire en SUBLOTOS le comportement "B_1 >> B_2" il faut créer une nouvelle porte de terminaison, qui est une duplication de "δ". Les comportements B_1 et B_2 seront composés en parallèle et synchronisés uniquement sur cette porte. Tous les opérateurs "**exit**" figurant dans B_1 seront traduits par des rendez-vous sur cette porte. Il faut préfixer B_2 par un rendez-vous sur cette porte. Enfin cette porte de terminaison doit être cachée par un opérateur "**hide**" afin de la rendre inapte à toute synchronisation extérieure.

Exemple 5-7

$$\frac{\ldots\ \textbf{exit} >> \ \ldots}{\begin{array}{l}\textbf{hide}\ \delta_{\bullet 55}\ \textbf{in}\\ \quad (\ldots\ \delta_{\bullet 55}\ ;\ \textbf{stop}\ |[\delta_{\bullet 55}]|\ \delta_{\bullet 55}\ ;\ \ldots)\end{array}}$$

$$\frac{(\textbf{exit} >> \textbf{exit}) >> \textbf{exit}}{\begin{array}{l}\textbf{hide}\ \delta_{\bullet 56}\ \textbf{in}\\ \quad (\textbf{hide}\ \delta_{\bullet 57}\ \textbf{in}\\ \quad\quad (\delta_{\bullet 57}\ ;\ \textbf{stop}\ |[\delta_{\bullet 57}]|\ \delta_{\bullet 57}\ ;\ \delta_{\bullet 56}\ ;\ \textbf{stop})\ |[\delta_{\bullet 56}]|\ \delta_{\bullet 56}\ ;\ \delta_{\bullet 55}\ ;\ \textbf{stop})\end{array}}$$

$$\frac{\textbf{exit} >> (\textbf{exit} >> \textbf{exit})}{\begin{array}{l}\textbf{hide}\ \delta_{\bullet 56}\ \textbf{in}\ (\delta_{\bullet 56}\ ;\ \textbf{stop}\ |[\delta_{\bullet 56}]|\ \delta_{\bullet 56}\ ;\\ \quad (\textbf{hide}\ \delta_{\bullet 57}\ \textbf{in}\ (\delta_{\bullet 57}\ ;\ \textbf{stop}\ |[\delta_{\bullet 57}]|\ \delta_{\bullet 57}\ ;\ \delta_{\bullet 55}\ ;\ \textbf{stop})))\end{array}}$$

■

Si l'opérateur ">>" possède une clause "**accept**", les variables qu'elle définit sont converties en offres de rendez-vous.

Exemple 5-8

$$\frac{\ldots\ \textbf{exit}\ (V,\ \textbf{any}\ S) >> \textbf{accept}\ X_1, X_2\!:\!S\ \textbf{in}\ \ldots}{\begin{array}{l}\textbf{hide}\ \delta_{\bullet 55}\ \textbf{in}\\ \quad (\ldots\ \delta_{\bullet 55}\ !V\ ?\xi_{\bullet 66}\!:\!S\ ;\ \textbf{stop}\ |[\delta_{\bullet 55}]|\ \delta_{\bullet 55}\ ?X_{1\bullet 77}\!:\!S\ ?X_{2\bullet 88}\!:\!S\ ;\ \ldots)\end{array}}$$

■

5.4.3 Expansion de l'opérateur "`[>`"

L'opérateur LOTOS d'interruption, noté "`[>`", cesse d'être effectif dès que son opérande gauche a effectué une transition "δ". Lorsqu'on traduit cet opérateur en SUBLOTOS on doit indiquer sur quelle duplication de la porte "δ" prend fin l'effet d'interruption.

Exemple 5-9

$$\frac{\texttt{(...\ [>\ ...)\ >>\ ...}}{\begin{array}{l}\textbf{hide}\ \delta\bullet 55\ \textbf{in}\\ \quad \texttt{((...\ [}\delta\bullet 55\texttt{>\ ...)\ |[}\delta\bullet 55\texttt{]|\ }\delta\bullet 55\texttt{\ ;\ ...)}\end{array}}$$

■

5.4.4 Expansion des opérateurs "`||`", "`|||`" et "`|[...]|`"

Les opérateurs LOTOS de composition parallèle effectuent une synchronisation implicite sur la porte de terminaison "δ". En SUBLOTOS on doit préciser sur quelle duplication de la porte "δ" a lieu la synchronisation. Pour cela il faut ajouter cette porte à la liste des portes de l'opérateur parallèle.

Exemple 5-10

$$\frac{\texttt{...\ |[}G_0,\ldots\ G_n\texttt{]|\ ...}}{\texttt{...\ |[}\delta\bullet 55, G_0\bullet 66,\ldots\ G_n\bullet 77\texttt{]|\ ...}}$$

$$\frac{\texttt{...\ |||\ ...}}{\texttt{...\ |[}\delta\bullet 55\texttt{]|\ ...}}$$

$$\frac{\texttt{...\ ||\ ...}}{\texttt{...\ ||\ ...}}$$

■

On peut constater que cette transformation élimine l'opérateur "`|||`" qui n'existe donc plus en SUBLOTOS.

5.4.5 Expansion des listes de paramètres portes

En LOTOS chaque processus est implicitement paramétré par une porte de terminaison. En SUBLOTOS cette porte doit être mentionnée explicitement : c'est pourquoi il faut ajouter, à chaque définition de processus et à chaque instanciation de processus, un paramètre porte supplémentaire qui est une duplication de la porte de terminaison "δ".

5.5 Expansion des processus

5.5.1 Développement des processus non récursifs

L'expansion *développe* les processus, c'est-à-dire que chaque instanciation d'un processus P est remplacée par le corps de P (*inline expansion*). Bien entendu cette transformation n'est possible que pour les processus P non récursifs.

Exemple 5-11

$$
\begin{array}{l}
\quad G_1 \ ; \ P \ [\,] \ G_2 \ ; \ P \\
\textbf{where} \\
\quad \textbf{process } P \ : \ \textbf{exit} \ (S) \ := \\
\qquad \textbf{exit} \ (F) \\
\quad \textbf{endproc} \\
\hline
G_1 \ ; \ \delta_{\bullet 55} \ !F \ ; \ \textbf{stop} \ [\,] \ G_2 \ ; \ \delta_{\bullet 55} \ !F \ ; \ \textbf{stop}
\end{array}
$$

■

Lorsque le processus développé possède des portes formelles, on leur substitue les portes paramètres effectifs fournies par l'instanciation. Cette transformation permet d'éliminer les portes formelles qui sont, par définition, génériques.

Exemple 5-12

$$
\begin{array}{l}
\quad G_1 \ ; \ P \ [G_1, G_2] \ [\,] \ G_2 \ ; \ \textbf{stop} \\
\textbf{where} \\
\quad \textbf{process } P \ [G, G'] \ : \ \textbf{noexit} \ := \\
\qquad G \ ; \ G' \ ; \ \textbf{stop} \\
\quad \textbf{endproc} \\
\hline
G_{1\bullet 55} \ ; \ G_{1\bullet 55} \ ; \ G_{2\bullet 66} \ ; \ \textbf{stop} \ [\,] \ G_{2\bullet 66} \ ; \ \textbf{stop}
\end{array}
$$

$$
\begin{array}{l}
\quad P \ [G_1] \ [\,] \ P \ [G_2] \\
\textbf{where} \\
\quad \textbf{process } P \ [G] \ : \ \textbf{noexit} \ := \\
\qquad G \ ; \ (\textbf{hide } G' \textbf{ in } G' \ ; \ \textbf{stop}) \\
\quad \textbf{endproc} \\
\hline
G_{1\bullet 55} \ ; \ (\textbf{hide } G'_{\bullet 77} \textbf{ in } G'_{\bullet 77} \ ; \ \textbf{stop}) \ [\,] \\
G_{2\bullet 66} \ ; \ (\textbf{hide } G'_{\bullet 78} \textbf{ in } G'_{\bullet 78} \ ; \ \textbf{stop})
\end{array}
$$

$$
\begin{array}{l}
\quad P \ [G] \ \texttt{>>} \ \textbf{accept } X\!:\!S \textbf{ in } G \ !X \ ; \ \textbf{exit} \\
\textbf{where} \\
\quad \textbf{process } P \ [G'] \ : \ \textbf{exit} \ (S) \ := \\
\qquad G' \ ?X'\!:\!S \ ; \ \textbf{exit} \ (X') \\
\quad \textbf{endproc} \\
\hline
\textbf{hide } \delta_{\bullet 56} \textbf{ in} \\
\quad (G_{\bullet 66} \ ?X'_{\bullet 77}\!:\!S \ ; \ \delta_{\bullet 56} \ !X'_{\bullet 77} \ ; \ \textbf{stop} \ |[\delta_{\bullet 56}]| \\
\quad \delta_{\bullet 56} \ ?X_{\bullet 88}\!:\!S \ ; \ G_{\bullet 66} \ !X_{\bullet 88} \ ; \ \delta_{\bullet 55} \ ; \ \textbf{stop})
\end{array}
$$

■

Remarque 5-1
Le remplacement des portes formelles par les portes effectives est conforme à la sémantique de LOTOS quand les paramètres effectifs sont deux à deux distincts. Mais, dans le cas contraire, cette substitution conduit à une interprétation différente, puisqu'elle implémente un *pré-renommage* au lieu du *post-renommage*. Par exemple, le comportement LOTOS suivant :

$$
\begin{array}{l}
\quad P \ [G, G] \\
\textbf{where}
\end{array}
$$

$$\begin{array}{l}\textbf{process}\ P\ \texttt{[}G_1, G_2\texttt{]}\ \texttt{:}\ \textbf{noexit}\ \texttt{:=}\\ \quad G_1\ \texttt{;}\ \textbf{stop}\ \texttt{||}\ G_2\ \texttt{;}\ \textbf{stop}\\ \textbf{endproc}\end{array}$$

n'est pas équivalent à "G **; stop**" mais à "**stop**". Pour le montrer il suffit d'appliquer les règles de sémantique dynamique de LOTOS :

$$\begin{array}{ll} & P\ \texttt{[}G, G\texttt{]}\\ \longrightarrow & \textbf{rename}\ \mathit{gate}_1(P)\texttt{:=}G, \mathit{gate}_2(P)\texttt{:=}G\ \textbf{in}\ \mathit{behaviour}(P)\\ \longrightarrow & \textbf{rename}\ G_1\texttt{:=}G, G_2\texttt{:=}G\ \textbf{in}\ (G_1\ \texttt{;}\ \textbf{stop}\ \texttt{||}\ G_2\ \texttt{;}\ \textbf{stop})\\ \longrightarrow & \textbf{rename}\ G_1\texttt{:=}G, G_2\texttt{:=}G\ \textbf{in stop}\\ \longrightarrow & \textbf{stop}\end{array}$$

On ne peut donc pas toujours remplacer une instanciation de processus par le corps du processus : la sémantique de LOTOS ne garantit pas le "principe de substitution". Ce problème se pose uniquement lorsque l'on instancie deux portes formelles G_1 et G_2 avec la même porte G. Formellement cette situation se produit lorsqu'un renommage de portes n'est pas une application injective.

On retrouve un problème classique des langages de programmation, celui de la synonymie (*aliasing*) entre paramètres effectifs d'une procédure lors d'un passage par référence. Pour LOTOS la sémantique du pré-renommage aussi bien que celle du post-renommage présentent des inconvénients : la première complique la génération de code pour un processus, parce qu'il faut prévoir tous les cas possibles de synonymies ; la seconde est contraire à l'intuition et crée des blocages difficilement décelables. La meilleure solution serait de considérer comme illégaux les programmes qui possèdent des synonymies de portes ; pour les autres programmes, les définitions avec pré-renommage ou post-renommage sont équivalentes.

Pour des raisons "historiques", CÆSAR adopte la sémantique du pré-renommage, mais détecte les instanciations de processus ayant des portes synonymes. Tous les risques d'erreur et de déviation par rapport à la définition standard de LOTOS sont ainsi signalés à l'utilisateur. ■

Lorsque le processus développé possède des variables formelles, on ne les remplace pas par les valeurs qui figurent comme paramètres effectifs de l'instanciation : une telle substitution serait source d'inefficacité lorsque les variables formelles sont utilisées fréquemment et que les paramètres effectifs correspondants sont des expressions complexes. On utilise au contraire un opérateur "**let**" dont le rôle est d'affecter les valeurs des paramètres effectifs aux variables formelles.

Exemple 5-13

$$\frac{\begin{array}{l}\quad P\ \texttt{[}G_0\texttt{]}\ \texttt{(}V_1, V_2\texttt{)}\\ \textbf{where}\\ \quad \textbf{process}\ P\ \texttt{[}G\texttt{]}\ \texttt{(}X_1\texttt{:}S_1, X_2\texttt{:}S_2\texttt{)}\ \texttt{:}\ \textbf{noexit}\ \texttt{:=}\\ \quad\quad G\ \texttt{!}X_1\ \texttt{!}X_2\ \texttt{!}X_2\ \texttt{;}\ \textbf{stop}\\ \quad \textbf{endproc}\end{array}}{\textbf{let}\ X_1{\bullet}66\texttt{:}S_1\texttt{=}V_1, X_2{\bullet}77\texttt{:}S_2\texttt{=}V_2\ \textbf{in}\ G_0{\bullet}55\ \texttt{!}X_1{\bullet}66\ \texttt{!}X_2{\bullet}77\ \texttt{!}X_2{\bullet}77\ \texttt{;}\ \textbf{stop}}$$

$$\frac{\begin{array}{l} \quad P\ [G_1]\ (V_1)\ []\ P\ [G_2]\ (V_2) \\ \textbf{where} \\ \quad \textbf{process}\ P\ [G]\ (X{:}S)\ :\ \textbf{noexit}\ := \\ \qquad [F\ (X)]\ \text{->}\ G\ !X\ ;\ \textbf{stop} \\ \quad \textbf{endproc} \end{array}}{\begin{array}{l} \textbf{let}\ X{\bullet}77{:}S{=}V_1\ \textbf{in}\ [F\ (X{\bullet}77)]\ \text{->}\ G_1{\bullet}55\ !X{\bullet}77\ ;\ \textbf{stop}\ [] \\ \textbf{let}\ X{\bullet}78{:}S{=}V_2\ \textbf{in}\ [F\ (X{\bullet}78)]\ \text{->}\ G_2{\bullet}66\ !X{\bullet}78\ ;\ \textbf{stop} \end{array}}$$

■

5.5.2 Duplication des processus récursifs

Lorsqu'un processus est récursif on ne peut évidemment pas développer à l'infini ses instanciations. C'est pourquoi on choisit de conserver en SubLotos les notions de processus et d'instanciation, mais uniquement pour les processus récursifs, les autres processus étant développés.

Exemple 5-14

$$\frac{\begin{array}{l} \quad P\ [G_0] \\ \textbf{where} \\ \quad \textbf{process}\ P\ [G]\ :\ \textbf{noexit}\ := \\ \qquad G\ ;\ P\ [G] \\ \quad \textbf{endproc} \end{array}}{\begin{array}{l} \quad P{\bullet}55\ [\delta{\bullet}66, G_0{\bullet}77] \\ \textbf{where} \\ \quad \textbf{process}\ P{\bullet}55\ [\delta{\bullet}66, G_0{\bullet}77]\ := \\ \qquad G_0{\bullet}77\ ;\ P{\bullet}55\ [\delta{\bullet}66, G_0{\bullet}77] \\ \quad \textbf{endproc} \end{array}}$$

■

On veut néanmoins, y compris dans le cas des processus récursifs, supprimer les portes génériques. Cette élimination est obtenue en imposant que, pour chaque instanciation d'un processus SubLotos, les portes paramètres effectifs fournies par l'instanciation soient égales aux portes formelles du processus.

Pour cela on est conduit à dupliquer plusieurs fois un même processus Lotos récursif ; un processus Lotos sera dupliqué en autant de processus SubLotos qu'il existe d'instanciations avec des portes effectives différentes.

Exemple 5-15

$$\frac{\begin{array}{l} \quad P\ \texttt{[}G_1\texttt{]}\ \texttt{[]}\ P\ \texttt{[}G_2\texttt{]} \\ \textbf{where} \\ \quad \textbf{process}\ P\ \texttt{[}G\texttt{]}\ :\ \textbf{exit}\ := \\ \qquad G\ ;\ P\ \texttt{[}G\texttt{]}\ \texttt{[]}\ \textbf{exit} \\ \quad \textbf{endproc} \end{array}}{\begin{array}{l} \quad P{\bullet}55\ \texttt{[}\delta{\bullet}66, G_1{\bullet}77\texttt{]}\ \texttt{[]}\ P{\bullet}56\ \texttt{[}\delta{\bullet}66, G_2{\bullet}88\texttt{]} \\ \textbf{where} \\ \quad \textbf{process}\ P{\bullet}55\ \texttt{[}\delta{\bullet}66, G_1{\bullet}77\texttt{]}\ := \\ \qquad G_1{\bullet}77\ ;\ P{\bullet}55\ \texttt{[}\delta{\bullet}66, G_1{\bullet}77\texttt{]}\ \texttt{[]}\ \delta{\bullet}66\ ;\ \textbf{stop} \\ \quad \textbf{endproc} \\ \quad \textbf{process}\ P{\bullet}56\ \texttt{[}\delta{\bullet}66, G_2{\bullet}88\texttt{]}\ := \\ \qquad G_2{\bullet}88\ ;\ P{\bullet}56\ \texttt{[}\delta{\bullet}66, G_2{\bullet}88\texttt{]}\ \texttt{[]}\ \delta{\bullet}66\ ;\ \textbf{stop} \\ \quad \textbf{endproc} \end{array}}$$

■

Pour que les portes effectives soient égales aux portes formelles il peut s'avérer nécessaire de développer partiellement un processus récursif.

Exemple 5-16

$$\frac{\begin{array}{l} \quad P\ \texttt{[}G_1, G_2\texttt{]} \\ \textbf{where} \\ \quad \textbf{process}\ P\ \texttt{[}G'_1, G'_2\texttt{]}\ :\ \textbf{noexit}\ := \\ \qquad G'_1\ ;\ P\ \texttt{[}G'_2, G'_1\texttt{]} \\ \quad \textbf{endproc} \end{array}}{\begin{array}{l} \quad P{\bullet}55\ \texttt{[}\delta{\bullet}66, G_1{\bullet}77, G_2{\bullet}88\texttt{]} \\ \textbf{where} \\ \quad \textbf{process}\ P{\bullet}55\ \texttt{[}\delta{\bullet}66, G_1{\bullet}77, G_2{\bullet}88\texttt{]}\ := \\ \qquad G_1{\bullet}77\ ;\ G_2{\bullet}88\ ;\ P{\bullet}55\ \texttt{[}\delta{\bullet}66, G_1{\bullet}77, G_2{\bullet}88\texttt{]} \\ \quad \textbf{endproc} \end{array}}$$

■

A un instant donné, on appelle *trace* l'ensemble des processus dont on a commencé, sans l'avoir encore achevée, l'expansion du corps ; il s'agit d'une pile.

En première approximation, le développement de la récursion doit s'arrêter lorsque l'on retrouve une instanciation avec des portes effectives identiques aux portes formelles d'un processus déjà rencontré. Plus précisément, lorsqu'on rencontre une instanciation d'un processus LOTOS P ayant comme paramètres effectifs les portes $\delta{\bullet}i_0, G_1{\bullet}i_1, \ldots\ G_m{\bullet}i_m$, on ne la développe pas s'il existe dans la trace un processus SUBLOTOS qui est une duplication de P et dont les portes formelles sont $\delta{\bullet}i_0, G_1{\bullet}i_1, \ldots\ G_m{\bullet}i_m$.

Cependant cette condition d'arrêt n'est pas satisfaisante car elle peut, dans certains cas, provoquer un développement illimité, comme le montre l'exemple suivant où il y a création d'un nombre infini de portes et de processus SUBLOTOS.

Exemple 5-17

$$\frac{\begin{array}{l} P\ [G_0] \\ \textbf{where} \\ \quad \textbf{process}\ P\ [G]\ :\ \textbf{noexit}\ := \\ \quad\quad G\ ;\ (\textbf{hide}\ G'\ \textbf{in}\ G'\ ;\ P\ [G]) \\ \quad \textbf{endproc} \end{array}}{\begin{array}{l} \quad P_{\bullet 55}\ [\delta_{\bullet 66}, G_0{}_{\bullet 77}] \\ \textbf{where} \\ \quad \textbf{process}\ P_{\bullet 55}\ [\delta_{\bullet 66}, G_0{}_{\bullet 77}]\ := \\ \quad\quad G_0{}_{\bullet 77}\ ;\ (\textbf{hide}\ G'_{\bullet 88}\ \textbf{in}\ G'_{\bullet 88}\ ;\ P_{\bullet 56}\ [\delta_{\bullet 66}, G'_{\bullet 88}]) \\ \quad \textbf{endproc} \\ \quad \textbf{process}\ P_{\bullet 56}\ [\delta_{\bullet 66}, G'_{\bullet 88}]\ := \\ \quad\quad G'_{\bullet 88}\ ;\ (\textbf{hide}\ G'_{\bullet 89}\ \textbf{in}\ G'_{\bullet 89}\ ;\ P_{\bullet 57}\ [\delta_{\bullet 66}, G'_{\bullet 89}]) \\ \quad \textbf{endproc} \\ \quad \textbf{process}\ P_{\bullet 57}\ [\delta_{\bullet 66}, G'_{\bullet 89}]\ := \\ \quad\quad G'_{\bullet 89}\ ;\ (\textbf{hide}\ G'_{\bullet 90}\ \textbf{in}\ G'_{\bullet 90}\ ;\ P_{\bullet 58}\ [\delta_{\bullet 66}, G'_{\bullet 90}]) \\ \quad \textbf{endproc} \\ \quad \ldots \end{array}}$$

■

Pour éviter cette situation, la condition d'arrêt doit être affinée de la manière suivante : lorsqu'on rencontre une instanciation d'un processus Lotos P ayant comme paramètres effectifs les portes $\delta_{\bullet i_0}, G_1{}_{\bullet i_1}, \ldots\ G_m{}_{\bullet i_m}$, on ne la développe pas s'il existe dans la trace un processus SubLotos qui est une duplication de P, notée $P_{\bullet k}$, et dont les portes formelles sont $\delta_{\bullet j_0}, G_1{}_{\bullet j_1}, \ldots\ G_m{}_{\bullet j_m}$. On n'exige plus que $(i_0 = j_0)\ \wedge\ \ldots\ (i_m = j_m)$; autrement dit on ne compare que les origines (§ 3.4.3, p. 53) des portes, sans considérer les indices (cette comparaison est inutile pour le premier paramètre puisqu'il s'agit toujours d'une duplication de la porte "δ").

Dans le programme SubLotos produit, l'instanciation du processus P est traduite par l'instanciation "$P_{\bullet k}\ [\delta_{\bullet j_0}, G_1{}_{\bullet j_1}, \ldots\ G_m{}_{\bullet j_m}]$" et non par une instanciation de la forme "$P_{\bullet l}\ [\delta_{\bullet i_0}, G_1{}_{\bullet i_1}, \ldots\ G_m{}_{\bullet i_m}]$". Ainsi on retrouve une instanciation connue, on ne crée pas de nouveau processus ni de nouvelle porte, ce qui évite la récursion à l'infini. Les paramètres effectifs sont alors égaux aux portes formelles.

Exemple 5-18

$$\frac{\begin{array}{l} P\ [G_0] \\ \textbf{where} \\ \quad \textbf{process}\ P\ [G]\ :\ \textbf{noexit}\ := \\ \quad\quad G\ ;\ (\textbf{hide}\ G'\ \textbf{in}\ G'\ ;\ P\ [G]) \\ \quad \textbf{endproc} \end{array}}{\begin{array}{l} \quad P_{\bullet 55}\ [\delta_{\bullet 66}, G_0{}_{\bullet 77}] \\ \textbf{where} \\ \quad \textbf{process}\ P_{\bullet 55}\ [\delta_{\bullet 66}, G_0{}_{\bullet 77}]\ := \\ \quad\quad G_0{}_{\bullet 77}\ ;\ (\textbf{hide}\ G'_{\bullet 88}\ \textbf{in}\ G'_{\bullet 88}\ ;\ P_{\bullet 56}\ [\delta_{\bullet 66}, G'_{\bullet 88}]) \\ \quad \textbf{endproc} \\ \quad \textbf{process}\ P_{\bullet 56}\ [\delta_{\bullet 66}, G'_{\bullet 88}]\ := \\ \quad\quad G'_{\bullet 88}\ ;\ (\textbf{hide}\ G'_{\bullet 89}\ \textbf{in}\ G'_{\bullet 89}\ ;\ P_{\bullet 56}\ [\delta_{\bullet 66}, G'_{\bullet 88}]) \\ \quad \textbf{endproc} \end{array}}$$

■

Il faut démontrer la terminaison et la correction de cette transformation. La preuve de terminaison est immédiate :

- pour avoir un développement illimité de la récursion, il faudrait qu'il existe un processus LOTOS récursif, possédant m portes formelles et ayant une infinité d'instanciations qui, comparées deux à deux, ne vérifient pas la condition d'arrêt
- il existerait donc une infinité de $(m + 1)$-uplets distincts de portes SUBLOTOS $(\delta{\bullet}i_0, G_1{\bullet}i_1, \ldots\ G_m{\bullet}i_m)$ qui constituent les paramètres effectifs de ces instanciations
- par définition de la condition d'arrêt et en considérant les origines de ces portes SUBLOTOS, il existerait donc une infinité de m-uplets de portes LOTOS $(G_1, \ldots\ G_m)$ deux à deux distincts
- or cette situation est impossible puisque le nombre de portes d'une spécification LOTOS est fini

On doit prouver ensuite que cette transformation est compatible avec la sémantique de LOTOS lorsque les programmes LOTOS considérés vérifient la propriété du contrôle statique (§ 3.1.5, p. 47) :

- supposons qu'il existe un processus LOTOS P dont on rencontre une instanciation ayant comme paramètres effectifs les portes $\delta{\bullet}i_0, G_1{\bullet}i_1, \ldots\ G_m{\bullet}i_m$ sachant qu'il existe déjà une duplication de P dont les portes formelles sont $\delta{\bullet}j_0, G_1{\bullet}j_1, \ldots\ G_m{\bullet}j_m$, avec $(i_0 \neq j_0)\ \vee\ \ldots\ (i_m \neq j_m)$
- montrons que $i_0 = j_0$. Dans le cas contraire, cela signifierait que le corps du processus P duplique la porte de terminaison "δ". Or seul l'opérateur ">>" permet de créer une nouvelle duplication de cette porte, duplication visible uniquement dans son opérande gauche. Il existerait donc une instanciation récursive de P en partie gauche de l'opérateur ">>", ce qui contredit l'hypothèse du contrôle statique
- supposons que $i_1 \neq j_1$ et montrons que la sémantique de LOTOS est respectée, c'est-à-dire que le remplacement de $G_1{\bullet}i_1$ par $G_1{\bullet}j_1$ dans l'instanciation récursive de P ne modifie pas le graphe obtenu par application des règles de sémantique statique.
  - on a forcément $i_1 \neq 0$ et $j_1 \neq 0$. S'il existait en effet une duplication $G_1{\bullet}0$ de la porte LOTOS G_1, cela signifierait que G_1 est égale, soit à "δ", soit à une porte formelle de la spécification LOTOS. Or G_1 ne peut pas être égale à "δ", puisqu'elle sert de paramètre effectif à un processus LOTOS, ni à une porte formelle de la spécification puisque ces portes n'admettent qu'une seule duplication et que la porte G_1 en possède deux : $G_1{\bullet}i_1$ et $G_1{\bullet}j_1$. Par conséquent les portes SUBLOTOS $G_1{\bullet}i_1$ et $G_1{\bullet}j_1$ sont cachées et les rendez-vous sur ces deux portes produisent dans le graphe des arcs étiquetés par le label "**i**", donc indiscernables les uns des autres
  - si le fait de substituer $G_1{\bullet}i_1$ par $G_1{\bullet}j_1$ change le graphe, cela signifie que la spécification contient un test d'égalité entre $G_1{\bullet}i_1$ et $G_1{\bullet}j_1$. Or le seul moyen en LOTOS comme en SUBLOTOS de comparer deux portes, c'est de les synchroniser : si elles sont identiques, le rendez-vous est autorisé, sinon il y a un blocage. L'instanciation de P considérée est donc nécessairement composée en parallèle avec un autre comportement, au moyen d'un opérateur parallèle, noté *op*, qui effectue la synchronisation, soit sur la porte $G_1{\bullet}i_1$, soit sur la porte $G_1{\bullet}j_1$. Dans le premier cas, le fait de remplacer le paramètre effectif $G_1{\bullet}i_1$ par $G_1{\bullet}j_1$ provoque un blocage qui n'existait pas, alors que dans le second cas, il permet un rendez-vous qui n'était pas possible auparavant
  - la porte G_1 est dupliquée entre deux instanciations successives du processus P dont elle est paramètre effectif. Elle est donc définie, soit dans le corps du processus P, soit dans le corps d'un autre processus qui est mutuellement récursif avec P.

Remarque 5-2
On peut même ajouter que la porte G_1 est déclarée par un opérateur "**hide**". Cet opérateur constitue en effet la seule construction LOTOS permettant de créer des duplications d'une porte cachée. On a donc une récursion à travers l'opérateur "**hide**", mais cette situation n'est pas interdite par la propriété du contrôle statique ■

– puisque l'opérateur de composition parallèle *op* impose la synchronisation sur une duplication de la porte G_1, il est nécessairement situé dans la portée de G_1, c'est-à-dire sur un chemin de récursion du processus P. Cette situation correspond très exactement à la présence dans la spécification LOTOS d'une récursion à travers un opérateur parallèle, ce qui est impossible puisque contraire à l'hypothèse du contrôle statique

- en se ramenant au cas précédent, on montrerait de la même manière que la sémantique de LOTOS est conservée lorsque $i_2 \neq j_2$ ou ... $i_m \neq j_m$

Enfin lorsqu'une instanciation possède des paramètres effectifs, leurs valeurs sont, soit affectées aux variables formelles correspondantes au moyen d'un opérateur "**let**" lorsque le corps du processus doit être développé, soit passées en paramètres dans le cas contraire.

Exemple 5-19

P [G_0] (0) [] P' [G_0] (0)
where
  process P [G] (X:S) :=
    P [G] ($X+1$) [] P' [G] ($X+1$)
  endproc
  process P' [G'] (X':S) :=
    G' !X' ; **stop**
  endproc

$P_{\bullet 55}$ [$\delta_{\bullet 66}, G_{0\bullet 77}$] (0) [] **let** $X'_{\bullet 88}$:S=0 **in** $G_{0\bullet 77}$!$X'_{\bullet 88}$; **stop**
where
  process $P_{\bullet 55}$ [$\delta_{\bullet 66}, G_{0\bullet 77}$] ($X_{\bullet 99}$:$S$) :=
    $P_{\bullet 55}$ [$\delta_{\bullet 66}, G_{0\bullet 77}$] ($X_{\bullet 99}+1$) [] **let** $X'_{\bullet 89}$:S=$X_{\bullet 99}+1$ **in** $G_{0\bullet 77}$!$X'_{\bullet 89}$; **stop**
  endproc

■

Pendant l'expansion les expressions de valeur qui figurent dans le programme LOTOS sont manipulées symboliquement, sans les évaluer. En particulier, contrairement à ce qui est fait pour les paramètres portes, on ne développe pas la récursion sur les paramètres variables.

Exemple 5-20
C'est ainsi que le comportement LOTOS suivant :

P [G] (*false*)
where
  process P [G] (X:$BOOL$) : **noexit** :=
    G !X ; P [G] (*not* (X))
  endproc

a pour traduction en SubLotos :

```
P•55 [δ•66, G•77] (false)
where
   process P•55 [δ•66, G•77] (X•88:BOOL) :=
      G•77 !X•88 ; P•55 [δ•66, G•77] (not (X•88))
   endproc
```

et non pas :

```
P•55 [δ•66, G•77]
where
   process P•55 [δ•66, G•77] :=
      G•77 !false ; G•77 !true ; P•55 [δ•66, G•77]
   endproc
```

car il faudrait pour cela être capable de décider que :

$$not\ (not\ (false)) = true$$

■

Remarque 5-3
Au cours de l'expansion des processus les définitions des processus Lotos inaccessibles ne sont ni examinées ni traduites en SubLotos ; toutefois leur existence a été auparavant détectée et signalée à l'utilisateur : *cf.* remarque 3-2 (p. 50). ■

5.6 Algorithme d'expansion

L'expansion se fait par une exploration en profondeur de l'arbre abstrait Lotos ; elle porte principalement sur les expressions de comportement. L'arbre abstrait SubLotos correspondant est construit de manière ascendante, au fur et à mesure de cette exploration.

Les règles d'expansion sont formulées par récurrence sur la complexité des constructions Lotos à traduire en SubLotos. Elles sont décrites au moyen de grammaires attribuées. Auparavant les attributs et les notations utilisés par l'algorithme d'expansion sont présentés et leur définition est accompagnée d'explications sur leur signification intuitive.

Dans toute la suite de ce chapitre, les symboles non-terminaux de Lotos (§ 2.1.1, p. 25) sont désignés par des lettres minuscules $(g, x, p, b, \ldots)$ alors que les symboles non-terminaux de SubLotos (§ 3.4.1, p. 53) le sont par des lettres majuscules $(G, X, P, B, \ldots)$.

5.6.1 Renommage des portes

On appelle *renommage des portes* une application partielle $\mathcal{G}$ de l'ensemble des portes Lotos vers l'ensemble des portes SubLotos. Pour manipuler les renommages on utilise les notations relatives aux applications partielles ainsi que les définitions suivantes :

- on note "$new\_gate(g)$" une nouvelle duplication de la porte Lotos g : il s'agit d'une porte SubLotos G qui est différente de toutes les portes SubLotos créées auparavant et qui vérifie $origin(G) = g$ (G est de la forme $g_{\bullet}i$)

- on note "$new\_gate(\widehat{g})$", où $\widehat{g}$ est une liste de portes LOTOS, une liste de portes SUBLOTOS définies comme suit :

$$new\_gate(\widehat{g}) = new\_gate(g_0), \ldots\ new\_gate(g_n)$$

Le concept de renommage des portes intervient dans l'algorithme d'expansion de la manière suivante :

- la traduction d'un comportement LOTOS b vers un comportement SUBLOTOS B s'effectue dans le contexte d'un renommage de portes $\mathcal{G}$ qui est reçu comme un attribut hérité
- chaque fois qu'on rencontre dans b une occurrence d'utilisation d'une porte g on lui fait généralement correspondre dans B une occurrence d'utilisation de la porte $\mathcal{G}(g)$
- chaque fois qu'on rencontre dans b une occurrence de définition d'une porte g :
  - on crée une nouvelle duplication G de g (on prend $G := new\_gate(g)$)
  - on fabrique un nouveau renommage $\mathcal{G}_0$ qui est identique à $\mathcal{G}$ sauf en la porte g pour laquelle il renvoie G (on prend $\mathcal{G}_0 \ := \ \mathcal{G} \oslash (g \rightsquigarrow G)$)
  - si b_0 est un sous-comportement de b dans lequel la porte g est visible, on effectue la traduction de b_0 dans le contexte du renommage $\mathcal{G}_0$
- l'expansion des opérateurs "**choice**" et "**par**" s'effectue aussi grâce au renommage des portes, mais sans créer de nouvelle duplication

5.6.2 Renommage des variables

On appelle *renommage des variables* une application partielle de l'ensemble des variables LOTOS vers l'ensemble des variables SUBLOTOS. La signification intuitive des renommages de variables est analogue à celles des renommages de portes. Il en est de même des notations :

- on note "$new\_var(x : s)$" une nouvelle duplication de la variable LOTOS x : il s'agit d'une variable SUBLOTOS X, de sorte s, qui est différente de toutes les variables SUBLOTOS créées auparavant et qui vérifie $origin(X) = x$ (X est de la forme $x{\bullet}i$)
- on note "$new\_var(\widehat{x} : s)$", où $\widehat{x}$ est une liste de variables LOTOS, une liste de variables SUBLOTOS définie comme suit :

$$new\_var(x_0, \ldots\ x_n : s) = new\_var(x_0 : s), \ldots\ new\_var(x_n : s)$$

5.6.3 Renommage des processus

On appelle *renommage des processus* une application totale de l'ensemble des processus LOTOS vers l'ensemble des parties de l'ensemble des processus SUBLOTOS.

Si $\mathcal{P}$ est un renommage des processus et p un processus LOTOS, $\mathcal{P}(p)$ vérifie plusieurs invariants :

- $(\forall P \in \mathcal{P}(p))\ \ origin(P) = p$
- $(\forall P \in \mathcal{P}(p))\ \ origin(gate_0(P)) = \delta$
- $(\forall P \in \mathcal{P}(p))\ \ (\forall i \in \{1, \ldots\ m\})\ \ origin(gate_i(P)) \neq \delta$
- $(\forall P, P' \in \mathcal{P}(p))\ \ ((\forall i \in \{0, \ldots\ m\})\ \ origin(gate_i(P)) = origin(gate_i(P'))) \Longrightarrow (P = P')$

Pour manipuler les renommages on utilise les définitions suivantes :

- on note "$\top$" l'application $\mathcal{P}$ qui à tout processus LOTOS p associe l'ensemble vide :

$$\mathcal{P}(p) = \emptyset$$

- on note "$p_0 \mapsto P_0$" l'application $\mathcal{P}$ qui renvoie l'ensemble vide sauf pour le processus LOTOS p_0 auquel elle fait correspondre le singleton contenant le processus LOTOS P_0 :

$$\mathcal{P}(p) = \begin{cases} \textbf{si } p = p_0 \textbf{ alors } \{P_0\} \\ \textbf{si } p \neq p_0 \textbf{ alors } \ \emptyset \end{cases}$$

- on note "$\uplus$" l'opération binaire qui à deux renommages de processus $\mathcal{P}_1$ et $\mathcal{P}_2$ associe le renommage $\mathcal{P}$ formé par l'union de $\mathcal{P}_1$ et $\mathcal{P}_2$:

$$\mathcal{P}(p) = \mathcal{P}_1(p) \cup \mathcal{P}_2(p)$$

- on note "*new_proc*(p)" une nouvelle duplication du processus LOTOS p : il s'agit d'un processus SUBLOTOS P qui est différent de tous les processus SUBLOTOS créés auparavant et qui vérifie *origin*$(P) = p$ (P est de la forme $p{\bullet}i$)
- on note "*old_proc*(P)" un commentaire qui indique que le processus SUBLOTOS P a été créé mais ne sera jamais utilisé dans la spécification SUBLOTOS ; on peut donc le détruire

En outre on associe à chaque processus SUBLOTOS P un attribut local booléen, noté *recursive*(P), qui vaut *true* si et seulement si P est récursif. La valeur de *recursive*(P) est calculée pendant l'expansion du comportement *behaviour*(*origin*(P)).

Le concept de renommage des processus intervient dans l'algorithme d'expansion de la manière suivante :

- l'expansion et le développement des processus peuvent être assimilés à un parcours en profondeur du *graphe des appels* qui est défini de la manière suivante :
  - chaque sommet correspond à un processus SUBLOTOS
  - il existe un arc du sommet P vers le sommet P' si et seulement si le corps du processus P contient une instanciation du processus P'
- on marque les sommets afin de détecter les circuits, c'est-à-dire les instanciations récursives. Ce marquage s'effectue au moyen d'un renommage des processus $\mathcal{P}$ qui est transmis comme un attribut hérité : l'ensemble des sommets marqués est égal à l'union des ensembles $\mathcal{P}(p)$ pour tous les processus LOTOS p. Cet attribut matérialise l'état de la trace à un instant donné
- quand on rencontre une instanciation LOTOS de la forme "p `[`$g_1, \ldots\ g_m$`]`" on recherche s'il existe dans $\mathcal{P}(p)$ un processus P dont les paramètres formels correspondent aux paramètres effectifs de cette instanciation : plus précisément on doit avoir, pour tout i dans $\{1, \ldots\ m\}$, *origin*(*gate*$_i(P)$) = *origin*$(\mathcal{G}(g_i))$, où $\mathcal{G}$ est le renommage des portes dans lequel s'effectue l'expansion de l'instanciation
  - si oui, on est en présence d'une instanciation LOTOS récursive que l'on traduit par l'instanciation SUBLOTOS récursive "P `[`*gate*$_0(P), \ldots$ *gate*$_m(P)$`]`". On donne à *recursive*(P) la valeur *true*

– sinon, on est en présence d'une instanciation LOTOS non récursive[28]. On crée alors une nouvelle duplication du processus p : on obtient ainsi un processus SUBLOTOS P auquel on donne comme paramètres formels $\mathcal{G}(\delta), \mathcal{G}(g_1), \ldots\ \mathcal{G}(g_m)$. On initialise *recursive*(P) à *false*, on ajoute P dans la trace et on traduit en SUBLOTOS le corps de p, ce qui produit un comportement B_0. Ce parcours récursif positionne le booléen *recursive*(P). Au retour on teste si *recursive*(P) est égal à *true* :
  * si oui on traduit l'instanciation LOTOS par une instanciation SUBLOTOS "P [$gate_0(P), \ldots\ gate_m(P)$]". Le corps de P est constitué par B_0
  * sinon il on traduit l'instanciation LOTOS par le comportement B_0. Ce développement fait que le processus SUBLOTOS P n'est jamais utilisé

5.6.4 Expressions de valeur

Chaque expression de valeur LOTOS v est traduite en une expression de valeur SUBLOTOS V correspondante. On note $\mathcal{X}$ le renommage des variables dans lequel s'effectue l'expansion de v. Par définition *sort*(v) est égal à *sort*(V).

$$
\begin{array}{l}
v \downarrow \mathcal{X} \uparrow V \equiv \\
\quad x \\
\qquad \{ \quad V \ := \ \mathcal{X}(x) \\
\mid f \ (v_1 \downarrow \mathcal{X} \uparrow V_1, \ldots\ v_n \downarrow \mathcal{X} \uparrow V_n) \\
\qquad \{ \quad V \ := \ f \ (V_1, \ldots\ V_n) \\
\mid v_1 \downarrow \mathcal{X} \uparrow V_1 = v_2 \downarrow \mathcal{X} \uparrow V_2 \\
\qquad \{ \quad V \ := \ V_1 = V_2
\end{array}
$$

5.6.5 Résultats

Chaque résultat LOTOS r est traduit en une offre SUBLOTOS O. On note $\mathcal{X}$ le renommage des variables dans lequel s'effectue l'expansion de r.

$$
\begin{array}{l}
r \downarrow \mathcal{X} \uparrow O \equiv \\
\quad v \downarrow \mathcal{X} \uparrow V \\
\qquad \{ \quad O \ := \ !V \\
\mid \textbf{any } s \\
\qquad \{ \quad O \ := \ ?new\_var(\xi : s) : s
\end{array}
$$

[28] bien que le processus p puisse être récursif

5.6.6 Offres

Chaque offre LOTOS o est traduite en une liste d'offres SUBLOTOS $\widehat{O}$. En effet les offres multiples "$?x_0, \ldots\ x_n : s$" sont linéarisées par l'expansion en "$?X_0 : s, \ldots\ ?X_n : s$". On note $\mathcal{X}$ le renommage des variables dans lequel s'effectue l'expansion de o. On note $\mathcal{X}'$ le renommage obtenu en ajoutant à $\mathcal{X}$ les variables définies par l'offre o.

$$
\begin{array}{l}
o \downarrow \mathcal{X} \uparrow \mathcal{X}' \uparrow \widehat{O} \equiv \\
\quad \mathtt{!}v \downarrow \mathcal{X} \uparrow V \\
\quad\quad \left\{ \begin{array}{lcl} \mathcal{X}' & := & \perp \\ \widehat{O} & := & \mathtt{!}V \end{array} \right. \\
\mid\ \mathtt{?}x_0, \ldots\ x_n : s \\
\quad\quad \left\{ \begin{array}{l} (\forall i \in \{0, \ldots\ n\})\ \ X_i \ := \ \mathit{new\_var}(x_i : s) \\ \mathcal{X}' \ := \ \bigoplus_{i \in \{0, \ldots\ n\}} (x_i \rightsquigarrow X_i) \\ \widehat{O} \ := \ \mathtt{?}X_0 : s, \ldots\ \mathtt{?}X_n : s \end{array} \right.
\end{array}
$$

5.6.7 Opérateurs parallèles

Chaque opérateur parallèle LOTOS op est traduit en un opérateur parallèle SUBLOTOS OP. On note $\mathcal{G}$ le renommage des portes dans lequel s'effectue l'expansion de op.

$$
\begin{array}{l}
op \downarrow \mathcal{G} \uparrow OP \equiv \\
\quad \mathtt{||} \\
\quad\quad \left\{ \begin{array}{lcl} OP & := & \mathtt{||} \end{array} \right. \\
\mid\ \mathtt{|||} \\
\quad\quad \left\{ \begin{array}{lcl} \Delta & := & \mathcal{G}(\delta) \\ OP & := & \mathtt{|[}\Delta\mathtt{]|} \end{array} \right. \\
\mid\ \mathtt{|[}g_0, \ldots\ g_n\mathtt{]|} \\
\quad\quad \left\{ \begin{array}{l} \Delta \ := \ \mathcal{G}(\delta) \\ (\forall i \in \{0, \ldots\ n\})\ \ G_i \ := \ \mathcal{G}(g_i) \\ OP \ := \ \mathtt{|[}\Delta, G_0, \ldots\ G_n\mathtt{]|} \end{array} \right.
\end{array}
$$

5.6.8 Expressions de comportement

Chaque expression de comportement LOTOS b est traduite en une expression de comportement SUBLOTOS B correspondante. On note $\mathcal{G}$ (*resp.* $\mathcal{X}$ et $\mathcal{P}$) le renommage des portes (*resp.* variables et processus) dans lequel s'effectue l'expansion de b.

$$
\begin{array}{l}
b \downarrow \mathcal{G} \downarrow \mathcal{X} \downarrow \mathcal{P} \uparrow B \equiv \\
\quad \textbf{stop} \\
\quad\quad \left\{ \begin{array}{lcl} B & := & \textbf{stop} \end{array} \right.
\end{array}
$$

$| \ \mathbf{i} \ ; \ b_0 \downarrow \mathcal{G} \downarrow \mathcal{X} \downarrow \mathcal{P} \uparrow B_0$
$$\left\{ \ B \ := \ \mathbf{i} \ ; \ B_0 \ \textit{(false, false)} \right.$$

$| \ g \ o_1 \downarrow \mathcal{X} \uparrow \mathcal{X}'_1 \uparrow \widehat{O_1}, \ldots \ o_n \downarrow \mathcal{X} \uparrow \mathcal{X}'_n \uparrow \widehat{O_n} \ ; \ b_0 \downarrow \mathcal{G} \downarrow \mathcal{X}' \downarrow \mathcal{P} \uparrow B_0$
$$\left\{ \begin{array}{lcl} \mathcal{X}' & := & \mathcal{X} \oslash \bigoplus_{i \in \{1,\ldots\ n\}} \mathcal{X}'_i \\ G & := & \mathcal{G}(g) \\ B & := & G \ \widehat{O_1}, \ldots \ \widehat{O_n} \ ; \ B_0 \ \textit{(false, false)} \end{array} \right.$$

$| \ g \ o_1 \downarrow \mathcal{X} \uparrow \mathcal{X}'_1 \uparrow O_1, \ldots \ o_n \downarrow \mathcal{X} \uparrow \mathcal{X}'_n \uparrow O_n \ [v_0 \downarrow \mathcal{X}' \uparrow V_0] \ ; \ b_0 \downarrow \mathcal{G} \downarrow \mathcal{X}' \downarrow \mathcal{P} \uparrow B_0$
$$\left\{ \begin{array}{lcl} \mathcal{X}' & := & \mathcal{X} \oslash \bigoplus_{i \in \{1,\ldots\ n\}} \mathcal{X}'_i \\ G & := & \mathcal{G}(g) \\ B & := & G \ O_1, \ldots \ O_n \ [V_0] \ ; \ B_0 \ \textit{(false, false)} \end{array} \right.$$

$| \ b_1 \downarrow \mathcal{G} \downarrow \mathcal{X} \downarrow \mathcal{P} \uparrow B_1 \ [] \ b_2 \downarrow \mathcal{G} \downarrow \mathcal{X} \downarrow \mathcal{P} \uparrow B_2$
$$\left\{ \ B \ := \ B_1 \ [] \ B_2 \right.$$

$| \ \mathbf{choice} \ \widehat{g_0} \ \mathbf{in} \ [\widehat{g'_0}], \ldots \ \widehat{g_n} \ \mathbf{in} \ [\widehat{g'_n}] \ [] \ b_0$
$$\left\{ \begin{array}{l} (\forall i \in \{0,\ldots\ n\}) \ \widehat{G_i} \ := \ \mathcal{G}(\widehat{g'_i}) \\ B \ := \ \textit{choice}(b_0, \mathcal{G}, \mathcal{X}, \mathcal{P}, \{\widehat{g_0}, \ldots \ \widehat{g_n}\}, \{\widehat{G_0}, \ldots \ \widehat{G_n}\}) \\ \mathbf{avec} \\ \textit{choice}(b_0, \mathcal{G}, \mathcal{X}, \mathcal{P}, \{\widehat{g_0}, \ldots \ \widehat{g_n}\}, \{\widehat{G_0}, \ldots \ \widehat{G_n}\}) = \\ \quad \mathbf{soient} \ g_0, \ldots \ g_p \ \mathbf{tels\ que} \ \widehat{g_0} \equiv g_0, \ldots \ g_p \\ \quad \mathbf{soient} \ G_0, \ldots \ G_q \ \mathbf{tels\ que} \ \widehat{G_0} \equiv G_0, \ldots \ G_q \\ \quad \mathbf{si} \ (p = 0) \ \wedge \ (n = 0) \ \mathbf{alors} \\ \qquad []_{i \in \{0,\ldots\ q\}} B_i \\ \qquad \mathbf{où} \ (\forall i \in \{0,\ldots\ q\}) \ B_i \ \mathbf{est\ défini\ par} \ b_0 \downarrow \mathcal{G} \oslash (g_0 \rightsquigarrow G_i) \downarrow \mathcal{X} \downarrow \mathcal{P} \uparrow B_i \\ \quad \mathbf{sinon\ si} \ (p = 0) \ \wedge \ (n > 0) \ \mathbf{alors} \\ \qquad []_{i \in \{0,\ldots\ q\}} \textit{choice}(b_0, \mathcal{G} \oslash (g_0 \rightsquigarrow G_i), \mathcal{X}, \mathcal{P}, \{\widehat{g_1}, \ldots \ \widehat{g_n}\}, \{\widehat{G_1}, \ldots \ \widehat{G_n}\}) \\ \quad \mathbf{sinon\ si} \ p > 0 \ \mathbf{alors} \\ \qquad []_{i \in \{0,\ldots\ q\}} \textit{choice}(b_0, \mathcal{G} \oslash (g_0 \rightsquigarrow G_i), \mathcal{X}, \mathcal{P}, \{\widehat{g'_0}, \widehat{g_1}, \ldots \ \widehat{g_n}\}, \{\widehat{G_0}, \ldots \ \widehat{G_n}\}) \\ \qquad \mathbf{où} \ \widehat{g'_0} \ := \ g_1, \ldots \ g_p \end{array} \right.$$

$| \ b_1 \downarrow \mathcal{G} \downarrow \mathcal{X} \downarrow \mathcal{P} \uparrow B_1 \ \textit{op} \downarrow \mathcal{G} \uparrow \textit{OP} \ b_2 \downarrow \mathcal{G} \downarrow \mathcal{X} \downarrow \mathcal{P} \uparrow B_2$
$$\left\{ \ B \ := \ B_1 \ \textit{OP} \ B_2 \ \textit{(false)} \right.$$

$| \ \mathbf{par} \ \widehat{g_0} \ \mathbf{in} \ [\widehat{g'_0}], \ldots \ \widehat{g_n} \ \mathbf{in} \ [\widehat{g'_n}] \ \textit{op} \downarrow \mathcal{G} \uparrow \textit{OP} \ b_0$
$$\left\{ \begin{array}{l} (\forall i \in \{0,\ldots\ n\}) \ \widehat{G_i} \ := \ \mathcal{G}(\widehat{g'_i}) \\ B \ := \ \textit{par}(b_0, \textit{OP}, \mathcal{G}, \mathcal{X}, \mathcal{P}, \{\widehat{g_0}, \ldots \ \widehat{g_n}\}, \{\widehat{G_0}, \ldots \ \widehat{G_n}\}) \\ \mathbf{avec} \\ \textit{par}(b_0, \textit{OP}, \mathcal{G}, \mathcal{X}, \mathcal{P}, \{\widehat{g_0}, \ldots \ \widehat{g_n}\}, \{\widehat{G_0}, \ldots \ \widehat{G_n}\}) = \\ \quad \mathbf{soient} \ g_0, \ldots \ g_p \ \mathbf{tels\ que} \ \widehat{g_0} \equiv g_0, \ldots \ g_p \\ \quad \mathbf{soient} \ G_0, \ldots \ G_q \ \mathbf{tels\ que} \ \widehat{G_0} \equiv G_0, \ldots \ G_q \\ \quad \mathbf{si} \ (p = 0) \ \wedge \ (n = 0) \ \mathbf{alors} \\ \qquad \textit{OP}_{i \in \{0,\ldots\ q\}} B_i \ \textit{(false)} \\ \qquad \mathbf{où} \ (\forall i \in \{0,\ldots\ q\}) \ B_i \ \mathbf{est\ défini\ par} \ b_0 \downarrow \mathcal{G} \oslash (g_0 \rightsquigarrow G_i) \downarrow \mathcal{X} \downarrow \mathcal{P} \uparrow B_i \\ \quad \mathbf{sinon\ si} \ (p = 0) \ \wedge \ (n > 0) \ \mathbf{alors} \\ \qquad \textit{OP}_{i \in \{0,\ldots\ q\}} \textit{par}(b_0, \textit{OP}, \mathcal{G} \oslash (g_0 \rightsquigarrow G_i), \mathcal{X}, \mathcal{P}, \{\widehat{g_1}, \ldots \ \widehat{g_n}\}, \{\widehat{G_1}, \ldots \ \widehat{G_n}\}) \ \textit{(false)} \\ \quad \mathbf{sinon\ si} \ p > 0 \ \mathbf{alors} \\ \qquad \textit{OP}_{i \in \{0,\ldots\ q\}} \textit{par}(b_0, \textit{OP}, \mathcal{G} \oslash (g_0 \rightsquigarrow G_i), \mathcal{X}, \mathcal{P}, \{\widehat{g'_0}, \widehat{g_1}, \ldots \ \widehat{g_n}\}, \{\widehat{G_0}, \ldots \ \widehat{G_n}\}) \ \textit{(false)} \\ \qquad \mathbf{où} \ \widehat{g'_0} \ := \ g_1, \ldots \ g_p \end{array} \right.$$

$| \ \textbf{hide}\ g_0, \ldots\ g_n\ \textbf{in}\ b_0 \downarrow \mathcal{G}' \downarrow \mathcal{X} \downarrow \mathcal{P} \uparrow B_0$
$$\left\{\begin{array}{l} (\forall i \in \{0, \ldots\ n\})\ \ G_i\ := \ \textit{new\_gate}(g_i) \\ \mathcal{G}'\ :=\ \mathcal{G} \oslash \bigoplus_{i \in \{0, \ldots\ n\}} (g_i \rightsquigarrow G_i) \\ B\ :=\ \textbf{hide}\ G_0, \ldots\ G_n\ \textbf{in}\ B_0 \end{array}\right.$$

$| \ \texttt{[}v_0 \downarrow \mathcal{X} \uparrow V_0\texttt{]}\ \texttt{->}\ b_0 \downarrow \mathcal{G} \downarrow \mathcal{X} \downarrow \mathcal{P} \uparrow B_0$
$$\left\{\ B\ :=\ \texttt{[}V_0\texttt{]}\ \texttt{->}\ B_0\right.$$

$| \ \textbf{let}\ \widehat{x_0}\!:\!s_0\texttt{=}v_0 \downarrow \mathcal{X} \uparrow V_0, \ldots\ \widehat{x_n}\!:\!s_n\texttt{=}v_n \downarrow \mathcal{X} \uparrow V_n\ \textbf{in}\ b_0 \downarrow \mathcal{G} \downarrow \mathcal{X}' \downarrow \mathcal{P} \uparrow B_0$
$$\left\{\begin{array}{l} (\forall i \in \{0, \ldots\ n\})\ \ \widehat{X_i}\ :=\ \textit{new\_var}(\widehat{x_i} : s_i) \\ \mathcal{X}'\ :=\ \mathcal{X} \oslash \bigoplus_{i \in \{0, \ldots\ n\}} (\widehat{x_i} \rightsquigarrow \widehat{X_i}) \\ B\ :=\ \textbf{let}\ \widehat{X_0}\!:\!s_0\texttt{=}V_0, \ldots\ \widehat{X_n}\!:\!s_n\texttt{=}V_n\ \textbf{in}\ B_0 \end{array}\right.$$

$| \ \textbf{choice}\ \widehat{x_0}\!:\!s_0, \ldots\ \widehat{x_n}\!:\!s_n\ \texttt{[]}\ b_0 \downarrow \mathcal{G} \downarrow \mathcal{X}' \downarrow \mathcal{P} \uparrow B_0$
$$\left\{\begin{array}{l} (\forall i \in \{0, \ldots\ n\})\ \ \widehat{X_i}\ :=\ \textit{new\_var}(\widehat{x_i} : s_i) \\ \mathcal{X}'\ :=\ \mathcal{X} \oslash \bigoplus_{i \in \{0, \ldots\ n\}} (\widehat{x_i} \rightsquigarrow \widehat{X_i}) \\ B\ :=\ \textbf{choice}\ \widehat{X_0}\!:\!s_0, \ldots\ \widehat{X_n}\!:\!s_n\ \textbf{in}\ B_0 \end{array}\right.$$

$| \ \textbf{exit}\ \texttt{(}r_1 \downarrow \mathcal{X} \uparrow O_1, \ldots\ r_n \downarrow \mathcal{X} \uparrow O_n\texttt{)}$
$$\left\{\begin{array}{l} \Delta\ :=\ \mathcal{G}(\delta) \\ B\ :=\ \Delta\ O_1, \ldots\ O_n\ \ \texttt{;}\ \textbf{stop}\ \textit{(true, false)} \end{array}\right.$$

$| \ b_1 \downarrow \mathcal{G}_1 \downarrow \mathcal{X} \downarrow \mathcal{P} \uparrow B_1\ \texttt{>>}\ \textbf{accept}\ \widehat{x_1}\!:\!s_1, \ldots\ \widehat{x_n}\!:\!s_n\ \textbf{in}\ b_2 \downarrow \mathcal{G} \downarrow \mathcal{X}_2 \downarrow \mathcal{P} \uparrow B_2$
$$\left\{\begin{array}{l} \Delta\ :=\ \textit{new\_gate}(\delta) \\ \mathcal{G}_1\ :=\ \mathcal{G} \oslash (\delta \rightsquigarrow \Delta) \\ (\forall i \in \{0, \ldots\ n\})\ \ \widehat{X_i} = \textit{new\_var}(\widehat{x_i} : s_i) \\ \mathcal{X}_2\ :=\ \mathcal{X} \oslash \bigoplus_{i \in \{1, \ldots\ n\}} (\widehat{x_i} \rightsquigarrow \widehat{X_i}) \\ (\forall i \in \{0, \ldots\ n\})\ \ \textbf{si}\ \widehat{X_i} \equiv X_0, \ldots\ X_p\ \textbf{alors}\ \widehat{O_i}\ :=\ \texttt{?}X_0\!:\!s_i, \ldots\ X_p\!:\!s_i \\ B\ :=\ \textbf{hide}\ \Delta\ \textbf{in}\ \texttt{(}B_1\ \texttt{|[}\Delta\texttt{]|}\ \texttt{(}\Delta\ \widehat{O_1}, \ldots\ \widehat{O_n}\ \ \texttt{;}\ B_2\ \textit{(false, true)}\texttt{)}\ \textit{(true)}\texttt{)} \end{array}\right.$$

$| \ b_1 \downarrow \mathcal{G} \downarrow \mathcal{X} \downarrow \mathcal{P} \uparrow B_1\ \texttt{[>}\ b_2 \downarrow \mathcal{G} \downarrow \mathcal{X} \downarrow \mathcal{P} \uparrow B_2$
$$\left\{\begin{array}{l} \Delta\ :=\ \mathcal{G}(\delta) \\ B\ :=\ B_1\ \texttt{[}\Delta\texttt{>}\ B_2 \end{array}\right.$$

$$
\begin{array}{l}
\mid p\ \texttt{[}g_1, \ldots\ g_m\texttt{]}\ \texttt{(}v_1 \downarrow \mathcal{X} \uparrow V_1, \ldots\ v_n \downarrow \mathcal{X} \uparrow V_n\texttt{)} \\
\quad \left\{ \begin{array}{l}
\textbf{si } (\exists P \in \mathcal{P}(p))\ \ (\forall i \in \{1, \ldots\ m\})\ \ \mathit{origin}(\mathit{gate}_i(P)) = \mathit{origin}(\mathcal{G}(g_i))\ \textbf{alors} \\
\quad \mathit{recursive}(P)\ := \ \mathit{true} \\
\quad B\ :=\ P\ \texttt{[}\mathit{gate}_0(P), \ldots\ \mathit{gate}_m(P)\texttt{]}\ \texttt{(}V_1, \ldots\ V_n\texttt{)} \\
\textbf{sinon} \\
\quad P\ :=\ \mathit{new\_proc}(p) \\
\quad \mathit{gate}_0(P)\ :=\ \mathcal{G}(\delta) \\
\quad (\forall i \in \{1, \ldots\ m\})\ \ \mathit{gate}_i(P)\ :=\ \mathcal{G}(g_i) \\
\quad (\forall j \in \{1, \ldots\ n\})\ \ s_j\ :=\ \mathit{sort}(\mathit{var}_j(p)) \\
\quad (\forall j \in \{1, \ldots\ n\})\ \ \mathit{var}_j(P)\ :=\ \mathit{new\_var}(\mathit{var}_j(p) : s_j) \\
\quad \mathcal{G}'\ :=\ \mathcal{G} \oslash ((\delta \leadsto \mathit{gate}_0(P)) \oplus \bigoplus_{i \in \{1, \ldots\ m\}} (\mathit{gate}_i(p) \leadsto \mathit{gate}_i(P))) \\
\quad \mathcal{X}'\ :=\ \mathcal{X} \oslash \bigoplus_{j \in \{1, \ldots\ n\}} (\mathit{var}_j(p) \leadsto \mathit{var}_j(P)) \\
\quad \mathcal{P}'\ :=\ \mathcal{P} \uplus (p \mapsto P) \\
\quad \mathit{recursive}(P)\ :=\ \mathit{false} \\
\quad \textbf{soit } B_0 \textbf{ défini par } \mathit{behaviour}(p) \downarrow \mathcal{G}' \downarrow \mathcal{X}' \downarrow \mathcal{P}' \uparrow B_0 \\
\quad \textbf{si } \mathit{recursive}(P) = \mathit{true}\ \textbf{alors} \\
\quad\quad \mathit{behaviour}(P)\ :=\ B_0 \\
\quad\quad B\ :=\ P\ \texttt{[}\mathit{gate}_0(P), \ldots\ \mathit{gate}_m(P)\texttt{]}\ \texttt{(}V_1, \ldots\ V_n\texttt{)} \\
\quad \textbf{sinon si } n = 0\ \textbf{alors} \\
\quad\quad B\ :=\ B_0 \\
\quad\quad \mathit{old\_proc}(P) \\
\quad \textbf{sinon} \\
\quad\quad B\ :=\ \textbf{let } \mathit{var}_1(P) \mathbin{:} s_1 \texttt{=} V_1, \ldots\ \mathit{var}_n(P) \mathbin{:} s_n \texttt{=} V_n\ \textbf{in } B_0 \\
\quad\quad \mathit{old\_proc}(P)
\end{array} \right.
\end{array}
$$

5.6.9 Construction du programme SUBLOTOS

La spécification Lotos λ est traduite en une spécification SubLotos Λ correspondante. Pour qu'une simulation exhaustive soit possible, il faut que Λ soit une spécification *fermée*, c'est-à-dire non paramétrée par des valeurs. Si λ comporte des variables paramètres formels, l'expansion les élimine en les déclarant au moyen d'un opérateur "**choice**".

$$
\left\{ \begin{array}{l}
\mathit{gate}_0(\Lambda)\ :=\ \delta_{\bullet 0} \\
(\forall i \in \{1, \ldots\ m\})\ \ \mathit{gate}_i(\Lambda)\ :=\ \mathit{gate}_i(\lambda)_{\bullet 0} \\
(\forall j \in \{1, \ldots\ n\})\ \ s_j\ :=\ \mathit{sort}(\mathit{var}_j(\lambda)) \\
(\forall j \in \{1, \ldots\ n\})\ \ X_j\ :=\ \mathit{new\_var}(\mathit{var}_j(\lambda) : s_j) \\
\mathcal{G}\ :=\ (\mathbf{i} \leadsto \mathbf{i}_{\bullet 1}) \oplus (\delta \leadsto \mathit{gate}_0(\Lambda)) \oplus \bigoplus_{i \in \{1, \ldots\ m\}} (g_i \leadsto \mathit{gate}_i(\Lambda)) \\
\mathcal{X} = \bigoplus_{j \in \{1, \ldots\ n\}} (x_j \leadsto X_j) \\
\textbf{soit } B \textbf{ défini par } \mathit{behaviour}(\lambda) \downarrow \mathcal{G} \downarrow \mathcal{X} \downarrow \top \uparrow B \\
\textbf{si } n = 0\ \textbf{alors} \\
\quad \mathit{behaviour}(\Lambda)\ :=\ B \\
\textbf{sinon} \\
\quad \mathit{behaviour}(\Lambda)\ :=\ \textbf{choice } X_1 \mathbin{:} s_1, \ldots\ X_n \mathbin{:} s_n\ \texttt{[]}\ B
\end{array} \right.
$$

5.7 Implémentation

Le logiciel Cæsar comporte une phase d'expansion qui implémente exactement l'algorithme d'expansion défini ici. A partir de la grammaire attribuée, un programme en langage C a été dérivé manuellement, mais de manière systématique afin d'assurer la conformité entre la spécification du compilateur et sa réalisation. L'implémentation ainsi obtenue est fondée sur plusieurs constatations :

- tous les attributs, hérités, synthétisés et locaux, sont calculés en un seul passage
- pour chaque construction Lotos (expression de valeur, expression de comportement, ...) il n'y a qu'un seul attribut synthétisé. On crée donc, pour chacune de ces constructions, une *fonction de traduction* qui prend en paramètre une construction Lotos et renvoie comme résultat la construction SubLotos correspondante. Ces fonctions sont définies par récurrence sur la structure syntaxique de leurs arguments
- pour des raisons de performances les renommages des portes, des variables et des processus ne sont pas passés en paramètres des fonctions de traduction. Ils sont implémentés comme des variables globales

Chapitre 6

Génération

La phase de génération construit, à partir d'un programme SubLotos, un réseau équivalent.

On présente d'abord les principes généraux de cette traduction. Puis, pour chaque opérateur de comportement SubLotos, on explique comment produire le réseau correspondant. Enfin on décrit formellement l'algorithme de génération au moyen d'une grammaire attribuée.

Si le problème de la traduction des expressions régulières en automates d'états finis est bien connu — divers algorithmes existent pour cela [ASU86, p. 121–125] [BS87] — il n'en est pas de même pour la traduction en réseaux de Petri des langages comme Lotos, dont la partie contrôle est basée sur une algèbre de processus. Dans le cas de Lotos, il faut également prendre en compte la partie données, ainsi que certaines caractéristiques originales du langage : opérateurs "`>>`" et "`[>`", rendez-vous n-aire, mécanisme d'unification des offres au cours des rendez-vous, ... Les choix de conception effectués au moment de la définition du modèle réseau permettent d'avoir pour Lotos un algorithme de génération simple et efficace.

6.1 Principes de la génération

La méthode de génération conçue pour Cæsar présente certaines analogies avec d'autres algorithmes ; toutefois aucune des techniques existantes ne répond réellement au problème posé par la traduction de Lotos :

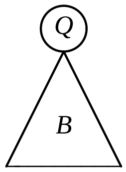
- génération de réseaux de Petri pour un langage avec valeurs proche de CSP [Que82, chapitre 3]. Ce principe de traduction, utilisé par le système Quasar, se retrouve, sous des formes diverses, dans tous les outils de la famille CESAR. Il ne peut toutefois pas être appliqué directement au cas du langage Lotos

- génération de réseaux de Petri pour CCS [GM84]. Outre les différences essentielles entre CCS et Lotos, notamment pour la composition parallèle, cette méthode ne prend pas en compte la présence de données

- génération de réseaux de Petri pour Lotos [ML88]. Ici encore, cette solution ne concerne qu'un sous-ensemble Lotos restreint à la seule partie contrôle, à l'exclusion de la partie données. Comme les réseaux engendrés ne comportent pas d'ε-transition, leur taille peut croître très vite, ce qui est contraire aux principes de Cæsar qui exigent que la forme intermédiaire produite soit compacte

Toutes ces méthodes ont un point commun : elles engendrent l'automate ou le réseau de Petri par récurrence sur la structure syntaxique du langage à traduire. Dans le cas de SubLotos, la construction, pour chaque expression de comportement B, du réseau associé se fait de manière ascendante par synthèse d'attributs : c'est ainsi que le réseau d'un comportement B comprenant des sous-comportements $B_1, \ldots\ B_n$ s'exprime en fonction des réseaux de $B_1, \ldots\ B_n$.

L'algorithme de génération mis en œuvre dans Cæsar est basé sur un invariant simple : pour tout comportement B, le réseau de B doit être engendré à partir d'une place initiale Q, ce que l'on représente par le schéma suivant :



Remarque 6-1
La sémantique de Lotos fait qu'en général le réseau d'un comportement B ne comporte pas de place finale : en effet la plupart des comportements ne se terminent pas et il est inutile de leur assigner une place finale inaccessible. Cette approche permet de minimiser le nombre de places et de transitions du réseau. Elle diffère de celle de [Que82] qui imposait aux réseaux d'être construits entre une demi-transition initiale et une demi-transition terminale. ■

Pendant la génération — tout comme pendant l'expansion — on ne cherche pas à évaluer les données : les variables, les valeurs et les actions sont construites et manipulées symboliquement.

6.1.1 Génération de l'opérateur "stop"

Le comportement "**stop**" est traduit par un réseau réduit à la place init dont aucune transition n'est issue, ce qui exprime le blocage :

6.1.2 Génération de l'opérateur ";"

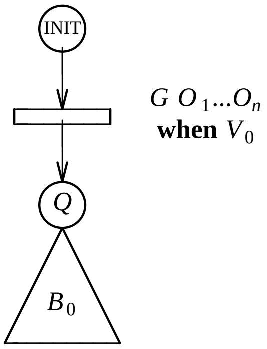
Lorsque l'on rencontre un comportement de la forme :

$$G\ O_1, \ldots\ O_n\ \texttt{[}V_0\texttt{]}\ \texttt{;}\ B_0\ \textit{(from\_exit, from\_enable)}$$

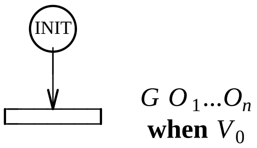
il faut distinguer trois cas, suivant la valeur des booléens *from_exit* et *from_enable* :

1. si *from_exit* = *false* et *from_enable* = *false* : l'opérateur SubLotos ";" provient de l'expansion d'un opérateur Lotos ";". On crée une place Q à partir de laquelle on engendre le réseau de B_0. On crée aussi une transition T allant de init à Q, ayant pour porte G, pour offre $\langle O_1, \ldots\ O_n \rangle$ et pour action "**when** V_0"[29].

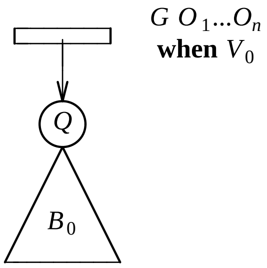
[29] si l'action V_0 est absente, la transition T est étiquetée par l'action "**none**"



2. si *from_exit* = *true* : l'opérateur SubLotos ";" provient de l'expansion d'un opérateur Lotos "**exit**". Dans ce cas, d'après l'algorithme d'expansion, le comportement B_0 est nécessairement réduit à "**stop**". On crée alors une transition T partant de INIT et n'ayant aucune place de sortie.



3. si *from_enable* = *true* : l'opérateur SubLotos ";" provient de l'expansion d'un opérateur Lotos ">>". Dans ce cas, la place INIT est indéfinie. On crée alors une place Q à partir de laquelle on engendre le réseau de B_0. On crée aussi une transition T arrivant vers Q et n'ayant aucune place d'entrée.

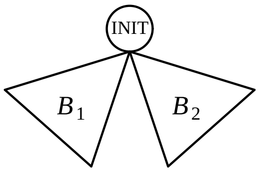


6.1.3 Génération de l'opérateur "[]"

Pour traduire un comportement de la forme :

$$B_1 \texttt{ [] } B_2$$

on engendre les réseaux de B_1 et B_2 à partir de la même place INIT.



6.1.4 Génération des opérateurs "||" et "|[...]|"

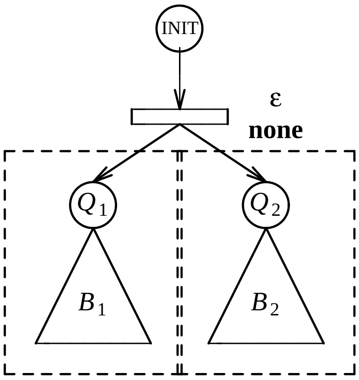
Lorsqu'on rencontre un comportement de la forme :

$$B_1 \ op \ B_2 \ (from\_enable)$$

il faut distinguer deux cas, suivant la valeur du booléen *from_enable* :

1. si *from_enable* = *false* : l'opérateur SubLotos "*op*" provient de l'expansion d'un opérateur Lotos "*OP*". La génération du réseau correspondant se fait en deux étapes :

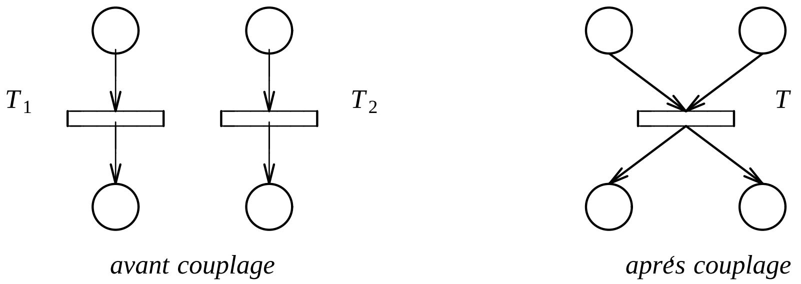
   - on commence par modéliser la *concurrence* entre B_1 et B_2. Pour exprimer le fait que B_1 et B_2 doivent être exécutés en parallèle, on crée deux places Q_1 et Q_2 à partir desquelles on engendre les réseaux respectifs de B_1 et de B_2. Puis on crée une ε-transition T, ayant une seule place d'entrée (INIT) et deux places de sortie (Q_1 et Q_2), dont l'effet est de démarrer simultanément (*fork*) B_1 et B_2. On crée enfin deux unités U_1 et U_2 destinées à contenir les places respectives des réseaux de B_1 et de B_2.



   - ensuite il faut modéliser la *synchronisation* et la *communication* entre B_1 et B_2. L'expression des rendez-vous dans le modèle réseau se fait par *couplage* des transitions de B_1 et de B_2. Pour cela les définitions suivantes sont nécessaires
   - on dit qu'une transition T_1 (*resp.* T_2) appartenant au réseau de B_1 (*resp.* B_2) est *en attente de rendez-vous* dans B_1 (*resp.* B_2) si sa porte fait partie de la liste des portes auxquelles l'opérateur "*op*" impose de se synchroniser
   - on dit que deux transitions T_1 et T_2 sont *synchronisables* si et seulement si :
     - T_1 est en attente de rendez-vous dans B_1
     - T_2 est en attente de rendez-vous dans B_2
     - les portes de T_1 et de T_2 sont identiques
     - les offres de T_1 et de T_2 sont compatibles, en nombre et en sortes, selon les règles du rendez-vous Lotos et SubLotos
   - *coupler* deux transitions synchronisables T_1 et T_2, c'est créer une nouvelle transition T dont les attributs sont définis comme suit :
     - l'ensemble des places d'entrée de T est égal à l'union des ensembles des places d'entrée de T_1 et de T_2, ce qui exprime que les comportements B_1 et B_2 doivent s'attendre afin de franchir la transition T de manière synchrone

– l'ensemble des places de sortie de T est égal à l'union des ensembles des places de sortie de T_1 et de T_2, ce qui signifie que les comportements B_1 et B_2, après avoir franchi simultanément la transition T, repartent de façon asynchrone

– la porte de T est celle de T_1 et de T_2



– l'offre de T, notée $\langle O_1, \ldots\ O_n\rangle$, est calculée à partir des offres de T_1 et de T_2, notées $\langle O_1^1, \ldots\ O_n^1\rangle$ et $\langle O_1^2, \ldots\ O_n^2\rangle$. Quatre cas peuvent se présenter :

  ∗ si $O_i^1 \equiv\ !V_1$ et $O_i^2 \equiv\ !V_2$: il s'agit d'une interaction de type *value matching*. Les valeurs V_1 et V_2 ont la même sorte. On prend $O_i \equiv\ !V_1$ mais il faut ajouter la condition $A_i \equiv$ **when** $V_1 = V_2$ à l'action de T, afin d'empêcher le franchissement de la transition si les valeurs V_1 et V_2 ne sont pas égales

  ∗ si $O_i^1 \equiv\ !V_1$ et $O_i^2 \equiv\ ?X_2 : S_2$: il s'agit d'une interaction de type *value passing*. La valeur V_1 a pour sorte S_2. On prend $O_i \equiv\ !V_1$; autrement dit, le couplage d'une émission et d'une réception produit une émission. On ajoute aussi l'affectation $A_i \equiv X_2 := V_1$ à l'action de T

  ∗ si $O_i^1 \equiv\ ?X_1 : S_1$ et $O_i^2 \equiv\ !V_2$: il s'agit d'une interaction de type *value passing*. La valeur V_2 a pour sorte S_1. Dans ce cas — dual du précédent — on prend $O_i \equiv\ !V_2$. On ajoute aussi l'affectation $A_i \equiv X_1 := V_2$ à l'action de T

  ∗ si $O_i^1 \equiv\ ?X_1 : S_1$ et $O_i^2 \equiv\ ?X_2 : S_2$: il s'agit d'une interaction de type *value generation*. Les sortes S_1 et S_2 sont identiques. On prend $O_i \equiv\ ?X_2 : S_2$. Il faut ajouter l'affectation $A_i \equiv X_1 := X_2$ à l'action de T afin de garantir que les deux variables X_1 et X_2 ont bien la même valeur.

  Remarque 6-2

  On avait a priori le choix entre prendre $O_i \equiv\ ?X_2 : S_2$ et prendre $O_i \equiv\ ?X_1 : S_1$. La seconde solution est préférable dans le cas où X_1 est une variable de la forme "$\xi_{\bullet p}$" introduite par l'algorithme d'expansion (§ 5.4.1, p. 98) : de cette façon la valeur de de la variable X_1 n'est pas utilisée. Ce procédé permettra à la phase d'optimisation d'éliminer simplement la plupart des variables "$\xi_{\bullet p}$". ■

  Pour toutes les actions d'affectation ainsi créées, on donne au booléen *from_sync* la valeur *true* qui indique qu'il s'agit d'une communication par rendez-vous

– l'action de T, notée A, est calculée à partir des actions de T_1 et de T_2, notées A^1 et A^2, et des actions $A_1, \ldots . A_n$ obtenues au moment du couplage des offres :

$$A \equiv \texttt{(} A_1\ \texttt{\&} \ldots\ A_n \texttt{)}\ \texttt{;}\ \texttt{(} A^1\ \texttt{\&}\ A^2 \texttt{)}$$

  On utilise l'opérateur "&" pour composer entre elles les actions qui portent sur des ensembles de variables disjoints

• pour exprimer la communication entre B_1 et B_2 on couple toutes les transitions (T_1, T_2) synchronisables. Une fois que tous ces couplages ont été effectués, toutes les transitions en attente de rendez-vous dans B_1 et dans B_2 sont détruites : elles ne feront pas partie du réseau de "B_1 *op* B_2".

Pour une même transition T_1 il peut exister plusieurs transitions T_2 telles que (T_1, T_2) soient synchronisables, et vice versa. On effectue tous les couplages possibles même si, ce faisant, on crée des transitions qui ne pourront jamais être franchies. La détection — même partielle — de ces transitions inutiles est un problème complexe, dont la solution passe par le calcul des propriétés de vivacité du réseau, et qui, pour cette raison, ne sera pas abordé au moment de la génération.

Exemple 6-1
Si B_1 et B_2 désignent le même comportement "G ; G ; **stop**", le réseau obtenu pour "B_1 || B_2" comportera quatre transitions étiquetées G dont deux sont inutiles, sans que cela reflète un blocage au niveau de la spécification LOTOS. ■

En revanche, si pour une transition T_1 en attente de rendez-vous dans B_1 il n'existe aucune transition T_2 de B_2 synchronisable avec T_1, ou vice versa, il s'agit certainement d'une erreur de spécification. A titre d'avertissement CÆSAR signale à l'utilisateur ces situations, caractéristiques de blocage, mais sans interrompre la génération.

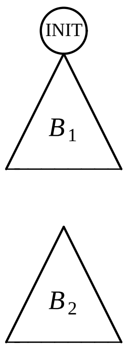
Exemple 6-2
Si *op* désigne l'opérateur "||", si B_1 est le comportement "G_1 ; G_2 ; **stop**" et si B_2 est le comportement "G_1 ; **stop**", on peut détecter le blocage par le fait que le réseau de B_2 ne comporte aucune transition étiquetée G_2. ■

Cette méthode ne suffit pas à trouver tous les blocages, mais seulement quelques-uns. Elle ne permet pas détecter les interblocages ni les blocages liés aux valeurs

Remarque 6-3
La prise en compte des rendez-vous n-aires s'effectue en appliquant successivement cette technique de génération à tous les opérateurs parallèles, sans qu'on ait à la modifier pour prendre en compte le cas où $n \neq 2$. ■

Remarque 6-4
On pourrait s'étonner qu'ayant créé une transition *fork* qui lance simultanément l'exécution B_1 et B_2, on ne crée aucune transition *join* pour terminer leur exécution. En effet si B_1 et B_2 doivent se terminer de manière synchrone, ils comportent des transitions étiquetées par une porte de la forme "$\delta_{\bullet p}$". Ces transitions sont en attente de rendez-vous et, après couplage, produisent des transitions de type *join*, c'est-à-dire ayant plus de places d'entrée que de places de sortie. ■

2. si *from_enable* = *true* : l'opérateur SUBLOTOS "*op*" provient de l'expansion d'un opérateur LOTOS ">>". Puisqu'il s'agit en fait d'une composition séquentielle, contrairement au cas précédent, on ne crée ni transition *fork* ni unités. On engendre le réseau de B_1 à partir de la place INIT. On engendre aussi le réseau de B_2 mais sans préciser à partir de quelle place, ce qui est inutile puisque, d'après l'algorithme d'expansion, B_2 commence avec un opérateur ";" pour lequel *from_enable* = *false*.



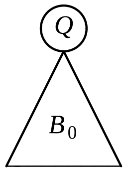
Le passage en séquence de B_1 à B_2 s'obtient simplement par application de la technique de couplage des transitions. L'opérateur "*op*" ne requiert la synchronisation que sur une seule porte, de la forme "$\delta_{\bullet p}$". Dans B_1 il peut y avoir plusieurs transitions T_1 en attente de rendez-vous (à la fin de B_1) alors que dans B_2 il n'y en a qu'une seule, T_2 (au début de B_2). Par couplage ces transitions sont fusionnées

6.1.5 Génération de l'opérateur "hide"

Le réseau correspondant à un comportement de la forme :

$$\textbf{hide } G_0, \ldots\ G_n \textbf{ in } B_0$$

est identique au réseau de B_0. Mais les transitions dont la porte fait partie de $\{G_0, \ldots\ G_n\}$ deviennent inaptes à la synchronisation et ne pourront désormais plus participer à des rendez-vous.

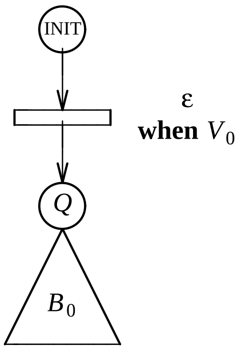


6.1.6 Génération de l'opérateur "->"

Pour traduire un comportement de la forme :

$$\texttt{[}V_0\texttt{] ->}\ B_0$$

on crée une place Q à partir de laquelle on engendre le réseau de B_0. On crée aussi une ε-transition allant de INIT à Q et portant l'action "**when** V_0".

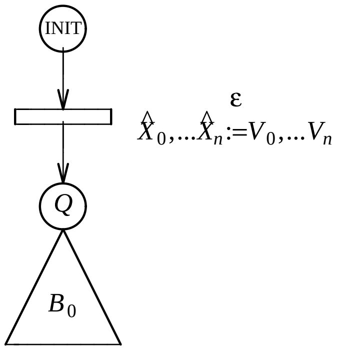


6.1.7 Génération de l'opérateur "let"

Pour traduire un comportement de la forme :

$$\textbf{let } \widehat{X_0}\!:\!S_0\texttt{=}V_0, \ldots\ \widehat{X_n}\!:\!S_n\texttt{=}V_n \textbf{ in } B_0$$

on crée une place Q à partir de laquelle on engendre le réseau de B_0. On crée aussi une ε-transition allant de INIT à Q et portant l'action "$\widehat{X_0}, \ldots\ \widehat{X_n} := V_0, \ldots\ V_n$". On donne au booléen *from_sync* la valeur *false* puisque cette affectation n'est pas due à une communication entre comportements concurrents.

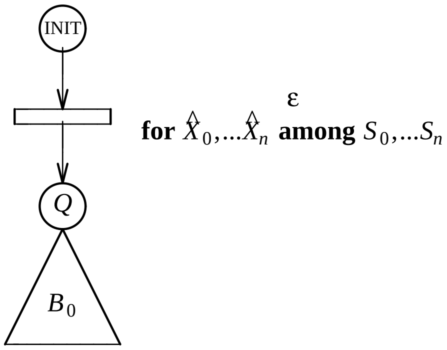


6.1.8 Génération de l'opérateur "choice"

Pour traduire un comportement de la forme :

$$\textbf{choice}\ \widehat{X_0}:S_0,\ldots\ \widehat{X_n}:S_n\ \texttt{[]}\ B_0$$

on crée une place Q à partir de laquelle on engendre le réseau de B_0. On crée aussi une ε-transition allant de INIT à Q et portant l'action "**for** $\widehat{X_0},\ldots\ \widehat{X_n}$ **among** $S_0,\ldots\ S_n$".

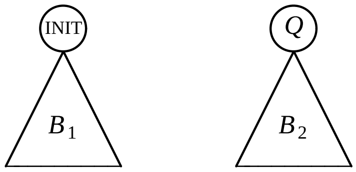


6.1.9 Génération de l'opérateur "[. . . >"

Pour traduire un comportement de la forme :

$$B_1\ \texttt{[}G\texttt{>}\ B_2$$

on crée une place Q. Dans un premier temps, on commence par engendrer le réseau de B_1 à partir de INIT et le réseau de B_2 à partir de Q.

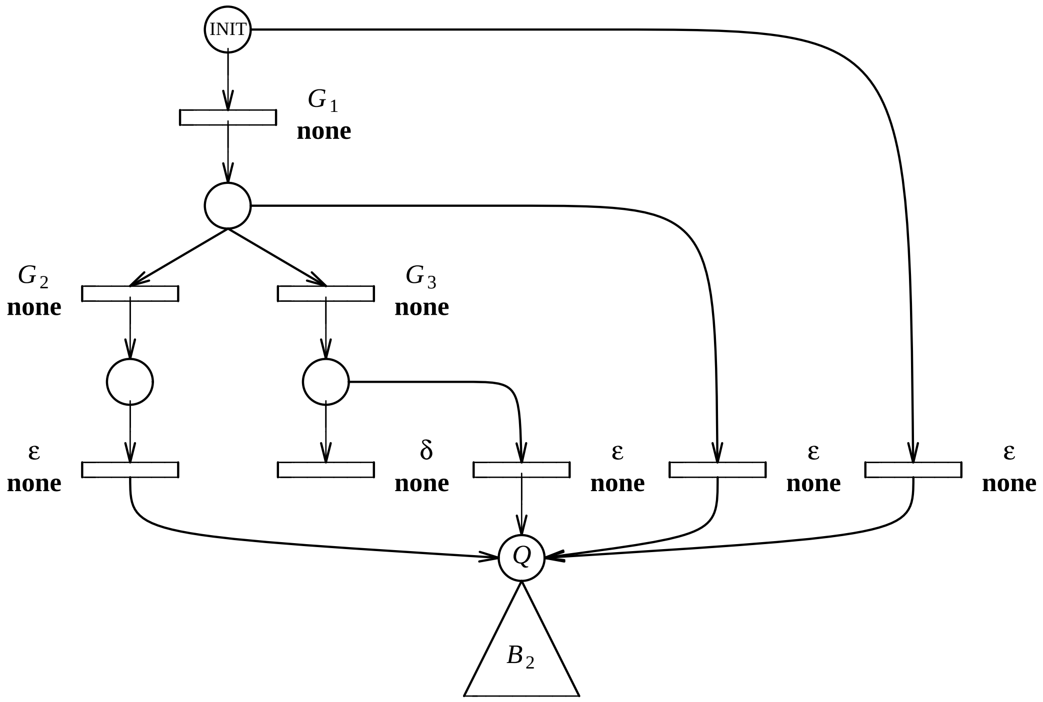


Puis, pour exprimer le fait que B_1 peut être interrompu pour exécuter B_2, on crée des ε-transitions allant des places de B_1 vers Q. Le nombre et la nature de ces transitions sont déterminés par le comportement de B_1.

Remarque 6-5
En revanche ces ε-transitions ne dépendent pas de la porte G (cette porte est toujours de la forme "$\delta_{\bullet p}$"). En effet, grâce au booléen *from_exit*, toutes les transitions de B_1 dont la porte est G n'ont aucune place de sortie. De fait cette porte ne sert qu'au moment de la définition formelle de la sémantique de SubLotos. ∎

- si B_1 est un comportement strictement séquentiel, il n'y a qu'une seule marque dans le réseau de B_1. Il suffit donc de créer autant d' ε-transitions qu'il y a de places dans le réseau de B_1.

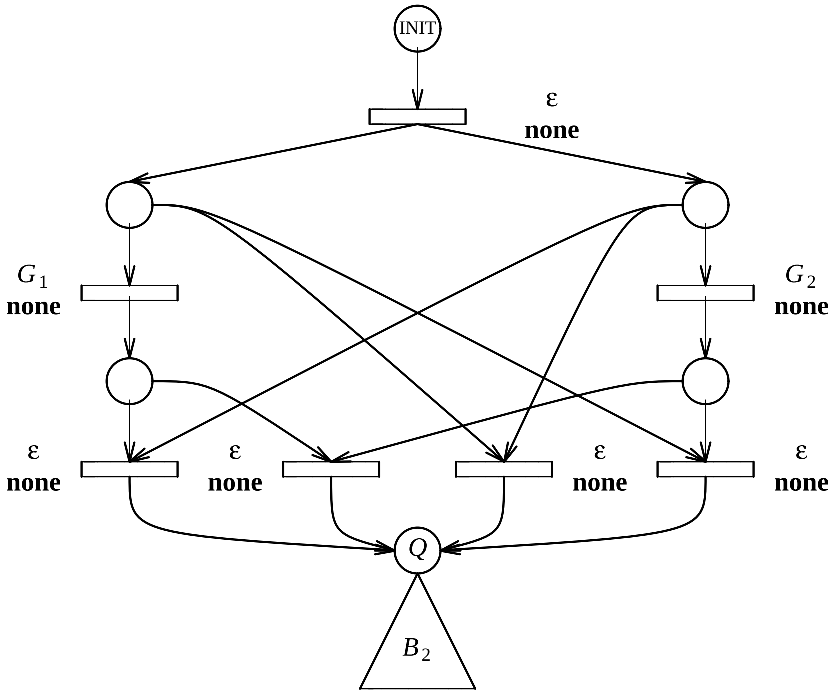
  Exemple 6-3
  Si B_1 est le comportement "G_1 **;** (${G_2}$ **; stop** [] G_3 **; exit**)" et si B_2 est un comportement quelconque on obtient le réseau suivant :



∎

- si B_1 est un comportement qui contient des sous-comportements devant être exécutés en parallèle, le problème est plus complexe puisqu'il peut y avoir plusieurs marques dans le réseau de B_1. Pour interrompre B_1 il faut enlever *simultanément* toutes les marques. Les ε-transitions que l'on crée ont alors plusieurs places d'entrée.

  Exemple 6-4
  Si B_1 est le comportement "G_1 **; stop** ||| G_2 **; stop**" et si B_2 est un comportement quelconque on obtient le réseau suivant :



Cette solution s'appuie sur l'absence de récursion à gauche de l'opérateur "`[>`" et de récursion à gauche et à droite des opérateurs parallèles, garantie par la propriété du contrôle statique (§ 3.1.5, p. 47) ■

- plus généralement, lorsque B_1 est un comportement quelconque, il faut déterminer un ensemble d'ensembles de places $\{\widetilde{Q_1}, \ldots \widetilde{Q_n}\}$ tel que, pour chaque $\widetilde{Q_i}$, on doive créer une ε-transition dont l'ensemble des places d'entrée est $\widetilde{Q_i}$. Cet ensemble est défini par récurrence sur la complexité du comportement B_1 ; les règles de calcul seront détaillées plus loin (§ 6.2, p. 129)

6.1.10 Génération de l'instanciation

On rappelle que tous les processus SubLotos sont récursifs. A chaque processus P on associe la place, notée *init*(P), à partir de laquelle le réseau correspondant au corps de P a été construit. Initialement cette quantité est indéfinie.

Lorsque l'on rencontre un comportement de la forme :

$$P \texttt{ [}G_0, \ldots\ G_m\texttt{] (}V_1, \ldots\ V_n\texttt{)}$$

il faut distinguer deux cas, suivant la valeur de *init*(P) :

1. si *init*(P) est encore indéfini : l'instanciation de P n'est pas récursive. On crée une place Q à partir de laquelle on engendre le réseau de *behaviour*(P). On donne à *init*(P) la valeur Q. On crée aussi une ε-transition T allant de INIT à Q et portant l'action "$var_1(P), \ldots\ var_n(P) := V_1, \ldots\ V_n$" en donnant au booléen *from_sync* la valeur *false*

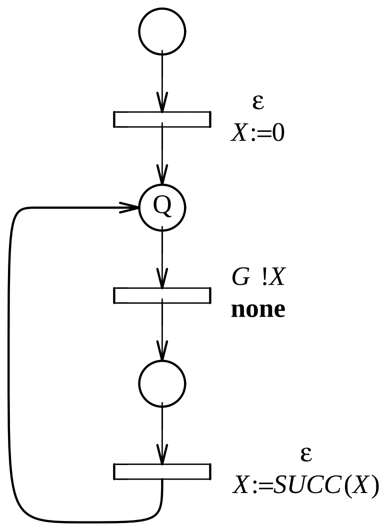
2. si *init*(P) est égal à une place Q : le réseau de *behaviour*(P) a déjà été construit à partir de la place Q. On crée alors une ε-transition T allant de INIT à Q et portant l'action "$var_1(P), \ldots\ var_n(P) := V_1, \ldots\ V_n$" en donnant au booléen *from_sync* la valeur *false*. On dit

que cette transition T est une *boucle* sur la place Q ; seules les instanciations récursives sont capables de créer des boucles dans le réseau

Exemple 6-5
On peut illustrer ces deux cas de figure avec le comportement :

P [G] (0)
where
process P [G] ($X : NAT$) :
G !X ; P [G] ($X + 1$)
endproc

dont le réseau correspondant est :



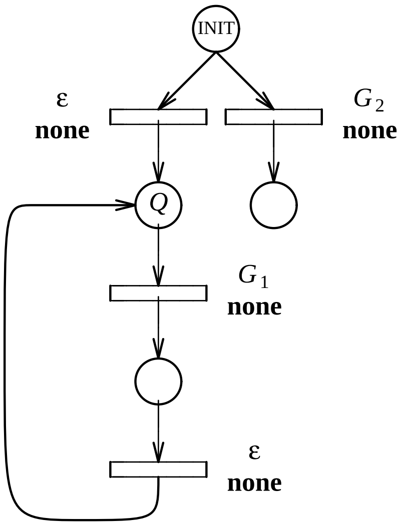
■

Dans le cas d'une instanciation non récursive sans paramètres valeurs (avec $n = 0$), il est possible d'engendrer le réseau de *behaviour*(P) directement à partir de la place INIT sans créer ni la place Q ni la transition T, ce qui constitue un gain appréciable. Attention ! Cette optimisation, parce qu'elle crée une boucle sur INIT, n'est pas correcte si plusieurs transitions sont issues de la place INIT, c'est-à-dire dans le cas où l'instanciation apparaît comme opérande de l'opérateur "[]".

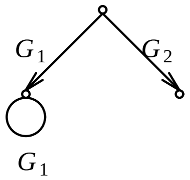
Exemple 6-6
Pour le comportement :

P [G_1] [] G_2 ; **stop**
where
process P [G_1] :
G_1 ; P [G_1]
endproc

on doit produire le réseau et le graphe suivants :

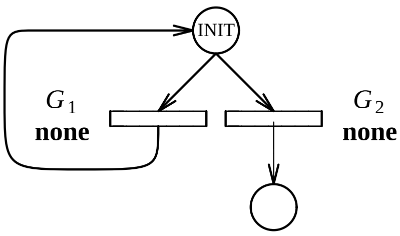


rèseau correct

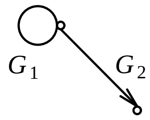


graphe correct

alors qu'en appliquant — à tort — l'optimisation précédente, on obtiendrait :



rèseau incorrect



graphe incorrect

■

6.1.11 Clôture des rendez-vous

A la fin de l'algorithme de génération, il faut transformer le réseau obtenu afin que les transitions ne comportent plus aucune offre de la forme "$\texttt{?}X\texttt{:}S$" puisque la sémantique du réseau n'a été définie que pour des offres de la forme "$\texttt{!}V$". La solution de ce problème consiste à *clore* les rendez-vous en procédant de la manière suivante :

- pour chaque transition T dont l'offre comporte des éléments de la forme "$\texttt{?}X_1\texttt{:}S_1, \ldots\ \texttt{?}X_n\texttt{:}S_n$" on préfixe l'action de T par "$\texttt{(}$**for** $X_1, \ldots\ X_n$ **among** $S_1, \ldots\ S_n\texttt{) \&}$"
- on remplace chaque offre de la forme "$\texttt{?}X_i\texttt{:}S_i$" par "$\texttt{!}X_i$". Ainsi le fait que la variable X_i décrive toutes les valeurs appartenant au domaine de la sorte S_i n'est plus exprimé par l'offre de T mais par l'action de T
- on remplace l'offre des transitions cachées par l'offre vide $\langle\rangle$

6.2 Algorithme de génération

Les règles de génération sont formulées par récurrence sur la complexité du comportement SubLotos dont on engendre le réseau. Elles sont décrites au moyen de grammaires attribuées. Auparavant les attributs et les notations utilisés par l'algorithme d'expansion sont présentés.

6.2.1 Attributs

L'algorithme de génération associe à chaque comportement SubLotos B un système de dix attributs ; pour chacun on indique son nom, son type, son mode d'évaluation (hérité, synthétisé ou local) et on précise sa signification intuitive :

- l'attribut hérité INIT est la place à partir de laquelle le réseau de B doit être construit. La valeur de INIT est indéfinie quand B est un opérateur "`;`" avec *from_enable = true*

- l'attribut synthétisé PLAC est l'ensemble des places du réseau de B, hormis INIT. Cet attribut est *croissant au sens de l'inclusion*, ce qui signifie que l'attribut PLAC renvoyé par un nœud père contient toujours les attributs PLAC synthétisés par ses nœuds fils

- l'attribut synthétisé UNIT est l'ensemble des unités du réseau de B. Cet attribut est croissant au sens de l'inclusion

- l'attribut synthétisé WAIT est l'ensemble des transitions visibles du réseau de B, c'est-à-dire les transitions en attente de rendez-vous. Cet attribut n'est pas croissant au sens de l'inclusion, à cause des opérateurs "`||`" et "**hide**"

- l'attribut synthétisé HIDE est l'ensemble des transitions cachées du réseau de B, c'est-à-dire les transitions dont la porte ne peut participer à aucune synchronisation avec l'environnement de B. Cet attribut est croissant au sens de l'inclusion

- l'attribut synthétisé NONE est l'ensemble des ε-transitions du réseau de B. Cet attribut est croissant au sens de l'inclusion. Les ensembles WAIT, HIDE et NONE constituent une partition de l'ensemble des transitions du réseau de B

- l'attribut synthétisé HOLD est l'ensemble des ensembles des places dont il faut "prendre" les marques pour exprimer l'interruption de B par un opérateur "`[`G`>`" (§ 6.1.9, p. 124).

  Les règles de calcul de cet attribut sont optimisées afin de minimiser le cardinal de cet ensemble, parfois très important puisqu'il nécessite des produits cartésiens d'ensembles. On n'y met pas toutes les places que l'on crée, notamment celles qui proviennent de la génération des opérateurs "**let**". "**choice**" et "`->`".

  De plus on n'évalue l'attribut HOLD que lorsque sa valeur est nécessaire, c'est-à-dire quand B apparaît en partie gauche d'un opérateur d'interruption (évaluation "paresseuse"). Dans le cas contraire la valeur de HOLD est indéfinie

- l'attribut hérité ABLE a une valeur booléenne qui vaut *true* si et seulement si l'attribut HOLD doit être calculé. Dans le cas contraire la valeur de HOLD doit être laissée indéfinie

- l'attribut hérité LOOP a une valeur booléenne qui vaut *true* si et seulement on peut optimiser la génération des instanciations non récursives (§ 6.1.10, p. 127). En revanche si LOOP vaut *false*, c'est-à-dire que B apparaît directement en partie gauche ou droite d'un opérateur "`[]`", cette optimisation ne doit pas être appliquée parce que le réseau de B ne doit pas contenir de boucle sur la place INIT

- un attribut local $init(P)$ est attaché à chaque processus SubLotos P qui est soit la place à partir de laquelle le réseau de $behaviour(P)$ a été engendré, soit la valeur spéciale "*indéfini*" si ce réseau n'a pas encore été construit. Cet attribut est initialisé à "*indéfini*"

6.2.2 Notations préliminaires

L'algorithme de génération utilise les notations suivantes :

- la fonction "$new\_place()$" crée et renvoie une nouvelle place, différente de toutes les places déjà créées

- la fonction "$new\_unit(\{Q_1, \dots\ Q_m\}, Q, \{U_1, \dots\ U_n\})$" renvoie une nouvelle unité U différente de toutes les unités déjà créées. Cette unité U contient les places $Q_1, \dots\ Q_m$, a pour place initiale Q ($Q \in \{Q_1, \dots\ Q_m\}$) et contient les unités $\{U_1, \dots\ U_n\}$:

$$\begin{cases} places^\star(U) = \{Q_1, \dots\ Q_m\} \\ first(U) = Q \\ units^\star(U) = \{U_1, \dots\ U_n\} \end{cases}$$

- la fonction "$new\_trans(\{Q_1, \dots\ Q_m\}, \{Q'_1, \dots\ Q'_n\}, G, \widehat{O}, A)$" crée et renvoie une nouvelle transition T, différente de toutes les transitions déjà créées. Cette transition T a pour places d'entrée $Q_1, \dots\ Q_m$, pour places de sortie $Q'_1, \dots\ Q'_n$, pour porte G, pour offre $\widehat{O}$ et pour action A :

$$\begin{cases} in(T) = \{Q_1, \dots\ Q_m\} \\ out(T) = \{Q'_1, \dots\ Q'_n\} \\ gate(T) = G \\ offer(T) = \widehat{O} \\ action(T) = A \end{cases}$$

- la fonction "$kind(\widehat{O})$" renvoie le n-uplet formé par les sortes des offres de la liste $\widehat{O}$:

$$kind(\langle O_1, \dots\ O_n\rangle) = \langle S'_1, \dots\ S'_n\rangle$$

  où :

$$S'_i = \begin{cases} \textbf{si } O_i \equiv\ !V_i \textbf{ alors } sort(V_i) \\ \textbf{si } O_i \equiv\ ?X_i\!:\!S_i \textbf{ alors } S_i \end{cases}$$

- la fonction "$sync(T_1, T_2)$" renvoie un résultat booléen qui vaut *true* si et seulement les transitions T_1 et T_2 peuvent se synchroniser au sens du rendez-vous :

$$sync(T_1, T_2) = (gate(T_1) = gate(T_2)) \ \wedge\ (kind(offer(T_1)) = kind(offer(T_2)))$$

- la fonction "$couple(T_1, T_2)$" crée et renvoie une nouvelle transition T, différente de toutes les transitions déjà créées. Cette transition T est obtenue par *couplage* des transitions T_1 et T_2 — qui doivent pouvoir être synchronisées :

$$couple(T_1, T_2) = new\_trans(\widetilde{Q}, \widetilde{Q'}, G, \widehat{O}, A)$$

où :

$$\left\{\begin{array}{l}
\widetilde{Q} = in(T_1) \cup in(T_2) \\
\widetilde{Q'} = out(T_1) \cup out(T_2) \\
G = gate(T_1) = gate(T_2) \\
\textbf{soient } O^1_1, \ldots\ O^1_n \textbf{ tels que } \mathit{offer}(T_1) = \langle O^1_1, \ldots\ O^1_n \rangle \\
\textbf{soient } O^2_1, \ldots\ O^2_n \textbf{ tels que } \mathit{offer}(T_2) = \langle O^2_1, \ldots\ O^2_n \rangle \\
\textbf{soient } O_1, \ldots\ O_n, A_1, \ldots\ A_n \textbf{ tels que} \\
\quad \textbf{si } (O^1_i \equiv \texttt{!}V_1) \ \wedge \ (O^2_i \equiv \texttt{!}V_2) \textbf{ alors} \\
\quad\quad O_i \equiv \texttt{!}V_1 \\
\quad\quad A_i \equiv (\textbf{when } V_1 \texttt{ = } V_2) \\
\quad \textbf{sinon si } (O^1_i \equiv \texttt{!}V_1) \ \wedge \ (O^2_i \equiv \texttt{?}X_2\texttt{:}S_2) \textbf{ alors} \\
\quad\quad O_i \equiv \texttt{!}V_1 \\
\quad\quad A_i \equiv (X_2 := V_1 \ \mathit{(true)}) \\
\quad \textbf{sinon si } (O^1_i \equiv \texttt{?}X_1\texttt{:}S_1) \ \wedge \ (O^2_i \equiv \texttt{!}V_2) \textbf{ alors} \\
\quad\quad O_i \equiv \texttt{!}V_2 \\
\quad\quad A_i \equiv (X_1 := V_2 \ \mathit{(true)}) \\
\quad \textbf{sinon si } (O^1_i \equiv \texttt{?}X_1\texttt{:}S_1) \ \wedge \ (O^2_i \equiv \texttt{?}X_2\texttt{:}S_2) \textbf{ alors} \\
\quad\quad O_i \equiv \texttt{?}X_2\texttt{:}S_2 \\
\quad\quad A_i \equiv (X_1 := X_2 \ \mathit{(true)}) \\
\widehat{O} = \langle O_1, \ldots\ O_n \rangle \\
A \equiv (A_1 \ \texttt{\&} \ldots\ A_n) \ \texttt{;} \ (\mathit{action}(T_1) \ \texttt{\&} \ \mathit{action}(T_2))
\end{array}\right.$$

- la fonction "*flat_wait*(T)" renvoie une transition T' obtenue à partir de la transition T — dont la porte doit être visible — après clôture des rendez-vous. On remplace chaque offre de la forme "$\texttt{?}X_i\texttt{:}S_i$" par une offre de la forme "$\texttt{!}X_i$" et on fait décrire à la variable X_i le domaine de la sorte S_i :

$$\mathit{flat\_wait}(T) = T'$$

où :

$$\left\{\begin{array}{l}
in(T') = in(T) \\
out(T') = out(T) \\
gate(T') = gate(T) \\
\textbf{soient } O_1, \ldots\ O_n \textbf{ tels que } \mathit{offer}(T) = \langle O_1, \ldots\ O_n \rangle \\
\textbf{soient } O'_1, \ldots\ O'_n, A'_1, \ldots\ A'_n \textbf{ tels que} \\
\quad \textbf{si } O_i \equiv \texttt{!}V_i \textbf{ alors} \\
\quad\quad O'_i \equiv \texttt{!}V_i \\
\quad\quad A'_i \equiv \textbf{none} \\
\quad \textbf{sinon si } O_i \equiv \texttt{?}X_i\texttt{:}S_i \textbf{ alors} \\
\quad\quad O'_i \equiv \texttt{!}X_i \\
\quad\quad A'_i \equiv (\textbf{for } X_i \textbf{ among } S_i) \\
\mathit{offer}(T') = \langle O'_1, \ldots\ O'_n \rangle \\
\mathit{action}(T') = (A'_1 \ \texttt{\&} \ldots\ A'_n) \ \texttt{;} \ \mathit{action}(T)
\end{array}\right.$$

- la fonction "*flat_hide*(T)" renvoie une transition T' obtenue à partir de la transition T — dont la porte doit être cachée — après clôture des rendez-vous. On élimine chaque offre de la forme "$\texttt{?}X_i\texttt{:}S_i$" et on fait décrire à la variable X_i le domaine de la sorte S_i :

$$\mathit{flat\_hide}(T) = T'$$

où :

$$\begin{cases} in(T') = in(T) \\ out(T') = out(T) \\ gate(T') = gate(T) \\ \textbf{soient } O_1, \ldots\ O_n \textbf{ tels que } \mathit{offer}(T) = \langle O_1, \ldots\ O_n \rangle \\ \quad \textbf{si } O_i \equiv \texttt{!}V_i \textbf{ alors} \\ \quad\quad A'_i \equiv \textbf{none} \\ \quad \textbf{sinon si } O_i \equiv \texttt{?}X_i \texttt{:} S_i \textbf{ alors} \\ \quad\quad A'_i \equiv (\textbf{for } X_i \textbf{ among } S_i) \\ \mathit{offer}(T') = \langle\rangle \\ action(T') = (A'_1\ \texttt{\&} \ldots\ A'_n)\ \texttt{;}\ action(T) \end{cases}$$

Enfin on peut remarquer que tous les opérateurs d'union "$\cup$" figurant dans l'algorithme de génération sont en fait des opérateurs d'*union disjointe* : lorsque l'on écrit $X \cup Y$, l'invariant $X \cap Y = \emptyset$ est implicitement vérifié.

6.2.3 Expressions de comportement

L'algorithme de génération est formellement défini par la grammaire attribuée suivante qui définit les règles d'évaluation des attributs sur les expressions de comportement SubLotos :

$B \downarrow \textsc{init} \downarrow \textsc{loop} \downarrow \textsc{able} \uparrow \textsc{plac} \uparrow \textsc{unit} \uparrow \textsc{wait} \uparrow \textsc{hide} \uparrow \textsc{none} \uparrow \textsc{hold} \equiv$

stop

$$\begin{cases} \textsc{plac} := \emptyset \\ \textsc{unit} := \emptyset \\ \textsc{wait} := \emptyset \\ \textsc{hide} := \emptyset \\ \textsc{none} := \emptyset \\ \textbf{si } \textsc{able} \textbf{ alors } \textsc{hold} := \{\{\textsc{init}\}\} \end{cases}$$

$|\ G\ O_1, \ldots\ O_n\ \texttt{[}[V_0]\texttt{]}\ \texttt{;}$

$B_0 \downarrow Q \downarrow true \downarrow \textsc{able} \uparrow \textsc{plac}_0 \uparrow \textsc{unit}_0 \uparrow \textsc{wait}_0 \uparrow \textsc{hide}_0 \uparrow \textsc{none}_0 \uparrow \textsc{hold}_0$ *(from_exit, from_enable)*

$$\begin{cases} \textbf{si } V_0 \textit{ existe} \textbf{ alors } A := (\textbf{when } V_0) \textbf{ sinon } A := \textbf{none} \\ \textbf{si } \textit{from\_exit} \textbf{ alors} \\ \quad T := \textit{new\_trans}(\{\textsc{init}\},\ \emptyset, G, \langle O_1, \ldots\ O_n \rangle, A) \\ \quad \textsc{plac} := \textsc{plac}_0 \\ \quad \textbf{si } \textsc{able} \textbf{ alors } \textsc{hold} := \textsc{hold}_0 \cup \{\{\textsc{init}\}\} \\ \textbf{sinon si } \textit{from\_enable} \textbf{ alors} \\ \quad Q := \textit{new\_place}() \\ \quad T := \textit{new\_trans}(\ \emptyset, \{Q\}, G, \langle O_1, \ldots\ O_n \rangle, A) \\ \quad \textsc{plac} := \textsc{plac}_0 \cup \{Q\} \\ \quad \textbf{si } \textsc{able} \textbf{ alors } \textsc{hold} := \textsc{hold}_0 \\ \textbf{sinon} \\ \quad Q := \textit{new\_place}() \\ \quad T := \textit{new\_trans}(\{\textsc{init}\}, \{Q\}, G, \langle O_1, \ldots\ O_n \rangle, A) \\ \quad \textsc{plac} := \textsc{plac}_0 \cup \{Q\} \\ \quad \textbf{si } \textsc{able} \textbf{ alors } \textsc{hold} := \textsc{hold}_0 \cup \{\{\textsc{init}\}\} \\ \textsc{unit} := \textsc{unit}_0 \\ \textbf{si } G = \mathbf{i}_{\bullet 1} \textbf{ alors } \textsc{wait} := \textsc{wait}_0 \textbf{ sinon } \textsc{wait} := \textsc{wait}_0 \cup \{T\} \\ \textbf{si } G = \mathbf{i}_{\bullet 1} \textbf{ alors } \textsc{hide} := \textsc{hide}_0 \cup \{T\} \textbf{ sinon } \textsc{hide} := \textsc{hide}_0 \\ \textsc{none} := \textsc{none}_0 \end{cases}$$

$| \ B_1 \downarrow \textsc{init} \downarrow \mathit{false} \downarrow \textsc{able} \uparrow \textsc{plac}_1 \uparrow \textsc{unit}_1 \uparrow \textsc{wait}_1 \uparrow \textsc{hide}_1 \uparrow \textsc{none}_1 \uparrow \textsc{hold}_1$ `[]`
$B_2 \downarrow \textsc{init} \downarrow \mathit{false} \downarrow \textsc{able} \uparrow \textsc{plac}_2 \uparrow \textsc{unit}_2 \uparrow \textsc{wait}_2 \uparrow \textsc{hide}_2 \uparrow \textsc{none}_2 \uparrow \textsc{hold}_2$

$$\left\{\begin{array}{l}
\textsc{plac} := \textsc{plac}_1 \cup \textsc{plac}_2 \\
\textsc{unit} := \textsc{unit}_1 \cup \textsc{unit}_2 \\
\textsc{wait} := \textsc{wait}_1 \cup \textsc{wait}_2 \\
\textsc{hide} := \textsc{hide}_1 \cup \textsc{hide}_2 \\
\textsc{none} := \textsc{none}_1 \cup \textsc{none}_2 \\
\textbf{si } \textsc{able} \textbf{ alors} \\
\quad \textbf{si } \textsc{hold}_1 = \{\{\textsc{init}\}\} \textbf{ alors } \textsc{hold} := \textsc{hold}_2 \\
\quad \textbf{sinon si } \textsc{hold}_2 = \{\{\textsc{init}\}\} \textbf{ alors } \textsc{hold} := \textsc{hold}_1 \\
\quad \textbf{sinon si } (\{\textsc{init}\} \notin \textsc{hold}_1) \ \vee \ (\{\textsc{init}\} \notin \textsc{hold}_2) \textbf{ alors } \textsc{hold} := \textsc{hold}_1 \cup \textsc{hold}_2 \\
\quad \textbf{sinon } \textsc{hold} := (\textsc{hold}_1 - \{\{\textsc{init}\}\}) \cup (\textsc{hold}_2 - \{\{\textsc{init}\}\}) \cup \{\{\textsc{init}\}\}
\end{array}\right.$$

$| \ B_1 \downarrow Q_1 \downarrow \textsc{loop}_1 \downarrow \textsc{able} \uparrow \textsc{plac}_1 \uparrow \textsc{unit}_1 \uparrow \textsc{wait}_1 \uparrow \textsc{hide}_1 \uparrow \textsc{none}_1 \uparrow \textsc{hold}_1$ *op*
$B_2 \downarrow Q_2 \downarrow \textsc{loop}_2 \downarrow \textsc{able} \uparrow \textsc{plac}_2 \uparrow \textsc{unit}_2 \uparrow \textsc{wait}_2 \uparrow \textsc{hide}_2 \uparrow \textsc{none}_2 \uparrow \textsc{hold}_2$ *(from_enable)*

$$\left\{\begin{array}{l}
\textbf{si } \mathit{from_enable} \textbf{ alors} \\
\quad Q_1 := \textsc{init} \\
\quad Q_2 := \mathit{indéfini} \\
\quad \textsc{loop}_1 := \textsc{loop} \\
\quad \textsc{loop}_2 := \mathit{indéfini} \\
\quad \textsc{plac} := \textsc{plac}_1 \cup \textsc{plac}_2 \\
\quad \textsc{unit} := \textsc{unit}_1 \cup \textsc{unit}_2 \\
\quad \textsc{none} := \textsc{none}_1 \cup \textsc{none}_2 \\
\quad \textbf{si } \textsc{able} \textbf{ alors } \textsc{hold} := \textsc{hold}_1 \cup \textsc{hold}_2 \\
\textbf{sinon} \\
\quad Q_1 := \mathit{new_place}() \\
\quad Q_2 := \mathit{new_place}() \\
\quad \textsc{loop}_1 := \mathit{true} \\
\quad \textsc{loop}_2 := \mathit{true} \\
\quad U_1 := \mathit{new_unit}(\textsc{plac}_1 \cup \{Q_1\}, Q_1, \textsc{unit}_1) \\
\quad U_2 := \mathit{new_unit}(\textsc{plac}_2 \cup \{Q_2\}, Q_2, \textsc{unit}_2) \\
\quad T := \mathit{new_trans}(\{\textsc{init}\}, \{\textsc{init}_1, \textsc{init}_2\}, \varepsilon, \langle\rangle, \textbf{none}) \\
\quad \textsc{plac} := \textsc{plac}_1 \cup \textsc{plac}_2 \cup \{Q_1, Q_2\} \\
\quad \textsc{unit} := \textsc{unit}_1 \cup \textsc{unit}_2 \cup \{U_1, U_2\} \\
\quad \textsc{none} := \textsc{none}_1 \cup \textsc{none}_2 \cup \{T\} \\
\quad \textbf{si } \textsc{able} \textbf{ alors } \textsc{hold} := \{\widetilde{Q_1} \cup \widetilde{Q_2} \mid (\widetilde{Q1}, \widetilde{Q_2}) \in \textsc{hold}_1 \times \textsc{hold}_2\} \\
\textbf{si } op \equiv \texttt{||} \textbf{ alors} \\
\quad \textsc{wait} := \{\mathit{couple}(T_1, T_2) \mid ((T_1, T_2) \in \textsc{wait}_1 \times \textsc{wait}_2) \ \wedge \ \mathit{sync}(T_1, T_2)\} \\
\textbf{sinon si } op \equiv \texttt{|[} G_0, \dots \ G_n \texttt{]|} \textbf{ alors} \\
\quad \widetilde{T_1} := \{T \mid (T \in \textsc{wait}_1) \ \wedge \ (\mathit{gate}(T) \notin \{G_0, \dots \ G_n\})\} \\
\quad \widetilde{T_2} := \{T \mid (T \in \textsc{wait}_2) \ \wedge \ (\mathit{gate}(T) \notin \{G_0, \dots \ G_n\})\} \\
\quad \textsc{wait} := \widetilde{T_1} \cup \widetilde{T_2} \cup \\
\quad \{\mathit{couple}(T_1, T_2) \mid ((T_1, T_2) \in (\textsc{wait}_1 - \widetilde{T_1}) \times (\textsc{wait}_2 - \widetilde{T_2})) \ \wedge \ \mathit{sync}(T_1, T_2)\} \\
\textsc{hide} := \textsc{hide}_1 \cup \textsc{hide}_2
\end{array}\right.$$

$| \; \textbf{hide}\ G_0, \ldots\ G_n\ \textbf{in}\ B_0 \downarrow \textsc{init} \downarrow \textsc{loop} \downarrow \textsc{able} \uparrow \textsc{plac}_0 \uparrow \textsc{unit}_0 \uparrow \textsc{wait}_0 \uparrow \textsc{hide}_0 \uparrow \textsc{none}_0 \uparrow \textsc{hold}_0$

$$\left\{ \begin{array}{l} \textsc{plac} \ := \ \textsc{plac}_0 \\ \textsc{unit} \ := \ \textsc{unit}_0 \\ \widetilde{T} \ := \ \{T \mid (T \in \textsc{wait}_0) \ \wedge \ (\mathit{gate}(T) \in \{G_0, \ldots\ G_n\})\} \\ \textsc{wait} \ := \ \textsc{wait}_0 - \widetilde{T} \\ \textsc{hide} \ := \ \textsc{hide}_0 \cup \widetilde{T} \\ \textsc{none} \ := \ \textsc{none}_0 \\ \textbf{si}\ \textsc{able}\ \textbf{alors}\ \textsc{hold} \ := \ \textsc{hold}_0 \end{array} \right.$$

$| \; \texttt{[}V_0\texttt{] ->}\ B_0 \downarrow Q \downarrow \mathit{true} \downarrow \textsc{able} \uparrow \textsc{plac}_0 \uparrow \textsc{unit}_0 \uparrow \textsc{wait}_0 \uparrow \textsc{hide}_0 \uparrow \textsc{none}_0 \uparrow \textsc{hold}_0$

$$\left\{ \begin{array}{l} Q \ := \ \mathit{new\_place}() \\ T \ := \ \mathit{new\_trans}(\{\textsc{init}\}, \{Q\}, \varepsilon, \langle\rangle, (\textbf{when}\ V_0)) \\ \textsc{plac} \ := \ \textsc{plac}_0 \cup \{Q\} \\ \textsc{unit} \ := \ \textsc{unit}_0 \\ \textsc{wait} \ := \ \textsc{wait}_0 \\ \textsc{hide} \ := \ \textsc{hide}_0 \\ \textsc{none} \ := \ \textsc{none}_0 \cup \{T\} \\ \textbf{si}\ \textsc{able}\ \textbf{alors}\ \textsc{hold} \ := \ \textsc{hold}_0 \cup \{\{\textsc{init}\}\} \end{array} \right.$$

$| \; \textbf{let}\ \widehat{X_0}\!:\!S_0\texttt{=}V_0, \ldots\ \widehat{X_n}\!:\!S_n\texttt{=}V_n\ \textbf{in}$
$B_0 \downarrow Q \downarrow \mathit{true} \downarrow \textsc{able} \uparrow \textsc{plac}_0 \uparrow \textsc{unit}_0 \uparrow \textsc{wait}_0 \uparrow \textsc{hide}_0 \uparrow \textsc{none}_0 \uparrow \textsc{hold}_0$

$$\left\{ \begin{array}{l} Q \ := \ \mathit{new\_place}() \\ T \ := \ \mathit{new\_trans}(\{\textsc{init}\}, \{Q\}, \varepsilon, \langle\rangle, (\widehat{X_0}, \ldots\ \widehat{X_n} := V_0, \ldots\ V_n\ \ \mathit{(false)})) \\ \textsc{plac} \ := \ \textsc{plac}_0 \cup \{Q\} \\ \textsc{unit} \ := \ \textsc{unit}_0 \\ \textsc{wait} \ := \ \textsc{wait}_0 \\ \textsc{hide} \ := \ \textsc{hide}_0 \\ \textsc{none} \ := \ \textsc{none}_0 \cup \{T\} \\ \textbf{si}\ \textsc{able}\ \textbf{alors}\ \textsc{hold} \ := \ \textsc{hold}_0 \end{array} \right.$$

$| \; \textbf{choice}\ \widehat{X_0}\!:\!S_0, \ldots\ \widehat{X_n}\!:\!S_n\ \texttt{[]}$
$B_0 \downarrow Q \downarrow \mathit{true} \downarrow \textsc{able} \uparrow \textsc{plac}_0 \uparrow \textsc{unit}_0 \uparrow \textsc{wait}_0 \uparrow \textsc{hide}_0 \uparrow \textsc{none}_0 \uparrow \textsc{hold}_0$

$$\left\{ \begin{array}{l} Q \ := \ \mathit{new\_place}() \\ T \ := \ \mathit{new\_trans}(\{\textsc{init}\}, \{Q\}, \varepsilon, \langle\rangle, (\textbf{for}\ \widehat{X_0}, \ldots\ \widehat{X_n}\ \textbf{among}\ S_0, \ldots\ S_n)) \\ \textsc{plac} \ := \ \textsc{plac}_0 \cup \{Q\} \\ \textsc{unit} \ := \ \textsc{unit}_0 \\ \textsc{wait} \ := \ \textsc{wait}_0 \\ \textsc{hide} \ := \ \textsc{hide}_0 \\ \textsc{none} \ := \ \textsc{none}_0 \cup \{T\} \\ \textbf{si}\ \textsc{able}\ \textbf{alors}\ \textsc{hold} \ := \ \textsc{hold}_0 \end{array} \right.$$

$| \; B_1 \downarrow \textsc{init} \downarrow \textsc{loop} \downarrow \mathit{true} \uparrow \textsc{plac}_1 \uparrow \textsc{unit}_1 \uparrow \textsc{wait}_1 \uparrow \textsc{hide}_1 \uparrow \textsc{none}_1 \uparrow \textsc{hold}_1$ `[G>`
$B_2 \downarrow Q \downarrow \mathit{true} \downarrow \textsc{able} \uparrow \textsc{plac}_2 \uparrow \textsc{unit}_2 \uparrow \textsc{wait}_2 \uparrow \textsc{hide}_2 \uparrow \textsc{none}_2 \uparrow \textsc{hold}_2$

$$\left\{\begin{array}{l}
Q \;:=\; \mathit{new_place}() \\
\textsc{plac} \;:=\; \textsc{plac}_1 \cup \textsc{plac}_2 \cup \{Q\} \\
\textsc{unit} \;:=\; \textsc{unit}_1 \cup \textsc{unit}_2 \\
\textsc{wait} \;:=\; \textsc{wait}_1 \cup \textsc{wait}_2 \\
\textsc{hide} \;:=\; \textsc{hide}_1 \cup \textsc{hide}_2 \\
\textsc{none} \;:=\; \textsc{none}_1 \cup \textsc{none}_2 \cup \left(\bigcup_{\widetilde{Q} \in \textsc{hold}_1} \{\mathit{new_trans}(\widetilde{Q}, \{Q\}, \varepsilon, \langle\rangle, \mathbf{none})\}\right) \\
\textbf{si } \textsc{able} \textbf{ alors} \\
\quad \textbf{si } \{\textsc{init}\} \in \textsc{hold}_1 \textbf{ alors } \textsc{hold} \;:=\; (\textsc{hold}_1 - \{\{\textsc{init}\}\}) \cup \textsc{hold}_2 \\
\quad \textbf{sinon } \textsc{hold} \;:=\; \textsc{hold}_1 \cup \textsc{hold}_2
\end{array}\right.$$

$| \; P$ `[`$G_0, \ldots\ G_m$`]` `(`$V_1, \ldots\ V_n$`)`

$$\left\{\begin{array}{l}
A \;:=\; \mathit{var}_1(P), \ldots\ \mathit{var}_n(P) := V_1, \ldots\ V_n \ \mathit{(false)} \\
\textbf{si } \mathit{init}(P) = \mathit{indéfini} \textbf{ alors} \\
\quad \textbf{si } \textsc{loop} \wedge (n = 0) \textbf{ alors} \\
\quad\quad \mathit{init}(P) \;:=\; \textsc{init} \\
\quad\quad \textbf{soient } \textsc{plac}, \textsc{unit}, \textsc{wait}, \textsc{hide}, \textsc{none}, \textsc{hold} \textbf{ définis par} \\
\quad\quad \mathit{behaviour}(P) \downarrow \textsc{init} \downarrow \mathit{true} \downarrow \textsc{able} \uparrow \textsc{plac} \uparrow \textsc{unit} \uparrow \textsc{wait} \uparrow \textsc{hide} \uparrow \textsc{none} \uparrow \textsc{hold} \\
\quad \textbf{sinon} \\
\quad\quad Q \;:=\; \mathit{new_place}() \\
\quad\quad \mathit{init}(P) \;:=\; Q \\
\quad\quad T \;:=\; \mathit{new_trans}(\{\textsc{init}\}, \{Q\}, \varepsilon, \langle\rangle, A) \\
\quad\quad \textbf{soient } \textsc{plac}_0, \textsc{unit}, \textsc{wait}, \textsc{hide}, \textsc{none}_0, \textsc{hold} \textbf{ définis par} \\
\quad\quad \mathit{behaviour}(P) \downarrow Q \downarrow \mathit{true} \downarrow \textsc{able} \uparrow \textsc{plac}_0 \uparrow \textsc{unit} \uparrow \textsc{wait} \uparrow \textsc{hide} \uparrow \textsc{none}_0 \uparrow \textsc{hold} \\
\quad\quad \textsc{plac} \;:=\; \textsc{plac}_0 \cup \{Q\} \\
\quad\quad \textsc{none} \;:=\; \textsc{none}_0 \cup \{T\} \\
\textbf{sinon} \\
\quad T \;:=\; \mathit{new_trans}(\{\textsc{init}\}, \{\mathit{init}(P)\}, \varepsilon, \langle\rangle, A) \\
\quad \textsc{plac} \;:=\; \emptyset \\
\quad \textsc{unit} \;:=\; \emptyset \\
\quad \textsc{wait} \;:=\; \emptyset \\
\quad \textsc{hide} \;:=\; \emptyset \\
\quad \textsc{none} \;:=\; \{T\} \\
\quad \textbf{si } \textsc{able} \textbf{ alors } \textsc{hold} \;:=\; \{\{\textsc{init}\}\}
\end{array}\right.$$

6.2.4 Construction du réseau

A partir de la spécification SubLotos Λ on construit le réseau $(\mathcal{Q}, \mathcal{U}, \mathcal{T}, \mathcal{G}, \mathcal{X}, \mathcal{S}, \mathcal{F})$ où :

$$\left\{\begin{array}{l}
Q \;:=\; \mathit{new_place}() \\
\textbf{soient } \textsc{plac}, \textsc{unit}, \textsc{wait}, \textsc{hide}, \textsc{none}, \textsc{hold} \textbf{ définis par} \\
\quad \mathit{behaviour}(\Lambda) \downarrow Q \downarrow \mathit{true} \downarrow \mathit{false} \uparrow \textsc{plac} \uparrow \textsc{unit} \uparrow \textsc{wait} \uparrow \textsc{hide} \uparrow \textsc{none} \uparrow \textsc{hold} \\
\mathcal{Q} \;:=\; \textsc{plac} \cup \{Q\} \\
\mathcal{U} \;:=\; \mathit{new_unit}(\mathcal{Q}, Q, \textsc{unit}) \\
\mathcal{T} \;:=\; \{\mathit{flat_wait}(T) \mid T \in \textsc{wait}\} \cup \{\mathit{flat_hide}(T) \mid T \in \textsc{hide}\} \cup \textsc{none}
\end{array}\right.$$

et où :

- $\mathcal{G}$ est l'ensemble des portes SubLotos présentes dans Λ
- $\mathcal{X}$ est l'ensemble des variables SubLotos présentes dans Λ

- $\mathcal{S}$ est l'ensemble des sortes SubLotos présentes dans Λ
- $\mathcal{F}$ est l'ensemble des opérations SubLotos présentes dans Λ

6.3 Implémentation

Comme pour l'algorithme d'expansion, la phase de génération du logiciel Cæsar, programmée en langage C, a été dérivée de la grammaire attribuée proposée ici. Pour assurer l'efficacité de la réalisation, les propriétés suivantes de l'algorithme ont été mises à profit :

- tous les attributs sont évalués en un seul passage sur l'arbre abstrait SubLotos
- pour éviter le coût des recopies de paramètres, tous les attributs synthétisés sont implémentés comme des variables globales
- le fait que toutes les unions d'ensembles soit disjointes permet d'implémenter efficacement ces opérations lorsque les ensembles sont représentés par des listes chaînées

Chapitre 7

Optimisation

La phase d'optimisation, qui prend place immédiatement après la phase de génération, a pour objectif de transformer le réseau obtenu pour en réduire la taille (c'est-à-dire le nombre d'unités, de places, de transitions, de variables) tout en préservant la sémantique du réseau, définie pour la relation d'équivalence forte (§ 1.3.1, p. 23). L'optimisation permet donc d'obtenir un réseau plus simple, ce qui rend la phase de simulation moins coûteuse en temps et en mémoire.

On présente une liste non exhaustive d'optimisations dont on décrit les conditions d'application. Des données numériques sur les simplifications obtenues, fournies par le système Cæsar, justifient l'intérêt de ces optimisations. On suggère enfin d'autres optimisations susceptibles d'être appliquées au réseau.

7.1 Principes de l'optimisation

L'ensemble des différentes optimisations mises en œuvre ne procède pas d'une théorie unifiée mais d'une approche pragmatique. Sur des exemples précis on cherche les améliorations pouvant être apportées à la structure du réseau. Elles sont de deux ordres :

partie contrôle : ces optimisations (notées E0, E1, E2, ...) visent à supprimer des unités, des places et des transitions du réseau. La destruction de places et d'unités permet de diminuer la place mémoire occupée par les marquages pendant la simulation, tandis que la réduction du nombre de transitions, notamment des ε-transitions, produit des gains importants en temps de calcul

partie données : ces optimisations (notées V0, V1, V2, ...) tendent à simplifier les actions qui étiquettent les transitions et à supprimer certaines variables d'état, élimination qui entraîne une diminution substantielle de la taille mémoire nécessaire pour représenter les contextes

Pourquoi dissocier génération et optimisation ? N'est-il pas possible d'engendrer directement un réseau optimal ? A ces questions, les éléments de réponse suivants peuvent être apportés :

- la génération effectue déjà certaines optimisations ; par exemple l'attribut loop permet de ne créer des ε-transitions pour traduire l'opérateur "`[]`" que lorsque c'est indispensable
- certaines optimisations locales, dont celles qui éliminent les ε-transitions, pourraient probablement être faites dès la génération. Mais il faudrait pour cela modifier l'algorithme de génération,

en ajoutant de nouveaux attributs, en faisant plusieurs passages et/ou en ayant un coût en mémoire plus important : *cf.* remarque 4-4 (p. 74)

- en revanche d'autres optimisations sont globales, notamment celles qui portent sur les données ; leur application requiert la connaissance de la totalité du réseau, c'est-à-dire d'informations qui ne sont disponibles qu'une fois le réseau complètement construit

- en outre, les optimisations doivent être appliquées de manière itérative, car l'application d'une optimisation peut en rendre d'autres possibles. On aboutit ainsi à des transformations assez profondes du réseau, difficiles à mettre en œuvre au niveau de la génération

- le fonctionnement de l'algorithme de génération est simple et systématique ; pour effectuer des optimisations, il faudrait nécessairement augmenter sa complexité et prendre en compte divers cas particuliers. De même, chacune des optimisations, prise isolément, est simple et il est aisé de se convaincre de sa justesse. Dans un souci de modularité, il est sans doute préférable d'avoir plusieurs algorithmes relativement simples plutôt qu'un seul algorithme complexe, dont la correction pourrait poser problème

- grâce à ce choix, CÆSAR est un logiciel ouvert et extensible ; il est facile de rajouter de nouvelles optimisations pour remédier à certaines situations, sans remettre en cause le reste du compilateur. C'est d'ailleurs ainsi que les optimisations présentées ici ont été introduites

- enfin la séparation entre génération et optimisation ne s'accompagne d'aucune dégradation sensible des performances. Pour tous les exemples traités, ces deux phases sont exécutées en quelques secondes

Les optimisations décrites ici et implémentées dans le logiciel CÆSAR vérifient deux hypothèses :

- toute transformation doit faire décroître la taille du réseau. En particulier on ne supprime pas une ε-transition lorsque cela doit conduire à une augmentation du nombre de transitions[30]

- pendant toute la phase d'optimisation, les données sont manipulées de manière purement symbolique : on prend comme principe de n'évaluer aucune expression de valeur

7.1.1 Notations préliminaires

Dans ce chapitre on considère un réseau $(\mathcal{Q}, \mathcal{U}, \mathcal{T}, \mathcal{G}, \mathcal{X}, \mathcal{S}, \mathcal{F})$ que l'on va être amené à transformer. Les différentes optimisations utilisent les notations et les définitions suivantes :

- on appelle *prédécesseur* d'une place Q toute transition T dont Q est une place de sortie. On note "$pred(Q)$" l'ensemble des prédécesseurs de Q :

$$pred(Q) = \{T \mid Q \in out(T)\}$$

- on appelle *successeur* d'une place Q toute transition T dont Q est une place d'entrée. On note "$succ(Q)$" l'ensemble des successeurs de Q :

$$succ(Q) = \{T \mid Q \in in(T)\}$$

- on appelle *occurrence de définition* d'une variable X toute action de la forme "$X := V$" ou "**for** X **among** S" figurant dans l'action d'une transition de réseau

- on appelle *occurrence d'utilisation* d'une variable X toute valeur V figurant dans l'action ou l'offre d'une transition du réseau et telle que la variable X soit présente dans V

[30] ce choix pourrait être remis en question dans les futures versions de CÆSAR

7.2 Optimisation E0 : places et transitions improductives

On dit qu'une place Q est *improductive* lorsqu'aucune transition n'en est issue ($succ(Q) = \emptyset$).

On dit qu'une transition T est *improductive* lorsque :

- la porte de T est "ε"
- l'action de T ne contient aucune occurrence de définition de variable
- soit T n'a aucune place de sortie, soit toutes les places de sortie de T sont improductives

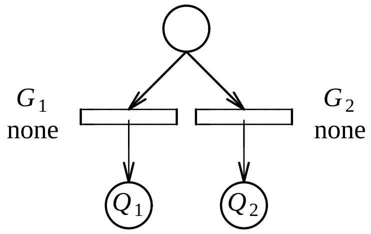
On peut détruire toutes les places improductives, à l'exception de la place initiale du réseau. On peut détruire toutes les transitions improductives : en effet à partir de ces transitions il est impossible de franchir une transition dont la porte est visible ou cachée, donc d'engendrer un arc dans le graphe.

La présence de transitions improductives est souvent due à une erreur de spécification. En revanche les réseaux possèdent généralement, même s'ils sont corrects, des places improductives dont la suppression provoque une diminution de la taille du graphe correspondant.

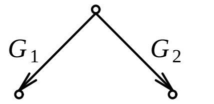
Exemple 7-1
Le comportement LOTOS suivant :

$$G_1 \ ; \ \textbf{stop} \ [] \ G_2 \ ; \ \textbf{stop}$$

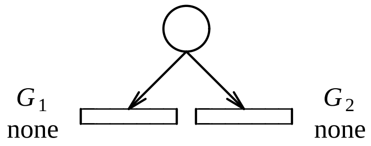
produit le réseau ci-dessous dans lequel les places Q_1 et Q_2 sont improductives :



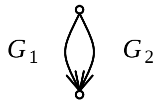
réseau avant optimisation



graphe avant optimisation



réseau aprés optimisation



graphe aprés optimisation

■

7.3 Optimisation E1 : places et transitions inaccessibles

On dit qu'une place Q est *inaccessible* lorsqu'elle est différente de la place d'entrée du réseau et qu'aucune transition n'arrive vers cette place ($pred(Q) = \emptyset$).

On dit qu'une transition T est *inaccessible* soit lorsqu'elle n'a aucune place d'entrée, soit lorsqu'une au moins de ses places d'entrée est inaccessible.

On peut détruire toutes les places et toutes les transitions inaccessibles du réseau, dont la présence est généralement due à l'existence de blocages dans la spécification LOTOS.

7.4 Optimisation E2 : places parallèles

On dit que deux places Q_1 et Q_2 sont *parallèles* lorsque :

- Q_1 et Q_2 ont le même ensemble, éventuellement vide, de prédécesseurs ($pred(Q_1) = pred(Q_2)$)
- Q_1 et Q_2 ont le même ensemble, éventuellement vide, de successeurs ($succ(Q_1) = succ(Q_2)$)

On peut détruire une place Q_1 lorsqu'il ne s'agit pas de la place initiale du réseau et qu'il existe une place Q_2 qui lui est parallèle. En effet pour tout marquage accessible M, on a la propriété :

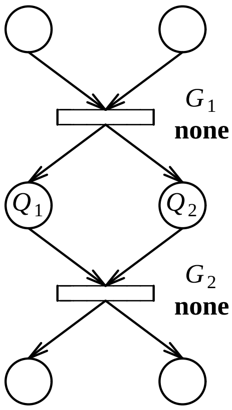
$$Q_1 \in M \Longleftrightarrow Q_2 \in M$$

La présence de places parallèles dans le réseau est une conséquence de l'algorithme de couplage des transitions.

Exemple 7-2
Le comportement LOTOS suivant :

$$\ldots \ \ ; \ G_1 \ ; \ G_2 \ ; \ \ldots \ \ \texttt{|[}G_1, G_2\texttt{]|} \ \ \ldots \ \ ; \ G_1 \ ; \ G_2 \ ; \ \ldots$$

produit le réseau ci-dessous dans lequel on peut ainsi supprimer soit Q_1 soit Q_2 :



■

Remarque 7-1
Pour une place Q_1 fixée, l'ensemble des places Q_2 qui lui sont parallèles est inclus dans l'ensemble :

$$\left(\bigcap_{T \in pred(Q_1)} out(T) \right) \cap \left(\bigcap_{T \in succ(Q_1)} in(T) \right) - \{Q_1\}$$

En pratique, cette formule permet de déterminer rapidement un ensemble restreint de candidats Q_2 possibles. ■

7.5 Optimisation E3 : pré-compactage

Cette optimisation permet de fusionner une ε-transition avec les transitions qui la précèdent immédiatement. Un triplet (T_1, Q, T_2) est *pré-compactable* lorsque :

- Q est une place de sortie de T_1 ($Q \in out(T_1)$)
- T_2 est la seule transition issue de Q ($succ(Q) = \{T_2\}$)
- Q est l'unique place d'entrée de T_2 ($in(T_2) = \{Q\}$)
- la porte de T_2 est "ε"
- l'action de T_2 est "**none**"

Si (T_1, Q, T_2) est pré-compactable, on peut détruire la transition T_2 ainsi que la place Q, si elle est différente de la place d'entrée du réseau, après avoir modifié l'ensemble des places de sortie de la transition T_1 de la façon suivante :

$$out(T_1) := (out(T_1) - \{Q\}) \cup out(T_2)$$

Remarque 7-2
Cette transformation est encore correcte quand Q fait partie des places de sortie de T_2 puisque, d'après la propriété du marquage sauf (§ 4.6.1, p. 89). si T_2 est franchissable, on a :

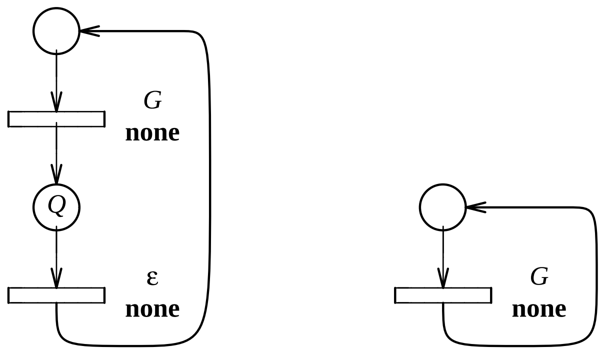
$$(Q \in in(T_2)) \ \wedge \ (Q \in out(T_2)) \Longrightarrow out(T_2) = \{Q\}$$

Dans ce cas, la boucle formée par la transition T_2 sur la place Q est détruite. ■

Cette optimisation apporte une solution uniforme à plusieurs problèmes distincts :

- elle permet d'éliminer les ε-transitions créées par les instanciations récursives (la place Q du triplet (T_1, Q, T_2) est détruite)

  Exemple 7-3

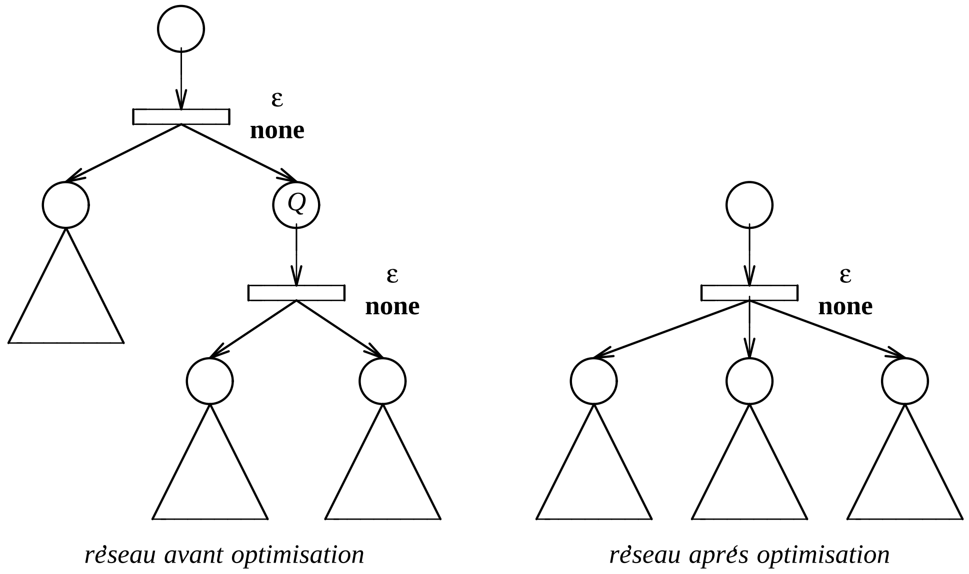


rèseau avant optimisation *rèseau aprés optimisation*

■

- elle permet la fusion des transitions *fork* créées par les opérateurs parallèles. Grâce à l'optimisation E3, la structure du réseau reflète l'associativité de la composition parallèle puisque les transitions *fork* peuvent désormais avoir plus de deux places de sortie

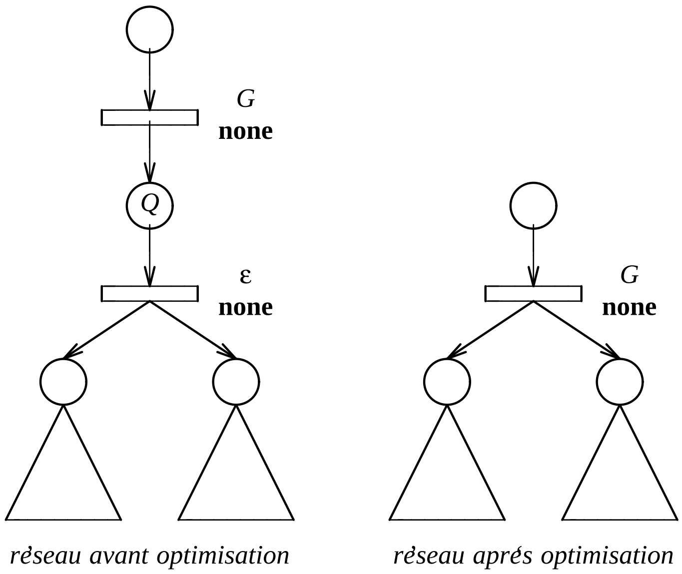
  Exemple 7-4



■

- cette optimisation permet aussi la destruction d'ε-transitions *fork*

  Exemple 7-5



■

7.6 Optimisation E4 : post-compactage $1 \rightarrow n$

Cette optimisation permet de fusionner une ε-transition avec une transition qui la suit. On dit qu'un triplet (T_1, Q, T_2) est *post-compactable* $1 \rightarrow n$ lorsque :

- la porte de T_1 est "ε"
- Q est l'unique place de sortie de T_1 $(out(T_1) = \{Q\})$
- Q n'est pas la place d'entrée du réseau
- T_1 est la seule transition arrivant vers Q $(pred(Q) = \{T_1\})$
- Q n'est pas une place d'entrée de T_1 $(Q \notin in(T_1))$
- T_2 est une transition issue de Q $(T_2 \in succ(Q))$
- les transitions T_1 et T_2 sont distinctes

La notation "$1 \rightarrow n$" signifie que la place Q peut avoir n successeurs alors qu'elle ne doit avoir qu'un seul prédécesseur.

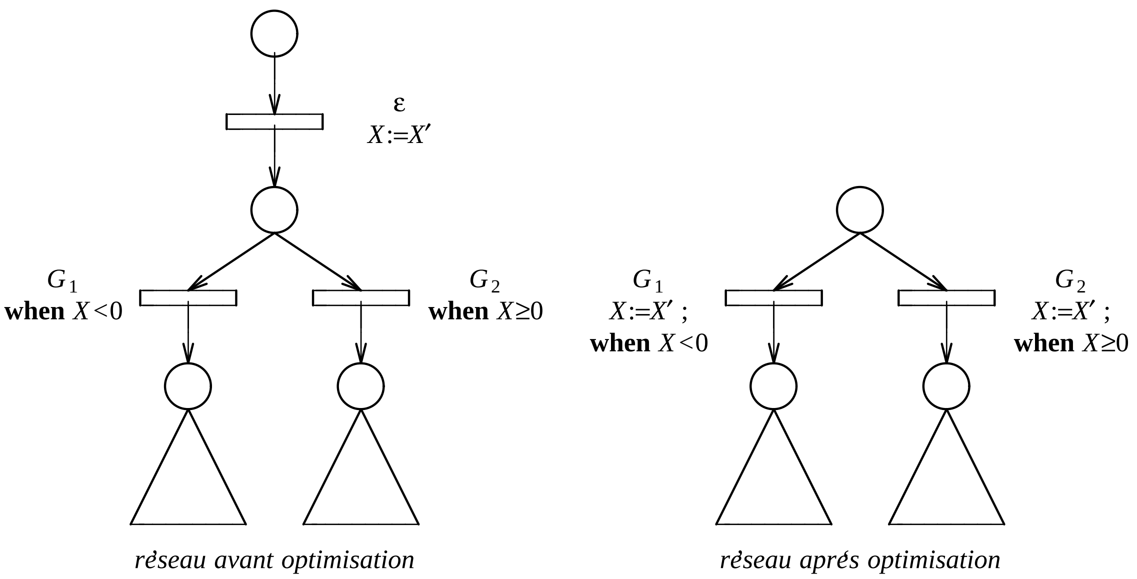
Si (T_1, Q, T_2) est post-compactable $1 \rightarrow n$ on peut modifier les attributs de la transition T_2 de la manière suivante :

- $in(T_2) := in(T_1) \cup (in(T_2) - \{Q\})$
- $action(T_2) := (action(T_1)\ ;\ action(T_2))$

Si tous les successeurs T_2 de Q forment avec T_1 des triplets post-compactables $1 \rightarrow n$, la place Q et la transition T_2 deviennent improductives et peuvent être détruites par application de l'optimisation E4.

Cette optimisation permet souvent de réduire les ε-transitions créées pour engendrer les opérateurs "**let**", "**choice**" et "**->**".

Exemple 7-6



■

7.7 Optimisation E5 : post-compactage $n \rightarrow 1$

Complémentaire de la précédente, cette optimisation permet de fusionner une ε-transition avec une transition qui la suit. On dit qu'un triplet (T_1, Q, T_2) est *post-compactable* $n \rightarrow 1$ lorsque :

- la porte de T_1 est "ε"
- Q est l'unique place de sortie de T_1 $(out(T_1) = \{Q\})$
- Q n'est pas une place d'entrée de T_1 $(Q \notin in(T_1))$
- T_2 est la seule transition issue de Q $(succ(Q) = \{T_2\})$
- les transitions T_1 et T_2 sont distinctes

La notation "$n \rightarrow 1$" signifie que la place Q peut avoir n prédécesseurs alors qu'elle ne doit avoir qu'un seul successeur.

Si (T_1, Q, T_2) est post-compactable $n \rightarrow 1$ on peut modifier les attributs de la transition T_1 de la manière suivante :

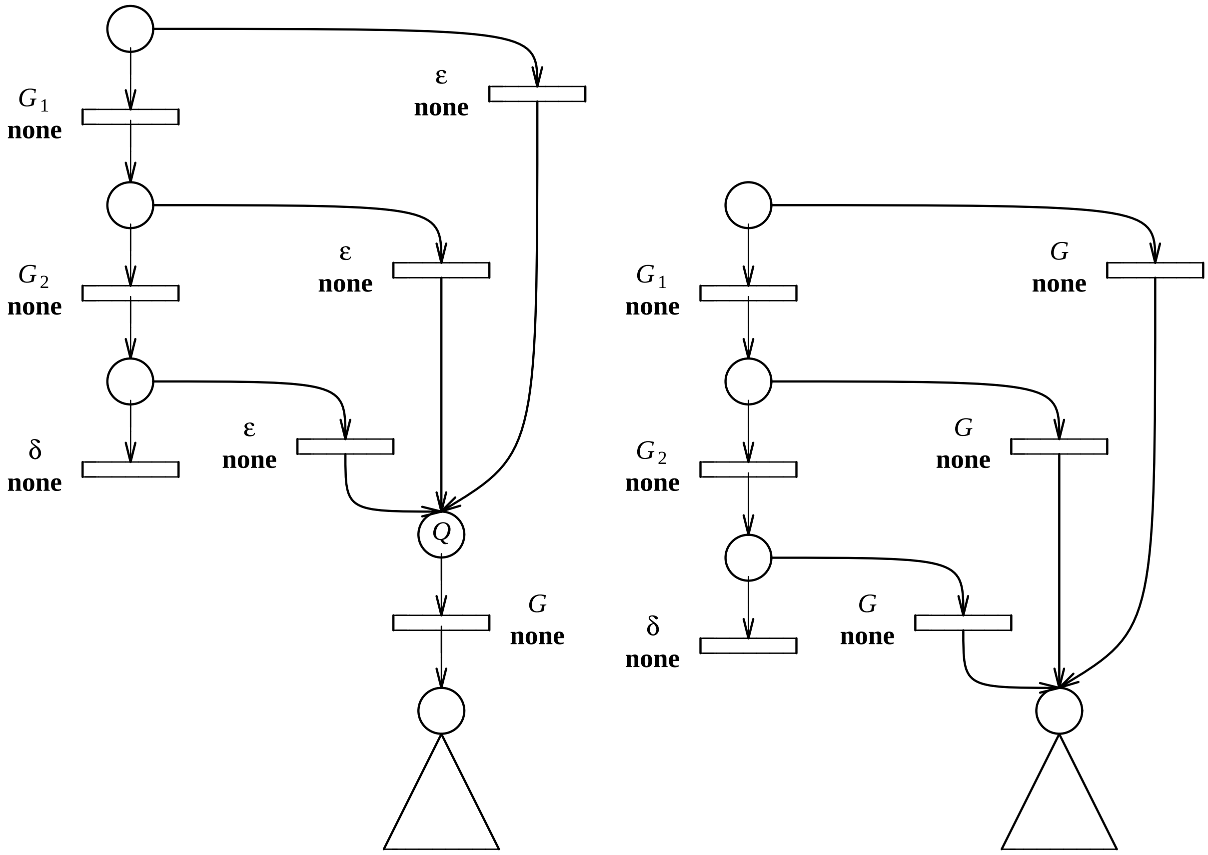
- $in(T_1) := in(T_1) \cup (in(T_2) - \{Q\})$
- $out(T_1) := out(T_2)$
- $gate(T_1) := gate(T_2)$
- $offer(T_1) := offer(T_2)$
- $action(T_1) := (action(T_1) \ ; \ action(T_2))$

Ainsi, quand T_2 n'est pas une ε-transition, cette transformation remplace une ε-transition (T_1 avant modification) par une transition visible ou cachée (T_1 après modification).

Si tous les prédécesseurs T_1 de Q forment avec T_2 des triplets post-compactables $n \rightarrow 1$, la place Q devient inaccessible et peut être éliminée, s'il ne s'agit pas de la place d'entrée du réseau, par application de l'optimisation E1. De même la transition T_2 devient inaccessible et peut également être détruite.

Cette optimisation permet, entre autres, de réduire certaines ε-transitions créées pour engendrer l'opérateur "`[>`".

Exemple 7-7



rèseau avant optimisation *rèseau aprés optimisation*

■

7.8 Optimisation E6 : unités vides

On dit qu'une unité U est *vide* lorsqu'elle ne comporte plus aucune place propre ($places(U) = \emptyset$), ce qui est la conséquence des destructions de places effectuées par l'une des optimisations précédentes.

On doit détruire toute unité vide U après avoir modifié l'ensemble des sous-unités de l'unité U_0 qui englobe immédiatement U ($U \in units(U_0)$) de la façon suivante :

$$units(U_0) := (units(U_0) - \{U\}) \cup units(U)$$

L'unité racine (c'est-à-dire l'unité qui englobe toutes les autres) ne peut jamais devenir vide car elle contient la place initiale du réseau. En revanche la place initiale des autres unités peut être détruite.

7.9 Optimisation V0 : affectations redondantes

Il est possible que les actions des transitions du réseau contiennent des affectations de la forme "$X := X$". On peut remplacer ces affectations inutiles par l'action "**none**" et simplifier ensuite les actions attachées aux transitions du réseau en utilisant le fait que "**none**" est élément neutre pour "`;`" et "`&`".

Cette situation se produit fréquemment lorsque X est paramètre formel d'un processus P et qu'il existe un appel récursif de P qui ne modifie par la valeur de X.

Exemple 7-8
Le comportement suivant :

$$
\begin{array}{l}
\quad P\ \texttt{[}G\texttt{]}\ \texttt{(}V_1, V_2\texttt{)} \\
\textbf{where} \\
\quad \textbf{process}\ P\ \texttt{[}G\texttt{]}\ \texttt{(}X_1, X_2 \texttt{:} S\texttt{)}\ \texttt{:=} \\
\qquad G\ \texttt{!}X_2\ \texttt{;}\ P\ \texttt{[}G\texttt{]}\ \texttt{(}X_1, -X_2\texttt{)} \\
\quad \textbf{endproc}
\end{array}
$$

produit un réseau contenant une ε-transition dont l'action est "$X_1, X_2 := X_1, -X_2$". L'optimisation V0 remplace cette action par "$X_2 := -X_2$". ■

7.10 Optimisation V1 : variables inutilisées

On dit qu'une variable X est *inutilisée* lorsque le réseau ne contient aucune occurrence d'utilisation de cette variable. Si X est une variable inutilisée on peut remplacer toutes les occurrences de définition de X par l'action "**none**" et simplifier ensuite les actions attachées aux transitions du réseau en utilisant le fait que "**none**" est élément neutre pour "`;`" et "`&`".

La présence de variables inutilisées peut avoir des causes diverses :

- il s'agit d'une intention délibérée : dans de nombreux protocoles, il existe des situations dans lesquelles on accepte de recevoir un message dont le contenu n'est jamais pris en compte
- le programme LOTOS contient un comportement de la forme :

  $$G_1\ \texttt{?}X \texttt{:} S\ \texttt{;}\ G_2\ \texttt{!}X\ \texttt{;}\ \ldots$$

  Si les portes G_1 et G_2 sont masquées par un opérateur "**hide**" la variable X est inutilisée car les transitions correspondantes ont une offre vide. On peut alors détruire l'itération "**for** X **among** S", ce qui fait que le graphe correspondant peut être fini même si le domaine de la sorte S n'est pas borné
- le programme LOTOS contient un comportement de la forme :

  $$\textbf{exit any}\ S\ \texttt{>>}\ \textbf{accept}\ X \texttt{:} S\ \textbf{in}\ \ldots$$

  qui produit une transition dont l'action est "$\xi_{\bullet p} := X$" ; la variable $\xi_{\bullet p}$ n'est jamais utilisée. On peut donc éliminer cette affectation

7.11 Optimisation V2 : variables référencées

On dit qu'une variable X_1 *référence* une variable X_2 lorsque :

- X_1 et X_2 sont différentes

- toutes les occurrences de définition de X_1 sont de la forme "$X_1 := X_2$"
- le booléen *from_sync* attaché à chacune de ces affectations est égal à *false*

Si X_1 référence X_2, on peut remplacer X_1 par X_2 dans toutes les occurrences d'utilisation de X_1. On peut aussi remplacer toutes les occurrences de définition de X_1 par l'action "**none**" et simplifier ensuite les actions attachées aux transitions du réseau en utilisant le fait que "**none**" est élément neutre pour "`;`" et "`&`".

Cette optimisation est correcte parce que LOTOS est un langage fonctionnel, ce qui, au niveau du modèle réseau, se traduit par la propriété suivante : il est impossible d'avoir une évolution du réseau qui, après avoir effectué l'affectation "$X_1 := X_2$", modifie la valeur de X_2 tout en continuant d'utiliser celle de X_1, auquel cas la substitution de X_1 par X_2 serait incorrecte. En effet, d'après l'algorithme de génération, l'existence d'une affectation "$X_1 := X_2$" avec *from_sync* = *false* ne peut avoir que deux causes :

- le programme LOTOS contient un comportement de la forme "**let** X_1**:**S=X_2 **in** ..."
- le programme LOTOS contient une instanciation de la forme "P `[`... `]` `(`..., X_2, ...`)`" où X_1 est la variable formelle qui reçoit le paramètre effectif X_2

Or, pour modifier, dans le langage SUBLOTOS et dans le modèle réseau, la variable X_2, dont la valeur antérieure n'était pas indéfinie — puisqu'elle a été affectée à X_1 — il faut nécessairement, au niveau du programme LOTOS, qu'une instanciation de processus récursive soit intervenue. Les règles de sémantique statique de LOTOS font qu'après cette instanciation récursive la valeur que dénotait la variable X_1 avant l'appel n'est plus accessible[31] En SUBLOTOS et dans le modèle réseau, la variable X_1 a une portée globale : l'ancienne valeur subsiste donc après l'instanciation récursive, mais elle ne sera jamais utilisée avant d'avoir été "écrasée" par une nouvelle valeur calculée en fonction de la nouvelle valeur de X_2.

Attention ! Ce raisonnement ne s'applique pas aux actions "$X_1 := X_2$" pour lesquelles *from_sync* = *true*. Ces affectations sont produites par les échanges de valeurs qui ont lieu au moment des rendez-vous. En effet on peut imaginer qu'un comportement B_2 possédant une variable interne X_2 modifie cette variable, après avoir communiqué l'ancienne valeur de X_2 à un comportement concurrent B_1 qui l'a conservée dans une variable X_1.

Exemple 7-9
Le comportement suivant :

$$
\begin{array}{l}
\quad (G\ ?X_1\!:\!S\ ;\ G'\ !X_1\ ;\ \mathbf{stop})\ |[G]|\ P\ [G]\ (V) \\
\mathbf{where} \\
\quad \mathbf{process}\ P\ [G]\ (X_2\!:\!S)\ := \\
\quad\quad \mathbf{i}\ ;\ G\ !X_2\ ;\ P\ [G]\ (-\,X_2) \\
\quad \mathbf{endproc}
\end{array}
$$

engendre une action "$X_1 := X_2$". Mais il serait erroné de remplacer X_1 par X_2. ■

7.12 Ordonnancement des optimisations

Le choix de l'ordre dans lequel les différentes simplifications doivent être effectuées dépend de deux facteurs :

[31] tout comme, dans les langages algorithmiques, une procédure appelée ne peut pas accéder aux variables locales de la procédure appelante

- les modifications apportées au réseau par une optimisation sont susceptibles de rendre possibles d'autres optimisations qui ne l'étaient pas auparavant. Par exemple V0, en simplifiant les actions, permet d'appliquer E3.

  Pour résoudre ce problème on tient à jour l'ensemble des optimisations restant à effectuer (*candidates*). Initialement toutes les optimisations sont candidates. On en sélectionne une parmi l'ensemble des candidates et on essaie de l'appliquer. Si elle échoue, on la retire de l'ensemble. Si elle réussit, toutes les optimisations deviennent candidates

- pour un réseau donné, plusieurs optimisations peuvent s'appliquer simultanément. On est donc confronté à un choix, sachant qu'une transformation effectuée par une optimisation peut ensuite empêcher l'application d'autres optimisations.

  Pour résoudre ce problème, on donne à chaque optimisation une priorité ; ces priorités déterminent une relation d'ordre total entre les optimisations. En effet on constate expérimentalement qu'il est préférable de respecter un certain ordre et, par exemple, d'appliquer E3 avant E4 et E4 avant E5. Lorsque l'ensemble des candidates contient plusieurs éléments on choisit toujours l'optimisation la plus prioritaire. Dans le système CÆSAR, l'ordre suivant a été retenu :

  $$V0 \rightarrow V1 \rightarrow V2 \rightarrow E0 \rightarrow E1 \rightarrow E2 \rightarrow E3 \rightarrow E4 \rightarrow E5 \rightarrow E6$$

  où V0 est l'optimisation la plus prioritaire et E6 celle qui l'est le moins

7.13 Résultats de l'optimisation

En pratique les gains résultant de l'optimisation sont appréciables. L'optimisation produit une réduction sensible de la taille du réseau et parfois de la taille du graphe correspondant. Les chiffres suivants — relevés sur les exemples présentés en annexe — donnent une idée de l'ordre de grandeur des gains obtenus :

nombre d'unités détruites	29%–30%
nombre de places détruites	25%–53%
nombre de transitions détruites	23%–46%
nombre de variables détruites	18%–32%

7.14 Autres optimisations

Les optimisations proposées ne constituent pas une liste exhaustive. De nouvelles optimisations pourraient être introduites, si le besoin s'en fait sentir :

- détection et fusion des parties identiques du réseau, en s'inspirant de technique de construction des graphes orientés sans circuits (*dags* [ASU86, p. 290–293])

- utilisation de techniques d'analyse des réseaux de Petri, notamment les méthodes d'analyse structurelle. C'est ainsi que l'étude des propriétés de vivacité permettrait de supprimer certaines transitions infranchissables créées par l'algorithme de couplage : *cf.* exemple 6-1 (p. 122)

- utilisation de techniques d'analyse de flux mises en œuvre dans les compilateurs qui effectuent des optimisations globales. Nombre de variables d'état demeurent, qui pourraient être éliminées après une analyse poussée du flux des données dans le réseau. On obtiendrait aussi des gains en mémoire importants en déterminant les variables qui ne peuvent jamais être utilisées simultanément — par exemple les variables déclarées dans les deux opérandes d'un choix non-déterministe — ce qui permettrait d'effectuer des recouvrements

- élimination totale des ε-transitions. Cette transformation accélère notablement la rapidité de la simulation, mais elle présente l'inconvénient d'augmenter la taille du réseau. Elle devrait donc être appliquée en dernier afin de ne pas pénaliser les autres optimisations

- évaluation des certaines expressions que l'on remplace par leur valeur si celle-ci est constante. Par exemple les rendez-vous avec communication de type *value matching* engendrent fréquemment des actions de la forme "**when** V_1 = V_2" pour lesquelles l'évaluation de V_1 et V_2 est susceptible de renvoyer des valeurs constantes ; si V_1 et V_2 sont égales on peut remplacer ces actions par "**none**", sinon on peut détruire les transitions qui portent ces actions

Chapitre 8

Simulation

L'objectif de la simulation, ultime phase de CÆSAR, est de produire le graphe à partir du réseau, en parcourant exhaustivement toutes les séquences possibles.

Cette traduction soulève plusieurs difficultés que ce chapitre expose et pour lesquelles il propose des solutions. Le premier problème rencontré est la présence en LOTOS de types abstraits algébriques. Il faut ensuite trouver un algorithme performant pour implémenter la sémantique du réseau, ce qui pose plusieurs problèmes dérivés, notamment pour le calcul efficace des successeurs des états, des positions, des marquages et des contextes.

Enfin, l'approche exhaustive étant coûteuse en temps de calcul et en espace mémoire, on conçoit aisément que les décisions d'implémentation aient une importance cruciale sur les performances, pour la simulation plus encore que pour les autres phases. C'est pourquoi ce chapitre donne aussi quelques indications sur les techniques mises en œuvre dans CÆSAR pour représenter en mémoire les structures d'informations.

8.1 Traitement des données

Dans toutes les phases antérieures (expansion, génération et optimisation) l'évaluation des expressions de valeur a été constamment différée afin d'éviter l'explosion combinatoire. C'est donc à la phase de simulation qu'il appartient de traiter ce problème.

On doit pour cela calculer les expressions de valeur qui figurent dans les actions et les offres attachées aux transitions du réseau. Toutes ces expressions sont des termes algébriques clos : chaque variable possède une valeur que l'on peut lui substituer. Il n'y a donc aucune variable libre.

Pour évaluer une expression de valeur, on peut la transmettre à un moteur de réécriture qui la réduit, en appliquant dynamiquement les équations. Le moteur peut être, soit indépendant des équations, comme c'est le cas dans les outils HIPPO [Tre87] et SPIDER [Joh88], soit engendré spécifiquement à partir des équations, comme le fait le compilateur LOTOS→C [ndM88]. Dans les deux cas, la méthode est lente, puisque l'application des règles de réécriture se fait de manière non-déterministe et nécessite un mécanisme de chaînage arrière (*backtrack*). En outre elle consomme beaucoup de mémoire, puisque les valeurs sont représentées par des termes algébriques. C'est pourquoi cette solution n'a pas été retenue pour CÆSAR.

Une autre approche consiste à traduire l'expression à évaluer dans un langage impératif ; pour CÆSAR il s'agit du langage C. Pour cela il faut :

- associer à chaque sorte LOTOS une implémentation sous forme de *type* C. On note "*implementation*[S]" le type qui correspond à la sorte S
- associer à chaque opération LOTOS une implémentation sous forme de *fonction* C. On note "*implementation*[F]" la fonction qui correspond à l'opération F

On dit alors que les types abstraits de LOTOS sont implémentés de manière *concrète*. Cette solution est efficace parce que les données ont une représentation compacte en mémoire et parce que les opérations sont effectuées de manière déterministe. Par exemple le nombre 1789 sera probablement codé par un entier et non par un terme algébrique à base de `0` et de `SUCC` ; de même l'addition de deux nombres ne nécessitera pas de réécriture de termes.

En outre, pour chaque sorte S, il faut disposer :

- d'une *relation de comparaison* qui est une fonction C qui à deux arguments V_1 et V_2 de sorte S associe un résultat booléen égal à *true* si et seulement si V_1 et V_2 sont congrues selon la relation définie par les équations algébriques. Cette fonction, notée "*comparison*[S]", permet d'implémenter les valeurs de la forme "V_1 = V_2" car, en LOTOS, le symbole "=" ne correspond pas à une opération définie par l'utilisateur
- d'un *itérateur* qui est une macro-définition[32] C prenant comme argument une variable X de sorte S ; l'expansion de cette macro-définition doit créer une boucle de programme dans laquelle la variable X décrit successivement toutes les valeurs du domaine de S. Cet itérateur, noté "*iteration*[S]", sert à traduire les actions de la forme "**for** X **among** S". Si le domaine de S est potentiellement infini, il faut en restreindre l'énumération à un sous-ensemble fini
- d'une *procédure d'impression* qui est une fonction C, qui convertit une valeur V_0 de sorte S en une chaîne de caractères ASCII et l'imprime sur un fichier F, afin de permettre l'affichage des valeurs contenues dans les offres qui étiquettent les arcs du graphe

Pour passer des types abstraits LOTOS aux types concrets correspondants, deux solutions peuvent être envisagées :

implémentation manuelle : l'utilisateur écrit "à la main" pour chaque sorte et opération LOTOS les types et les fonctions C correspondantes. Dans ce cas le système CÆSAR "fait confiance" à l'utilisateur en supposant que l'implémentation concrète est correcte, c'est-à-dire conforme aux équations de la spécification abstraite. CÆSAR suppose que les types abstraits sont correctement définis par l'utilisateur, c'est-à-dire que les équations apparaissant dans la spécification LOTOS sont complètes et consistantes

implémentation automatique : étant donné un type abstrait, il est possible d'en dériver automatiquement une implémentation concrète, sous réserve que les équations aient une forme convenable. Une tentative a été faite dans ce sens avec le système CÆSAR.ADT [Bar88] [Gar89b], basé sur un algorithme de *compilation du pattern-matching* [Sch88b] [Sch88a].

A partir d'un type abstrait LOTOS, CÆSAR.ADT produit une bibliothèque de types et de fonctions en langage C. La version actuelle de l'outil suppose que les équations vérifient certaines conditions :

- les opérations doivent être divisées en deux catégories, les *constructeurs*, qui sont primitifs, et les *non-constructeurs*, qui sont définis en fonction des constructeurs
- les équations sont orientées : on considère que leur membre gauche est défini en fonction du membre droit et qu'on peut toujours le remplacer par celui-ci (LOTOS ne spécifie aucune orientation pour les équations)

[32] construction `#define`, en C

– toutes les équations doivent avoir la forme :

$$[V_1, \ldots\ V_m \texttt{ =>}]\ F\ \texttt{(}F_1\ \texttt{(}V_1^1, \ldots\ V_1^{n_1}\texttt{)}, \ldots\ F_p\ \texttt{(}V_p^1, \ldots\ V_p^{n_p}\texttt{))}\ \texttt{=}\ V$$

où F est un opérateur non-constructeur, $F_1, \ldots\ F_p$ sont des opérateurs constructeurs et $V_1, \ldots\ V_m, V_1^1, \ldots\ V_p^{n_p}, V$ sont des expressions quelconques. Cette restriction interdit qu'un opérateur constructeur soit défini en partie gauche d'une équation

Ces restrictions permettent à CÆSAR.ADT d'engendrer un code C de bonne qualité car l'évaluation d'une opération est faite de manière déterministe. En outre certains cas particuliers (types énumérés, types isomorphes aux entiers naturels) sont automatiquement reconnus et implémentés de manière optimale

Quelle que soit la solution retenue, traduction manuelle ou automatique, il faut établir la liaison entre les noms de types (*resp.* fonctions) C et les noms des sortes (*resp.* opérations) LOTOS correspondantes. Il n'est pas possible de donner aux objets C le nom des objets LOTOS qu'ils implémentent : en effet LOTOS autorise la surcharge des noms et l'emploi de caractères spéciaux (comme "-", "&", ...), toutes facilités absentes du langage C.

Il n'est pas davantage souhaitable d'engendrer automatiquement des identificateurs C uniques, dont l'utilisateur ne comprendrait pas l'origine et qui ne permettraient pas d'interfacer des bibliothèques prédéfinies.

C'est donc l'utilisateur qui doit indiquer la correspondance entre les noms abstraits et les noms concrets. Dans le système CÆSAR, ceci est réalisé au moyen de *commentaires spéciaux* qui constituent un sous-ensemble des commentaires LOTOS. Ils existent sous deux formes :

- lorsque la définition d'une sorte S est immédiatement suivie du commentaire :

  (*! **implementedby** N_1 **comparedby** N_2 **enumeratedby** N_3 **printedby** N_4 *)

  cela signifie que :

  – la sorte S est implémentée par un type C de nom N_1 ($\mathit{implementation}[S] = N_1$)

  – la relation de comparaison sur S est implémentée par une fonction C de nom N_2 ($\mathit{comparison}[S] = N_2$)

  – l'itérateur sur le domaine de S est une macro-définition de nom N_3 ($\mathit{iteration}[S] = N_3$)

  – la procédure d'impression des valeurs de S est une fonction C de nom N_4

- lorsque la définition d'une opération F est immédiatement suivie du commentaire :

  (*! **implementedby** N_0 [**constructor**] *)

  cela signifie que :

  – l'opération F est implémentée par une fonction C de nom N_0 ($\mathit{implementation}[F] = N_0$)

  – la présence du mot "**constructor**" indique que l'opérateur F est un constructeur. Cette information n'est pas exploitée par CÆSAR mais elle l'est par CÆSAR.ADT, qui partage avec CÆSAR les phases d'analyse syntaxique et sémantique statique

Remarque 8-1
Les identificateurs $N_0, \ldots\ N_4$ introduits dans les commentaires spéciaux doivent respecter les conventions lexicographiques du langage *C*. En outre ils doivent être deux à deux distincts et ne pas être égaux à l'un des mots-réservés du langage C (y compris "**main**"). ■

Remarque 8-2
Les commentaires spéciaux sont aussi employés dans le compilateur LOTOS→C [ndM88] (*annotations*). Dans l'ensemble ils coïncident avec les commentaires spéciaux de CÆSAR, à deux différences près :

- à chaque sorte S on peut associer, grâce à une annotation, une fonction permettant de récupérer la place mémoire occupée par une valeur de sorte S
- en revanche la notion d'itération sur le domaine d'une sorte n'est pas prévue

■

L'utilisateur doit fournir à CÆSAR, pour chaque type abstrait T, un fichier de nom "T.h" qui contient les définitions nécessaires pour implémenter le type T. Ce fichier peut être engendré automatiquement par CÆSAR.ADT.

Exemple 8-1
Si une spécification LOTOS contient le type abstrait suivant :

```
type NATURAL is
   sorts
      NAT (*! implementedby Nat
              comparedby EqNat
              enumeratedby LoopNat
              printedby PrintNat *)
   opns
      0    (*! implementedby Zero constructor *) : -> NAT
      SUCC (*! implementedby Succ constructor *) : NAT -> NAT
      _+_  (*! implementedby Plus *) : NAT, NAT -> NAT
   eqns
      forall X,Y: NAT
      ofsort NAT
      X + 0 = X;
      X + SUCC (Y) = SUCC (X+Y);
endtype
```

le fichier `NATURAL.h` pourra avoir le contenu suivant (pour optimiser le temps d'exécution, certaines définitions de fonctions sont remplacées par des macro-définitions) :

```
typedef unsigned short Nat;

#define EqNat(V1,V2) ((V1) == (V2))

#define LoopNat(X) for (X = 0; X < 100; ++X)

void PrintNat (F, V0)
   FILE *F;
   Nat V0;
{ fprintf (F, "%d", V0); }

#define Zero() 0

#define Succ(V0) ((V0)+1)

Nat Plus (V1, V2)
  Nat V1, V2;
{ return V1 + V2; }
```

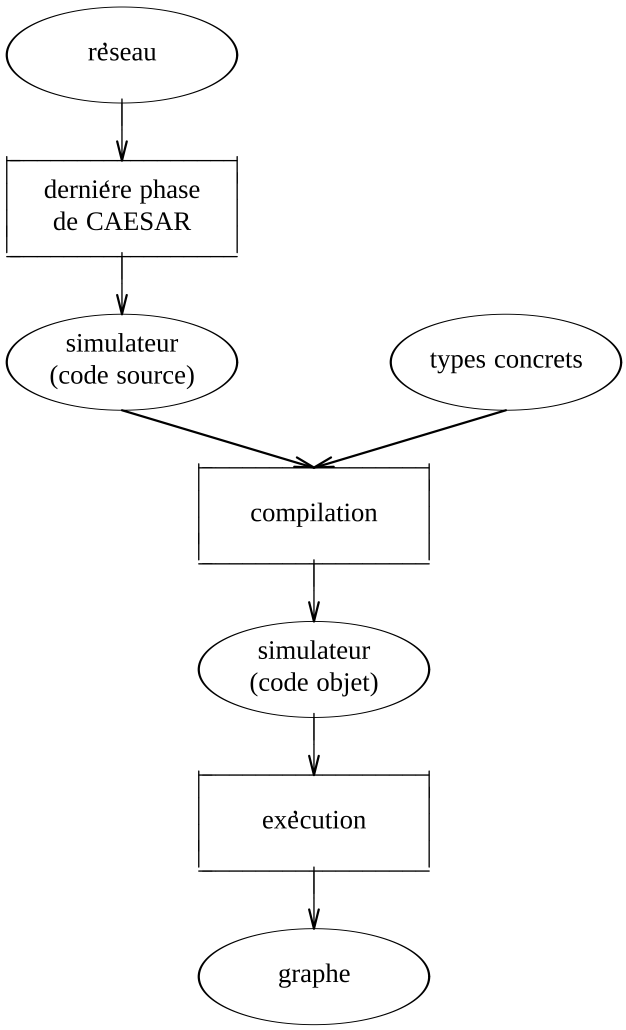
■

8.2 Algorithme de simulation

8.2.1 Programme simulateur

Contrairement aux phases d'expansion, de génération et d'optimisation la phase de simulation ne peut pas être effectuée directement : il n'est pas possible d'écrire un programme P qui, à partir d'un réseau, engendre le graphe correspondant. En effet, pour évaluer les expressions présentes dans le réseau, il faut utiliser les types et les fonctions C fournis par l'utilisateur, ce qui implique qu'ils fassent partie de P. Or ces types et ces fonctions dépendent de la spécification LOTOS à analyser et il est impossible de les connaître à l'avance, donc de les intégrer à P (ceci pourrait être remis en cause si l'on disposait d'un système d'exploitation, comme MULTICS, permettant l'édition de liens dynamique). La phase de simulation se déroule donc en trois étapes :

1. à partir du réseau, on engendre un programme C, appelé *simulateur*
2. ce programme est compilé donnant naissance à un programme objet
3. l'exécution du programme objet produit le graphe qui est écrit dans un fichier au fur et à mesure de sa construction



Remarque 8-3
Le fait de produire un simulateur séparé a une heureuse incidence sur les performances, en permettant de produire des graphes plus importants : en effet le simulateur est un programme exécuté dans un espace mémoire initialement vide qui n'est pas, comme celui de CÆSAR, occupé par les structures de données qui servent à représenter l'arbre abstrait LOTOS, l'arbre abstrait SUBLOTOS et le réseau. ■

8.2.2 Exploration et construction du graphe

On considère un réseau $(\mathcal{Q}, \mathcal{U}, \mathcal{T}, \mathcal{G}, \mathcal{X}, \mathcal{S}, \mathcal{F})$ à partir duquel on veut produire le programme simulateur.

On présente, par raffinements successifs, la structure de ce programme qui, pour plus de généralité, sera décrite à l'aide des notations algorithmiques classiques et non pas avec les constructions du langage C. Pour bien distinguer le texte du programme simulateur et celui de l'algorithme qui l'engendre, les fragments de programme appartenant au programme simulateur sont entourés d'un cadre rectangulaire.

Dans son principe, le fonctionnement du simulateur est assimilable à un parcours en largeur de graphe orienté, avec plusieurs différences :

- initialement seul l'état initial σ_0 est connu ; il est formé du marquage initial M_0 et du contexte initial C_0
- les autres états et les arcs sont créés au fur et à mesure de l'exploration du graphe
- pour détecter les circuits, on ne marque pas les états et on n'utilise pas de pile ; en revanche on mémorise les états dans un ensemble divisé en deux parties :
  - un sous-ensemble, noté Σ^+, qui contient les états visités et dont les successeurs ont été déterminés
  - un sous-ensemble, noté Σ^-, qui contient les états visités mais dont on n'a pas encore calculé les successeurs
- les arcs sont progressivement rangés dans un ensemble d'arcs, noté $\mathcal{E}$

Le simulateur a donc, en notant *oneof* (Σ^-) un élément quelconque choisi dans l'ensemble Σ^- , la structure suivante :

> **début**
> $M_0 \ := \ \{\mathit{first}(\mathcal{U})\}$
> $C_0 \ := \ \perp$
> $\sigma_0 \ := \ \langle M_0, C_0 \rangle$
> $\Sigma^+ \ := \ \emptyset$
> $\Sigma^- \ := \ \{\sigma_0\}$
> $\mathcal{E} \ := \ \emptyset$
> **tant_que** $\Sigma^- \neq \emptyset$ **faire**
>   **début**
> $\sigma_1 \ := \ \mathit{oneof}(\Sigma^-)$
> $\Sigma^+ \ := \ \Sigma^+ \cup \{\sigma_1\}$
> $\Sigma^- \ := \ \Sigma^- - \{\sigma_1\}$
>   **pour_tous** L, σ_2 **tels_que** $\sigma_1 \xrightarrow{L} \sigma_2$ **faire**
>   **début**
> $\mathcal{E} \ := \ \mathcal{E} \cup \{(\sigma_1, L, \sigma_2)\}$
>   **si** $\sigma_2 \notin (\Sigma^+ \cup \Sigma^-)$ **alors** $\Sigma^- \ := \ \Sigma^- \cup \{\sigma_2\}$
>   **fin**
>   **fin**
> **fin**

Le graphe produit pour le réseau $(\mathcal{Q}, \mathcal{U}, \mathcal{T}, \mathcal{G}, \mathcal{X}, \mathcal{S}, \mathcal{F})$ est ainsi défini :

- l'ensemble des états est la valeur de Σ^+ lorsque l'exécution se termine
- l'état initial est σ_0
- l'ensemble des arcs est la valeur de $\mathcal{E}$ lorsque l'exécution se termine
- l'ensemble des portes est $\mathcal{G}$
- l'ensemble des sortes est $\mathcal{S}$
- l'ensemble des opérations est $\mathcal{F}$

L'algorithme de simulation ne peut pas “boucler” indéfiniment puisqu'à chaque itération un nouvel état est créé. La quantité de mémoire disponible pour mémoriser les états étant limitée, la simulation se termine toujours en un temps fini, soit normalement, soit par épuisement de la mémoire.

Lorsque le simulateur s'arrête par manque de mémoire, il est impossible de savoir si l'automate correspondant au programme LOTOS est infini ou s'il fini mais simplement trop volumineux pour être représenté en mémoire. Le problème de l'arrêt du simulateur n'est pas décidable statiquement.

Remarque 8-4
A la fin de la simulation, on pourrait communiquer à l'utilisateur la liste des transitions qui n'ont jamais été franchies. Toutefois la présence de telles transitions n'est pas forcément synonyme d'erreur de conception, puisque CÆSAR lui-même engendre parfois des transitions infranchissables : *cf.* exemple 6-1 (p. 122). ■

8.2.3 Calcul des états successeurs

Dans l'algorithme de simulation (p. 156), le calcul des successeurs σ_2 d'un état σ_1 est décrit par l'itération suivante (dans laquelle on fait abstraction du corps de boucle) :

pour_tous L, σ_2 **tels_que** $\sigma_1 \xrightarrow{L} \sigma_2$ **faire**
  début
  (traitement de* (σ_1, L, σ_2) **)*
  ...
  fin

Cette itération nécessite d'être mieux détaillée. En effet la relation de transition entre états, telle qu'elle est donnée dans la définition sémantique du réseau, n'est pas directement exécutable. Il faut la mettre sous une forme plus opérationnelle.

Pour calculer les successeurs d'un état σ_1 on doit implémenter l'algorithme "$\mu\varepsilon^* L$" (§ 4.5.6, p. 89). Ceci est fait en trois phases successives :

1. on construit, dans un premier temps, un graphe orienté $\Gamma(\sigma_1)$ sans circuit (mais pouvant comporter des cycles) :

   - l'ensemble Π des sommets de $\Gamma(\sigma_1)$ est l'ensemble des positions que l'on peut atteindre à partir de σ_1 (y compris σ_1) en franchissant un nombre quelconque d'ε-transitions
   - l'ensemble des arcs de $\Gamma(\sigma_1)$ est représenté par une fonction de succession : pour chaque position π_1 de Π, on note "$succ_\varepsilon[\pi_1]$" l'ensemble des positions π_2 de Π telles qu'il existe un arc allant de π_1 vers π_2. On construit cet ensemble de la manière suivante : π_2 appartient à $succ_\varepsilon[\pi_1]$ s'il existe une ε-transition allant de π_1 à π_2 :

   $$(\exists T) \;\; (gate(T) = \varepsilon) \;\wedge\; ([\pi_1 \;,\; T] \xrightarrow{p} [\pi_2 \;,\; \varepsilon])$$

   Cette condition n'est toutefois pas suffisante car il faut "casser" les arcs qui créent des circuits. Dans ce but on effectue un parcours en profondeur en partant de σ_1 et en suivant les ε-transitions. Pour détecter les circuits, on marque et on démarque successivement chaque position π grâce à un attribut booléen noté "$mark[\pi]$"

   La construction du graphe est réalisée par le fragment de programme suivant :

   $\Pi \;:=\; \{\sigma_1\}$
   explore (σ_1)

   où la procédure récursive "*explore*", qui calcule Π et $succ_\varepsilon[\pi]$ pour tout π de Π, est définie comme suit :

```
procédure explore (π₁)
  début
  mark[π₁] := true
  succ_ε[π₁] := ∅
  soient M₁, C₁ tels_que π₁ = ⟨M₁, C₁⟩
  pour_tous T tels_que (gate(T) = ε) ∧ (marking(T) ⊆ M₁) faire
    début
    soit M₂ tel_que [M₁ , in(T) , out(T)] --m--> M₂
    pour_tous C₂ tels_que [C₁ , action(T)] --c--> C₂ faire
      début
      π₂ := ⟨M₂, C₂⟩
      si π₂ ∉ Π alors
        début
        Π := Π ∪ {π₂}
        succ_ε[π₁] := succ_ε[π₁] ∪ {π₂}
        explore (π₂)
        fin
      sinon si mark[π₂] = false alors
        succ_ε[π₁] := succ_ε[π₁] ∪ {π₂}
      fin
    fin
  mark[π₂] := false
  fin
```

2. pour chaque position π de Π, on détermine l'ensemble noté "$pass[\pi]$" des transitions T visibles ou cachées pouvant être franchies à partir de π au sens de la relation de transition entre marquages :

$$in(T) \subseteq marking(\pi)$$

et ne pouvant pas être franchies à partir d'une position π_i, antérieure à π, dont le marquage est différent de celui de π :

$$\begin{array}{c}(\textit{pour tout chemin } \pi_1, \ldots\ \pi_n \textit{ du graphe } \Gamma(\sigma_1) \textit{ allant de } \sigma_1 \textit{ à } \pi)\ \ (\forall i \in \{1, \ldots\ n-1\}) \\ (in(T) \not\subseteq marking(\pi_i))\ \vee\ (marking(\pi_i) = marking(\pi))\end{array}$$

Pour calculer $pass[\pi]$ on effectue un parcours en profondeur du graphe $\Gamma(\sigma_1)$. Lorsqu'une transition T appartient à l'attribut $pass[\pi_1]$ d'une position π_1, elle ne peut pas figurer dans l'attribut $pass[\pi_2]$ d'une position π_2 accessible à partir de π_1 en suivant les arcs du graphe $\Gamma(\sigma_1)$, sauf si π_1 et π_2 ont le même marquage.

Pour représenter cette contrainte on propage pendant l'exploration en profondeur un attribut hérité : il s'agit d'un tableau qui, à toute transition T dont la porte n'est pas ε fait correspondre un marquage, éventuellement indéfini, noté "$base[T]$". Lorsque $base[T]$ n'est pas indéfini, cela signifie que la transition T ne doit être franchie qu'à partir des positions dont le marquage est égal à $base[T]$.

Le graphe $\Gamma(\sigma_1)$ comportant des cycles, il peut exister plusieurs chemins allant de σ_1 à une position π donnée. Pour détecter les cycles, on marque les positions π explorées grâce à l'attribut $mark[\pi]$. Lorsqu'une position marquée π est atteinte, cette position a été visitée antérieurement en empruntant un autre chemin que le chemin actuel. C'est pourquoi on doit "corriger" $pass[\pi]$, c'est-à-dire en retirer les transitions qui ne satisfont pas les contraintes nouvelles imposées par le chemin actuel. De même il est nécessaire, pour chaque position π' accessible à partir de π, de corriger $succ_\varepsilon[\pi']$.

Le calcul des attributs $pass[\pi]$, pour toute position π de Π est effectué par le fragment de programme suivant :

```
pour_tous π ∈ Π faire mark[π] := false
pour_tous T tels_que gate(T) ≠ ε faire base[T] := indéfini
fire (σ1, base)
```

où la procédure récursive "*fire*", qui calcule l'attribut $pass[\pi]$ de toutes les positions π accessibles à partir d'une position π_1 en respectant les contraintes imposées par le paramètre $base_1$, est définie comme suit :

```
procédure fire (π1, base1)
   début
   mark[π1] := true
   pass[π1] := ∅
   base2 := base1
   M1 := marking(π1)
   pour_tous T tels_que (gate(T) ≠ ε) ∧ (marking(T) ⊆ M1) faire
      si (base1[T] = indéfini) ∨ (base1[T] = M1) alors
         début
         pass[π1] := pass[π1] ∪ {π2}
         base2[T] := M1
         fin
   pour_tous π2 ∈ succε[π1] faire
   si mark[π2] = false alors
      fire (π2, base2)
   sinon
      unfire (π2, base2)
   fin
```

et où la procédure récursive "*unfire*", dont le rôle est de corriger les attributs $pass[\pi]$ de toutes les positions π accessibles à partir d'une position π_1 en en retirant les transitions qui ne satisfont pas les contraintes imposées par le paramètre *base*, est définie comme suit :

```
procédure unfire (π1, base)
   début
   pour_tous T ∈ pass[π1] faire
      si (base[T] ≠ indéfini) ∧ (base[T] ≠ marking(π1)) alors
         pass[π1] := pass[π1] − {T}
   pour_tous π2 ∈ succε[π1] faire
      unfire (π2, base)
   fin
```

Le fait qu'une transition T appartienne à $pass[\pi]$ constitue une condition nécessaire, mais non suffisante, pour que la transition T puisse être franchie à partir de la position π : encore faut-il que le contexte de π soit compatible avec l'action de T

3. pour chaque position π_1 de Π et pour chaque transition T de $pass[\pi_1]$, on calcule les états σ_2 obtenus à partir de π_1 en franchissant la transition T. Ces états constituent les successeurs de σ_1 dans le graphe des états.

pour_tous $\pi_1 \in \Pi$ **faire**
début
soient M_1, C_1 **tels_que** $\pi_1 = \langle M_1, C_1 \rangle$
pour_tous $T \in \mathit{pass}[\pi_1]$ **faire**
début
soit M_2 **tel_que** $[M_1\ ,\ \mathit{in}(T)\ ,\ \mathit{out}(T)] \xrightarrow{m} M_2$
pour_tous C_2 **tels_que** $[C_1\ ,\ \mathit{action}(T)] \xrightarrow{c} C_2$ **faire**
début
$\sigma_2\ :=\ \langle M_2, C_2 \rangle$
$L\ :=\ \mathit{eval\_label}(\mathit{gate}(T), \mathit{offer}(T), C_2)$
(traitement de* (σ_1, L, σ_2) **)*
...
fin
fin
fin

8.3 Implémentation des marquages

Cette section détaille la réalisation de l'algorithme de simulation en proposant des méthodes efficaces pour représenter en mémoire les marquages et pour calculer la relation de transition entre marquages.

8.3.1 Représentation des marquages en mémoire

Le simulateur conserve la trace de tous les états rencontrés ; chaque marquage est représenté par une chaîne de bits. Ces chaînes de bits sont conservées dans des tables ; pour gérer efficacement ces tables, on impose aux chaînes de bits d'avoir toutes la même longueur. Afin d'économiser la mémoire, il est impératif que cette longueur soit minimale.

Une solution évidente consiste à représenter un marquage par une chaîne de bits telle qu'à chaque place correspond un bit, qui vaut 1 si cette place est marquée et 0 sinon. Mais il est possible d'obtenir un codage plus compact en exploitant le fait que n'importe quel ensemble de places ne constitue pas forcément un marquage accessible.

Soient $U_1, \ldots\ U_n$ les unités qui composent le réseau. Chaque unité U_i possède N_i places propres notées $Q_i^1, \ldots\ Q_i^{N_i}$. On appelle *projection* d'un marquage M sur une unité U_i l'ensemble de places, noté $M \rhd U_i$, défini par :

$$M \rhd U_i = M \cap \mathit{places}(U_i)$$

D'après la propriété de séquentialité des unités (§ 4.6.2, p. 91), tout marquage M contient au plus une place propre de chaque unité :

$$(\forall i \in \{1, \ldots\ n\})\ \ \mathrm{card}(M \rhd U_i) \leq 1$$

On peut donc représenter tout marquage M par ses projections sur les unités du réseau :

$$M = \bigcup_{i \in \{1, \ldots\ n\}} M \rhd U_i$$

Une unité U_i ayant N_i places propres peut donc se trouver dans $N_i + 1$ situations possibles : soit une seule place est marquée, soit aucune place n'est marquée. Pour coder ces $N_i + 1$ possibilités il suffit d'une chaîne de bits de longueur $L(N_i)$, où :

$$L(N_i) = \lceil \log_2(N_i + 1) \rceil$$

Si m et l sont deux entiers tels que m appartient à $\{0, \ldots\ 2^l - 1\}$, on note $\ll m \gg_l$ la chaîne de bits de longueur l dont la valeur est le codage binaire de m. Soit M un marquage et soit U_i une unité possédant N_i places propres $Q_1, \ldots\ Q_{N_i}$. On code $M \rhd U_i$ par une chaîne de bits de longueur $L(N_i)$, notée $b(M, U_i)$, définie par :

$$b(M, U_i) = \begin{cases} \textbf{si } M \rhd U_i = \emptyset \textbf{ alors } \ll 0 \gg_{L(N_i)} \\ \textbf{si } M \rhd U_i = \{Q_1\} \textbf{ alors } \ll 1 \gg_{L(N_i)} \\ \ldots \\ \textbf{si } M \rhd U_i = \{Q_{N_i}\} \textbf{ alors } \ll N_i \gg_{L(N_i)} \end{cases}$$

Finalement chaque marquage M est représenté par une chaîne $B(M)$ résultant de la concaténation des chaînes de bits $b(M, U_i)$ lorsque U_i décrit toutes les unités du réseau.

Par rapport à la solution évidente, cette technique conduit à une réduction logarithmique du nombre de bits utilisés :

$$\lceil \log_2(N_1 + 1) \rceil + \ldots\ \lceil \log_2(N_n + 1) \rceil$$

au lieu de :

$$N_1 + \ldots\ N_n$$

En pratique cependant, pour respecter les contraintes de cadrage on doit représenter un marquage sur un nombre entier d'octets. Les bits inutilisés sont systématiquement mis à 0.

Le calcul des projections d'un marquage sur les unités du réseau est implémenté par des masquages logiques et des décalages. L'égalité de deux marquages se fait par comparaison des chaînes de bits correspondantes.

8.3.2 Calcul des transitions franchissables

Les procédures *explore* (p. 158) et *fire* (p. 160) comportent chacune une itération pour calculer tous les marquages successeurs d'un marquage M_1, soit pour l'ensemble $\mathcal{T}_\varepsilon$ des ε-transitions, soit pour l'ensemble $\overline{\mathcal{T}_\varepsilon}$ des transitions dont la porte n'est pas "ε" :

pour_tous T **tels_que** $(gate(T) = \varepsilon) \wedge (marking(T) \subseteq M_1)$ **faire**
  début
  (traitement de* (T, M_1) **)*
  ...
  fin

et :

pour_tous T **tels_que** $(gate(T) \neq \varepsilon) \wedge (marking(T) \subseteq M_1)$ **faire**
  début
  (traitement de* (T, M_1) **)*
  ...
  fin

On cherche à implémenter efficacement ces itérations. Les deux cas étant similaires, on traitera uniquement le cas de l'ensemble $\mathcal{T}_\varepsilon$, le traitement de $\overline{\mathcal{T}_\varepsilon}$ étant identique. On note $T_1, \ldots\ T_p$ les transitions de $\mathcal{T}_\varepsilon$. Il s'agit de déterminer l'ensemble des ε-transitions qui peuvent être franchies à partir d'un marquage M_1. Une solution évidente est :

```
si in(T_1) ⊆ M_1 alors
   début
   (* traitement de (T_1, M_1) *)
   ...
   fin
...
si in(T_p) ⊆ M_1 alors
   début
   (* traitement de (T_p, M_1) *)
   ...
   fin
```

Remarque 8-5
La duplication des fragments de code associés à chaque transition augmente la taille du simulateur ; elle est cependant inévitable à cause du traitement des contextes car, pour chaque transition T il faut produire un fragment de code spécifique dépendant de $action(T)$ (§ 8.4.4, p. 172). ■

Il existe une meilleure implémentation, basée sur la propriété suivante :

$$in(T) \subseteq M_1 \iff T \notin h_1(M_1 \rhd U_1) \cup \ldots\ h_n(M_1 \rhd U_n)$$

où les fonctions h_i sont définies par :

$$\begin{cases} h_i(\emptyset) & = \{T \in \mathcal{T}_\varepsilon \mid in(T) \rhd U_i \neq \emptyset\} \\ h_i(\{Q\}) & = \{T \in \mathcal{T}_\varepsilon \mid (in(T) \rhd U_i \neq \emptyset)\ \wedge\ (in(T) \rhd U_i \neq \{Q\})\} \end{cases}$$

On constate qu'il est possible de déterminer statiquement (c'est-à-dire au moment de la construction du programme simulateur) les ensembles $h_i(\emptyset)$ et $h_i(\{Q\})$, i appartenant à $\{1, \ldots\ n\}$ et Q étant une place propre de U_i.

Dans le simulateur on associe à chaque transition T de $\mathcal{T}_\varepsilon$ une variable booléenne, notée H_T, qui vaut *true* si et seulement si la transition T est franchissable à partir de M_1. Chaque variable H_T est initialisée à *true* mais, par la suite, on lui affecte *false* s'il existe i dans $\{1, \ldots\ n\}$ tel que T appartienne à $h_i(M_1 \rhd U_i)$.

En pratique, on remplace le fragment de programme :

```
pour_tous T tels_que (gate(T) = ε) ∧ (marking(T) ⊆ M_1) faire
   début
   (* traitement de (T, M_1) *)
   ...
   fin
```

par :

$H_{T_1}, \ldots\ H_{T_n}\ :=\ true$
choisir $M_1 \rhd U_1$ **parmi**
  si $\emptyset$ **alors pour_tous** $T \in h_1(\emptyset)$ **faire** $H_T\ :=\ false$
  si $\{Q_1^1\}$ **alors pour_tous** $T \in h_1(\{Q_1^1\})$ **faire** $H_T\ :=\ false$
  ...
  si $\{Q_1^{N_1}\}$ **alors pour_tous** $T \in h_1(\{Q_1^{N_1}\})$ **faire** $H_T\ :=\ false$
...
choisir $M_1 \rhd U_n$ **parmi**
  si $\emptyset$ **alors pour_tous** $T \in h_n(\emptyset)$ **faire** $H_T\ :=\ false$
  si $\{Q_n^1\}$ **alors pour_tous** $T \in h_n(\{Q_n^1\})$ **faire** $H_T\ :=\ false$
  ...
  si $\{Q_n^{N_n}\}$ **alors pour_tous** $T \in h_n(\{Q_n^{N_n}\})$ **faire** $H_T\ :=\ false$
si H_{T_1} **alors**
  début
  (traitement de* (T_1, M_1) **)*
  ...
  fin
...
si H_{T_p} **alors**
  début
  (traitement de* (T_p, M_1) **)*
  ...
  fin

Comme les ensembles $h_i(\ldots)$ sont calculables statiquement, chaque boucle :

pour_tous $T \in h_i(\ldots)$ **faire** $H_T\ :=\ false$

se ramène en fait à l'affectation d'un petit nombre de variables H_T.

Exemple 8-2
Considérons un réseau composé de quatre unités U_1, U_2, U_3 et U_4 comportant respectivement 1, 2, 3 et 3 places propres, notées comme suit :

$$\begin{array}{lcl} places(U_1) & = & \{Q_1\} \\ places(U_2) & = & \{Q_2, Q_3\} \\ places(U_3) & = & \{Q_4, Q_5, Q_6\} \\ places(U_4) & = & \{Q_7, Q_8, Q_9\} \end{array}$$

Considérons ensuite six transitions $T_1, \ldots\ T_6$ dont les places d'entrées et de sortie sont ainsi définies :

T_i	$in(T_i)$	$out(T_i)$
T_1	$\{Q_1\}$	$\{Q_2, Q_4, Q_7\}$
T_2	$\{Q_2\}$	$\{Q_3\}$
T_3	$\{Q_3, Q_5\}$	$\{Q_2, Q_6\}$
T_4	$\{Q_4, Q_7\}$	$\{Q_5, Q_8\}$
T_5	$\{Q_8\}$	$\{Q_9\}$
T_6	$\{Q_9\}$	$\{Q_7\}$

Le fragment de programme correspondant est :

$H_1, H_2, H_3, H_4, H_5, H_6 := true$
choisir $M_1 \rhd U_1$ **parmi**
  si ε **alors** $H_1 := false$
  si $\{Q_1\}$ **alors rien**
choisir $M_1 \rhd U_2$ **parmi**
  si ε **alors** $H_2, H_3 := false$
  si $\{Q_2\}$ **alors** $H_3 := false$
  si $\{Q_3\}$ **alors** $H_2 := false$
choisir $M_1 \rhd U_3$ **parmi**
  si ε **alors** $H_3, H_4 := false$
  si $\{Q_4\}$ **alors** $H_3 := false$
  si $\{Q_5\}$ **alors** $H_4 := false$
  si $\{Q_6\}$ **alors** $H_3, H_4 := false$
choisir $M_1 \rhd U_4$ **parmi**
  si ε **alors** $H_4, H_5, H_6 := false$
  si $\{Q_7\}$ **alors** $H_6 := false$
  si $\{Q_8\}$ **alors** $H_4, H_5 := false$
  si $\{Q_9\}$ **alors** $H_4, H_5, H_6 := false$
...

Les variables $H_1, \ldots H_6$ sont implémentées par une chaîne de bits que l'on manipule globalement. ■

8.3.3 Calcul des marquages successeurs

L'algorithme de simulation (p. 158 et p. 160) contient des instructions :

$$\boxed{\textbf{soit } M_2 \textbf{ tel\_que } [M_1 \ , \ in(T) \ , \ out(T)] \xrightarrow{m} M_2}$$

qui sont équivalentes à :

$$\boxed{M_2 := (M_1 - in(T)) \cup out(T)}$$

Du fait de la duplication des fragments de code associés à chaque transition, ces instructions seront reproduites autant de fois qu'il existe de transitions T dans le réseau. Dans chaque cas les quantités $in(T)$ et $out(T)$ sont connues, ce dont on tire parti pour implémenter efficacement le calcul de M_2. En utilisant la notion de projection, on remplace l'instruction ci-dessus par :

si $out(T) \rhd U_1 \neq \emptyset$ **alors**
  $\boxed{M_2 \rhd U_1 := out(T) \rhd U_1}$
sinon si $in(T) \rhd U_1 \neq \emptyset$ **alors**
  $\boxed{M_2 \rhd U_1 := \emptyset}$
sinon
  $\boxed{M_2 \rhd U_1 := M_1 \rhd U_1}$
...
si $out(T) \rhd U_n \neq \emptyset$ **alors**
  $\boxed{M_2 \rhd U_n := out(T) \rhd U_n}$
sinon si $in(T) \rhd U_n \neq \emptyset$ **alors**
  $\boxed{M_2 \rhd U_n := \emptyset}$

sinon

$\boxed{M_2 \rhd U_n \ := \ M_1 \rhd U_n}$

Exemple 8-3
Pour le réseau décrit dans l'exemple 8-2 (p. 164) le calcul des marquages M_2 successeurs de M_1 pour les transitions T_1 et T_2 est implémenté par :

si H_1 **alors**
  début
  $M_2 \rhd U_1 \ := \ \emptyset$
  $M_2 \rhd U_2 \ := \ \{Q_2\}$
  $M_2 \rhd U_3 \ := \ \{Q_4\}$
  $M_2 \rhd U_4 \ := \ \{Q_7\}$
  ...
  fin
si H_2 **alors**
  début
  $M_2 \rhd U_1 \ := \ M_1 \rhd U_1$
  $M_2 \rhd U_2 \ := \ \{Q_3\}$
  $M_2 \rhd U_3 \ := \ M_1 \rhd U_3$
  $M_2 \rhd U_4 \ := \ M_1 \rhd U_4$
  ...
  fin

■

8.4 Implémentation des contextes

Comme cela a été fait pour les marquages, cette section détaille la réalisation de l'algorithme de simulation en indiquant comment représenter en mémoire les contextes et comment implémenter la relation de transition entre contextes.

8.4.1 Représentation des contextes en mémoire

Comme les marquages, les contextes sont codés par des chaînes de bits, mémorisées dans des tables. De même, on impose à ces chaînes de bits d'avoir toutes la même longueur.

Remarque 8-6
Cette restriction interdit l'emploi de types concrets contenant des structures dynamiques (listes, ...) qui nécessitent la création et la destruction d'éléments. En effet l'utilisation de tels types nécessiterait que l'on mémorise, pour chaque contexte, le tas (*heap*) dans lequel les éléments sont alloués. Cette possibilité aurait un coût en mémoire prohibitif. ■

Compte tenu de cette contrainte, on peut représenter un contexte par un type structuré dont chaque champ correspond à une variable du réseau. Si on note $X_1, \ldots\ X_n$ les variables du réseau et $S_1, \ldots\ S_n$ leurs sortes respectives, un contexte C est déclaré par un type structuré :

$$\boxed{\begin{array}{l} C\ :\ \textbf{structure} \\ \quad X_1\ :\ \textit{implementation}[S_1] \\ \quad \ldots \\ \quad X_n\ :\ \textit{implementation}[S_n] \\ \quad \textbf{fin} \end{array}}$$

On note "$C \circ X_i$" le champ X_i du contexte C.

Remarque 8-7
D'autres représentations des contextes pourraient être envisagées, notamment le codage incrémental qui consiste à représenter un contexte C_2 sur la base d'un autre contexte C_1, en donnant la liste des différences (*deltas*) entre C_2 et C_1. Cette possibilité intéressante n'a pas été expérimentée dans la version actuelle de CÆSAR. ■

Pour décider si deux contextes C_1 et C_2 sont identiques, il ne faut pas comparer les chaînes de bits qui implémentent C_1 et C_2, mais utiliser les relations de comparaison associées aux sortes des champs de C_1 et C_2. L'égalité entre contextes est implémentée par le fragment de code suivant :

$$\boxed{\textit{comparison}[S_1]\ (C_1 \circ X_1, C_2 \circ X_1)\ \ \wedge\ \ldots\ \ \textit{comparison}[S_n]\ (C_1 \circ X_n, C_2 \circ X_n)}$$

Pour évaluer une valeur V dans un contexte C il faut engendrer un fragment de code, noté $\textit{translate}_V(C, V,)$, défini par :

$$\begin{array}{l} \textit{translate}_V(C, V) = \\ \quad \textbf{si } V \equiv X \textbf{ alors} \\ \qquad \boxed{C \circ X} \\ \quad \textbf{sinon si } V \equiv V_1 \texttt{ = } V_2 \textbf{ alors} \\ \qquad \boxed{\textit{comparison}[S]\ (\textit{translate}_V(C, V_1), \textit{translate}_V(C, V_2))} \\ \qquad \textbf{avec } S \textbf{ défini par } S = \textit{sort}(V_1) = \textit{sort}(V_2) \\ \quad \textbf{sinon si } V \equiv F\ \texttt{(}V_1, \ldots\ V_n\texttt{)} \textbf{ alors} \\ \qquad \boxed{\textit{implementation}[F]\ (\textit{translate}_V(C, V_1), \ldots\ \textit{translate}_V(C, V_n))} \end{array}$$

Il est possible de réduire la taille des contextes ; en effet il n'est pas nécessaire de mettre toutes les variables du réseau dans les contextes. On distingue deux sortes de variables :

variables globales : une variable X est *globale* s'il existe une transition T du réseau telle que la valeur de X soit utilisée dans l'action ou l'offre de T, sans avoir été préalablement affectée par l'action de T. Autrement dit la valeur d'une variable globale est affectée au moment du franchissement de certaines transitions et utilisée, plus tard, pour d'autres transitions ; c'est pourquoi il faut conserver la valeur des variables globales dans les contextes

variables locales : une variable X est *locale* si elle n'est pas globale. Si X est locale alors, pour chaque transition T du réseau telle que l'offre ou l'action de T utilise la valeur de X, l'action de T commence par affecter X. Il est inutile de faire figurer une variable locale X dans les contextes ; il suffit de la représenter par une variable temporaire qui n'a d'existence que pendant le franchissement des transitions T dont l'action comporte des occurrences de X. Le gain en mémoire ainsi occasionné est considérable

Exemple 8-4
Considérons un réseau comportant six variables $X_1, \ldots\ X_6$ et cinq transitions $T_1, \ldots\ T_5$ définies par :

T_i	$action(T_i)$	$offer(T_i)$
T_1	$X_1 := F_1$; $X_2 := X_1$	$\langle\rangle$
T_2	**when** X_1 ; $X_3 := F_2\ (X_2)$	$\langle !X_3\rangle$
T_3	$X_3 := F_3\ (X_2)$; $X_4 := X_3$	$\langle !X_4\rangle$
T_4	**for** X_5 **among** S ; $X_6 := X_5$	$\langle !X_5\rangle$
T_5	$X_4 := F_4\ (X_6)$	$\langle !X_4, !X_6\rangle$

Les variables locales sont X_3, X_4 et X_5. Les contextes n'auront que trois champs : X_1, X_2 et X_6. ■

Les variables locales proviennent généralement des variables "de travail" utilisée dans les programmes LOTOS pour conserver des résultats intermédiaires. Bien souvent le compactage des ε-transitions effectué par les optimisations E3 (§ 7.5, p. 140), E4 (§ 7.6, p. 142) et E5 (§ 7.7, p. 144) rend ces variables locales.

La détermination des variables locales nécessite le calcul de cinq attributs :

- on note $\text{USE}(V)$ l'ensemble des variables utilisées dans l'expression de valeur V. La définition de cet attribut est donnée par la grammaire attribuée suivante :

$$
\begin{array}{l}
V \uparrow \text{USE} \equiv \\
\quad X \\
\qquad \{\ \text{USE} := \{X\} \\
\ \mid F\ (V_1 \uparrow \text{USE}_1, \ldots\ V_n \uparrow \text{USE}_n) \\
\qquad \{\ \text{USE} := \text{USE}_1 \cup \ldots\ \text{USE}_n \\
\ \mid V_1 \uparrow \text{USE}_1 = V_2 \uparrow \text{USE}_2 \\
\qquad \{\ \text{USE} := \text{USE}_1 \cup \text{USE}_2
\end{array}
$$

- on note $\text{USE}(\widehat{O})$ l'ensemble des variables utilisées dans la liste d'offres $\widehat{O}$. Si $\widehat{O}$ est de la forme $\langle V_1, \ldots\ V_n\rangle$ on a :

$$\text{USE}(\widehat{O}) = \text{USE}(V_1) \cup \ldots\ \text{USE}(V_n)$$

- on note $\text{USE}(A)$ l'ensemble des variables utilisées dans l'action A :

$$
\begin{array}{l}
A \uparrow \text{USE} \equiv \\
\quad \textbf{none} \\
\qquad \{\ \text{USE} := \emptyset \\
\ \mid \textbf{when}\ V \uparrow \text{USE}_0 \\
\qquad \{\ \text{USE} := \text{USE}_0 \\
\ \mid \widehat{X_0}, \ldots\ \widehat{X_n} := V_0 \uparrow \text{USE}_0, \ldots\ V_n \uparrow \text{USE}_n \\
\qquad \{\ \text{USE} := \text{USE}_0 \cup \ldots\ \text{USE}_n \\
\ \mid \textbf{for}\ \widehat{X_0}, \ldots\ \widehat{X_n}\ \textbf{among}\ S_0, \ldots\ S_n \\
\qquad \{\ \text{USE} := \emptyset \\
\ \mid A_1 \uparrow \text{USE}_1\ ;\ A_2 \uparrow \text{USE}_2 \\
\qquad \{\ \text{USE} := \text{USE}_1 \cup \text{USE}_2 \\
\ \mid A_1 \uparrow \text{USE}_1\ \&\ A_2 \uparrow \text{USE}_2 \\
\qquad \{\ \text{USE} := \text{USE}_1 \cup \text{USE}_2
\end{array}
$$

- on note $\text{DEF}(A)$ l'ensemble des variables modifiées par l'action A :

$A \uparrow \textsc{def} \equiv$
none
$\{\ \textsc{def} := \emptyset$
| **when** V
$\{\ \textsc{def} := \emptyset$
| $\widehat{X_0}, \ldots \widehat{X_n} := V_0, \ldots V_n$
$\{\ \textsc{def} := \widehat{X_0} \cup \ldots \widehat{X_n}$
| **for** $\widehat{X_0}, \ldots \widehat{X_n}$ **among** $S_0, \ldots S_n$
$\{\ \textsc{def} := \widehat{X_0} \cup \ldots \widehat{X_n}$
| $A_1 \uparrow \textsc{def}_1$; $A_2 \uparrow \textsc{def}_2$
$\{\ \textsc{def} := \textsc{def}_1 \cup \textsc{def}_2$
| $A_1 \uparrow \textsc{def}_1$ & $A_2 \uparrow \textsc{def}_2$
$\{\ \textsc{def} := \textsc{def}_1 \cup \textsc{def}_2$

- on note $\textsc{loc}(A)$ l'ensemble des variables locales à l'action A, c'est-à-dire non utilisées dans A ou bien modifiées dans A avant d'être utilisées. Ces trois attributs sont définis simultanément par la grammaire attribuée suivante (on rappelle que $\mathcal{X}$ désigne l'ensemble de toutes les variables du réseau) :

$A \uparrow \textsc{loc} \equiv$
none
$\{\ \textsc{loc} := \mathcal{X}$
| **when** $V \uparrow \textsc{use}_0$
$\{\ \textsc{loc} := \mathcal{X} - \textsc{use}_0$
| $\widehat{X_0}, \ldots \widehat{X_n} := V_0 \uparrow \textsc{use}_0, \ldots V_n \uparrow \textsc{use}_n$
$\{\ \textsc{loc} := \mathcal{X} - (\textsc{use}_0 \cup \ldots \textsc{use}_n)$
| **for** $\widehat{X_0}, \ldots \widehat{X_n}$ **among** $S_0, \ldots S_n$
$\{\ \textsc{loc} := \mathcal{X}$
| $A_1 \uparrow \textsc{loc}_1$; $A_2 \uparrow \textsc{loc}_2$
$\{\ \textsc{loc} := \textsc{loc}_1 \cap (\textsc{loc}_2 \cup (\mathcal{X} - \textsc{use}(A_2)) \cup \textsc{def}(A_1))$
| $A_1 \uparrow \textsc{loc}_1$ & $A_2 \uparrow \textsc{loc}_2$
$\{\ \textsc{loc} := \textsc{loc}_1 \cap \textsc{loc}_2$

L'ensemble des variables locales du réseau est défini de la manière suivante (on rappelle que $\mathcal{T}$ désigne l'ensemble de toutes les transitions du réseau) :

$$\left(\bigcap_{T \in \mathcal{T}} \textsc{loc}(action(T))\right) \cap \left(\bigcap_{T \in \mathcal{T}} (\mathcal{X} - \textsc{use}(offer(T))) \cup \textsc{def}(action(T))\right)$$

8.4.2 Linéarisation des actions d'affectation

Les actions d'affectation ont une interprétation vectorielle et ne peuvent pas être traduites telles quelles en un fragment de code séquentiel. Par exemple, l'action :

$$X, Y := Y + 1, X + 1$$

requiert, pour être implémentée, une variable *temporaire* X' ayant même sorte que X :

$$X' := Y + 1 \ ; \ Y := X + 1 \ ; \ X := X'$$

C'est pourquoi, avant d'engendrer le programme simulateur, on *linéarise* les affectations présentes dans les actions du réseau, c'est-à-dire qu'on les transforme en *affectations simples* de la forme $X_0, \ldots\ X_n := V$ composées séquentiellement par l'opérateur "**;**". Il y a éventuellement introduction de variables temporaires[33].

Lorsque l'on linéarise une affectation, il y a plusieurs façons de composer séquentiellement les affectations simples et on constate que certains ordonnancements sont préférables à d'autres parce qu'ils nécessitent moins de variables temporaires.

Exemple 8-5
L'action :

$$X, Y \ := \ X+1, Y-X$$

peut être traduite soit par :

$$X' := X+1 \ ; \ Y := Y-X \ ; \ X := X'$$

soit par :

$$Y := Y-X \ ; \ X := X+1$$

la seconde solution étant évidemment meilleure. ■

Pour déterminer l'ordonnancement optimal associé à une action :

$$\widehat{X_0}, \ldots\ \widehat{X_n} \ := \ V_0, \ldots\ V_n$$

on définit une *relation de dépendance* entre indices de $\{0, \ldots\ n\}$, notée "$i \longrightarrow j$", par :

$$(i \neq j) \ \wedge \ (\widehat{X_j} \cap \textsc{use}(V_i) \neq \emptyset)$$

La condition $\widehat{X_j} \cap \textsc{use}(V_i) \neq \emptyset$ exprime qu'il faut calculer V_i *avant* de modifier les variables de la liste $\widehat{X_j}$. La condition $i \neq j$ élimine les dépendances de la forme $X := X+1$ qui ne posent aucun problème de traduction.

Cette relation n'étant pas une relation d'ordre à cause de la présence de dépendances cycliques, on ne peut pas utiliser la méthode du tri topologique. L'introduction à bon escient de variables temporaires permet de "casser" ces cycles. On utilise une heuristique qui consiste à placer d'abord les indices dont le nombre de prédécesseurs selon la relation de dépendance est minimal.

L'action obtenue après linéarisation est le résultat d'une fonction récursive "*linearize*" définie ci-dessous, dont le paramètre I représente l'ensemble des indices de $\{0, \ldots\ n\}$ restant à ordonner. L'appel initial a lieu en donnant au paramètre I la valeur $\{0, \ldots\ n\}$. Si $\widehat{X_i}$ est une liste de variables, on note X'_i une variable temporaire dont la sorte est celle des variables de $\widehat{X_i}$.

$$
\begin{array}{l}
\mathit{linearize}(I) = \\
\quad \textbf{soit } i_0 \textbf{ tel\_que } (\forall i \in I) \ \ \mathrm{card}(\{j \in I \mid j \longrightarrow i_0\}) \leq \mathrm{card}(\{j \in I \mid j \longrightarrow i\}) \\
\quad \textbf{si } I = \{i_0\} \textbf{ alors} \\
\qquad (\widehat{X_{i_0}} := V_{i_0}) \\
\quad \textbf{sinon si } \ \mathrm{card}(\{j \in I \mid j \longrightarrow i_0\}) = 0 \textbf{ alors} \\
\qquad (\widehat{X_{i_0}} := V_{i_0}) \ ; \ \mathit{linearize}(I - \{i_0\}) \\
\quad \textbf{sinon} \\
\qquad (X'_{i_0} := V_{i_0}) \ ; \ \mathit{linearize}(I - \{i_0\}) \ ; \ (\widehat{X_{i_0}} := X'_{i_0})
\end{array}
$$

[33]qui sont des variables locales

8.4.3 Linéarisation des actions collatérales

Pour simplifier la création du programme simulateur, on cherche à mettre toutes les actions du réseau sous la forme canonique suivante :

$$A_0 \ \texttt{;} \ \ldots \ A_n$$

où chaque A_i est une *action simple* de la forme :

$$\begin{array}{rcl} A_i & \equiv & \textbf{none} \\ & | & \textbf{when } V \\ & | & X_0, \ldots \ X_n := V \\ & | & \textbf{for } \widehat{X_0}, \ldots \ \widehat{X_n} \textbf{ among } S_0, \ldots \ S_n \end{array}$$

Il faut donc remplacer tous les opérateurs "`&`" par des opérateurs "`;`". L'opérateur de composition collatérale "`&`", commutatif et associatif, n'impose aucun ordre d'évaluation sur ses opérandes. Comme souvent on peut tirer parti de ce non-déterminisme pour optimiser.

Exemple 8-6
Quand on remplace "`&`" par "`;`" dans l'action suivante :

$$X_2 := F_1 \ \texttt{(}X_1\texttt{)} \ \texttt{\&} \ \textbf{when } F_2 \ \texttt{(}X_1\texttt{)}$$

il est préférable d'en permuter les opérandes, puisque, si la condition portant sur X_1 n'est pas vérifiée, la transition n'est pas franchissable et l'affectation de X_2 est alors inutile. ■

Exemple 8-7
De même, il vaut mieux permuter les opérandes de l'action :

$$\textbf{for } X_1 \textbf{ among } S \ \texttt{\&} \ X_2 := F \ \texttt{(}X_3\texttt{)}$$

ce qui permet de n'affecter X_2 qu'une seule fois, avant le commencement de l'itération de X_1 parmi les valeurs du domaine de S. ■

Exemple 8-8
L'associativité de l'opérateur "`&`", comme sa commutativité, doit être mise à profit. Pour l'action suivante :

$$X_1 := V_2 \ \texttt{\& (}\textbf{when } V_2 \ \texttt{\&} \ \textbf{for } X_2 \textbf{ among } S\texttt{)}$$

le meilleur ordonnancement est :

$$\textbf{when } V_2 \ \texttt{;} \ X_1 := V_2 \ \texttt{;} \ \textbf{for } X_2 \textbf{ among } S$$

■

Pour déterminer le meilleur ordonnancement, dans l'hypothèse d'une simulation exhaustive, on associe à chaque action A un coût, noté $\mathit{weight}(A)$, qui correspond au nombre *approximatif* d'affectations et d'expressions évaluées pendant l'exécution de A. Ce coût n'est défini que pour les actions qui ont la forme "$A_0 \ \texttt{;} \ \ldots \ A_n$". On note W_0 le coût estimé des instructions que le simulateur doit exécuter après une action et K le nombre supposé d'éléments dans le domaine d'une sorte. Dans CÆSAR on prend $W_0 = 128$ et $K = 16$, ces valeurs étant parfaitement arbitraires.

$$
\begin{array}{l}
\mathit{weight}(A_1\ \texttt{;}\ \ldots\ A_m) = \\
\quad \textbf{si } m = 0 \textbf{ alors} \\
\qquad W_0 \\
\quad \textbf{sinon si } A_1 \equiv \textbf{none alors} \\
\qquad \mathit{weight}(A_2\ \texttt{;}\ \ldots\ A_m) \\
\quad \textbf{sinon si } A_1 \equiv \textbf{when } V \textbf{ alors} \\
\qquad 1 + \dfrac{\mathit{weight}(A_2\ \texttt{;}\ \ldots\ A_m)}{2} \\
\quad \textbf{sinon si } A_1 \equiv X_0, \ldots\ X_n := V \textbf{ alors} \\
\qquad n + 2 + \mathit{weight}(A_2\ \texttt{;}\ \ldots\ A_m) \\
\quad \textbf{sinon si } A_1 \equiv \textbf{for } \widehat{X_0}, \ldots\ \widehat{X_n} \textbf{ among } S_0, \ldots\ S_n \textbf{ alors} \\
\qquad f(\mathit{weight}(A_2\ \texttt{;}\ \ldots\ A_m), (N_0+1) + \ldots\ (N_n+1)) \\
\qquad \textbf{où } (\forall i \in \{0, \ldots\ n\})\ \widehat{X_i} \equiv X_i^0, \ldots\ X_i^{N_i} \\
\qquad \textbf{et } f(w,k) = \left\{ \begin{array}{l} \textbf{si } k = 0 \textbf{ alors } w \\ \textbf{si } k > 0 \textbf{ alors } (f(w,k-1)+2)K \end{array} \right. \quad = \left(w + \dfrac{2K}{K-1}\right) K^k - \dfrac{2K}{K-1}
\end{array}
$$

La linéarisation de la composition collatérale consiste à remplacer chaque action de la forme :

$$A_0\ \texttt{\&}\ \ldots\ A_n$$

où n a une valeur maximale (afin d'utiliser au mieux l'associativité) et où chaque action A_i est sous forme canonique, par une action :

$$A_{\sigma_0(0)}\ \texttt{;}\ \ldots\ A_{\sigma_0(n)}$$

où σ_0 est une permutation sur $\{0, \ldots\ n\}$ qui minimise le coût total ; autrement dit, pour toute permutation σ sur $\{0, \ldots\ n\}$, on doit avoir :

$$\mathit{weight}(A_{\sigma_0(0)}\ \texttt{;}\ \ldots\ A_{\sigma_0(n)}) \leq \mathit{weight}(A_{\sigma(0)}\ \texttt{\&}\ \ldots\ A_{\sigma(n)})$$

Cette transformation se fait par un parcours en profondeur de chaque action, en commençant par les opérateurs "&" les plus imbriqués.

8.4.4 Calcul des contextes successeurs

La procédure *explore* (p. 158) et l'algorithme de simulation (p. 160) contiennent chacun une itération pour calculer tous les contextes C_2 successeurs d'un contexte C_1 après exécution de l'action attachée à une transition T :

$$
\boxed{
\begin{array}{l}
\textbf{pour\_tous } C_2 \textbf{ tels\_que } [C_1\ ,\ \mathit{action}(T)] \xrightarrow{c} C_2 \textbf{ faire} \\
\quad \textbf{début} \\
\quad \textit{(* traitement de } (T, C_2) \textit{ *)} \\
\quad \ldots \\
\quad \textbf{fin}
\end{array}
}
$$

En pratique cette itération est dupliquée en autant de fragments de code qu'il existe de transitions T dans le réseau. Dans chaque fragment la valeur de T est connue statiquement. C'est une conséquence de la duplication qui a servi à implémenter la relation de transition entre marquages (§ 8.3.2, p. 162).

Dans cette itération deux contextes C_1 et C_2 sont utilisés. Cependant la présence de C_1 n'est pas indispensable puisque ce contexte n'est pas utilisé dans le corps de boucle. Le seul contexte C_2 suffit donc : il est initialisé à C_1 avant l'itération et l'exécution de l'action de T le fait évoluer.

Par suite de la linéarisation (§ 8.4.2, p. 169) (§ 8.4.3, p. 171), toutes les actions sont sous forme canonique "A_0 ; ... A_n" où chaque A_i est une action simple dont la traduction en un fragment de programme ne pose guère de problèmes :

- pour l'action "**none**" on n'engendre aucune instruction
- pour l'action "**when** V" on engendre un test "**si**... **alors**..."
- pour l'action "$X_0, \dots\ X_n := V$" on engendre une affectation
- pour l'action "**for** $\widehat{X_0}, \dots\ \widehat{X_n}$ **among** $S_0, \dots\ S_n$" on engendre une série de boucles imbriquées.

  Exemple 8-9
  Si la transition T comporte l'action suivante :

  $$\begin{array}{l} X_1, X_3 := X_2, X_4\ \texttt{;} \\ \textbf{for}\ X_5\ \textbf{among}\ S\ \texttt{;} \\ X_6 := F_1\ \texttt{(}X_3, X_5\texttt{)}\ \texttt{;} \\ \textbf{when}\ F_2\ \texttt{(}X_6\texttt{)}\ \texttt{=}\ X_5\ \texttt{;} \\ X_7 := F_3\ \texttt{(}X_1, X_6\texttt{)} \end{array}$$

  on produira le fragment de code ci-dessous :

$C_2 \circ X_1\ :=\ C_2 \circ X_2$
$C_2 \circ X_3\ :=\ C_2 \circ X_4$
iteration$[S]$ $(C_2 \circ X_5)$
  début
  $C_2 \circ X_6\ :=\ \mathit{implementation}[F_1]\ (C_2 \circ X_3, C_2 \circ X_5)$
  si *comparison*$[S]$ $(\mathit{implementation}[F_2]\ (C_2 \circ X_6), C_2 \circ X_5)$ **alors**
    début
    $C_2 \circ X_7\ :=\ \mathit{implementation}[F_3]\ (C_2 \circ X_1, C_2 \circ X_6)$
    (traitement de* (T, C_2) **)*
    ...
    fin
  fin

  ■

  Toutefois, pour traduire les actions "**for**... **among**", la valeur de certaines variables doit être sauvegardée dans dans des variables temporaires.

  Exemple 8-10
  C'est ainsi que l'action :

  $$\begin{array}{l} \textbf{for}\ X_1\ \textbf{among}\ S\ \texttt{;} \\ X_3\ :=\ X_2 + 1\ \texttt{;} \\ X_2\ :=\ X_1 \end{array}$$

  ne doit pas être traduite en :

```
pour_tous C2 ∘ X1 ∈ domain(S) faire
   début
   C2 ∘ X3  :=  C2 ∘ X2 + 1
   C2 ∘ X2  :=  C2 ∘ X1
   (* traitement de (T, C2) *)
   ...
   fin
```

En effet la variable $C_2 \circ X_2$ est modifiée dans la boucle *après* avoir été utilisée. Dès la seconde itération elle n'aura plus la valeur qu'elle avait en entrant dans la boucle. C'est pourquoi il faut conserver cette valeur, avant de commencer les itérations, dans une variable temporaire X_2' et la restituer à $C_2 \circ X_2$ à chaque itération :

```
X2'  :=  C2 ∘ X2
pour_tous C2 ∘ X1 ∈ domain(S) faire
   début
   C2 ∘ X2  :=  X2'
   C2 ∘ X3  :=  C2 ∘ X2 + 1
   C2 ∘ X2  :=  C2 ∘ X1
   (* traitement de (T, C2) *)
   ...
   fin
```

■

Dans le programme simulateur on remplace donc :

```
pour_tous C2 tels_que [C1 , action(T)] --c--> C2 faire
   début
   (* traitement de (T, C2) *)
   ...
   fin
```

par le fragment de code $translate_A(C_2, action(T))$ défini comme suit :

```
translate_A(C2, A1 ; ... Am) =
   si m = 0 alors
      (* traitement de (T, C2) *)
      ...
   sinon si A1 ≡ none alors
      translate_A(C2, A2 ; ... Am)
   sinon si A1 ≡ when V alors
      si translate_V(C2, V) alors
         début
         translate_A(C2, A2 ; ... Am)
         fin
   sinon si A1 ≡ X0, ... Xn := V alors
      X0, ... Xn  :=  translate_V(C2, V)
      translate_A(C2, A2 ; ... Am)
   sinon si A1 ≡ for X̂0, ... X̂n among S0, ... Sn alors
```

soient $X_0^0, \ldots\ X_n^{N_n}$ **définis par** $(\forall i \in \{0, \ldots\ n\})\ \widehat{X_i} \equiv X_i^0, \ldots\ X_i^{N_i}$
soient $X_1, \ldots\ X_p$ **définis par**
$\{X_1, \ldots\ X_p\} = (\textsc{def}(A_2\ ;\ \ldots\ A_m) \cap \textsc{use}(A_2\ ;\ \ldots\ A_m)) - \textsc{loc}(A_2\ ;\ \ldots\ A_m)$

$X_1'\ :=\ X_1$
...
$X_p'\ :=\ X_p$
$\mathit{iteration}[S_0]\ (X_0^0)\ \ldots\ \mathit{iteration}[S_0]\ (X_0^{N_0})$
...
$\mathit{iteration}[S_n]\ (X_n^0)\ \ldots\ \mathit{iteration}[S_n]\ (X_n^{N_n})$
    début
    $X_1\ :=\ X_1'$
    ...
    $X_p\ :=\ X_p'$
    $\mathit{translate}_A(C_2, A_2\ ;\ \ldots\ A_m)$
    fin

8.5 Implémentation de la table des états

L'algorithme de simulation effectue une exploration du graphe en conservant en mémoire l'ensemble des états $\Sigma^+ \cup \Sigma^-$ dans une structure de données appelée *table des états*. On donne ici quelques indications sur la manière dont cette table est gérée par le système Cæsar :

- la table contient tous les états appartenant à $\Sigma^+ \cup \Sigma^-$. Chaque état est constitué d'un marquage et d'un contexte, représentés en mémoire principale selon les techniques décrites ci-dessus (§ 8.3.1, p. 161) et (§ 8.4.1, p. 166)

- on n'impose pas de borne a priori sur le nombre d'états. La table est extensible : elle peut s'agrandir au fur et à mesure des besoins jusqu'à occuper la totalité de la mémoire principale disponible. En pratique, la table est formée d'une liste chaînée de pages de mémoire, chaque page pouvant contenir un nombre fixé d'états

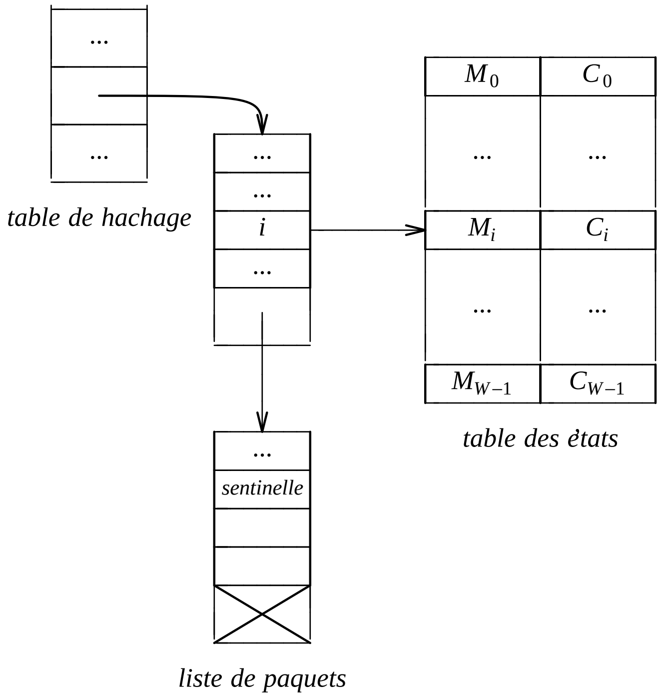
- le nombre d'états dans la table est donné dans une variable W qui est incrémentée chaque fois qu'on insère un nouvel état qui ne figure pas déjà dans la table. Chaque état est repéré par un indice variant de 0 à $W - 1$. On peut retrouver rapidement un état dont on connaît l'indice

- la table est partagée en deux zones par un indice R vérifiant $R \in \{0, \ldots\ W\}$. Les états dont l'indice appartient à $\{0, \ldots\ R - 1\}$ sont ceux de Σ^+. Les états dont l'indice appartient à $\{R, \ldots\ W - 1\}$ sont ceux de Σ^-. A tout instant la variable R contient l'indice du prochain état à traiter ; elle est incrémentée dès que les successeurs de cet état ont été calculés. La simulation est finie lorsque R et W sont égaux

- lorsque le simulateur construit un nouvel état, il doit rechercher s'il est présent dans la table, à défaut l'insérer, et dans tous les cas déterminer son indice. La table des états doit implémenter une correspondance bi-univoque entre les couples $\langle M, C\rangle$ et les indices d'états

- bien entendu, la recherche d'un état dont on connaît le marquage et le contexte ne se fait pas séquentiellement. On utilise une structure de données auxiliaire qui permet un accès rapide par hachage[34] [Knu73, § 6.4, pp. 506–549] [AHU83, pp. 122–125 et 363–365] : on a donc une table de hachage ayant H entrées et une fonction de hachage qui à un couple $\langle M, C\rangle$ associe un entier dans $\{0, \ldots\ H-1\}$

- pour tout h dans $\{0, \ldots\ H-1\}$, les indices des états qui ont h comme image par la fonction de hachage (*ensemble de collisions*) sont mémorisés hors de la table de hachage (*open hashing*), sous forme d'une liste chaîne de *paquets* (*buckets*). L'entrée h de la table de hachage est un pointeur vers la tête de cette liste. Chaque paquet comporte P indices dans la table des états et un pointeur permettant le chaînage vers le paquet suivant de la liste. Un paquet peut être incomplètement rempli : pour délimiter la fin de la zone effectivement utilisée, on utilise une valeur *sentinelle*



- à l'intérieur d'un ensemble de collisions, la recherche est séquentielle. Il est préférable de l'effectuer dans l'ordre inverse de l'ancienneté des états dans la table, c'est-à-dire suivant les indices décroissants.

  En effet, une étude statistique des automates produits par CÆSAR a permis de dégager un *principe de localité* : les états insérés récemment sont recherchés avec succès plus fréquemment que les autres. Cette constatation s'explique sans doute par le fait que le graphe est engendré en largeur et non en profondeur.

  Les listes de paquets doivent donc être construites à l'envers : chaque nouveau paquet est systématiquement inséré en tête de liste. A l'intérieur d'un paquet, la recherche doit commencer par les indices les plus récemment insérés

- il est recommandé de donner à H une valeur qui soit un nombre premier. La valeur de H a

[34] on trouve dans [Knu73] une intéressante étude sur la reconnaissance tardive du mot *hashing* ; le terme français prôné par [Gin83] est *adressage calculé*

une grande influence sur le temps d'exécution : elle doit être choisie en fonction du nombre N d'états des graphes à traiter[35] et du nombre moyen de comparaisons moyen que l'on vise.

Exemple 8-11
Pour CÆSAR on prend $H = 8\,329$ ce qui, pour $N = 100\,000$ fait environ 13 comparaisons en cas de recherche infructueuse et 7 en cas de recherche réussie ■

- lorsque N et H sont fixés, la valeur de P peut être ajustée pour minimiser la taille mémoire consommée par le hachage. Cette taille est la somme de deux termes :
  - une partie fixe, due à la table de hachage elle-même et indépendante du nombre d'états. En supposant qu'un pointeur tient sur 4 octets, la table occupe $4H$ octets
  - une partie variable, imputable aux paquets et qui augmente en fonction du nombre d'états. Si la fonction de hachage est équitable, le nombre N_P de paquets nécessaires pour ranger N états vérifie les inégalités suivantes :

    $$\frac{N}{P} \leq N_P \leq \frac{N-H}{P} + H$$

    Le terme droit correspond au cas le plus défavorable, celui du gaspillage maximal, où chacune des H listes de paquets contient un certain nombre de paquets complètement remplis, plus un paquet réduit à seul état. En supposant que les indices des états soient codés sur 3 octets (ce qui autorise en principe 16 millions d'états) la taille d'un paquet est de $(3P + 4)$ octets

Le nombre total d'octets utilisés pour l'accès secondaire est donc :

$$N_M = 4H + N_P(3P + 4)$$

En reprenant l'encadrement de N_P donné ci-dessus, N_M vérifie les inégalités suivantes :

$$\frac{(3N + 4H)P + 4N}{P} \leq N_M \leq \frac{(3H)P^2 + (3N + 5H)P + (4N - 4H)}{P}$$

La valeur optimale de P est celle qui minimise la valeur du terme droit :

$$P_{opt} = \sqrt{\frac{4(N-H)}{3H}}$$

Exemple 8-12
En prenant $N = 100\,000$ et $H = 8\,329$, la valeur optimale de P, arrondie au meilleur entier voisin, est 4. Pour mémoriser la structure d'accès secondaire il faut alors entre 433316 et 499769 octets ■

8.6 Implémentation de la table des positions

L'algorithme de simulation gère un ensemble de positions, noté Π, dont l'implémentation est assez proche de celle de la table des états, à quelques différences près :

[35] on ne connaît pas la valeur exacte de N mais seulement son ordre de grandeur

- chaque élément π de la table des positions est un quintuplet $(M, C, succ_\varepsilon[\pi], pass[\pi], mark[\pi])$ où M est la représentation en mémoire du marquage de π (§ 8.3.1, p. 161), C est la représentation en mémoire du contexte de π (§ 8.4.1, p. 166), $succ_\varepsilon[\pi]$ est la tête d'une liste chaînée de positions, $pass[\pi]$ est un tableau de booléens indexé par les transitions visibles ou cachées et $mark[\pi]$ est une valeur booléenne

- il est inutile d'attribuer un indice aux positions

- la table des positions contient beaucoup moins d'éléments que la table des états puisqu'elle est vidée et ré-initialisée avant le traitement de chaque état. Il est toutefois impossible de connaître à l'avance le nombre maximal de positions présentes dans la table : il faut prévoir une table dynamiquement extensible

- on utilise aussi un accès secondaire par hachage ouvert, avec des paquets de taille 1 pour la table des positions

8.7 Implémentation de la table des arcs

On constate que le contenu de l'ensemble $\mathcal{E}$ des arcs n'est jamais consulté et que le simulateur se contente d'y ranger les arcs produits :

$$\boxed{\mathcal{E} \ := \ \emptyset}$$

et :

$$\boxed{\mathcal{E} \ := \ \mathcal{E} \cup \{(\sigma_1, L, \sigma_2)\}}$$

C'est pourquoi on peut, sans perte d'efficacité, implémenter $\mathcal{E}$ en mémoire secondaire par un fichier à accès séquentiel. Au fur et à mesure que les arcs sont créés, ils sont ajoutés à la fin de ce fichier.

En pratique ce fichier constitue, à lui seul, le résultat produit par CÆSAR puisqu'il contient toutes les informations nécessaires pour définir complètement le graphe. C'est ce fichier qui est utilisé par les outils de vérification comme, par exemple, ALDEBARAN [Fer88], AUTO [LMV87]. CÆSAR est capable d'engendrer ce fichier sous les formats requis par ces outils.

Chapitre 9

Vérification

Le but de la vérification est de comparer un programme — qu'on suppose ici décrit en langage LOTOS — avec sa spécification. Selon la nature de la spécification, on distingue deux types d'approches pour la vérification [Pnu86] [Sif86] :

- si la spécification est également faite en LOTOS, il s'agit de comparer deux programmes LOTOS. Pour cela on utilise des équivalences sur les graphes, comme celles décrites précédemment (§ 1.3, p. 22)
- cependant certaines propriétés s'expriment mieux et peuvent être vérifiées de manière plus efficace lorsque la spécification est faite avec d'autres formalismes. Parmi ceux-ci, les logiques temporelles sont bien adaptées à l'expression de propriétés portant sur les comportements parallèles

Une logique temporelle dépend nécessairement du modèle qu'elle entend décrire. Or les logiques classiques — notamment celles utilisées dans les outils de la famille CESAR, à savoir CTL [CES85], STL [GS86a] et LTAC [Gra84] — conviennent mal à LOTOS car elles ont été élaborées pour des modèles sémantiques différents des graphes utilisés pour LOTOS. C'est pourquoi une logique temporelle originale et adaptée à LOTOS a été conçue ; elle est baptisée R$\star$ICO (*Regular Information Chronological Ordering*).

Ce chapitre définit la syntaxe et la sémantique formelle de cette logique, en séparant clairement les aspects temporels des aspects logiques. Des exemples illustrent les possibilités de ce langage de spécification ; ils sont consacrés à des propriétés communément rencontrées dans la vérification de protocoles.

On donne ensuite des points de comparaison qui permettent de situer la logique R$\star$ICO par rapport aux autres logiques temporelles existantes. Pour finir, le problème de l'évaluation des formules de cette logique est abordé de manière partielle, à travers plusieurs considérations d'implémentation.

9.1 Valeurs étendues

Pour améliorer les moyens d'expression fournis à l'utilisateur, on introduit la notion de *valeur étendue*, dénotée par le non-terminal W, qui généralise la notion d'expression de valeur LOTOS. Les valeurs étendues sont définies par la syntaxe suivante, dans laquelle les non-terminaux X, F et S ont leur sens usuel (§ 2.1.1, p. 25) :

$$
\begin{array}{rcl}
W & \equiv & X \\
 & | & F\ \texttt{(}W_1, \ldots\ W_n\texttt{)} \\
 & | & W_1\ \texttt{=}\ W_2 \\
 & | & \textbf{if}\ W_0\ \textbf{then}\ W_1\ \textbf{else}\ W_2 \\
 & | & \textbf{let}\ \widehat{X_0}\,\texttt{:}\,S_0\texttt{=}W_0, \ldots\ \widehat{X_n}\,\texttt{:}\,S_n\texttt{=}W_n\ \textbf{in}\ W'
\end{array}
$$

On note "$eval(W, C)$", où W est une valeur étendue et C un contexte (c'est-à-dire une application partielle qui à chaque variable de W associe la valeur qu'elle dénote), le résultat de l'évaluation de W dans C ; il s'agit d'une valeur LOTOS sans variable et sous forme normale. Cette fonction est définie par récurrence sur la complexité de W :

$$
\begin{array}{l}
eval(W, C) = \\
\quad \textbf{si}\ (W \equiv X)\ \textbf{alors}\ C(X) \\
\quad \textbf{sinon si}\ (W \equiv F\ \texttt{(}W_1, \ldots\ W_n\texttt{)})\ \textbf{alors}\ F\ (eval(W_1, C), \ldots\ eval(W_n, C)) \\
\quad \textbf{sinon si}\ (W \equiv W_1\ \texttt{=}\ W_2)\ \textbf{alors}\ eval(W_1, C) = eval(W_2, C) \\
\quad \textbf{sinon si}\ (W \equiv \textbf{if}\ W_0\ \textbf{then}\ W_1\ \textbf{else}\ W_2)\ \textbf{alors} \\
\quad\quad \textbf{si}\ eval(W_0, C)\ \textbf{alors}\ eval(W_1, C)\ \textbf{sinon}\ eval(W_2, C) \\
\quad \textbf{sinon si}\ (W \equiv \textbf{let}\ \widehat{X_0}\,\texttt{:}\,S_0\texttt{=}W_0, \ldots\ \widehat{X_n}\,\texttt{:}\,S_n\texttt{=}W_n\ \textbf{in}\ W')\ \textbf{alors} \\
\quad\quad eval(W', C \oslash ((\widehat{X_0} \rightsquigarrow W_0) \oplus \ldots\ (\widehat{X_n} \rightsquigarrow W_n)))
\end{array}
$$

9.2 Aspects logiques

La logique R$\overset{\star}{\text{I}}$CO comporte deux classes d'objets : les *formules logiques* et les *expressions temporelles*, qui permettent d'exprimer des propriétés sur les graphes LOTOS. Dans un premier temps, on s'intéresse uniquement aux formules. Celles-ci n'utilisent pas les informations fournies par la relation de transition : les propriétés qu'elles caractérisent portent exclusivement sur les labels qui étiquettent les arcs.

9.2.1 Syntaxe des formules logiques

Les formules logiques sont dénotées par le non-terminal φ. Il est possible de nommer les formules à l'aide d'identificateurs dénotés par le non-terminal θ, tout comme on peut donner un nom aux valeurs grâce aux variables. Ceci améliore l'efficacité en permettant à l'utilisateur de factoriser les sous-formules identiques, qui ne seront ainsi évaluées qu'une seule fois.

Hormis φ, θ et W, la syntaxe des formules logiques utilise les non-terminaux F, G, O[36], X, $\widehat{X}$ et S définis précédemment (§ 2.1.1, p. 25), ainsi que le non-terminal ψ qui dénote les expressions

[36] dans la définition des offres, il convient de remplacer "$\texttt{!}V$" par "$\texttt{!}W$"

temporelles et qui sera présenté ultérieurement (§ 9.3.1, p. 188). La syntaxe d'une formule φ est définie de la manière suivante :

$$
\begin{array}{rcl}
\varphi & \equiv & \textbf{false} \\
 & | & \textbf{true} \\
 & | & \textbf{not } \varphi_0 \\
 & | & \varphi_1 \textbf{ and } \varphi_2 \\
 & | & \varphi_1 \textbf{ or } \varphi_2 \\
 & | & \varphi_1 \textbf{ xor } \varphi_2 \\
 & | & \varphi_1 \texttt{ => } \varphi_2 \\
 & | & \varphi_1 \texttt{ = } \varphi_2 \\
 & | & \textbf{start} \\
 & | & \textbf{stop} \\
 & | & G\ O_1, \ldots\ O_n\ [\textbf{any}]\ \texttt{[}[W_0]\texttt{]} \\
 & | & \textbf{all } \widehat{X_0} : S_0, \ldots\ \widehat{X_n} : S_n \textbf{ in } \varphi' \\
 & | & \textbf{some } \widehat{X_0} : S_0, \ldots\ \widehat{X_n} : S_n \textbf{ in } \varphi' \\
 & | & \textbf{empty } \varphi_0 \\
 & | & \textbf{full } \varphi_0 \\
 & | & \textbf{if } W_0 \textbf{ then } \varphi_1 \textbf{ else } \varphi_2 \\
 & | & \textbf{let } \widehat{X_0} : S_0 \texttt{=} W_0, \ldots\ \widehat{X_n} : S_n \texttt{=} W_n \textbf{ in } \varphi' \\
 & | & \textbf{eval } \theta_0 \texttt{=} \varphi_0, \ldots\ \theta_n \texttt{=} \varphi_n \textbf{ in } \varphi' \\
 & | & \theta \\
 & | & \textbf{first } \texttt{(}\psi\texttt{)} \\
 & | & \textbf{last } \texttt{(}\psi\texttt{)}
\end{array}
$$

Les opérateurs unaires (y compris "**all**", "**some**", "**let**" et "**eval**") sont les plus prioritaires ; ensuite ce sont les opérateurs binaires, et finalement l'opérateur "**if**".

Les variables des listes $\widehat{X_0}, \ldots\ \widehat{X_n}$ présentes dans "**all**", "**some**" et "**let**" constituent des occurrences de définition ; elles ne sont visibles que dans φ'.

Les variables éventuellement contenues dans les offres O_i ayant la forme "$\texttt{?}\widehat{X_i} : S_i$" constituent des occurrences de définition ; elles ne sont visibles que dans W_0.

Les identificateurs de formules $\theta_0, \ldots\ \theta_n$ présentes dans "**eval**" constituent des occurrences de définition ; ils ne sont visibles que dans φ'.

Les constructions syntaxiques définissant les formules se répartissent en six catégories :

- les opérateurs logiques : "**false**", "**true**", "**not**", "**and**", "**or**", "**xor**", "**=>**" et "**=**"
- les prédicats de base : "**start**", "**stop**" et "$G \ldots$"
- les quantificateurs sur les valeurs : "**all**" et "**some**"
- les opérateur méta-logiques : "**empty**" et "**full**"

- les opérateurs de paramétrisation : "**if**", "**let**" et "**eval**"
- les opérateurs temporels : "**first**" et "**last**"

9.2.2 Sémantique des formules logiques

Etant donné un graphe $(\Sigma, \sigma_0, \mathcal{E}, \mathcal{G}, \mathcal{S}, \mathcal{F})$ on doit définir l'interprétation des formules de la logique R$\overset{\star}{\text{I}}$CO sur ce modèle. Le principe est simple : une formule φ représente un ensemble d'arcs du graphe, c'est-à-dire un sous-ensemble de $\mathcal{E}$.

De la même manière qu'un contexte C associe à une variable la valeur LOTOS qu'elle dénote, on définit la notion d'*environnement* Θ : il s'agit d'une application partielle qui, à un identificateur θ fait correspondre un ensemble d'arcs. Les contextes et les environnements servent à exprimer l'action des opérateurs "**all**", "**some**", "**let**" et "**eval**".

La sémantique des formules logiques s'appuie sur une *relation de satisfaction*, notée "$E \overset{\Theta,C}{\models} \varphi$", qui signifie : "l'arc E satisfait la formule φ évaluée dans l'environnement Θ et dans le contexte C". Cette relation, qui caractérise la manière dont les formules sont interprétées, est définie ci-dessous.

Remarque 9-1
Initialement et au plus haut niveau, les formules sont toujours évaluées dans l'environnement $\Theta = \perp$ et dans le contexte $C = \perp$. Il n'en est pas de même des sous-formules qui sont opérandes de "**all**", "**some**", "**let**" et "**eval**". ■

Remarque 9-2
Si θ est un identificateur dénotant une formule φ, alors $\Theta(\theta)$ est l'ensemble des arcs qui satisfont φ (dans un environnement et un contexte donnés). ■

Une formule φ est *valide*, pour un graphe donné, dans un environnement Θ et un contexte C, lorsqu'elle est satisfaite par tous les arcs du graphe :

$$(\forall E \in \mathcal{E}) \;\; E \overset{\Theta,C}{\models} \varphi$$

Remarque 9-3
La détermination des formules valides pour tout graphe n'est pas abordée ici ; cette question n'a qu'un intérêt théorique dans le cadre du *model checking*. ■

L'ensemble des arcs qui satisfont une formule φ est défini par récurrence sur la complexité des formules :

- la formule "**false**" dénote l'ensemble vide ; aucune règle ne lui est associée
- la formule "**true**" dénote l'ensemble $\mathcal{E}$ de tous les arcs :

$$\frac{\mathit{true}}{E \overset{\Theta,C}{\models} \textbf{true}}$$

- la formule "**not** φ_0" dénote le complémentaire de l'ensemble des arcs qui satisfont φ_0 :

$$\frac{\neg(E \overset{\Theta,C}{\models} \varphi_0)}{E \overset{\Theta,C}{\models} \textbf{not } \varphi_0}$$

- la formule "φ_1 **and** φ_2" dénote l'intersection de l'ensemble des arcs qui satisfont φ_1 et de l'ensemble de ceux qui satisfont φ_2 :

$$\frac{(E \overset{\Theta,C}{\models} \varphi_1) \ \wedge \ (E \overset{\Theta,C}{\models} \varphi_2)}{E \overset{\Theta,C}{\models} \varphi_1 \ \textbf{and} \ \varphi_2}$$

- la formule "φ_1 **or** φ_2" dénote l'union de l'ensemble des arcs qui satisfont φ_1 et de l'ensemble de ceux qui satisfont φ_2 :

$$\frac{(E \overset{\Theta,C}{\models} \varphi_1) \ \vee \ (E \overset{\Theta,C}{\models} \varphi_2)}{E \overset{\Theta,C}{\models} \varphi_1 \ \textbf{or} \ \varphi_2}$$

- la formule "φ_1 **xor** φ_2" dénote la différence symétrique de l'ensemble des arcs qui satisfont φ_1 et de l'ensemble de ceux qui satisfont φ_2 :

$$\frac{\neg((E \overset{\Theta,C}{\models} \varphi_1) \Longleftrightarrow (E \overset{\Theta,C}{\models} \varphi_2))}{E \overset{\Theta,C}{\models} \varphi_1 \ \textbf{xor} \ \varphi_2}$$

- la formule "φ_1 `=>` φ_2" dénote l'inclusion de l'ensemble des arcs qui satisfont φ_1 dans l'ensemble de ceux qui satisfont φ_2 :

$$\frac{(E \overset{\Theta,C}{\models} \varphi_1) \Longrightarrow (E \overset{\Theta,C}{\models} \varphi_2)}{E \overset{\Theta,C}{\models} \varphi_1 \ \texttt{=>} \ \varphi_2}$$

- la formule "φ_1 `=` φ_2" dénote l'égalité de l'ensemble des arcs qui satisfont φ_1 et de l'ensemble de ceux qui satisfont φ_2 :

$$\frac{(E \overset{\Theta,C}{\models} \varphi_1) \Longleftrightarrow (E \overset{\Theta,C}{\models} \varphi_2)}{E \overset{\Theta,C}{\models} \varphi_1 \ \texttt{=} \ \varphi_2}$$

- la formule "**start**" dénote l'ensemble des arcs issus de l'état initial σ_0 du graphe

$$\frac{(E = (\sigma_1, L, \sigma_2)) \ \wedge \ (\sigma_1 = \sigma_0)}{E \overset{\Theta,C}{\models} \textbf{start}}$$

- la formule "**stop**" dénote l'ensemble des arcs arrivant dans un état qui n'a aucun successeur

$$\frac{(E = (\sigma_1, L, \sigma_2)) \ \wedge \ (\forall (\sigma'_1, L', \sigma'_2) \in \mathcal{E}) \ \ (\sigma'_1 \neq \sigma_2)}{E \overset{\Theta,C}{\models} \textbf{stop}}$$

- la formule "$G\ O_1, \ldots\ O_n$ [**any**] [`[`W_0`]`]" dénote l'ensemble des arcs dont le label est *compatible* avec la porte G, les offres $O_1, \ldots\ O_n$, et la garde W_0 si elle est présente. Le symbole "**any**", lorsqu'il est présent, signifie que l'on accepte aussi les labels qui ont davantage d'offres, à condition que les premières offres soient compatibles avec $O_1, \ldots\ O_n$ et la garde (en particulier la formule "G **any**" dénote l'ensemble des arcs dont la porte est G, quels que soient le nombre et la nature des offres) :

$$\frac{(E = (\sigma_1, L, \sigma_2)) \ \wedge \ \mathit{compatible}(L, G, (O_1, \ldots\ O_n), W_0, \Theta, C)}{E \overset{\Theta,C}{\models} G\ O_1, \ldots\ O_n\ [\mathbf{any}]\ [[W_0]]}$$

avec :

$\mathit{compatible}(L, G, (O_1, \ldots\ O_n), W_0, \Theta, C) =$
  soient $g, v_1, \ldots\ v_q$ **tels que** $L = g\ v_1, \ldots\ v_q$
  soient $O'_1, \ldots\ O'_p$ **obtenus par linéarisation (§ 2.2.7, p. 35) de** $O_1, \ldots\ O_n$
  si $g \neq G$ **alors false**
  sinon si $\neg((p = q) \ \vee \ (\text{“}\mathbf{any}\text{”}\ \mathit{existe} \ \wedge \ (p < q)))$ **alors false**
  sinon si $(\exists i \in \{1, \ldots\ p\})\ (O'_i \equiv \texttt{!}W_i) \ \wedge \ (\mathit{eval}(W_i, C) \neq v_i)$ **alors false**
  sinon si $(\exists i \in \{1, \ldots\ p\})\ (O'_i \equiv \texttt{?}X_i\!:\!S_i) \ \wedge \ (S_i \neq \mathit{sort}(v_i))$ **alors false**
  sinon si W_0 *existe* **alors**
    soient $C_1, \ldots\ C_p$ **tels que** $C_i = \begin{cases} \textbf{si } O'_i \equiv \texttt{!}W_i \textbf{ alors } \bot \\ \textbf{si } O'_i \equiv \texttt{?}X_i\!:\!S_i \textbf{ alors } (X_i \rightsquigarrow v_i) \end{cases}$
    si $\mathit{eval}(W_0, C \oslash (C_1 \oplus \ldots\ C_p)) \neq \mathit{true}$ **alors false**
    sinon true

- la formule “**all** $\widehat{X_0}\!:\!S_0, \ldots\ \widehat{X_n}\!:\!S_n$ **in** φ'” dénote l'intersection généralisée des ensembles d'arcs satisfaisant la formule φ' lorsque les variables $\widehat{X_0}, \ldots\ \widehat{X_n}$ décrivent respectivement le domaine[37] des sortes $S_0, \ldots\ S_n$; il s'agit du quantificateur universel du 1[er] ordre :

$$\frac{(\forall C' \in \mathit{iterate}(\widehat{X_0}\!:\!S_0, \ldots\ \widehat{X_n}\!:\!S_n))\ \ E \overset{\Theta,C\oslash C'}{\models} \varphi'}{E \overset{\Theta,C}{\models} \mathbf{all}\ \widehat{X_0}\!:\!S_0, \ldots\ \widehat{X_n}\!:\!S_n\ \mathbf{in}\ \varphi'}$$

avec :

$\mathit{iterate}(\widehat{X_0}\!:\!S_0, \ldots\ \widehat{X_n}\!:\!S_n) =$
  soient $X_0, \ldots\ X_p$ **tels que** $\widehat{X_0} \equiv X_0, \ldots\ X_p$
  si $(p = 0) \ \wedge \ (n = 0)$ **alors**
    $\{X_0 \rightsquigarrow V \mid V \in \mathit{domain}(S_0)\}$
  sinon si $(p = 0) \ \wedge \ (n > 0)$ **alors**
    $\{(X_0 \rightsquigarrow V) \oplus C \mid (V \in \mathit{domain}(S_0)) \ \wedge \ (C \in \mathit{iterate}(\widehat{X_1}\!:\!S_1, \ldots\ \widehat{X_n}\!:\!S_n))\}$
  sinon si $p > 0$ **alors**
    $\{(X_0 \rightsquigarrow V) \oplus C \mid (V \in \mathit{domain}(S_0)) \ \wedge \ (C \in \mathit{iterate}(\widehat{X'_0}\!:\!S_0, \widehat{X_1}\!:\!S_1, \ldots\ \widehat{X_n}\!:\!S_n))\}$
    où $\widehat{X'_0} \equiv X_1, \ldots\ X_p$

[37]que l'on suppose fini par hypothèse

- la formule "**some** $\widehat{X_0}:S_0,\dots\ \widehat{X_n}:S_n$ **in** φ'" dénote l'union généralisée des ensembles d'arcs satisfaisant la formule φ' lorsque les variables $\widehat{X_0},\dots\ \widehat{X_n}$ décrivent respectivement le domaine des sortes $S_0,\dots\ S_n$; il s'agit du quantificateur existentiel du 1[er] ordre :

$$\frac{(\exists C' \in \mathit{iterate}(\widehat{X_0}:S_0,\dots\ \widehat{X_n}:S_n))\ \ E \overset{\Theta,C\oslash C'}{\models} \varphi'}{E \overset{\Theta,C}{\models} \textbf{some}\ \widehat{X_0}:S_0,\dots\ \widehat{X_n}:S_n\ \textbf{in}\ \varphi'}$$

- la formule "**empty** φ_0" dénote, soit l'ensemble $\mathcal{E}$ si aucun arc ne satisfait φ_0 (c'est-à-dire si **not** φ_0 est valide), soit l'ensemble vide dans le cas contraire :

$$\frac{(\forall E' \in \mathcal{E})\ \ \neg E' \overset{\Theta,C}{\models} \varphi_0}{E \overset{\Theta,C}{\models} \textbf{empty}\ \varphi_0}$$

- la formule "**full** φ_0" dénote, soit l'ensemble $\mathcal{E}$ si tous les arcs satisfont φ_0 (c'est-à-dire si φ_0 est valide), soit l'ensemble vide dans le cas contraire :

$$\frac{(\forall E' \in \mathcal{E})\ \ E' \overset{\Theta,C}{\models} \varphi_0}{E \overset{\Theta,C}{\models} \textbf{full}\ \varphi_0}$$

- la formule "**if** W_0 **then** φ_1 **else** φ_2" dénote l'ensemble des arcs qui satisfont, soit la formule φ_1 si la condition W_0 est vraie, soit la formule φ_2 sinon :

$$\frac{((\mathit{eval}(W_0,C) = \mathit{true})\ \wedge\ (E \overset{\Theta,C}{\models} \varphi_1))\ \vee\ ((\mathit{eval}(W_0,C) = \mathit{false})\ \wedge\ (E \overset{\Theta,C}{\models} \varphi_2))}{E \overset{\Theta,C}{\models} \textbf{if}\ W_0\ \textbf{then}\ \varphi_1\ \textbf{else}\ \varphi_2}$$

- la formule "**let** $\widehat{X_0}:S_0$=$W_0,\dots\ \widehat{X_n}:S_n$=$W_n$ **in** φ'" dénote l'ensemble des arcs qui satisfont la formule φ' dans laquelle les variables $\widehat{X_0},\dots\ \widehat{X_n}$, de sortes respectives $S_0,\dots\ S_n$ sont remplacées par les valeurs de $W_0,\dots\ W_n$:

$$\frac{(C' = (\widehat{X_0} \rightsquigarrow \mathit{eval}(W_0,C)) \oplus \dots\ (\widehat{X_n} \rightsquigarrow \mathit{eval}(W_n,C)))\ \wedge\ (E \overset{\Theta,C\oslash C'}{\models} \varphi')}{E \overset{\Theta,C}{\models} \textbf{let}\ \widehat{X_0}:S_0{=}W_0,\dots\ \widehat{X_n}:S_n{=}W_n\ \textbf{in}\ \varphi'}$$

- la formule "**eval** θ_0=$\varphi_0,\dots\ \theta_n$=$\varphi_n$ **in** φ'" dénote l'ensemble des arcs qui satisfont la formule φ' dans laquelle les identificateurs $\theta_0,\dots\ \theta_n$ représentent les ensembles d'arcs qui satisfont les formules $\varphi_0,\dots\ \varphi_n$, respectivement :

$$\frac{(\Theta' = (\theta_0 \rightsquigarrow \{E' \mid E' \overset{\Theta,C}{\models} \varphi_0\}) \oplus \dots\ (\theta_n \rightsquigarrow \{E' \mid E' \overset{\Theta,C}{\models} \varphi_n\}))\ \wedge\ (E \overset{\Theta\oslash\Theta',C}{\models} \varphi')}{E \overset{\Theta,C}{\models} \textbf{eval}\ \theta_0{=}\varphi_0,\dots\ \theta_n{=}\varphi_n\ \textbf{in}\ \varphi'}$$

- la formule "θ" dénote l'ensemble des arcs appartenant à $\Theta(\mathit{theta})$:

$$\frac{E \in \Theta(\theta)}{E \overset{\Theta,C}{\models} \theta}$$

- la sémantique des formules "**first** `(`ψ`)`" et "**last** `(`ψ`)`", qui permettent de prendre en compte les aspects temporels, sera définie ultérieurement (§ 9.3.3, p. 190)

9.2.3 Applications

Les formules de la logique R$\overset{\star}{\text{I}}$CO obtenues en se restreignant aux aspects logiques (sans utiliser "**first**" et "**last**"), permettent d'exprimer simplement des propriétés utiles. En voici quelques exemples :

absence de terminaison : l'exécution ne se termine jamais ; tous les arcs du graphe ont au moins un successeur :

not stop

absence de blocage : il n'y a pas de *blocage* (*deadlock*) si l'exécution, lorsqu'elle se termine, le fait de manière conforme aux conventions du langage LOTOS ; tout arc qui n'a pas de successeur doit être étiqueté par la porte "δ", que l'utilisateur note "**exit**" :

stop = exit any

propriétés de sûreté : ces propriétés (*safety properties*) expriment que certaines situations fâcheuses ne peuvent jamais se produire. Un cas simple est l'absence de certains événements tels que, par exemple, l'occurrence d'un signal pour lequel l'adresse de l'expéditeur est identique à celle du destinataire :

empty `SIGNAL ?SOURCE,DESTINATION:ADDRESS [SOURCE = DESTINATION]`

De même on peut vérifier que tous les signaux comportent exactement deux adresses :

empty `(SIGNAL` **any and not** `SIGNAL ?SOURCE,DESTINATION:ADDRESS)`

propriétés de vivacité : ces propriétés (*liveness properties*) expriment que certaines situations favorables peuvent ou doivent se produire. Par exemple, on peut vérifier qu'il est possible d'avoir un signal avec deux adresses `SOURCE` et `DESTINATION` fixées :

not empty `SIGNAL !SOURCE !DESTINATION`

Bien entendu, pour exprimer des propriétés de sûreté ou de vivacité plus élaborées (c'est-à-dire ne portant plus sur des événements isolés, mais sur des séquences plus complexes d'événements) il faudra avoir recours aux aspects temporels

propriétés du 1$^{\text{er}}$ ordre : les logiques temporelles usuelles ne possèdent pas de quantificateurs sur les valeurs, ce qui oblige en pratique l'utilisateur à écrire des suites fastidieuses de formules à vérifier (une formule par valeur possible de la variable quantifiée). Les opérateurs "**all**" et "**some**" — qui représentent les opérations "**and**" et "**or**" généralisées — de la logique R$\overset{\star}{\text{I}}$CO autorisent des descriptions plus concises.

Outre ces quantificateurs sur les valeurs, la logique R$\overset{\star}{\text{I}}$CO possède des quantificateurs sur les arcs : "**not empty** φ" et "**full** φ" signifient respectivement "il existe un arc satisfaisant φ" et "tous les arcs satisfont φ".

Remarque 9-4
Les opérateurs "**empty**" et "**full**" sont appelés *méta-logiques* parce qu'ils permettent de convertir des méta-formules ("φ est valide", "**not** φ est valide") comme de simples formules ; ils établissent une "passerelle" entre la logique R$\overset{\star}{\text{I}}$CO et la méta-logique employée pour la définir. ■

La combinaison des quatre quantificateurs autorise l'expression d'une large classe de modalités :

– s'il existe un arc "SEND", alors il existe un arc "RECV" :

not empty (SEND) => **not empty** (RECV)

– les arcs initiaux sont tous étiquetés par "SEND" ou "RECV" :

full (**start** => (SEND **or** RECV))

– les arcs initiaux sont tous étiquetés par "SEND" ou bien tous par "RECV" :

full (**start** => SEND) **or full** (**start** => RECV)

– pour tout entier N, il existe un arc "DATA" portant le numéro N :

all N:NUM **in not empty** (DATA !N)

– il existe un entier N tel qu'au moins un arc "DATA" porte le numéro N :

some N:NUM **in not empty** (DATA !N)

– il existe un entier N tel que tous les arcs "DATA" portent le numéro N :

some N:NUM **in full** (DATA **any** => DATA !N)

– il existe un entier N qui majore tous les numéros portés par les arcs "DATA" :

some N:NUM **in full** (DATA **any** => DATA ?M:NUM [M $<$ N])

– pour tout entier N, il existe un arc "SEND !N" si et seulement si il existe un arc "RECV !N" :

all N:NUM **in** (**not empty** (SEND !N) = **not empty** (RECV !N))

– pour tout entier N non nul, s'il existe un arc "DATA !N" alors il existe un arc "DATA !N-1" :

all N:NUM **in**
  if N $=$ 0 **then true**
  else (**not empty** (DATA !N) => **not empty** (DATA !N-1))

– pour tous entiers M et N distincts, il existe un arc "DATA !M !N" :

all M,N:NUM **in**
  if M $=$ N **then true**
  else not empty (DATA !M !N)

9.3 Aspects temporels

Les formules logiques expriment des propriétés ensemblistes sur les arcs du graphe, sans faire intervenir la relation de transition. Elles ne suffisent pas, à elles seules, à décrire toutes les propriétés utiles, notamment celles qui portent sur les chemins du graphe. C'est pourquoi la logique R$\star$ICO comporte une autre classe d'objets : les *expressions temporelles*. Il s'agit essentiellement d'une généralisation des expressions régulières construites sur le vocabulaire des formules.

9.3.1 Syntaxe des expressions temporelles

Les expressions temporelles sont dénotées par le non-terminal ψ qui est défini comme suit :

$$
\begin{array}{rcl}
\psi & \equiv & \mathbf{null} \\
 & | & \varphi \\
 & | & \psi_1 \ \mathtt{.} \ \psi_2 \\
 & | & \psi_1 \ \mathtt{;} \ \psi_2 \\
 & | & \psi_1 \ \mathtt{|} \ \psi_2 \\
 & | & \psi_1 \ \mathtt{\&} \ \psi_2 \\
 & | & \psi_0\mathtt{*} \\
 & | & \psi_0\mathtt{+} \\
 & | & \psi_0\mathtt{@}
\end{array}
$$

Les opérateurs unaires "`*`", "`+`" et "`@`" sont les plus prioritaires ; ensuite ce sont les opérateurs binaires "`.`" et "`;`", et enfin par les opérateurs "`|`" et "`&`".

9.3.2 Sémantique des expressions temporelles

Comme pour les formules logiques, la sémantique des expressions temporelles est définie sur le modèle graphe, en termes de séquences d'exécution.

Deux arcs E et E' sont *consécutifs*, ce que l'on note "$E \rightarrow E'$", si et seulement E' est un successeur de E, c'est-à-dire :

$$(\exists \sigma, \sigma', \sigma'') \ (\exists L, L') \ (E = (\sigma, L, \sigma')) \ \wedge \ (E' = (\sigma', L', \sigma''))$$

Un ensemble *fini* d'arcs $E_1, \ldots \ E_n$ de $\mathcal{E}$ forme une *séquence*, ce que l'on note "$\ll E_1, \ldots \ E_n \gg$", si et seulement si ces arcs sont deux à deux consécutifs :

$$(\forall i \in \{1, \ldots \ n-1\}) \ \ E_i \rightarrow E_{i+1}$$

La séquence vide, obtenue pour $n = 0$, est notée "$\ll \gg$".

Remarque 9-5
Il ne faut pas confondre une séquence d'arcs $\ll (\sigma_1, L_1, \sigma_2), \ldots \ (\sigma_n, L_n, \sigma_{n+1}) \gg$ avec le mot constitué par les labels de ces arcs $L_1, \ldots \ L_n$. ■

Etant donné un graphe, une expression temporelle représente un ensemble — éventuellement infini — de séquences du graphe. Plus précisément on définit une relation notée "$\ll E_1, \ldots \ E_n \gg \overset{\Theta,C}{\sqsubset} \ \psi$" qui signifie "la séquence $\ll E_1, \ldots \ E_n \gg$ vérifie l'expression temporelle ψ, évaluée dans l'environnement Θ et dans le contexte C". Cette relation est définie par récurrence sur la complexité des expressions temporelles :

- l'expression "**null**" dénote l'ensemble réduit à la séquence vide :

$$\frac{true}{\ll \gg \overset{\Theta,C}{\sqsubset} \ \mathbf{null}}$$

- l'expression "φ" dénote l'ensemble des séquences de la forme $\ll E \gg$ où E est un arc qui satisfait la formule φ :

$$\frac{E \overset{\Theta,C}{\models} \varphi}{\ll E \gg \overset{\Theta,C}{\sqsubset} \varphi}$$

- l'expression "ψ_1 . ψ_2" dénote l'ensemble des séquences obtenues par concaténation de deux séquences, l'une qui satisfait ψ_1 et l'autre ψ_2 :

$$\frac{(\ll E_1, \ldots\ E_n \gg \overset{\Theta,C}{\sqsubset} \psi_1) \wedge (\ll E'_1, \ldots\ E'_{n'} \gg \overset{\Theta,C}{\sqsubset} \psi_2) \wedge ((n \geq 1) \wedge (n' \geq 1) \Longrightarrow E_n \rightarrow E'_1)}{\ll E_1, \ldots\ E_n, E'_1, \ldots\ E'_{n'} \gg \overset{\Theta,C}{\sqsubset} \psi_1 \texttt{ . } \psi_2}$$

- l'expression "ψ_1 ; ψ_2" dénote l'ensemble des séquences obtenues par concaténation de deux séquences, l'une qui satisfait ψ_1, l'autre ψ_2, telles que le dernier arc de la première soit identique au premier arc de la seconde :

$$\frac{(\ll E_1, \ldots\ E_n \gg \overset{\Theta,C}{\sqsubset} \psi_1) \wedge (\ll E'_1, \ldots\ E'_{n'} \gg \overset{\Theta,C}{\sqsubset} \psi_2) \wedge ((n \geq 1) \wedge (n' \geq 1) \Longrightarrow E_n = E'_1)}{\ll E_1, \ldots\ E_{n-1}, E'_1, \ldots\ E'_{n'} \gg \overset{\Theta,C}{\sqsubset} \psi_1 \texttt{ ; } \psi_2}$$

- l'expression "ψ_1 | ψ_2" dénote l'ensemble des séquences qui satisfont ψ_1 ou ψ_2 :

$$\frac{(\ll E_1, \ldots\ E_n \gg \overset{\Theta,C}{\sqsubset} \psi_1) \vee (\ll E_1, \ldots\ E_n \gg \overset{\Theta,C}{\sqsubset} \psi_2)}{\ll E_1, \ldots\ E_n \gg \overset{\Theta,C}{\sqsubset} \psi_1 \texttt{ | } \psi_2}$$

- l'expression "ψ_1 & ψ_2" dénote l'ensemble des séquences qui satisfont ψ_1 et ψ_2 :

$$\frac{(\ll E_1, \ldots\ E_n \gg \overset{\Theta,C}{\sqsubset} \psi_1) \wedge (\ll E_1, \ldots\ E_n \gg \overset{\Theta,C}{\sqsubset} \psi_2)}{\ll E_1, \ldots\ E_n \gg \overset{\Theta,C}{\sqsubset} \psi_1 \texttt{ \& } \psi_2}$$

- l'expression "ψ_0*" dénote l'ensemble des séquences obtenues par concaténation d'un nombre fini, éventuellement nul, de séquences qui satisfont ψ_0 :

$$\frac{\mathit{true}}{\ll \gg \overset{\Theta,C}{\sqsubset} \psi_0\texttt{*}}$$

$$\frac{(\ll E_1, \ldots\ E_n \gg \overset{\Theta,C}{\sqsubset} \psi_0) \wedge (\ll E'_1, \ldots\ E'_{n'} \gg \overset{\Theta,C}{\sqsubset} \psi_0\texttt{*}) \wedge ((n \geq 1) \wedge (n' \geq 1) \Longrightarrow E_n \rightarrow E'_1)}{\ll E_1, \ldots\ E_n, E'_1, \ldots\ E'_{n'} \gg \overset{\Theta,C}{\sqsubset} \psi_0\texttt{*}}$$

- l'expression "ψ_0+" dénote l'ensemble des séquences obtenues par concaténation d'un nombre fini non nul de séquences qui satisfont ψ_0 :

$$\frac{(\ll E_1, \ldots\ E_n \gg \overset{\Theta,C}{\sqsubset} \psi_0) \wedge (\ll E'_1, \ldots\ E'_{n'} \gg \overset{\Theta,C}{\sqsubset} \psi_0\texttt{*}) \wedge ((n \geq 1) \wedge (n' \geq 1) \Longrightarrow E_n \rightarrow E'_1)}{\ll E_1, \ldots\ E_n, E'_1, \ldots\ E'_{n'} \gg \overset{\Theta,C}{\sqsubset} \psi_0\texttt{+}}$$

- l'expression "ψ_0@" dénote l'ensemble des séquences cycliques qui satisfont ψ_0+ :

$$\frac{(\ll E_1, \ldots\ E_n \gg \overset{\Theta,C}{\sqsubset} \psi_0\texttt{+}) \wedge (n \geq 1 \Longrightarrow E_n \rightarrow E_1)}{\ll E_1, \ldots\ E_n \gg \overset{\Theta,C}{\sqsubset} \psi_0\texttt{@}}$$

9.3.3 Lien entre formules logiques et expressions temporelles

Les relations entre les formules φ et les expressions ψ sont complètement déterminées par la définition des opérateurs "**first**" et "**last**" :

- la formule "**first** (ψ)" dénote l'ensemble des arcs E tels qu'il existe une séquence non vide qui satisfasse ψ et dont E soit le premier arc :

$$\frac{(\exists \ll E_0, \ldots\ E_n \gg)\ \ (\ll E_0, \ldots\ E_n \gg \overset{\Theta,C}{\sqsubset}\ \psi)\ \wedge\ (E = E_0)}{E\ \overset{\Theta,C}{\models}\ \textbf{first}\ (\psi)}$$

- la formule "**last** (ψ)" dénote l'ensemble des arcs E tels qu'il existe une séquence non vide qui satisfasse ψ et dont E soit le dernier arc :

$$\frac{(\exists \ll E_0, \ldots\ E_n \gg)\ \ (\ll E_0, \ldots\ E_n \gg \overset{\Theta,C}{\sqsubset}\ \psi)\ \wedge\ (E = E_n)}{E\ \overset{\Theta,C}{\models}\ \textbf{last}\ (\psi)}$$

Malgré les apparences, il existe une différence fondamentale entre les formules et les expressions : seules les formules peuvent être évaluées. Concrètement cela implique que l'utilisateur ne peut pas demander au système de vérifier une expression temporelle ψ. En effet l'ensemble des séquences qui satisfont ψ pourrait être infini et il n'existerait peut-être pas de procédure de décision. En revanche l'utilisateur peut évaluer les formules "**first** (ψ)" et "**last** (ψ)" ce qui permet de savoir, en particulier, s'il existe ou non une séquence satisfaisant ψ.

9.3.4 Applications

L'utilisation des expressions temporelles permet la spécification de propriétés plus élaborées que ne l'autorisent les seules formules logiques (sans "**first**" et "**last**") : ces dernières expriment uniquement des propriétés sur les arcs — c'est-à-dire les séquences de longueur 1 — alors que les expressions temporelles portent sur des séquences de longueur quelconque.

L'expérience montre qu'il faut donner à l'utilisateur la possibilité d'enrichir la syntaxe de base de la logique en y ajoutant ses propres constructions, adaptées aux propriétés qu'il veut vérifier. La méthode la plus simple pour rendre la logique R$\overset{\star}{\text{I}}$CO *extensible* est d'utiliser un macro-processeur existant[38].

La nature même de la logique R$\overset{\star}{\text{I}}$CO induit un style de spécification assez différent de celui des logiques classiques. On propose ici, à travers différents exemples, des macros-définitions générales permettant d'exprimer facilement les propriétés usuelles :

absence de famine : il n'y a pas de *famine* (*livelock*) si le graphe ne comporte aucun circuit dont tous les arcs sont étiquetés par l'action interne "**i**" :

empty first `(i@)`

propriétés de sûreté : les expressions temporelles sont particulièrement bien adaptées aux propriétés de sûreté, qui expriment le fait que certaines évolutions sont impossibles. Dans la logique

[38] par exemple `m4(1)` ou `/lib/cpp` sous UNIX

R$\stackrel{\star}{\text{I}}$CO, ces propriétés ont généralement la forme "**never** ψ", où "**never**" est un macro-opérateur qui peut être défini de deux manières équivalentes :

macro	*définition*
never (ψ)	(**empty first** (ψ))
never (ψ)	(**empty last** (ψ))

Voici quelques exemples de propriétés de sûreté :

- il est impossible qu'une action "`SEND !NO`" soit suivie, après un nombre quelconque d'actions internes, d'une action "`RECV !N`" où `N` est différent de `NO` :

  never (`SEND !NO .` **i**∗ `. (RECV ?N:NUM [N` $\neq$ `NO]))`

- il est impossible d'avoir deux actions "`SEND`" successives, sans qu'une action "`RECV`" n'intervienne entre temps :

  never `(SEND . (`**not** `RECV)* . SEND)`

propriétés de vivacité : les propriétés ayant la forme "il existe une séquence qui vérifie ψ" s'expriment aisément à l'aide du macro-opérateur "**sometimes**" défini par :

macro	*définition*
sometimes (ψ)	(**not never** (ψ))

C'est ainsi que l'on peut spécifier aisément des propriétés comme :

- il est possible d'effectuer une séquence de quatre actions visibles[39] `CONREQ`, `CONIND`, `CONRESP` et `CONCONF`, entre lesquelles peuvent intervenir des actions invisibles "**i**" :

  sometimes `(CONREQ .` **i**∗ `. CONIND .` **i**∗ `. CONRESP .` **i**∗ `. CONCONF)`

- après une action "`SEND !N`" il est possible d'avoir une action "`RECV !N`", quel que soit `N` :

  all `N:NUM` **in sometimes** `(SEND !N .` **i**∗ `. RECV !N)`

- avant une action "`SEND !N`", il est possible d'avoir une action "`SEND !N-1`", si `N` non nul :

  all `N:NUM` **in**
  if `N` $=$ `0` **then true**
  else sometimes `(SEND !N-1 .` **i**∗ `. SEND !N)`

En revanche d'autres propriétés de vivacité ne se formulent pas aussi directement : il faut les exprimer de manière duale en les remplaçant par des propriétés de sûreté. Par exemple il faut transformer "toute séquence vérifie ψ"en "il n'existe aucune séquence qui ne vérifie pas ψ". Les exemples suivants illustrent cette démarche :

- pour exprimer qu'après une action "`QUIT`" la terminaison est inévitable (autrement dit qu'après une action "`QUIT`" suivie d'un nombre fini d'actions quelconques, on arrive inévitablement à un arc satisfaisant "**stop**"), on spécifie qu'il n'est pas possible d'atteindre, à partir d'un arc "`QUIT`", un circuit dans le graphe :

  never `(QUIT .` **true**∗ `.` **true**`@)`

[39]selon le principe d'ouverture d'une connexion OSI

- pour exprimer que toute action "RECV" est inévitablement précédée d'une action "SEND", on spécifie qu'il est impossible d'atteindre un arc "RECV" à partir d'un arc satisfaisant "**start**" (mais pas "SEND") et après avoir franchi un nombre fini d'arcs autres que "SEND"

  eval NOT_SEND = **not** SEND **in**
  never (**start** ; NOT_SEND* ; RECV)

  L'opérateur ";" sert à détecter le cas où un arc satisferait simultanément "**start**" et "RECV". En règle générale, il faut utiliser ";" au lieu de "." lorsque l'un des opérandes est "**start**" ou "**stop**"

- pour exprimer que toute action "SEND" est inévitablement suivie d'une action "RECV", on spécifie qu'il est impossible d'atteindre, à partir d'un arc "SEND" et après avoir franchi un nombre fini d'arcs autres que "RECV", soit un arc satisfaisant "**stop**" (mais pas "RECV") (*deadlock*), soit un circuit dont tous les arcs sont autres que "RECV" (*livelock*) :

  eval NOT_RECV = **not** RECV **in**
  never (SEND ; NOT_RECV* ; (**stop** | NOT_RECV@))

9.4 Comparaison avec les logiques temporelles existantes

La logique R$\overset{\star}{\text{I}}$CO possède des aspects originaux qui sont mis en évidence par la comparaison avec les logiques existantes, par rapport auxquelles elle présente des différences sensibles :

propriétés sur les états ou sur les arcs : la logique R$\overset{\star}{\text{I}}$CO diffère en cela des logiques temporelles classiques (notamment CTL, STL et LTAC) dont les formules dénotent des ensembles d'états. En effet pour le modèle graphe sous-jacent à LOTOS, aucune information n'est associée aux états ; le contenu des marquages et des contextes n'est pas exploitable car il n'est pas facile de retrouver la correspondance entre les places, les variables d'état du réseau et le texte source LOTOS. Pour l'utilisateur, les seules informations pertinentes sont les labels qui étiquettent les arcs du graphe.

Ce choix donne aux formules R$\overset{\star}{\text{I}}$CO une certaine abstraction par rapport aux spécifications LOTOS qu'elles décrivent. Les propriétés que l'on peut exprimer concernent seulement les actions observables du système, c'est-à-dire l'interface présenté aux utilisateurs, et non pas l'état des composants internes du système. La possibilité de consulter la valeur des variables d'état[40] est souvent nuisible car elle rend les formules étroitement dépendantes du programme à vérifier, ce qui encourage le manque d'abstraction et la sur-spécification. En outre l'utilisation d'informations aussi détaillées constitue un obstacle aux techniques de réduction des graphes

propriétés linéaires et arborescentes : on divise usuellement les logiques temporelles en deux classes selon que leurs formules caractérisent des séquences du graphe (*logiques linéaires*) ou des états (*logiques arborescentes*).

Comme CTL, STL et LTAC, la logique R$\overset{\star}{\text{I}}$CO est, dans son principe, arborescente (bien que ses formules s'appliquent à des ensembles d'arcs et non d'états). Pour s'en convaincre il suffit de constater que l'on peut exprimer la propriété "le graphe comporte au moins un état duquel partent deux arcs étiquetés, l'un par G_1, l'autre par G_2", à l'aide de la formule suivante :

not empty (**first** (**true** . G_1) **and first** (**true** . G_2))

[40]sauf évidemment les constantes qui décrivent les paramètres du système (nombre de tâches, taille des files d'attente, ...) ainsi que les variables qui, dans certains langages comme ESTELLE, correspondent aux informations échangées via les canaux de communication, et dont la connaissance est indispensable

Typiquement il s'agit d'une propriété qui ne peut pas être caractérisée en logique linéaire.

Bien que la logique R$\overset{\star}{\text{I}}$CO soit arborescente, elle possède les moyens d'expression des logiques linéaires. En effet, une expression temporelle définit un ensemble de séquences, caractéristique propre aux logiques linéaires. On peut ainsi décrire avec précision des séquences complexes d'événements, chose particulièrement malaisée en logique arborescente. La dualité φ/ψ permet de concilier les mérites des approches arborescente/linéaire respectivement.

Remarque 9-6
Les propriétés des opérateurs "|" et "&" illustrent ce propos. Comme en logique linéaire on a :

$$\textbf{first}\ \texttt{(}G_1\ \texttt{.}\ \texttt{(}G_2\ \texttt{|}\ G_3\texttt{))} = \textbf{first}\ \texttt{(}G_1\ \texttt{.}\ G_2\ \texttt{|}\ G_1\ \texttt{.}\ G_3\texttt{)}$$

alors que, comme en logique arborescente :

$$\textbf{first}\ \texttt{(}G_1\ \texttt{.}\ \texttt{(}G_2\ \texttt{\&}\ G_3\texttt{))} \subseteq \textbf{first}\ \texttt{(}G_1\ \texttt{.}\ G_2\ \texttt{\&}\ G_1\ \texttt{.}\ G_3\texttt{)}$$

■

μ-calculs ou expressions régulières : l'expérience montre qu'il faut enrichir les logiques arborescentes classiques, jugées peu adaptées à la spécification des séquences d'actions. Par exemple, il n'est pas possible d'exprimer simplement avec les seuls opérateurs temporels de la logique CTL qu'après une émission S, il est possible d'avoir successivement un nombre quelconque d'actions internes "**i**" entre lesquelles intervient une réception R et une seule, puis un acquittement A.

Pour résoudre ce problème, les logiques STL et LTAC sont basées sur le μ-calcul [Koz82] ; c'est un formalisme puissant mais auquel on peut reprocher sa complexité. En μ-calcul la propriété précédente s'écrit :

$$\textbf{after}\ \texttt{(}S\texttt{)}\ \texttt{=>}\ \mu x\ \texttt{(}<\textbf{i}>\ x\ \textbf{or}\ \texttt{(}<R>\ \mu y\ \texttt{(}<\textbf{i}>\ y\ \textbf{or}\ <A>\ \textbf{true}\texttt{)))}$$

Au lieu du μ-calcul, la logique R$\overset{\star}{\text{I}}$CO est basée sur des expressions régulières, qui présentent l'avantage d'être simples d'emploi et bien connues[41]. Dans la logique R$\overset{\star}{\text{I}}$CO la propriété précédente se formule beaucoup plus simplement :

$$\textbf{sometimes}\ \texttt{(}S\ \texttt{.}\ \textbf{i}\texttt{*}\ \texttt{.}\ R\ \texttt{.}\ \textbf{i}\texttt{*}\ \texttt{.}\ A\texttt{)}$$

puissance d'expression : la logique R$\overset{\star}{\text{I}}$CO permet de retrouver comme cas particuliers, les opérateurs de la logique CTL (étant entendu que l'on manipule des ensembles d'arcs et non des ensembles d'états) :

- l'opérateur "**pre** (φ)" caractérise l'ensemble des arcs qui possèdent un arc consécutif satisfaisant φ :

macro	*définition*
pre (φ)	(**first** (**true** . (φ)))

- l'opérateur "**pot** (φ_1, φ_2)" caractérise l'ensemble des arcs à partir desquels il est *possible* d'atteindre un arc vérifiant une formule φ_2 après avoir franchi une séquence pendant laquelle une formule φ_1 est constamment vérifiée :

macro	*définition*
pot (φ_1, φ_2)	(**first** (**always** (φ_1) . φ_2))

[41] les expressions régulières sont d'un usage courant dans les langages de commandes (*shells*), les éditeurs de texte, les générateurs de compilateurs, ...

– l'opérateur "**finev** (φ_1, φ_2)" caractérise l'ensemble des arcs à partir desquels il est *inévitable* d'atteindre un arc vérifiant une formule φ_2 après avoir franchi une séquence pendant laquelle une formule φ_1 est constamment vérifiée ; cet opérateur, défini dans [QS83], est *équitable* car il suppose que l'on ne reste pas indéfiniment dans un circuit du graphe :

macro	*définition*
finev (φ_1, φ_2)	(**not pot** (**not** (φ_2), **not pot** (φ_1, φ_2)))

– l'opérateur "**inev** (φ_1, φ_2)" caractérise l'ensemble des arcs à partir desquels il est *inévitable* d'atteindre un arc vérifiant une formule φ_2 après avoir franchi une séquence pendant laquelle une formule φ_1 est constamment vérifiée ; cet opérateur est plus fort que "**finev**" car il exige que toutes les séquences conduisant à φ_2 soient finies, ce qui exclut la présence de circuits :

macro	*définition*
inev (φ_1, φ_2)	(**finev** (φ_1, φ_2) **and not first** ((**not** (φ_2))* . (**not** φ_2)@))

On peut enfin remarquer que les opérateurs "**empty**" et "**full**" possèdent des équivalents en logique CTL ; la correspondance est la suivante :

full φ	**init** => **all** φ
empty φ	**init** => **not pot** φ

propriétés sur le passé et sur le futur : les logiques CTL, STL et LTAC permettent d'exprimer des propriétés portant sur le futur. On peut ainsi spécifier qu'une action X a comme conséquence (possible, inévitable, ...) une action postérieure Y, par exemple *après avoir fait une requête, on accède inévitablement à la ressource* :

after (X) => **inev enable** (Y)

En revanche elles n'autorisent pas directement les propriétés portant sur le passé comme, par exemple, le fait qu'une action Y ait comme cause (possible, inévitable, ...) une action X. Ces propriétés, du style *pour accéder à la ressource, il faut au préalable avoir fait une requête*, doivent être exprimées de manière totalement différente des précédentes :

init => **not pot** [**not after** (X)] **after** (Y)

Au contraire la logique R$\overset{\star}{\text{I}}$CO permet de manipuler passé et futur de manière parfaitement symétrique et d'avoir, pour les deux situations, des formules duales. Par exemple, l'opérateur "**post**", dual de l'opérateur "**pre**", est défini par :

macro	*définition*
post (φ)	(**last** ((φ) . **true**))

et l'on pourrait faire de même avec les autres opérateurs

propriétés du 1$^{\text{er}}$ ordre : tout comme les langages de description du parallélisme sont inutilisables lorsqu'ils ne comportent pas de valeurs, les logiques temporelles qui ne permettent pas d'intégrer pleinement les données sont d'un emploi malaisé. Cela conduit en pratique à des situations absurdes dans lesquelles le programme à valider est correctement paramétré par des constantes (taille des tampons, taille des fenêtres d'émission et de réception, ...) alors que sa spécification en logique temporelle ne l'est pas, ce qui oblige à la modifier chaque fois que l'on change la valeur d'un paramètre.

La logique R$\overset{\star}{\text{I}}$CO comporte des constructions qui répondent à ce besoin : elle permet de donner des noms aux valeurs ("**let**") et aux formules ("**eval**") ; elle possède en outre les quantificateurs universel ("**all**") et existentiel ("**some**") du 1^{er} ordre sur les valeurs. Ces facilités sont complétées par l'emploi d'un macro-processeur

formalismes mixtes : l'intérêt se porte actuellement vers des langages permettant à l'utilisateur de combiner propriétés en logique temporelle et comportements de type automate. La logique R$\overset{\star}{\text{I}}$CO en est un exemple : les comportements sont décrits par les expressions temporelles. Il aurait été possible d'utiliser directement LOTOS pour spécifier les automates, mais LOTOS est trop riche pour les propriétés que l'on veut décrire ; en outre il n'offre pas le moyen de distinguer "`|`" de "`&`", ni "`*`" de "`@`"

interprétation des résultats : lorsqu'une formule n'est pas valide, le système de vérification devrait fournir des explications permettant à l'utilisateur de comprendre pourquoi le modèle ne satisfait pas la propriété requise. Dans ce but, une méthode sophistiquée a été élaborée pour la logique LTAC utilisée dans XESAR [Ras88] : elle construit un ensemble de séquences du graphe qui invalident une formule donnée.

Pour la logique R$\overset{\star}{\text{I}}$CO, la production de contre-exemples est probablement plus aisée, puisque les propriétés sont directement exprimées en termes de séquences, ce qui élimine du même coup le hiatus qu'il y avait entre la signification intuitive d'une formule et l'algorithme permettant de l'évaluer

9.5 Implémentation

La puissance d'expression d'une logique est loin d'être le seul critère de choix ; il faut aussi tenir compte de la complexité de l'algorithme qui permet d'évaluer les formules sur le graphe.

Dans le cas de l'outil XESAR, le temps mis par le système pour évaluer une formule est de l'ordre de la seconde, alors que la réflexion nécessaire à l'écriture d'une formule peut prendre à l'utilisateur plusieurs heures. Ce dernier est souvent dans l'incertitude quand au sens exact des formules qu'il valide et la recherche d'une spécification correcte procède généralement par approximations successives.

Il semble donc que l'on doive procéder à un rééquibrage et donner à l'utilisateur un formalisme de plus haut niveau, quitte à dégrader les performances d'un ou deux ordres de grandeur.

On ne propose pas ici d'algorithme général pour l'évaluation des formules de la logique R$\overset{\star}{\text{I}}$CO sur un graphe donné. Un tel algorithme n'existe pas, à l'heure actuelle ; on peut toutefois en indiquer les grandes lignes.

Comme pour la phase de simulation (§ 8.2.1, p. 155) l'évaluation d'un ensemble de formules passe par la production d'un programme en langage C, ce programme étant ensuite compilé puis exécuté. En effet, il faut disposer des types concrets (§ 8.1, p. 152) pour implémenter les sortes et les opérations abstraites LOTOS utilisées dans les formules logiques.

Deux choix d'implémentation peuvent être envisagés, selon que ce programme est indépendant ou non du modèle graphe à vérifier. On considère ici le cas où il n'en dépend pas : le programme d'évaluation est construit à partir des formules uniquement et peut être appliqué indifféremment à tout graphe (pourvu que les labels figurant sur les arcs de ce graphe soient compatibles avec ceux utilisés dans les formules).

L'implémentation des aspects logiques s'effectue comme suit :

- le résultat de l'évaluation d'une formule logique est un ensemble d'arcs. Comme cela a été fait dans QUASAR et dans XESAR pour les ensembles d'états, cet ensemble peut être implémenté par une chaîne de bits : à chaque arc du graphe correspond un bit dans la chaîne
- les opérateurs ensemblistes "**false**", "**true**", "**not**", "**and**", "**or**", "**xor**", "**=>**" et "**=**" sont implémentés de manière classique, par des opérations booléennes sur les chaînes de bits
- les opérateurs "**empty**" et "**full**" se traitent de manière analogue. Appliqué à une chaîne de bits $b_0, \ldots\ b_n$, "**empty**" renvoie une chaîne de bits $b'_0, \ldots\ b'_n$ définie par :

  si $(b_0, \ldots\ b_n) = (0, \ldots\ 0)$ **alors**
  $\quad (b'_0, \ldots\ b'_n) := (1, \ldots\ 1)$
  sinon
  $\quad (b'_0, \ldots\ b'_n) := (0, \ldots\ 0)$

  De même le résultat de "**full**" est défini par :

  si $(b_0, \ldots\ b_n) = (1, \ldots\ 1)$ **alors**
  $\quad (b'_0, \ldots\ b'_n) := (1, \ldots\ 1)$
  sinon
  $\quad (b'_0, \ldots\ b'_n) := (0, \ldots\ 0)$

- les prédicats de base "**start**" et "**stop**" ne posent aucun problème. L'évaluation de "$G\ \ldots$" nécessite une recherche associative des arcs en fonction de leur label ; pour optimiser l'accès, on peut initialement trier l'ensemble des arcs ou construire une fonction de hachage
- les quantificateurs "**all**" et "**some**" s'implémentent par des itérations sur le domaine des sortes. Si φ est une formule, paramétrée par une variable X de sorte S, ayant pour résultat une chaîne de bits $b_0(X), \ldots\ b_n(X)$, alors "**all** $X : S$ **in** φ" renvoie une chaîne de bits $b'_0, \ldots\ b'_n$ calculée par itérations successives :

  $(b'_0, \ldots\ b'_n) := (1, \ldots\ 1)$
  pour_tous $V \in domain(S)$ **faire**
  $\quad$ **début**
  $\quad (b'_0, \ldots\ b'_n) := (b'_0\ \wedge\ b_0(V), \ldots\ b'_n\ \wedge\ b_n(V))$
  $\quad$ **fin**

  Cet algorithme peut être optimisé si φ est de la forme "**empty** φ_0" ou bien "**full** φ_0" puisque tous les bits $b_0(X), \ldots\ b_n(X)$ sont égaux. Il suffit alors de travailler sur un seul bit b' et l'on peut interrompre la boucle dès que b' devient égal à 0 :

  $b' := 1$
  pour_tous $V \in domain(S)$ **tant_que** $b' \neq 0$ **faire**
  $\quad$ **début**
  $\quad b' := b'\ \wedge\ b_0(V)$
  $\quad$ **fin**
  $(b'_0, \ldots\ b'_n) := (b', \ldots\ b')$

  Bien entendu, l'opérateur "**some**" doit être traité de manière duale
- les opérateurs "**if**", "**let**" et "**eval**" s'implémentent naturellement. Les résultats des calculs partiels sont conservés dans des variables, pour "**let**", et dans des chaînes de bits, pour "**eval**". Il est possible de gérer en pile l'allocation et la libération de ces zones de mémoire, car la visibilité des identificateurs respecte une structure de blocs imbriqués

Le traitement des aspects temporels se ramène au problème de l'évaluation des opérateurs "**first**" et "**last**". En effet, d'après la syntaxe de la logique R$\overset{\star}{\text{I}}$CO, les expressions temporelles ne peuvent intervenir que comme opérandes de "**first**" et "**last**". Un algorithme d'évaluation de "**first** (ψ)" et "**last** (ψ)" serait basé sur les principes suivants :

- si l'expression ψ comporte n formules distinctes $\varphi_1, \ldots\ \varphi_n$, elles déterminent une partition des arcs du graphe en catégories d'équivalences : deux arcs sont dans la même classe si et seulement si ils satisfont exactement le même sous-ensemble de formules parmi $\varphi_1, \ldots\ \varphi_n$. Intuitivement, cela signifie que les formules contenues dans ψ ne permettent pas de distinguer ces deux arcs

- le nombre de classes est au plus égal à 2^n. En pratique n ne devrait guère dépasser la dizaine, ce qui correspond au plus à un millier de classes, alors que les graphes produits par CÆSAR peuvent avoir plus d'un million d'arcs

- on construit ensuite un automate fini déterministe, appelé *observateur*, ayant comme vocabulaire l'ensemble des classes d'équivalences, reconnaissant l'ensemble des séquences satisfaisant par l'expression temporelle ψ. Cette technique est bien connue dans le cas où ψ est une expression régulière (c'est-à-dire ne comportant ni "`.`", ni "`&`", ni "`@`") : diverses solutions existent, la plus efficace étant l'algorithme de la *résiduelle* [BS87]. Il reste à généraliser la méthode au cas des expressions temporelles.L'opérateur "`&`" s'implémente en faisant un produit d'automates. La détection des circuits pour l'opérateur "`@`" nécessite probablement un automate à pile

Lorsque le programme d'évaluation a été ainsi construit, de manière indépendante du graphe, son exécution sur un graphe donné comporte les opérations suivantes :

- on commence par évaluer sur le graphe les formules $\varphi_1, \ldots\ \varphi_n$ apparaissant dans l'expression temporelle ψ ; ceci est possible, car la signification d'une formule ne dépend pas de la manière dont elle est utilisée dans une expression temporelle

- on partitionne l'ensemble des arcs du graphe en fonction de leur appartenance aux classes d'équivalence

- pour qu'une séquence $\ll E_1, \ldots\ E_n \gg$ du graphe satisfasse la formule ψ, il faut que le mot $\mathcal{C}_1, \ldots\ \mathcal{C}_n$ soit accepté par l'observateur, $\mathcal{C}_1, \ldots\ \mathcal{C}_n$ désignant les classes respectives des arcs $E_1, \ldots\ E_n$. Cette condition est nécessaire, mais probablement pas suffisante car l'opérateur "`@`" nécessite en outre la détection des circuits

- il faut noter que la sémantique donnée aux opérateurs "**first** (ψ)" et "**last** (ψ)" est "raisonnable", au sens où il suffit de prouver qu'il existe une séquence satisfaisant ψ ou qu'il n'en existe pas. Il n'est jamais nécessaire d'explorer toutes les séquences

Un tel algorithme général d'évaluation des formules pourrait être complété par des optimisations spécifiques permettant de traiter avec un maximum d'efficacité certaines classes de formules et de chemins fréquemment rencontrés :

- certaines transformations simples, basées sur des propriétés algébriques de la logique R$\overset{\star}{\text{I}}$CO, permettent une meilleure implémentation. C'est ainsi que l'on peut remplacer "**never** (φ_0)" par "**empty** φ_0", "**never** (φ_1`+` `.` φ_2`+`)" par "**never** (φ_1 `.` φ_2)", "**first** (φ_1 `|` φ_2)" par "φ_1 **or** φ_2", ...

- la théorie des graphes fournit, pour certaines expressions temporelles, une implémentation optimale. Par exemple la formule "**first** (φ`+` `.` φ`@`)" (qui caractérise les arcs à partir desquels il est possible d'atteindre un circuit tout en satisfaisant continuellement la formule φ) peut être évaluée par un simple parcours en profondeur

- c'est aussi le cas des formules "**pot** (φ_1, φ_2)" et "**inev** (φ_1, φ_2)" de la logique CTL qui peuvent être implémentées avec un coût linéaire [Rod88]

- pour calculer "**never** (ψ)", on peut choisir entre les deux définitions de "**never**" (celle avec "**first**" ou celle avec "**last**") selon la forme de l'expression temporelle ψ. Par exemple, la seconde forme est certainement préférable si ψ a la forme "φ_1@ . φ_2"

- certaines formules peuvent être traitées plus efficacement lorsqu'on les évalue sur le graphe inverse, c'est-à-dire le graphe obtenu en remplaçant chaque arc (σ_1, L, σ_2) par un arc (σ_2, L, σ_1). Par exemple, la formule "**first (true*** . φ)" devient, sur le graphe inverse, "**last** (φ . **true***)" et peut être évaluée avec un coût linéaire, puisqu'elle caractérise tous les arcs accessibles à partir d'un arc satisfaisant φ

L'utilisateur disposerait ainsi d'un langage simple et puissant qui lui éviterait de concevoir les spécifications en fonction des algorithmes d'évaluation et des diverses stratégies mises en œuvre par le système de vérification.

Conclusion

Bilan

L'objectif de cette étude a été la définition et la mise en œuvre de techniques de compilation pour la partie contrôle du langage LOTOS, en vue de la vérification formelle des spécifications. Dans ce but nous avons proposé des modèles intermédiaires et des algorithmes permettant de traiter un large sous-ensemble du langage LOTOS.

Comparé aux autres langages FDT (ESTELLE et SDL), LOTOS avait la réputation d'être un langage ésotérique et rebelle à toute tentative d'implémentation. Les résultats de cette étude montrent qu'il est possible de compiler LOTOS avec une efficacité comparable aux autres langages FDT.

Les algorithmes de traduction ont été définis de manière complètement formelle et non ambiguë, notamment à l'aide de grammaires attribuées. A partir de ces spécifications, ils ont été implémentés de manière systématique et rigoureuse. Ces algorithmes devraient, en toute rigueur, être accompagnés d'une preuve formelle de leur correction. Mais la plupart du temps, ils nous ont semblé suffisamment simples et sûrs pour qu'il soit permis de s'en dispenser. Lorsque cela s'est révélé nécessaire, les points délicats ont été soigneusement détaillés et étayés par des arguments de preuve.

Notre démarche permet de compiler LOTOS, langage récent et novateur, vers deux modèles classiques du parallélisme : les graphes et les réseaux.

Les graphes constituent certainement le modèle le mieux adapté aux techniques de vérification formelle. Notre méthode de traduction donne la possibilité de comparer entre eux deux programmes LOTOS, en utilisant les relations d'équivalence définies sur les graphes : équivalence forte, équivalence observationnelle, ... Mais il est également utile de valider sur les graphes des propriétés spécifiées dans un langage déclaratif de haut niveau et différent de LOTOS. Dans ce but, nous avons conçu une logique temporelle basée sur les expressions régulières, la logique R$\star$ICO, qui allie puissance d'expression et simplicité d'utilisation.

Contrairement aux graphes, les réseaux autorisent une représentation compacte du comportement des spécifications LOTOS ; sauf exception, le problème de l'explosion combinatoire ne concerne pas les réseaux. Ils possèdent en outre une sémantique opérationnelle qui peut donner lieu à une implémentation efficace ; les réseaux constituent donc un modèle d'exécution adéquat pour LOTOS.

Les techniques de compilation décrites dans cette étude ont permis la réalisation du système CÆSAR [Gar89a] destiné à la vérification formelle de spécifications LOTOS.

Le programme CÆSAR a été écrit en langage C et fonctionne actuellement sous UNIX 4.3BSD. La version 1.0 de CÆSAR a été développée dans le cadre du projet européen ESPRIT/SEDOS. L'analyse lexicographique et syntaxique a été réalisée à l'aide de SYNTAX[42] qui constitue un excellent système de production de compilateurs. La partie avant de CÆSAR a été développée par Ahmed Serhrouchni

[42]SYNTAX est une marque déposée de l'I.N.R.I.A.

puis reprise et complétée par Pascal Bouchon et Jean-Michel Houdouin [BH88]. Elle est également utilisée par le système CÆSAR.ADT, conçu par Christian Bard [Bar88] [Gar89b], qui traduit automatiquement les types abstraits de LOTOS en langage C.

L'objectif initial était de récupérer au maximum le travail qui avait été fait pour le système XESAR, mais les différences sémantiques entre LOTOS et ESTELLE/R n'ont permis de réutiliser ni les formes intermédiaires ni les algorithmes de traduction développés pour XESAR (à l'exception toutefois de la phase de simulation qui présente certaines analogies). Bien qu'inspirés tous deux du "principe CESAR" CÆSAR et XESAR sont donc deux systèmes profondément différents.

CÆSAR se spécialise dans la génération du graphe, laissant le soin de valider ce graphe à d'autres outils qu'il est capable d'interfacer, suivant la "philosophie de la boîte à outils". Selon le type d'outil choisi la vérification de propriétés sur le graphe peut être effectuée, soit au moyen d'équivalences d'automates, soit par des logiques temporelles :

outil	*origine*	*type de vérification*
ALDEBARAN	LGI-IMAG (Grenoble)	équivalences
AUTO	INRIA (Sophia)	équivalences
MEC	Université de Bordeaux I	μ-calcul
PIPN	LAAS (Toulouse)	équivalences
SCAN	BULL	équivalences
SQUIGGLES	CNUCE (Pise)	équivalences
XESAR	LGI-IMAG (Grenoble)	logique temporelle

Le logiciel CÆSAR a subi avec succès plusieurs centaines de tests pour lesquels les graphes et les réseaux obtenus ont été vérifiés (cette suite de validation pour LOTOS constitue un sous-produit du développement de CÆSAR).

En ce qui concerne les performances, les objectifs initialement fixés ont été atteints puisque, sur une station de travail de type SUN 3/60, CÆSAR peut engendrer des graphes ayant plusieurs centaines de milliers d'états et d'arcs, avec un débit variant entre 100 et 200 états par seconde. Cette rapidité prouve la supériorité de l'approche statique sur l'approche dynamique pour la compilation de LOTOS. A titre d'indication le tableau suivant compare les temps mis par SQUIGGLES [BC88] et par la version 2.8 de CÆSAR pour engendrer de petits graphes :

exemple	*nombre d'états*	SQUIGGLES	CÆSAR
`max2`	5	6s	2s
`max3`	11	7s	2s
`max5`	47	47s	3s
`max7`	191	31mn	5s
`max9`	767	4h 41mn	14s

Notre expérience montre que CÆSAR et CÆSAR.ADT constituent une aide appréciable au développement de programmes LOTOS. En particulier, pour l'étude des exemples fournis en annexe, CÆSAR a permis de détecter un grand nombre de fautes de gravité diverse, allant des simples erreurs syntaxiques jusqu'aux incohérences de conception.

Perspectives

Pour conclure cette étude, il est temps d'indiquer les principales directions qui en constituent le prolongement.

La méthode de compilation proposée ici pour LOTOS trouve des applications dans des domaines autres que la vérification. Le système CÆSAR constitue une plate-forme logicielle à partir de laquelle d'autres types d'outils pourraient être développés, afin d'offrir à l'utilisateur un environnement complet. En modifiant seulement la manière dont la phase de simulation conserve les états rencontrés, on pourrait obtenir, au choix :

- **un générateur de code séquentiel :** pour produire automatiquement un prototype exécutable correspondant à une spécification LOTOS, il suffit de ne mémoriser que l'état courant et, lorsqu'un état comporte plusieurs successeurs, d'en choisir aléatoirement un

- **un interpréteur :** il faut conserver l'état courant et, si l'on autorise le retour en arrière, les états figurant sur le chemin allant de l'état initial à l'état courant. Le non-déterminisme peut être résolu, soit aléatoirement, soit de manière interactive en mode pas-à-pas. Par rapport aux interpréteurs classiques, la détection des boucles est automatique. En revanche la correspondance avec le texte source LOTOS est moins aisée

- **un générateur de séquences de test :** son fonctionnement est identique à celui de l'interpréteur, mais les boucles sont autorisées, ce qui revient à permettre qu'un même état soit exploré plusieurs fois, dans la limite d'un nombre maximum fixé par l'utilisateur

Le modèle réseau pourrait convenir à d'autres applications, parmi lesquelles l'évaluation de performances (méthodes probabilistes, chaînes de Markov, ...), la génération de code parallèle, la visualisation graphique de l'architecture des programmes LOTOS en tirant parti de la structuration du réseau en unités ...

En ce qui concerne la vérification, la réalisation d'un évaluateur pour les formules de la logique $\overset{\star}{\text{R}}$ICO permettrait de compléter les fonctionnalités du système CÆSAR. Cet évaluateur devrait intégrer une aide au diagnostic afin d'expliquer à l'utilisateur, à l'aide de contre-exemples, pourquoi une formule n'est pas valide.

A l'heure actuelle, la vérification par simulation exhaustive présente des limitations importantes. Lorsqu'elle est utilisée pour valider des applications réelles [GRRV89], on ne peut guère aller au-delà de quelques millions d'états alors qu'un "vrai" protocole en comporte plusieurs milliards.

Face à ce problème, l'utilisateur doit rechercher si le système étudié peut être décomposé en sous-systèmes assez petits pour être validés séparément par vérification exhaustive. Il faut ensuite tenter de remplacer chaque sous-système, après vérification, par une description équivalente mais plus simple ; la validation s'effectue ainsi de manière ascendante. Une telle décomposition est souvent rendue possible par la structuration modulaire des spécifications LOTOS.

Une autre approche, qui s'apparente davantage au test qu'à la vérification, consiste à n'engendrer qu'un graphe partiel, au lieu du graphe complet que l'on obtiendrait par simulation exhaustive. Lorsqu'un état possède $n > 1$ successeurs non encore visités, on n'en explore que n', avec $1 \leq n' \leq n$. La valeur de n' et le choix des successeurs peuvent être déterminés par diverses stratégies, les plus intéressantes étant celles qui font intervenir les propriétés à vérifier. Le graphe partiel ainsi obtenu possède des propriétés intéressantes ; en effet si une formule qui exprime la non-possibilité ou bien l'inévitabilité d'un événement est fausse sur le graphe partiel, alors elle est fausse sur le graphe complet. On peut ainsi détecter la présence de certaines erreurs, mais non pas prouver l'absence d'erreur.

Bien que ces techniques puissent rendre de nombreux services, l'objectif final demeure la vérification automatique d'un système complet, sans que l'utilisateur ait trop à intervenir. Pour cela il faut impérativement éviter l'explosion combinatoire : au lieu d'engendrer un graphe énorme pour le réduire ensuite selon des relations d'équivalence, il serait préférable de produire directement un graphe réduit, complètement ou partiellement.

Pour CÆSAR, cette réduction devrait avoir lieu au niveau du réseau, avant la phase de simulation. En effet l'expérience montre qu'une diminution, même minime, de la taille du réseau se traduit généralement par une réduction considérable de la taille du graphe obtenu après simulation et du temps consacré à l'engendrer.

C'est pourquoi d'autres optimisations du réseau devraient être mises en œuvre, en utilisant des techniques sophistiquées d'analyse de flux. En ramenant l'étude d'un programme LOTOS à celle de son réseau, on bénéficie des acquis de la théorie des réseaux de Petri et, en particulier, des techniques d'analyse et de transformation qui ont été développées pour les réseaux de Petri. En outre, le caractère purement fonctionnel du langage LOTOS fait que le flux des données dans les réseaux est suffisamment simple pour permettre des analyses élaborées mais efficaces.

Plus encore, on pourrait renoncer à produire le graphe pour la relation d'équivalence forte, et le faire pour des relations plus faibles. C'est ainsi que, pour l'équivalence observationnelle, on chercherait à compacter les chaînes de transitions étiquetées "**i**" de la même manière que l'optimisation élimine actuellement les ε-transitions. On pourrait aussi appliquer des critères d'abstraction au réseau : si l'utilisateur souhaite, par exemple, n'observer que les messages qui transitent sur la porte G, on pourrait renommer en "**i**" ou en "ε" toutes les autres portes du réseau.

Une expérience a été tentée en ce sens, basée sur la conjecture suivante : le fait de renommer en "ε" toutes les portes "**i**" du réseau préserve l'*équivalence de sûreté* [Rod88] qui est une relation compatible avec les propriétés de sûreté que l'on peut exprimer en logique temporelle de sûreté. Jusqu'à présent, cette conjecture n'a pas pu être prise en défaut. Son application produit des résultats spectaculaires. C'est ainsi qu'on a pu traiter l'exemple du contrôleur par jeton circulant (*scheduler* [Mil80, § 3.1]) avec 100 sites (*cyclers*) : sans réduction, l'explosion combinatoire rend la génération du graphe pratiquement impossible mais, après transformation du réseau, il n'a fallu que quelques minutes à CÆSAR pour produire le graphe réduit !

Les idées et les techniques développés pour CÆSAR permettent donc d'espérer des progrès en ce qui concerne la vérification formelle, mais leur portée n'est pas restreinte à ce seul domaine. Elles constituent une approche *unifiée* : sur cette base une large gamme d'outils utiles et performants peut être développée afin de construire un environnement logiciel complet pour le langage LOTOS.

Annexe A

Présentation du langage LOTOS : les structures de données

Comme la plupart des langages de programmation, LOTOS permet de définir et de manipuler des *structures de données*. Mais, à la différence des langages classiques, LOTOS utilise le formalisme des *types abstraits algébriques* (*abstract data types, ADT*), en s'inspirant largement du langage ACTONE [EFH83] [EM85]. Dans une large mesure, le choix des types abstraits pour la description des structures de données est conforme aux objectifs d'un langage de spécification :

- les spécifications algébriques constituent un modèle mathématique exprimant les propriétés que doit vérifier toute réalisation, sans imposer de contraintes d'implémentation superflues
- les propriétés des données et des opérations sont complètement décrites. En choisissant les types abstraits, LOTOS évite les difficultés rencontrées en ESTELLE où les données sont spécifiées au moyen des types du langage PASCAL [JW78], ce qui pose des problèmes bien connus (combien vaut $-7 \mathrm{mod} 3$?)

Cette annexe présente succinctement les principales caractéristiques de la description des données en LOTOS. Son propos n'est pas de se substituer à la définition formelle [ISO87] mais d'aider à sa compréhension. Pour une étude approfondie des types abstraits, [EM85] constitue une référence.

A.1 Présentation des types abstraits

Les définitions suivantes sont nécessaires à la compréhension des types abstraits, tels qu'ils figurent en LOTOS :

sorte : nom donné à un domaine de valeurs. Par exemple `BOOL` dénote la sorte des valeurs booléennes et `NAT` celle des entiers naturels. L'utilisateur n'a pas à spécifier explicitement le domaine des sortes qu'il déclare

domaine (*data-carrier*) : ensemble des valeurs d'une sorte

opérateur, opération : nom donné à une fonction qui à n arguments ($n \geq 0$) fait correspondre un résultat. Par exemple `false`, `true`, `not`, `and`, `or`, "`+`", "`*`", ... sont des opérations

arité : nombre d'arguments d'un opérateur. Par exemple, l'arité de `true` est 0, celle de `not` est 1, celle de "`+`" est 2

constante : opérateur d'arité nulle

profil : sortes des arguments et du résultat d'un opérateur. Par exemple le profil de `not` est `BOOL` $\rightarrow$ `BOOL` et celui de "`+`" est `NAT` $\times$ `NAT` $\rightarrow$ `NAT`

surcharge (*overloading*) **:** possibilité de définir plusieurs opérateurs possédant le même nom et des profils différents. Par exemple, l'opérateur d'égalité `eq` est surchargé puisqu'il peut avoir plusieurs profils : `NAT` $\times$ `NAT` $\rightarrow$ `BOOL` et `BOOL` $\times$ `BOOL` $\rightarrow$ `BOOL`

équation : égalité servant à définir la sémantique d'un opérateur. En LOTOS la sémantique des sortes et des opérateurs est exclusivement décrite par des équations. Par exemple, l'équation `P or Q = not (not (P) and not (Q))` exprime une relation qui peut constituer la définition de `or` à partir de `not` et `and`

type : nom donné à un ensemble de sortes, d'opérations et d'équations. On réunit dans un type des éléments qui décrivent un même concept : le type `BOOLEAN`, le type `NATURALNUMBER`

signature : ensemble des sortes et des opérations d'un type. Par exemple la signature du type `BOOLEAN` comprend, entre autres, `BOOL`, `false`, `true`, `not`, `and`, `or`. Celle du type `NATURALNUMBER` comprend `NAT`, "`+`", "`*`", ...

présentation : ensemble des sortes, des opérations et des équations d'un type

variable : nom donné à une valeur. LOTOS est un langage fonctionnel fortement typé : chaque variable ne peut prendre qu'une seule valeur d'une sorte déterminée par sa déclaration

expression de valeur, terme : expression construite à partir de variables et d'opérations. Par exemple, si `P` et `Q` sont des variables de sorte `BOOL` alors `P`, `not (P)`, `P and Q`, `not (P and Q)`, ... sont des termes

Les arguments des opérateurs unaires doivent être parenthésés (ainsi `not P` est syntaxiquement incorrect). Les opérateurs binaires peuvent être préfixés ou infixés. La surcharge d'opérateurs est autorisée, sous certaines conditions qui visent à interdire les ambiguïtés

congruence : deux termes sont congrus si l'on peut, en utilisant les propriétés et les transformations indiquées par les équations, démontrer qu'ils sont syntaxiquement identiques. Par exemple `true or false` et `false or true` sont congrus

algèbre quotient : ensemble quotient de l'ensemble des termes par la relation de congruence. Les valeurs sont des éléments de l'algèbre quotient, c'est-à-dire des classes d'équivalence. Par exemple la valeur *true* représente l'ensemble de termes : {`true, not (false), true or false`, ...}

A.2 Eléments lexicographiques et syntaxiques

En LOTOS les lettres majuscules et minuscules sont identiques. Les identificateurs sont composés de lettres, de chiffres et du caractère "_" suivant les règles usuelles. Pour les noms d'opérateurs LOTOS permet l'utilisation de caractères spéciaux : l'utilisateur peut ainsi créer des opérateurs baptisés "`+`", "`<=>`", "`//`", ... Il est toutefois interdit de redéfinir les mots-réservés du langage : "`=`", "`->`", "`=>`", ...

Identificateurs

Chaque classe d'identificateurs est dénotée par un symbole non-terminal défini comme suit :

- S : sortes
- F : opérateurs
- T : types
- X : variables

Déclarations de variables

La construction suivante déclare un ensemble de variables $X_0, \ldots\ X_n$ ayant la même sorte S :

$$X_0, \ldots\ X_n \mathtt{:} S$$

Pour alléger les notations, on emploie l'abréviation $\widehat{X}$ qui dénote une liste non vide de variables X_0, $\ldots\ X_n$.

Expressions de valeur

Les expressions de valeurs, dénotées par le non-terminal V, sont définies par les règles suivantes :

$$\begin{array}{rcl} V & \equiv & X \\ & | & F\ [\mathtt{(}V_0, \ldots\ V_n\mathtt{)}] \\ & | & V_1\ F\ V_2 \\ & | & V_0\ \textbf{of}\ S \end{array}$$

Les trois premières règles expriment qu'une expression de valeur peut être soit une variable, soit une opération préfixée (d'arité quelconque) ou infixée (d'arité 2). La quatrième règle introduit la notation "**of** S" qui permet de résoudre les ambiguïtés liées aux surcharges d'opérateurs en précisant que l'expression de valeur V_0 a pour sorte S.

Opérations

La construction suivante déclare un ensemble d'opérateurs $F_0, \ldots\ F_m$ ayant le même profil $S_1, \ldots\ S_n \rightarrow S$:

$$opns \quad \equiv \quad F_0, \ldots\ F_m\ \mathtt{:}\ S_1, \ldots\ S_n\ \mathtt{->}\ S$$

Pour spécifier qu'un opérateur binaire F est infixé il suffit, dans la déclaration *opns*, d'entourer F de deux caractères "`_`" : "`_+_`", "`_<=>_`", "`_//_`", …

Equations

Le non-terminal *seq* dénote une équation "simple" dont les membres sont deux expressions de valeurs V_1 et V_2 :

$$seq \quad \equiv \quad V_1\mathtt{=}V_2$$

Le non-terminal *peq* dénote une “prémisse” formée soit d'une équation simple soit d'une expression de valeur :

$$\begin{array}{rcl} peq & \equiv & seq \\ & \mid & V \end{array}$$

Le non-terminal *meq* dénote une équation “moyenne” formée d'une équation simple *seq*, éventuellement précédée d'un ensemble de prémisses (on parle alors d'*équation conditionnelle*) :

$$meq \quad \equiv \quad [peq_0, \ldots\ peq_n \texttt{ =>}]\ seq$$

Le non-terminal *ceq* dénote une équation “complexe” formée d'un ensemble d'équations moyennes $meq_0, \ldots\ meq_n$ dont les membres, gauches et droits, ont tous la même sorte S. Les membres des équations sont des termes du 1[er] ordre pouvant contenir des variables $\widehat{X_0}, \ldots\ \widehat{X_m}$ quantifiées universellement :

$$ceq \quad \equiv \quad \textbf{ofsort}\ S\ [\textbf{forall}\ \widehat{X_0}\!:\!S_0, \ldots\ \widehat{X_m}\!:\!S_m]\ meq_0, \ldots\ meq_n$$

Le non-terminal *eqns* dénote un ensemble d'équations complexes $ceq_0, \ldots\ ceq_n$ dans lesquelles peuvent figurer des variables $\widehat{X_0}, \ldots\ \widehat{X_m}$ quantifiées universellement :

$$eqns \quad \equiv \quad [\textbf{forall}\ \widehat{X_0}\!:\!S_0, \ldots\ \widehat{X_m}\!:\!S_m]\ ceq_0, \ldots\ ceq_n$$

A.3 Types importés

Un programme LOTOS peut importer des types prédéfinis. La construction suivante déclare les types $T_0, \ldots\ T_n$ dont la définition doit être recherchée dans la bibliothèque LOTOS :

$$\begin{array}{l} \textbf{library}\ T_0, \ldots\ T_n \\ \textbf{endlib} \end{array}$$

La définition de LOTOS n'impose aucune contrainte sur la façon dont cette bibliothèque doit être réalisée. En revanche elle fournit une liste de types prédéfinis qui constituent la *bibliothèque standard* du langage [ISO87, annexe A].

A.4 Types élémentaires

La construction suivante déclare un type élémentaire T en lui associant sa présentation, composée de sortes, d'opérations et d'équations :

$$\textbf{type}\ T\ \textbf{is}$$

> [**sorts** $S_0, \ldots\ S_p$]
> [**opns** $opns_0, \ldots\ opns_q$]
> [**eqns** $eqns$]
> **endtype**

Exemple A-1
L'exemple suivant est extrait de la bibliothèque standard du langage LOTOS [ISO87, annexe A.4]. Il définit le type abstrait `BOOLEAN` qui exporte une sorte `BOOL`, deux constantes `false` et `true`, un opérateur unaire `not` et sept opérateurs binaires infixés `and`, `or`, ... En appliquant les équations, n'importe quel terme peut être évalué soit à `false` soit à `true` :

```
type BOOLEAN is
   sorts BOOL
   opns true, false : -> BOOL
        not : BOOL -> BOOL
        _and_, _or_, _xor_, _implies_, _iff_ : BOOL, BOOL -> BOOL
        _eq_, _ne_ : BOOL, BOOL -> BOOL
   eqns forall X, Y :BOOL
        ofsort BOOL
           not (true) = false;
           not (false) = true;

           X and true = X;
           X and false = false;

           X or true = true;
           X or false = X;

           X xor Y = (X and not (Y)) or (Y and not (X));
           X implies Y = Y or not (X);
           X iff Y = (X implies Y) and (Y implies X);
           X eq Y = X iff Y;
           X ne Y = X xor Y
endtype
```

■

A.5 Types combinés

La combinaison (*combination*) permet de réutiliser des types déjà existants pour définir de nouveaux types. La construction suivante déclare un type T obtenu par combinaison d'un ensemble de types $T_0, \ldots\ T_n$ définis auparavant. Le type T est défini par sa présentation, de la même manière qu'un type élémentaire :

> **type** T **is** $T_0, \ldots\ T_n$
> [**sorts** $S_0, \ldots\ S_p$]
> [**opns** $opns_0, \ldots\ opns_q$]
> [**eqns** $eqns$]
> **endtype**

Les sortes et les opérations qui font partie de la signature de $T_0, \ldots\ T_n$ peuvent être utilisées dans la présentation de T. On dit que le type T *hérite*[43] des types $T_0, \ldots\ T_n$.

Exemple A-2
L'exemple suivant est une variante de la définition abstraite des nombres naturels donnée dans la bibliothèque standard [ISO87, annexe A.6.1.1]. Il définit le type `NATURAL` qui exporte la sorte `NAT`, la constante `0` et l'opérateur unaire `SUCC` ; chaque nombre possède une représentation canonique en base 1 définie comme suit :

$$\begin{array}{rcl} 0 & = & \texttt{0} \\ 1 & = & \texttt{SUCC (0)} \\ 2 & = & \texttt{SUCC (SUCC (0))} \\ & \ldots & \end{array}$$

Deux opérations plus complexes, l'addition (opérateur binaire infixé "`+`") et la multiplication (opérateur binaire infixé "`*`"), sont définies en fonction des opérateurs `0` et `SUCC` au moyen d'équations.

Le type `NATURAL` contient aussi les relations de comparaison entre nombres naturels, c'est-à-dire six opérateurs binaires infixés : `eq` (*equal*), `ne` (*not equal*), `lt` (*less than*), `le` (*less or equal*), `gt` (*greater than*) et `ge` (*greater or equal*).

Le résultat de ces opérateurs est de sorte `BOOL`. La définition du type `NATURAL` utilise celle du type `BOOLEAN` puisqu'elle importe la sorte `BOOL`.

```
type NATURAL is BOOLEAN
   sorts NAT
   opns 0 : -> NAT
        SUCC : NAT -> NAT
        _+_, _*_ : NAT, NAT -> NAT
        _eq_, _ne_, _lt_, _le_, _gt_, _ge_ : NAT, NAT -> BOOL
   eqns forall M, N:NAT
        ofsort NAT
           M + 0 = M;
           M + SUCC (N) = SUCC (M + N);

           M * 0 = 0;
           M * SUCC (N) = M + (M * N)
        ofsort BOOL
           0 eq 0 = true;
           0 eq SUCC (M) = false;
           SUCC (M) eq 0 = false;
           SUCC (M) eq SUCC (N) = M eq N;

           0 lt 0 = false;
           0 lt SUCC (M) = true;
           SUCC (M) lt 0 = false;
           SUCC (M) lt SUCC (N) = M lt N;

           M ne N = not (M eq N);
           M le N = (M lt N) or (M eq N);
           M gt N = not (M le N);
           M ge N = not (M lt N)
endtype
```

[43] il s'agit d'un héritage multiple

■

Exemple A-3
Il est possible de spécifier des structures de données plus complexes que les entiers et les booléens. L'exemple suivant montre comment décrire les listes LISP en LOTOS. On suppose que les éléments des listes (*atomes*) sont des nombres naturels. Le type `NATURAL_LIST` exporte une sorte `LIST` et les cinq opérations primitives de LISP : `NIL`, `CONS`, `CAR`, `CDR` et `ATOM`. Toute liste peut s'exprimer en fonction de `NIL` et `CONS` de la manière suivante :

$$\begin{array}{rcl} () & = & \texttt{NIL} \\ (n_1) & = & \texttt{CONS (}n_1\texttt{, NIL)} \\ (n_1, n_2) & = & \texttt{CONS (}n_1\texttt{, CONS (}n_2\texttt{, NIL))} \\ & \ldots & \end{array}$$

Les équations donnent aux fonctions `CAR`, `CDR` et `ATOM` leur sémantique usuelle. Les termes `CAR (NIL)` et `CDR (NIL)` sont indéfinis, donc irréductibles.

Ces opérations ont des paramètres et des résultats de sorte `NAT` et `BOOL`. La définition du type `NATURAL_LIST` utilise donc celles des types `NATURAL` et `BOOLEAN`.

```
type NATURAL_LIST is NATURAL, BOOLEAN
   sorts LIST
   opns NIL : -> LIST
        CONS : NAT, LIST -> LIST
        CAR : LIST -> NAT
        CDR : LIST -> LIST
        ATOM : LIST -> BOOL
   eqns forall N:NAT, L:LIST
        ofsort LIST
           CAR (CONS (N, L)) = N;
           CDR (CONS (N, L)) = L
        ofsort BOOL
           ATOM (NIL) = true;
           ATOM (CONS (N, L)) = false
endtype
```

■

La combinaison peut aussi servir à *enrichir* un type existant en le complétant par de nouvelles sortes et de nouveaux opérateurs.

Exemple A-4
Il est possible d'enrichir le type `NATURAL_LIST` défini dans l'exemple A-3 (p. 209) en lui ajoutant les relations d'égalité et d'inégalité sur les listes (opérateurs binaires infixés `eq` et `ne`). On obtient ainsi le type `ENRICHED_NATURAL_LIST` :

```
type ENRICHED_NATURAL_LIST is NATURAL_LIST
   opns _eq_, _ne_ : LIST, LIST -> BOOL
   eqns forall N, N1, N2:NAT, L, L1, L2:LIST
        ofsort BOOL
           NIL eq NIL = true;
           NIL eq CONS (N, L) = false;
           CONS (N, L) eq NIL = false;
           N1 eq N2 =>
              CONS (N1, L1) eq CONS (N2, L2) = (L1 eq L2);
           N1 ne N2 =>
              CONS (N1, L1) eq CONS (N2, L2) = false;

           L1 ne L2 = not (L1 eq L2)
endtype
```

L'emploi d'équations conditionnelles pour la définition de l'égalité n'est pas indispensable ; on aurait pu remplacer les deux équations conditionnelles par l'équation suivante :

```
CONS (N1, L1) eq CONS (N2, L2) = (N1 eq N2) and (L1 eq L2)
```

■

A.6 Types paramétrés

LOTOS permet la définition de types paramétrés par des *sortes formelles* et des *opérateurs formels*.

Pour cela on commence par définir un *type formel* T dont la présentation contient les sortes formelles et les opérations formelles servant de paramètres. La présentation de T peut également posséder des *équations formelles* qui, en spécifiant les propriétés des opérateurs formels, imposent des contraintes supplémentaires :

type T **is** $T_1, \ldots\ T_n$
  [**formalsorts** $S_0, \ldots\ S_p$]
  [**formalopns** $opns_0, \ldots\ opns_q$]
  [**formaleqns** *eqns*]
endtype

Les sortes et les opérations utilisées dans $opns_0, \ldots\ opns_q$ et *eqns* doivent être formelles.

Exemple A-5
Il est possible de généraliser le type ENRICHED_NATURAL_LIST défini dans l'exemple A-4 (p. 209) pour manipuler des listes dont les éléments sont de sorte quelconque — et non plus seulement de sorte NAT. On suppose néanmoins que tous les atomes ont la même sorte.

Pour cela il faut modifier la définition de ENRICHED_NATURAL_LIST en remplaçant toutes les occurrences de la sorte NAT par une sorte formelle appelée, par exemple, ITEM. Ce n'est pas suffisant : il faut également remplacer les opérations :

```
_eq_, _ne_ :  NAT, NAT -> BOOL
```

dont le profil dépend de la sorte NAT par les opérations formelles correspondantes sur la sorte ITEM. Par commodité on conservera pour ces opérations formelles les noms eq et ne :

```
_eq_, _ne_ :  ITEM, ITEM -> BOOL
```

On peut spécifier certaines propriétés que doivent vérifier les opérations formelles, comme par exemple :

```
(forall N1, N2:ITEM) N1 ne N2 = not (N1 eq N2)
```

Au niveau des types, cette transformation revient à remplacer le type `NATURAL` par un type formel `FORMAL_ITEM` défini comme suit :

```
type FORMAL_ITEM is BOOLEAN
   formalsorts ITEM
   formalopns _eq_, _ne_ : ITEM, ITEM -> BOOL
   formaleqns forall N1, N2:ITEM
              ofsort BOOL
                 N1 ne N2 = not (N1 eq N2)
endtype
```

■

La définition d'un type paramétré s'effectue par combinaison avec un ou plusieurs types contenant des sortes (*resp.* opérations) formelles.

Exemple A-6
On peut alors définir le type abstrait `GENERIC_LIST` décrivant les listes LISP paramétrées :

```
type GENERIC_LIST is FORMAL_ITEM, BOOLEAN
   sorts LIST
   opns NIL : -> LIST
        CONS : ITEM, LIST -> LIST
        CAR : LIST -> ITEM
        CDR : LIST -> LIST
        ATOM : LIST -> BOOL
        _eq_, _ne_ : LIST, LIST -> BOOL
   eqns forall N, N1, N2:ITEM, L, L1, L2:LIST
        ofsort LIST
           CAR (CONS (N, L)) = N;
           CDR (CONS (N, L)) = L
        ofsort BOOL
           ATOM (NIL) = true;
           ATOM (CONS (N, L)) = false;

           NIL eq NIL = true;
           NIL eq CONS (N, L) = false;
           CONS (N, L) eq NIL = false;
           N1 eq N2 =>
              CONS (N1, L1) eq CONS (N2, L2) = (L1 eq L2);
           N1 ne N2 =>
              CONS (N1, L1) eq CONS (N2, L2) = false;

           L1 ne L2 = not (L1 eq L2)
endtype
```

■

Remarque A-1
En fait, il n'y a pas de distinction stricte entre types élémentaires et types formels puisqu'un même type peut simultanément contenir des sortes (*resp.* opérations, équations) formelles et non formelles. La construction générale permettant la déclaration d'un type est :

type T **is** $T_1, \dots\ T_n$
  [**formalsorts** $S_0, \dots\ S_p$]

[**formalopns** $opns_0, \ldots\ opns_q$]
[**formaleqns** $eqns$]
[**sorts** $S'_0, \ldots\ S'_{p'}$]
[**opns** $opns'_0, \ldots\ opns'_{q'}$]
[**eqns** $eqns'$]
endtype

Il n'est donc pas indispensable de définir séparément les types `FORMAL_ITEM` et `GENERIC_LIST` : les clauses "**formalsorts**", "**formalopns**" et "**formaleqns**" figurant dans `FORMAL_ITEM` peuvent être directement insérées dans `GENERIC_LIST`. ■

A.7 Types instanciés

La construction suivante déclare un type T obtenu par instanciation (*actualization*) d'un type T' par les sortes formelles $S'_0, \ldots\ S'_p$ et les opérations formelles $F'_0, \ldots\ F'_q$. Pour instancier T' on substitue aux paramètres formels $S'_0, \ldots\ S'_p$, $F'_0, \ldots\ F'_q$ des paramètres effectifs, respectivement $S_0, \ldots\ S_p$, $F_0, \ldots\ F_q$. On indique l'ensemble de types $T_0, \ldots\ T_n$ dont les signatures contiennent les sortes et les opérations effectives :

type T **is** T' **actualizedby** $T_0, \ldots\ T_n$ **using**
[**sortnames** S_0 **for** $S'_0, \ldots\ S_p$ **for** S'_p]
[**opnnames** F_0 **for** $F'_0, \ldots\ F_q$ **for** F'_q]
endtype

L'instanciation peut être partielle, c'est à dire que certaines sortes ou opérations formelles de T' peuvent ne pas être instanciées. Dans ce cas T est, lui aussi, un type paramétré.

Exemple A-7
Pour définir des listes dont les atomes sont booléens, il suffit d'instancier le type `GENERIC_LIST` défini dans l'exemple A-5 (p. 210) en établissant la correspondance suivante :

$$
\begin{array}{rcl}
\texttt{FORMAL\_ITEM} & \longrightarrow & \texttt{BOOLEAN} \\
\texttt{ITEM} & \longrightarrow & \texttt{BOOL} \\
\texttt{eq} & \longrightarrow & \texttt{iff} \\
\texttt{ne} & \longrightarrow & \texttt{xor}
\end{array}
$$

En Lotos, cette instanciation s'écrit ainsi :

```
type BOOLEAN_LIST is GENERIC_LIST actualizedby BOOLEAN using
   sortnames BOOL for ITEM
   opnnames iff for eq
            xor for ne
endtype
```

■

A.8 Types renommés

La construction suivante déclare un type T obtenu par renommage (*renaming*) d'un type T'. La présentation de T est identique à celle de T' dans laquelle les sortes $S'_0, \ldots S'_p$ et les opérations F'_0, $\ldots F'_q$ sont respectivement renommées en $S_0, \ldots S_p$, $F_0, \ldots F_q$:

> **type** T **is** T' **renamedby**
> [**sortnames** S_0 **for** $S'_0, \ldots S_p$ **for** S'_p]
> [**opnnames** F_0 **for** $F'_0, \ldots F_q$ **for** F'_q]
> **endtype**

Le renommage peut être utilisé pour changer les notations des sortes et des opérations d'un type existant.

Exemple A-8
Il est possible de donner aux opérateurs de comparaison du type `NATURAL` défini dans l'exemple A-2 (p. 208) une lexicographie plus familière (il n'est pas possible de définir le symbole "`=`" qui est un mot réservé) :

```
type RENAMED_NATURAL is NATURAL renamedby
   opnnames == for eq
            <> for ne
            <  for lt
            <= for le
            >  for gt
            >= for ge
endtype
```

■

Le renommage est aussi utilisé pour construire un nouveau type ayant la même structure qu'un type existant, mais destiné à modéliser d'autres objets que le type existant. On retrouve cette approche en ADA avec le mécanisme des types dérivés (**type** T **is new** T').

Exemple A-9
Mathématiquement parlant, les deux corps commutatifs $(\{\mathit{false}, \mathit{true}\}, \mathit{xor}, \mathit{and})$ et $(\{0, 1\}, +, .)$, où "$+$" dénote l'addition modulo 2 et "." la multiplication, sont isomorphes. En LOTOS il est possible de définir l'une de ces structures algébriques comme simple renommage de l'autre :

```
type BINARY is BOOLEAN renamedby
   sortnames BIN for BOOL
   opnnames 0 for false
            1 for true
            + for xor
            . for and
endtype
```

■

Annexe B

Présentation du langage LOTOS : les structures de contrôle

Après les structures de données, cette annexe présente, de manière simple et accessible, l'autre partie du langage LOTOS : la description du contrôle. On peut également consulter le *tutorial* LOTOS [BB88].

LOTOS est un langage parallèle qui s'inspire des *algèbres de processus*, notamment CCS [Mil80] et TCSP [BHR84] : le contrôle des programmes est décrit par des expressions algébriques appelées *comportements*. La synchronisation et la communication s'effectuent exclusivement par rendez-vous, sans partage de mémoire.

B.1 Eléments lexicographiques et syntaxiques

B.1.1 Expressions de comportement

On appelle *opérateurs de comportement* les structures de contrôle utilisées dans le langage : composition séquentielle, composition parallèle, ... Dans la suite ces opérateurs sont présentés l'un après l'autre.

On appelle *expression de comportement* (*behaviour expression*) — ou plus simplement *comportement* — un terme syntaxique obtenu par combinaison des opérateurs de comportement. Les expressions de comportement sont dénotées par le non-terminal B.

B.1.2 Expressions de valeur

On appelle *expression de valeur* (*value expression*) — ou plus simplement *valeur* — un terme algébrique construit à partir de variables et d'opérateurs. Les expressions de valeur ont été définies au chapitre précédent, où elles apparaissaient dans les équations algébriques ; elles sont aussi utilisées dans les expressions de comportement. Les expressions de valeur sont dénotées par le non-terminal V.

LOTOS est fortement typé : chaque valeur ne peut avoir qu'une seule sorte, qu'il est possible de déterminer statiquement.

Les sortes et les opérations qui sont utilisées dans les comportements LOTOS ne doivent pas être formelles ; elles doivent appartenir à des types complètement instanciés.

Si S est une sorte, on note $domain(S)$ l'ensemble quotient de tous les termes de sorte S par la relation de congruence définie par les équations associées à S. On fait l'hypothèse que le domaine de chaque sorte n'est pas vide.

Si V_1 et V_2 sont deux valeurs de sorte S, on note $V_1 = V_2$ le fait que V_1 et V_2 soient congrues modulo la relation de congruence définie par les équations associées à S.

B.1.3 Variables

Une *variable* est un nom donné à une valeur. LOTOS est un langage fonctionnel : chaque variable est initialisée dès sa déclaration et sa valeur ne peut pas être modifiée.

B.1.4 Portes

En LOTOS, on appelle *porte* (*gate*) un canal de communication permettant la synchronisation par rendez-vous et l'échange de valeurs entre plusieurs tâches qui se déroulent en parallèle.

On note Γ l'ensemble de tous les identificateurs de portes, définis par l'utilisateur, qui figurent dans une spécification LOTOS. Deux portes spéciales sont prédéfinies, qui n'appartiennent pas à Γ :

- la porte invisible, notée "**i**"[44]. Cette porte peut apparaître dans les programmes LOTOS, mais uniquement dans le contexte d'un opérateur "**;**"
- la porte de terminaison, notée "δ". Cette porte ne peut jamais être employée explicitement dans un programme LOTOS mais elle est utilisée dans la définition sémantique du langage

B.1.5 Identificateurs

Chaque classe d'identificateurs est dénotée par un symbole non-terminal défini comme suit :

- G : portes
- P : processus
- X : variables
- T : types
- S : sortes
- F : opérations

On emploie en outre les abréviations suivantes :

- $\widehat{G}$: liste non vide de portes $G_0, \ldots\ G_n$
- $\widehat{X}$: liste non vide de variables $X_0, \ldots\ X_n$

On introduit à présent tous les opérateurs de contrôle du langage LOTOS. Un même exemple, celui d'un distributeur de boissons, est conservé tout au long de la présentation.

[44]qui correspond à la porte "τ" de CCS

B.2 Opérateur "stop"

La construction suivante :

stop

dénote un comportement inactif, qui ne propose aucun rendez-vous avec l'environnement ni aucune transition "**i**" interne.

B.3 Opérateur ";"

L'opérateur ";" permet de spécifier le rendez-vous. Si G est une porte et B_0 un comportement, la construction suivante :

$$G \; ; \; B_0$$

dénote le comportement qui propose un rendez-vous sur la porte G et, une fois qu'il a eu lieu, exécute B_0. La notation ";" a une signification séquentielle : on dit que le comportement B est *préfixé* par la porte G. Les termes *événement* et *interaction* seront utilisés comme synonymes de rendez-vous.

Exemple B-1
Le comportement suivant effectue une interaction `MONEY` (acquisition de pièces de monnaie) puis une interaction `TEA` (distribution d'une tasse de thé), après quoi il s'arrête :

```
MONEY;
   TEA;
      stop
```

En fait cet exemple décrit également le comportement d'un utilisateur qui, après avoir payé, reçoit une tasse de thé. ■

Cette forme simple de rendez-vous, qui ne comporte pas d'émission ni de réception de valeurs, ne permet que la *synchronisation pure*. Il existe une construction plus générale qui prend en compte l'échange de valeurs :

$$G \; O_0, \ldots \; O_n \; ; \; B_0$$

où $O_0, \ldots \; O_n$ sont des *offres*, définies comme suit :

$$\begin{array}{lll} O & \equiv & \texttt{!}V \\ & | & \texttt{?}X_0, \ldots \; X_n \texttt{:} S \end{array}$$

Une offre de la forme "$\texttt{!}V$" correspond à l'émission sur la porte G de la valeur de l'expression V. Une offre de la forme "$\texttt{?}X_0, \ldots \; X_n \texttt{:} S$" correspond à la réception sur la porte G de $n+1$ valeurs $v_0, \ldots \; v_n$ de sorte S ; chacune de ces valeurs v_i est ensuite affectée à la variable X_i correspondante.

Le rendez-vous est bloquant aussi bien pour l'émission que la réception : l'exécution d'un comportement qui attend un rendez-vous est suspendue et ne reprend qu'après que le rendez-vous a eu lieu. Le rendez-vous LOTOS est absolument symétrique ; aucune distinction n'est faite entre émetteur et récepteur.

Un seul et même rendez-vous peut comporter plusieurs émissions et réceptions qui se déroulent simultanément. De plus, une même porte peut être successivement utilisée dans plusieurs rendez-vous, tantôt avec des émissions, tantôt avec des réceptions.

LOTOS permet de conditionner le rendez-vous par une *garde* qui est soit une expression booléenne "[V_0]", soit une équation simple "[V_1=V_2]". Le rendez-vous n'a pas lieu si la condition définie par la garde n'est pas satisfaite.

Exemple B-2
Le comportement suivant modélise un distributeur qui effectue successivement trois interactions :

- acquisition d'une somme d'argent (interaction MONEY) en dollars et en cents ; au moyen d'une garde on interdit le rendez-vous si cette somme est inférieure au prix attendu
- distribution d'une tasse de thé (interaction TEA)
- restitution de la monnaie (interaction CHANGE)

```
MONEY ?DOLLARS:NAT ?CENTS:NAT [TOTAL (DOLLARS, CENTS) ge COST];
   TEA;
      CHANGE !CHG_DOLLARS (DOLLARS, CENTS) !CHG_CENTS (DOLLARS, CENTS);
         stop
```

Les opérations comptables sont décrites à l'aide d'un type abstrait :

```
type CHANGE is NATURALNUMBER, BOOLEAN
   opns COST : -> NAT
        TOTAL : NAT, NAT -> NAT
        CHG_DOLLARS : NAT, NAT -> NAT
        CHG_CENTS : NAT, NAT -> NAT
   eqns forall DOLLARS, CENTS:NAT ofsort NAT
       COST = 25;   (* prix d'une boisson, exprime en cents *)
       TOTAL (DOLLARS, CENTS) = (100 * DOLLARS) + CENTS;
       CHG_DOLLARS (DOLLARS, CENTS) = (TOTAL (DOLLARS, CENTS) - COST) div 100;
       CHG_CENTS (DOLLARS, CENTS) = (TOTAL (DOLLARS, CENTS) - COST) mod 100;
endtype
```

■

On peut faire figurer la porte "**i**" à gauche de l'opérateur ";", mais elle ne doit comporter ni offre ni garde. Le préfixage par la porte "**i**" spécifie une évolution interne qui n'est jamais bloquante.

En résumé la syntaxe générale de l'opération de préfixage est donc :

$$\begin{array}{ll} & \mathbf{i} \;\; \texttt{;} \; B_0 \\ | & G \; [O_0, \ldots \; O_n \; [\texttt{[}V_0\texttt{]}]] \;\texttt{;}\; B_0 \\ | & G \; [O_0, \ldots \; O_n \; [\texttt{[}V_1\texttt{=}V_2\texttt{]}]] \;\texttt{;}\; B_0 \end{array}$$

Les variables éventuellement définies dans les offres O_0, ... O_n ne sont visibles que dans V_0, V_1, V_2 et B_0.

B.4 Opérateur "[]"

L'opérateur "[]" permet de spécifier le choix non-déterministe. Si B_1 et B_2 sont deux comportements, la construction suivante :

$$B_1 \;\texttt{[]}\; B_2$$

dénote le comportement qui peut exécuter soit B_1 soit B_2. La signification intuitive de cet opérateur est identique à celle de la barre carrée "[]" de Dijkstra.

Remarque B-1
Il n'est pas permis d'écrire en LOTOS des comportements de la forme :

$$(G_1 \texttt{ [] } G_2) \texttt{ ; } G_3 \texttt{ ; } \textbf{stop}$$

Il s'agit d'une erreur syntaxique car les opérandes de "[]" doivent être des comportements et non des portes ; de même l'opérande gauche de ";" doit être une porte et non un comportement. La manière correcte d'écrire le comportement ci-dessus est :

$$(G_1 \texttt{ ; } G_3 \texttt{ ; } \textbf{stop}) \texttt{ [] } (G_2 \texttt{ ; } G_3 \texttt{ ; } \textbf{stop})$$

■

Exemple B-3
Le comportement suivant modélise un distributeur qui, après avoir accepté le paiement (interaction `MONEY`) peut délivrer du thé, du café ou du chocolat chaud (interactions `TEA`, `COFFEE` et `CHOCOLATE`) :

```
MONEY;
   (
   TEA;
      stop
   []
   COFFEE;
      stop
   []
   CHOCOLATE;
      stop
   )
```

■

Le fait que le choix soit non-déterministe ne signifie pas que le distributeur décide arbitrairement de la boisson qu'il fournit. Le distributeur doit respecter les contraintes imposées par l'*environnement*, c'est-à-dire les rendez-vous proposés par les autres comportements avec lesquels il communique et se synchronise.

Dans l'exemple B-3 (p. 219) ce sont les utilisateurs du distributeur qui constituent cet environnement : après avoir payé (interaction `MONEY`), chaque consommateur sélectionne la boisson qu'il désire, par exemple en appuyant sur un bouton. S'il choisit le café, l'interaction `COFFEE` est imposée au distributeur.

Le non-déterminisme intervient effectivement lorsque les contraintes de l'environnement ne déterminent pas une possibilité unique. Pour reprendre l'exemple B-3 (p. 219), si le consommateur pouvait indiquer qu'il désire soit du café, soit du chocolat, le choix du distributeur (interaction `COFFEE` ou `CHOCOLATE`) serait imprévisible.

Il existe des comportements pour lesquels aucune interprétation déterministe ne peut être trouvée, quel que soit l'environnement. L'exemple le plus simple est de la forme :

$$(G \texttt{ ; } B_1) \texttt{ [] } (G \texttt{ ; } B_2)$$

Dans ce cas le choix entre l'exécution de B_1 ou de B_2 est purement arbitraire. La porte "**i**" introduit également le non-déterminisme. Quel que soit l'environnement, le comportement suivant :

$$B_1 \texttt{ [] } \textbf{i} \texttt{ ; } B_2$$

ne peut pas recevoir d'interprétation déterministe (sauf dans le cas où B_1 est égal à "**stop**"). En effet l'événement "**i**" est toujours possible car l'environnement n'a aucune influence sur lui. Pour exprimer un choix non contrôlable par l'environnement, entre deux comportements B_1 et B_2, il faut écrire :

$$\mathbf{i} \texttt{ ; } B_1 \texttt{ [] } \mathbf{i} \texttt{ ; } B_2$$

L'opérateur "`[]`" est commutatif et associatif ; il admet "**stop**" comme élément neutre à gauche (*resp.* à droite) ; tout comportement est idempotent pour "`[]`". Une erreur fréquente consiste à croire que "`;`" est distributif sur "`[]`" et à écrire :

$$\texttt{(}G \texttt{ ; } B_1\texttt{) [] (}G \texttt{ ; } B_2\texttt{)}$$

au lieu de :

$$G \texttt{ ; (}B_1 \texttt{ [] } B_2\texttt{)}$$

L'emploi de la première forme peut provoquer un blocage comme le montre l'exemple suivant.

Exemple B-4
Le distributeur de boissons présenté dans l'exemple B-3 (p. 219) ne doit pas être décrit ainsi :

```
MONEY;
   TEA;
      stop
[]
MONEY;
   COFFEE;
      stop
[]
MONEY;
   CHOCOLATE;
      stop
```

Au moment où le consommateur paie (interaction `MONEY`) le distributeur se trouve confronté à un choix non-déterministe, qu'il résout en sélectionnant arbitrairement une branche, au détriment des autres. S'il a choisi par exemple, la première il ne pourra plus délivrer que du thé ! En d'autres termes le choix proposé à l'utilisateur après paiement est restreint à une seule des trois interactions `TEA`, `COFFEE` et `CHOCOLATE`. ■

B.5 Opérateur "choice" sur les portes

Soit B_0 un comportement qui contient des occurrences d'utilisation d'une porte G. Soient $G_1, \ldots G_n$ des portes et soient $B_1, \ldots B_n$ les comportements définis de la manière suivante : B_i est obtenu à partir de B en remplaçant G par G_i. Pour exprimer le comportement :

$$B_1 \texttt{ [] } \ldots B_n$$

LOTOS permet d'utiliser une notation abrégée :

$$\textbf{choice } G \textbf{ in } \texttt{[}G_1, \ldots G_n\texttt{] [] } B_0$$

La porte G sert d'indice à cette itération. Il n'est pas indispensable que les portes $G_1, \ldots G_n$ soient deux à deux distinctes.

Exemple B-5
L'exemple B-3 (p. 219) peut être écrit de manière plus concise :

```
MONEY;
   (
   choice DRINK in [TEA, COFFEE, CHOCOLATE] []
      DRINK;
         stop
   )
```

■

L'opérateur "**choice**" possède une forme plus générale permettant d'itérer sur plusieurs portes :

$$\textbf{choice}\ \widehat{G_0}\ \textbf{in}\ \texttt{[}\widehat{G'_0}\texttt{]}, \dots\ \widehat{G_n}\ \textbf{in}\ \texttt{[}\widehat{G'_n}\texttt{]}\ \texttt{[]}\ B_0$$

Les portes définies dans les listes $\widehat{G_0}$, ... $\widehat{G_n}$ ne sont visibles que dans B_0.

B.6 Opérateurs "||", "|||" et "|[...]|"

Les opérateurs qui ont été présentés jusqu'ici sont strictement *séquentiels* ; LOTOS comprend aussi des opérateurs *parallèles*. Si B_1 et B_2 sont deux comportements et G_0, ... G_n une liste de portes, la construction suivante :

$$B_1\ \texttt{|[}G_0, \dots\ G_n\texttt{]|}\ B_2$$

dénote le comportement qui exécute B_1 et B_2 en parallèle. La synchronisation et la communication entre les opérandes B_1 et B_2 s'effectuent uniquement par rendez-vous sur les portes de l'ensemble $\{G_0, \dots\ G_n, \delta\}$.

Lorsqu'un des opérandes veut effectuer une transition étiquetée par une porte G de $\{G_0, \dots\ G_n, \delta\}$, il doit attendre que l'autre opérande puisse en faire autant. Lorsque le rendez-vous est possible, les deux opérandes effectuent simultanément une même transition *synchrone* étiquetée G ; puis ils reprennent chacun leur exécution.

En revanche si l'un des opérandes veut effectuer une transition étiquetée par une porte G qui n'appartient pas à $\{G_0, \dots\ G_n, \delta\}$, il le fait indépendamment de l'autre opérande, de manière *asynchrone*.

Remarque B-2
En CCS, le rendez-vous entre deux portes complémentaires est facultatif ; on peut le rendre obligatoire en utilisant l'opérateur de restriction, mais on ne peut jamais l'interdire.

En LOTOS si deux portes identiques sont composées de manière synchrone, le rendez-vous est obligatoire ; si elles sont composées de manière asynchrone, le rendez-vous est interdit. ■

Exemple B-6
Pour composer en parallèle le distributeur de boissons (exemple B-5 (p. 220)) et un consommateur de thé (exemple B-1 (p. 217)) il faut les synchroniser sur les quatre interactions `MONEY`, `TEA`, `COFFEE` et `CHOCOLATE`. Comme le client n'effectue pas les interactions `COFFEE` et `CHOCOLATE`, le distributeur, qui doit se synchroniser avec lui, ne le peut pas non plus.

```
MONEY;
   (
   choice DRINK in [TEA, COFFEE, CHOCOLATE] []
      DRINK;
         stop
   )
|[MONEY, TEA, COFFEE, CHOCOLATE]|
   MONEY;
      TEA;
         stop
```

■

Outre l'opérateur de synchronisation général "`|[`$G_0, \ldots\ G_n$`]|`" qui traduit la synchronisation sur les portes $G_0, \ldots\ G_n$ et "δ", LOTOS possède deux autres opérateurs de composition parallèle :

- le premier opérateur exprime la synchronisation sur aucune porte, sauf "δ" (*interleaving*). Sa syntaxe est :

  $$B_1 \texttt{ ||| } B_2$$

  Les deux comportements B_1 et B_2 sont exécutés de manière totalement indépendante (terminaison sur "δ" exceptée) : il ne se synchronisent ni ne communiquent l'un avec l'autre. En revanche ils sont capables d'interagir avec leur environnement commun : "B_1 `|||` B_2" peut participer à un rendez-vous si et seulement si B_1 *ou* B_2 le peut

- le second opérateur exprime la synchronisation sur toutes les portes, y compris "δ" (*full synchronisation*). Sa syntaxe est :

  $$B_1 \texttt{ || } B_2$$

  Les deux comportements B_1 et B_2 sont exécutés en parallèle de manière entièrement synchrone : ils doivent se synchroniser sur toutes leurs interactions. Ils peuvent interagir avec leur environnement commun : "B_1 `||` B_2" peut participer à un rendez-vous si et seulement si B_1 *et* B_2 le peuvent

Exemple B-7
Si l'on veut modéliser le comportement simultané de quatre utilisateurs du distributeur de boissons décrit dans l'exemple B-6 (p. 221), il faut employer l'opérateur "`|||`". En effet ces consommateurs (thé : 1, café : 2, chocolat : 1) sont en concurrence pour l'accès à la ressource commune constituée par la machine. En revanche, comme la machine ne peut servir qu'un seul client à la fois, le groupe des quatre consommateurs doit être synchronisé avec le distributeur sur toutes les portes (`MONEY`, `TEA`, `COFFEE` et `CHOCOLATE`) ; on peut donc employer l'opérateur "`||`" :

```
MONEY;
   (
   choice DRINK in [TEA, COFFEE, CHOCOLATE] []
      DRINK;
         stop
   )
||
   (
   MONEY;
      TEA;
         stop
   |||
   MONEY;
      COFFEE;
         stop
   |||
   MONEY;
      COFFEE;
         stop
   |||
   MONEY;
      CHOCOLATE;
         stop
   )
```

■

Le tableau suivant indique pour chaque opérateur de composition parallèle et pour chaque porte de $\Gamma \cup \{\delta\} \cup \{\mathbf{i}\}$ si deux comportements composés en parallèle par cet opérateur doivent, oui ou non, se synchroniser sur cette porte.

portes	"|||"	"|[$G_0, \ldots\ G_n$]|"	"||"
δ	oui	oui	oui
$G_0, \ldots\ G_n$	non	oui	oui
$\Gamma - \{G_0, \ldots\ G_n\}$	non	non	oui
$\mathbf{i}$	non	non	non

Lorsque les portes sont accompagnées d'offres, c'est-à-dire quand la composition parallèle a la forme suivante :

$$(G_1\ O_0^1 \ldots\ O_n^1\ \texttt{[}V_1^1\texttt{=}V_2^1\texttt{]}\ \texttt{;}\ B_1')\ \mathit{op}\ (G_2\ O_0^2 \ldots\ O_p^2\ \texttt{[}V_1^2\texttt{=}V_2^2\texttt{]}\ \texttt{;}\ B_2')$$

le rendez-vous n'a lieu que si les conditions suivantes sont vérifiées :

- les portes G_1 et G_2 sont égales et leur synchronisation est permise par l'opérateur *op*
- le nombre d'offres de part et d'autre est le même ($n = p$)
- les sortes des offres O_i^1 et O_i^2 sont deux à deux identiques
- les deux gardes sont vérifiées

Lorsqu'il y a confrontation entre une émission (offre "!") et une réception (offre "?") la valeur émise est affectée à la variable de réception (*value passing*).

Exemple B-8
C'est le cas lorsqu'on compose en parallèle un distributeur qui rend la monnaie (exemples B-2 (p. 218) et B-6 (p. 221)) et un buveur de thé qui fournit \$1.00 à la machine et reprend sa monnaie.

```
MONEY ?DOLLARS:NAT ?CENTS:NAT [TOTAL (DOLLARS, CENTS) ge COST];
   (
   choice DRINK in [TEA, COFFEE, CHOCOLATE] []
      DRINK;
         CHANGE !CHG_DOLLARS (DOLLARS, CENTS) !CHG_CENTS (DOLLARS, CENTS);
            stop
   )
|[MONEY, TEA, COFFEE, CHOCOLATE, CHANGE]|
MONEY !1 !0;
   TEA;
      CHANGE ?DOLLARS:NAT ?CENTS:NAT;
         stop
```

■

LOTOS autorise également les confrontations entre deux émissions ("!" et "!") ou deux réceptions ("?" et "?"). Dans le premier cas (*value matching*), le rendez-vous n'a lieu que si les deux valeurs émises sont égales. Dans le second cas (*value generation*), les deux variables de réception reçoivent une valeur identique, choisie de manière non-déterministe. Le tableau suivant résume ces différentes possibilités :

offre n° 1	*offre n° 2*	*condition*	*action*
$!V_1$	$!V_2$	$V_1 = V_2$	—
$!V_1$	$?X_2 : S_2$	$V_1 \in domain(S_2)$	$X_2 := V_1$
$?X_1 : S_1$	$!V_2$	$V_2 \in domain(S_1)$	$X_1 := V_2$
$?X_1 : S_1$	$?X_2 : S_2$	$S_1 = S_2$	$X_1, X_2 := V (V \in domain(S_1))$

LOTOS permet le rendez-vous n-aire, c'est-à-dire la synchronisation, sur un même événement, de n comportements concurrents.

Exemple B-9

On peut imaginer un percolateur qui délivre simultanément deux tasses de café après avoir accepté successivement deux pièces de monnaie. Il est utilisé par deux consommateurs qui se synchronisent sur la porte `COFFEE` (puisqu'ils doivent recevoir simultanément leur boisson) et ne se synchronisent pas sur la porte `MONEY` (puisqu'ils paient à tour de rôle). On a ainsi un rendez-vous à trois sur la porte `COFFEE` entre le percolateur et les deux clients :

```
MONEY;
   MONEY;
      COFFEE;
         stop
|[MONEY, COFFEE]|
   (
   MONEY;
      COFFEE;
         stop
   |[COFFEE]|
   MONEY;
      COFFEE;
         stop
   )
```

■

Au cours d'un rendez-vous n-aire, n offres $O_1, \ldots O_n$ peuvent s'unifier si et seulement si l'intersection des ensembles de valeurs permis par les offres $O_1, \ldots O_n$ est non vide (s'il y a des gardes, il faut se restreindre aux valeurs pour lesquelles les conditions des gardes sont vérifiées). La valeur échangée est choisie de manière non-déterministe dans cette intersection. En particulier ce mécanisme permet de spécifier la *diffusion*, c'est-à-dire l'émission d'un message par un comportement et sa réception par les autres comportements.

Chaque opérateur parallèle est commutatif et associatif (l'associativité n'est pas vérifiée pour deux opérateurs parallèles différents) ; l'opérande "**stop**" n'est élément absorbant à gauche (*resp.* à droite) que si l'autre opérande n'effectue pas de transition étiquetée "**i**".

B.7 Opérateur "par"

De la même manière qu'il existe un opérateur "**choice**" permettant d'écrire le choix non-déterministe sous une forme concise, LOTOS comporte un opérateur "**par**" adapté à la composition parallèle.

On note "*op*" un des trois opérateurs parallèle :

$$\begin{array}{rcl} op & \equiv & \texttt{||} \\ & | & \texttt{|||} \\ & | & \texttt{|[}\widehat{G}\texttt{]|} \end{array}$$

Soit B_0 un comportement qui contient des occurrences d'utilisation d'une porte G. Soient $G_1, \ldots$ G_n des portes et soient $B_1, \ldots$ B_n les comportements définis de la manière suivante : B_i est obtenu à partir de B en remplaçant G par G_i. Pour exprimer le comportement :

$$B_1 \ op \ \ldots \ B_n$$

LOTOS permet d'utiliser une notation abrégée :

$$\textbf{par } G \textbf{ in } [G_1, \ldots \ G_n] \ op \ B_0$$

La porte G sert d'indice à cette itération. Il n'est pas nécessaire que les portes $G_1, \ldots$ G_n soient deux à deux distinctes.

Exemple B-10
On peut écrire l'exemple B-7 (p. 222) de manière plus concise :

```
MONEY;
   (
   choice DRINK in [TEA, COFFEE, CHOCOLATE] []
      DRINK;
         stop
   )
||
   (
   par DRINK in [TEA, COFFEE, COFFEE, CHOCOLATE] |||
      MONEY;
         DRINK;
            stop
   )
```

■

Comme pour l'opérateur "**choice**", il existe une forme plus générale permettant d'itérer sur plusieurs portes :

$$\textbf{par } \widehat{G_0} \textbf{ in } [\widehat{G'_0}], \ldots \ \widehat{G_n} \textbf{ in } [\widehat{G'_n}] \ op \ B_0$$

Les portes définies dans les listes $\widehat{G_0}, \ldots$ $\widehat{G_n}$ ne sont visibles que dans B_0.

B.8 Opérateur "hide"

LOTOS possède un opérateur qui permet de cacher certaines portes d'un comportement. Si G_0, ... G_n sont des portes et B_0 un comportement, la construction suivante :

$$\textbf{hide}\ G_0, \ldots\ G_n\ \textbf{in}\ B_0$$

dénote le comportement B_0 dont les portes $G_0, \ldots\ G_n$ sont renommées en "**i**". Les portes $G_0, \ldots\ G_n$ ne sont visibles que dans B_0. Elles deviennent inaptes à la synchronisation pour l'environnement de B_0 avec lequel elles n'interfèrent plus. Vu de l'extérieur ces interactions sont invisibles (puisqu'étiquetées "**i**") et ont lieu spontanément sans aucune participation de l'environnement de B_0 : le rendez-vous sur une porte cachée n'est jamais bloquant.

Exemple B-11
Bien souvent un comportement est décrit comme la mise en parallèle de plusieurs sous-comportements qui se synchronisent sur un ensemble de portes qu'il convient de dissimuler vis-à-vis de l'environnement. Dans cet esprit, on peut décomposer le distributeur décrit dans l'exemple B-8 (p. 223) en deux sous-systèmes :

- le premier reçoit une somme d'argent, s'assure que le montant est suffisant, calcule la monnaie à rendre et envoie une autorisation à l'autre sous-système via une porte `GRANT`
- le second, lorsque l'autorisation est accordée, délivre une boisson (thé, café ou chocolat, au choix du client) et rend la monnaie (la somme qu'il faut restituer lui a été communiquée via la porte `GRANT`)

On cache la porte `GRANT` au moyen de l'opérateur "**hide**" car il s'agit d'un détail d'implémentation qui n'est pas pertinent pour un observateur extérieur.

Le distributeur est composé en parallèle avec un consommateur de thé qui fournit $1.00 pour payer. Noter que l'opérateur "`||`" impose la synchronisation sur les portes `MONEY`, `TEA`, `COFFEE`, `CHOCOLATE` et `CHANGE` mais pas `GRANT`, qui est cachée.

```
hide GRANT in
   (
   MONEY ?DOLLARS:NAT ?CENTS:NAT [TOTAL (DOLLARS, CENTS) ge COST];
      GRANT !CHG_DOLLARS (DOLLARS, CENTS) !CHG_CENTS (DOLLARS, CENTS);
         stop
   |[GRANT]|
   GRANT ?DOLLARS:NAT ?CENTS:NAT;
      (
      choice DRINK in [TEA, COFFEE, CHOCOLATE] []
         DRINK;
            CHANGE !DOLLARS !CENTS;
               stop
      )
   )
||
   MONEY !1 !0;
      TEA;
         CHANGE ?DOLLARS:NAT ?CENTS:NAT;
            stop
```

■

Le rendez-vous "unaire" est correct et non bloquant ; par exemple :

$$\textbf{hide}\ G\ \textbf{in}\ G\ ;\ \textbf{stop}$$

est équivalent, vu de l'extérieur, à :

i ; stop

Remarque B-3
On peut comparer les solutions retenues pour CCS et LOTOS : CCS ne possède pas d'opérateur d'abstraction ("**hide**" en LOTOS) et, inversement, LOTOS ne possède pas d'opérateur de restriction ("\" en CCS).

En CCS l'abstraction est couplée avec l'opérateur parallèle : après synchronisation les portes complémentaires G et $\overline{G}$ sont cachées ; ce choix interdit le rendez-vous n-aire mais permet les rendez-vous à 2 parmi n.

En LOTOS la restriction est couplée avec l'opérateur parallèle : si cet opérateur indique qu'une porte G doit être synchronisée, alors tout rendez-vous sur G est bloquant. Cette méthode autorise le rendez-vous n-aire ; en revanche le rendez-vous à 2 parmi n est plus difficile à obtenir, mais plusieurs solutions existent néanmoins. ■

B.9 Opérateur "->"

LOTOS permet de conditionner tout comportement par une garde qui est, soit une expression booléenne, soit une équation simple. Si V_0, V_1 et V_2 sont des expressions de valeur et B_0 une expression de comportement, les deux constructions :

$$[V_0] \texttt{ -> } B_0$$

et :

$$[V_1 = V_2] \texttt{ -> } B_0$$

dénotent un comportement qui est égal à B_0 si la condition de garde est vraie et à "**stop**" si elle est fausse.

LOTOS ne possède pas de clause "**if then else**" : il faut utiliser des *commandes gardées*, obtenues en combinant l'opérateur "->" avec l'opérateur "[]", comme le montre l'exemple suivant.

Exemple B-12
Le distributeur décrit dans l'exemple B-8 (p. 223) peut être modifié afin qu'il accepte tout paiement quel que soit son montant mais, s'il est insuffisant, il restitue la somme sans délivrer de boisson :

```
MONEY ?DOLLARS:NAT ?CENTS:NAT;
   (
   [TOTAL (DOLLARS, CENTS) lt COST] ->
      CHANGE !DOLLARS !CENTS;
         stop
   []
   [TOTAL (DOLLARS, CENTS) ge COST] ->
      (
      choice DRINK in [TEA, COFFEE, CHOCOLATE] []
         DRINK;
            CHANGE !CHG_DOLLARS (DOLLARS, CENTS) !CHG_CENTS (DOLLARS, CENTS);
               stop
      )
   )
```

■

Si, dans une commande gardée de la forme :

$$\begin{array}{ll} & \texttt{[}V_0\texttt{] ->}\ B_0 \\ \texttt{[]} & \texttt{[}V_1\texttt{] ->}\ B_1 \\ & \dots \\ \texttt{[]} & \texttt{[}V_n\texttt{] ->}\ B_n \end{array}$$

les conditions V_0, ... V_n ne sont pas mutuellement exclusives, le non-déterministe s'applique.

Remarque B-4
Le comportement :

$$G_1\ O_0^1 \dots\ O_n^1\ \texttt{[}V_1^1\texttt{=}V_2^1\texttt{]}\ \texttt{;}\ \textbf{stop}$$

n'est pas équivalent à :

$$G_1\ O_0^1 \dots\ O_n^1\ \texttt{;}\ \texttt{[}V_1^1\texttt{=}V_2^1\texttt{] ->}\ \textbf{stop}$$

car dans le second cas le rendez-vous sur la porte G a toujours lieu, même si la garde est fausse. ■

Remarque B-5
L'opérateur "`[]`" de Lotos possède de "bonnes propriétés" qui limitent le risque de blocage. En particulier la garde est distributive sur le choix non-déterministe. Autrement dit, le comportement :

$$\texttt{([}V\texttt{] ->}\ B_1\texttt{)}\ \texttt{[]}\ \texttt{([}V\texttt{] ->}\ B_2\texttt{)}$$

est équivalent à :

$$\texttt{[}V\texttt{] -> (}B_1\ \texttt{[]}\ B_2\texttt{)}$$

En CSP et dans les autres langages où la garde n'est pas distributive sur la composition non-déterministe, les comportements de ce genre sont généralement incorrects. En effet, si l'on écrit en CSP [Hoa78] un comportement de la forme :

$$\begin{array}{l} \textbf{true}\ \texttt{->}\ G_1\ ?X_1\ \texttt{;}\ B_1 \\ \texttt{[]} \\ \textbf{true}\ \texttt{->}\ G_2\ ?X_2\ \texttt{;}\ B_2 \end{array}$$

on obtient un choix non-déterministe qui n'est pas contrôlable par l'environnement. ■

B.10 Opérateur "let"

Lorsqu'une expression de valeur doit être utilisée plusieurs fois, il est possible de lui donner un nom grâce à une définition de variable. Si V est une expression de valeur de sorte S, la construction suivante :

$$\textbf{let}\ X\texttt{:}S\texttt{=}V\ \textbf{in}\ B_0$$

dénote le comportement obtenu à partir de B_0 dans lequel la variable X possède la valeur V.

Exemple B-13
Le distributeur décrit dans l'exemple B-12 (p. 227) peut être simplifié en utilisant une variable booléenne `OK` pour factoriser les expressions figurant dans les gardes :

```
MONEY ?DOLLARS:NAT ?CENTS:NAT;
   (
   let OK:BOOL=(TOTAL (DOLLARS, CENTS) ge COST) in
      (
      [not (OK)] ->
         CHANGE !DOLLARS !CENTS;
            stop
      []
      [OK] ->
         (
         choice DRINK in [TEA, COFFEE, CHOCOLATE] []
            DRINK;
               CHANGE !CHG_DOLLARS (DOLLARS, CENTS) !CHG_CENTS (DOLLARS, CENTS);
                 stop
         )
      )
   )
```

■

L'opérateur "**let**" possède une forme plus générale permettant de définir simultanément plusieurs variables :

$$\textbf{let } \widehat{X_0} : S_0 = V_0, \ldots \ \widehat{X_n} : S_n = V_n \textbf{ in } B_0$$

Les variables définies dans les listes $\widehat{X_0}, \ldots \ \widehat{X_n}$ ne sont visibles que dans B_0. Chaque variable de la liste $\widehat{X_i}$ est de sorte S_i et a pour valeur V_i.

B.11 Opérateur "choice" sur les valeurs

Soit B_0 un comportement qui contient des occurrences d'utilisation d'une variable X de sorte S. Soit B_V le comportement obtenu à partir de B_0 en donnant à X la valeur V. Pour exprimer le comportement :

$$B_{V_1} \texttt{ [] } \ldots B_{V_n} \texttt{ [] } \ldots$$

où $V_1, \ldots V_n, \ldots$ désignent les valeurs de *domain*(S), LOTOS permet d'utiliser une notation abrégée :

$$\textbf{choice } X : S \texttt{ [] } B_0$$

La variable X sert d'indice à cette itération sur les valeurs du domaine de S.

Exemple B-14
Il est possible de modifier la façon dont le distributeur présenté dans l'exemple B-8 (p. 223) rend la monnaie afin qu'il ne restitue pas systématiquement le nombre maximal `D_MAX` de dollars mais un nombre `D` choisi entre `0` et `D_MAX` de manière non-déterministe (le distributeur complète avec autant de cents qu'il le faut) :

```
MONEY ?DOLLARS:NAT ?CENTS:NAT [TOTAL (DOLLARS, CENTS) ge COST];
   (
   choice DRINK in [TEA, COFFEE, CHOCOLATE] []
      DRINK;
         (
         let D_MAX:NAT=CHG_DOLLARS (DOLLARS, CENTS) in
            choice D:NAT []
               [D le D_MAX] ->
                  (
                  let C:NAT=TOTAL (D_MAX - D, CHG_CENTS (DOLLARS, CENTS)) in
                     CHANGE !D !C;
                        stop
                  )
         )
   )
```

■

L'opérateur "**choice**" possède une forme plus générale permettant de définir plusieurs variables :

$$\textbf{choice}\ \widehat{X_0}\,{:}\,S_0, \ldots\ \widehat{X_n}\,{:}\,S_n\ \texttt{[]}\ B_0$$

Les variables définies dans les listes $\widehat{X_0}, \ldots\ \widehat{X_n}$ ne sont visibles que dans B_0. Chaque variable de la liste $\widehat{X_i}$ est de sorte S_i (les variables d'une même liste ne reçoivent pas forcément la même valeur).

Remarque B-6
Dans les offres de rendez-vous, une émission (offre "!") peut être vue comme un cas particulier de réception (offre "?") où le domaine des valeurs acceptées est réduit à un singleton. Réciproquement toute réception peut être exprimée comme une émission en utilisant l'opérateur "**choice**" sur les valeurs. C'est ainsi que le comportement :

$$G\ \texttt{?}X\,{:}\,S\ \texttt{[}V\texttt{]}\ \texttt{;}\ B_0$$

est équivalent à :

$$\textbf{choice}\ X\,{:}\,S\ \texttt{[]}\ \texttt{([}V\texttt{]}\ \texttt{->}\ G\ \texttt{!}X\ \texttt{;}\ B_0\texttt{)}$$

■

Remarque B-7
Il existe deux formes de l'opérateur "**choice**" : une pour les portes (§ B.5, p. 220) l'autre pour les valeurs (§ B.11, p. 229). En revanche l'opérateur "**par**" n'existe que pour les portes (§ B.7, p. 225).
■

B.12 Opérateur "exit"

L'opérateur "**stop**" permet de spécifier explicitement l'arrêt d'un comportement. Mais "**stop**" peut aussi apparaître de manière implicite, lorsqu'un comportement se bloque. Pour distinguer ces deux formes de terminaison, normale et anormale, on introduit un nouvel opérateur. La construction suivante :

exit

dénote un comportement qui se termine normalement. La terminaison avec succès s'exprime par le franchissement d'une transition "δ". De fait "**exit**" est équivalent à un rendez-vous sur la porte "δ" :

$$\delta\ \texttt{;}\ \textbf{stop}$$

Un comportement peut, lorsqu'il se termine par “**exit**”, transmettre des *résultats*. Cette possibilité correspond à l'ajout d'une liste d'offres au rendez-vous sur la porte “δ”, mais la syntaxe est différente :

$$\textbf{exit}\ (R_0, \ldots\ R_n)$$

où les résultats $R_0, \ldots\ R_n$ sont définis comme suit :

$$\begin{array}{rcl} R & \equiv & V \\ & | & \textbf{any}\ S \end{array}$$

Un résultat dénote donc, soit une valeur V déterminée, soit une valeur choisie de façon non-déterministe dans le domaine d'une sorte S.

La définition de l'opérateur de composition parallèle (§ B.6, p. 221) et de la synchronisation sur la porte “δ” implique que la composition parallèle de n comportements $B_1, \ldots\ B_n$ ne se termine par “**exit**” que si $B_1, \ldots\ B_n$ se terminent aussi par “**exit**”, de manière synchrone (*join*), en proposant des offres compatibles.

Certaines règles interdisent les constructions susceptibles de conduire à des blocages sur la porte “δ”. Savoir si un comportement se termine ou non étant un problème indécidable dans le cas général, LOTOS se contente d'imposer des contraintes “de bon sens”, qu'il est possible de vérifier statiquement et qui protègent l'utilisateur contre certaines erreurs. Chaque comportement possède une *fonctionnalité* qui spécifie si le comportement se termine et précise, dans ce cas, les sortes des valeurs qu'il renvoie par l'opérateur “**exit**”. Il faut respecter certaines règles quand on compose les comportements ; c'est ainsi qu'il est interdit d'écrire :

$$\textbf{exit}\ (V_1, \ldots\ V_n)\ |||\ \textbf{exit}\ (V'_1, \ldots\ V'_{n'})$$

lorsque n et n' ne sont pas égaux.

B.13 Opérateur “>>”

L'opérateur de préfixage “;” est asymétrique : son opérande gauche est une porte (éventuellement accompagnée d'offres) alors que son opérande droit est un comportement. LOTOS possède un autre opérateur de composition séquentielle dont les deux opérandes sont des comportements. Si B_1 et B_2 sont deux comportements, la construction suivante :

$$B_1 \gg B_2$$

dénote le comportement qui exécute séquentiellement B_1 puis B_2.

Intuitivement, la composition séquentielle est modélisée en LOTOS comme un cas particulier de composition parallèle. Les comportements B_1 et B_2 sont exécutés en parallèle mais B_2 ne peut pas commencer avant que B_1 ne se soit terminé. L'attente de B_2 s'obtient par un rendez-vous sur la porte “δ”, rendez-vous qui devient possible dès que B_1 exécute “**exit**”. Si B_1 boucle indéfiniment ou se bloque sans atteindre “**exit**” B_2 ne sera jamais exécuté.

L'opérateur “>>” est souvent appelé *enabling operator* ou *enable* puisque la terminaison avec succès de B_1 autorise l'exécution de B_2.

L'opérateur “>>” possède une forme plus générale permettant au processus qui commence de récupérer les résultats renvoyés par le processus qui se termine. Si B_1 et B_2 sont deux comportements, la construction suivante :

$$B_1 \texttt{ >> } \textbf{accept } \widehat{X_0} : S_0, \ldots\ \widehat{X_n} : S_n \textbf{ in } B_2$$

dénote le comportement formé par la composition séquentielle de B_1 de B_2 ; lorsque B_1 exécute une instruction "**exit**", les résultats qu'il renvoie sont affectés aux variables $\widehat{X_0}$, ... $\widehat{X_n}$ respectivement. Les variables définies dans les listes $\widehat{X_0}$, ... $\widehat{X_n}$ ne sont visibles que dans B_2. Chaque variable de la liste $\widehat{X_i}$ est de sorte S_i (les contraintes portant sur la fonctionnalité des comportements imposent que les résultats renvoyés par B_1 correspondent, par leur nombre et par leurs sortes, aux variables déclarées après le mot "**accept**").

Exemple B-15
On peut réécrire le distributeur présenté dans l'exemple B-13 (p. 228) en factorisant la restitution de monnaie (interaction `CHANGE`) grâce à l'opérateur de composition séquentielle :

```
MONEY ?DOLLARS:NAT ?CENTS:NAT;
   (
   let OK:BOOL=(TOTAL (DOLLARS, CENTS) ge COST) in
      (
      [not (OK)] ->
         exit (DOLLARS, CENTS)
      []
      [OK] ->
         (
         choice DRINK in [TEA, COFFEE, CHOCOLATE] []
            DRINK;
               exit (CHG_DOLLARS (DOLLARS, CENTS), CHG_CENTS (DOLLARS, CENTS))
         )
      )
   )
>> accept DOLLARS, CENTS:NAT in
   CHANGE !DOLLARS !CENTS;
      stop
```

■

L'opérateur "`>>`", avec ou sans "**accept**", est associatif.

B.14 Opérateur "`[>`"

LOTOS dispose d'un opérateur permettant de spécifier l'interruption d'un comportement par un autre. Si B_1 et B_2 sont deux comportements, la construction suivante :

$$B_1 \texttt{ [> } B_2$$

dénote le comportement qui exécute B_1 mais qui peut abandonner à tout instant l'exécution de B_1 pour commencer celle de B_2.

Intuitivement, il s'agit d'un mécanisme d'interruption avec terminaison : B_1 joue le rôle d'un traitement normal et B_2 celui d'un traitement d'exception. Initialement, B_1 est exécuté seul mais, tant qu'il n'a pas effectué de transition "δ", son exécution peut être interrompue au profit de celle de B_2. Si B_1 se bloque avant d'atteindre une instruction "**exit**" alors B_2 est inévitablement exécuté. Dans le cas contraire B_2 peut très bien ne jamais être exécuté.

L'opérateur "`[>`" est souvent appelé *disabling operator* ou *disable* puisque la terminaison avec succès de B_1 interdit l'exécution de B_2.

Exemple B-16
L'opérateur "`[>`" peut être utilisé pour décrire le comportement d'un consommateur de thé lorsque le distributeur comporte un bouton d'annulation (interaction `CANCEL`). Le comportement normal du consommateur est celui décrit dans l'exemple B-8 (p. 223) mais, à chaque instant, il peut s'interrompre et presser sur le bouton d'annulation pour tenter de se faire rembourser par le distributeur (qui n'est pas obligé d'accepter).

```
MONEY !1 !0;
   TEA;
      CHANGE ?DOLLARS:NAT ?CENTS:NAT;
         exit
[>
CANCEL;
   CHANGE ?DOLLARS:NAT ?CENTS:NAT;
      exit
```

■

L'opérateur "`[>`" est associatif.

B.15 Processus et instanciation

Dans les langages algorithmiques, il est possible de donner un nom à un bloc d'instructions, en définissant une *procédure* qui peut éventuellement être paramétrée. De manière analogue, LOTOS permet de nommer un comportement au moyen d'une définition de *processus* (*process*). Un processus est un objet qui dénote un comportement ; il peut être paramétré par une liste de *portes formelles* et/ou une liste de *variables formelles*.

Remarque B-8
Ce mécanisme est limité au 1$^{\text{er}}$ ordre : il n'existe pas d'objet qui dénote un processus. On n'a donc pas de variables ou de paramètres formels "de type processus". ■

Remarque B-9
En LOTOS le mot "**process**" n'a pas le même sens que dans d'autres langages ; il ne dénote pas forcément une activité concurrente. La création des tâches parallèles est dévolue à l'opérateur de composition parallèle et non à l'instanciation. ■

La construction suivante :

$$\begin{array}{l} \textbf{process } P\ [\texttt{[}G_0,\ldots\ G_m\texttt{]}]\ [\texttt{(}\widehat{X_0}\texttt{:}S_0,\ldots\ \widehat{X_n}\texttt{:}S_n\texttt{)}]\ \texttt{:}\ \mathit{func}\ \texttt{:=} \\ \quad B \\ \quad [\textbf{where } \mathit{block}_0,\ldots\ \mathit{block}_p] \\ \textbf{endproc} \end{array}$$

définit un processus P (éventuellement paramétré par les portes formelles $G_1, \ldots\ G_m$ et les listes de variables formelles $\widehat{X1}, \ldots\ \widehat{X_n}$ de sortes respectives $S_1, \ldots\ S_n$) dont le *corps* est le comportement B. Le non-terminal *func* spécifie la fonctionnalité de B afin de permettre une vérification statique des contraintes de fonctionnalité ; sa définition syntaxique est :

$$\begin{array}{rcl} \mathit{func} & \equiv & \textbf{noexit} \\ & | & \textbf{exit}\ [\texttt{(}S_0,\ldots\ S_n\texttt{)}] \end{array}$$

Le premier cas indique que B ne s'achève jamais par "**exit**" (ce qui signifie que B se bloque ou boucle indéfiniment) ; le second cas exprime que B se termine en renvoyant des résultats de sortes respectives

$S_0, \ldots S_n$. Chaque non-terminal $block_i$ dénote une définition de processus ou de type ; dans les deux cas il s'agit d'une définition locale dont la visibilité est limitée à la définition du processus P.

L'instanciation d'un processus s'effectue en substituant des paramètres effectifs aux paramètres formels. Si P est un processus, $G_0, \ldots G_m$ des portes et $V_0, \ldots V_n$ des expressions de valeur, la construction suivante :

$$P\ [\texttt{[}G_0, \ldots G_m\texttt{]}]\ [\texttt{(}V_0, \ldots V_n\texttt{)}]$$

dénote le corps de P dans lequel les portes formelles sont renommées par $G_0, \ldots G_m$ et les variables formelles sont instanciées avec les valeurs $V_0, \ldots V_n$.

L'emploi de la récursion est autorisé ; en LOTOS la récursion est d'ailleurs le seul moyen pour créer des comportements cycliques.

Exemple B-17
L'exemple suivant décrit un distributeur relativement élaboré. Le consommateur peut effectuer autant de paiements qu'il le souhaite (plusieurs interactions `MONEY` successives) jusqu'à ce que le prix d'une boisson `COST` soit atteint ; il peut même dépasser ce montant.

Dès que la somme versée est suffisante, le consommateur peut choisir une boisson et récupérer sa monnaie, comme dans l'exemple B-8 (p. 223). L'utilisateur a aussi la possibilité de récupérer l'argent versé en appuyant sur le bouton `CANCEL` comme dans l'exemple B-16 (p. 233).

Le fonctionnement du distributeur est cyclique : il retourne dans son état initial et peut donc servir successivement plusieurs clients.

```
   SELL [MONEY, TEA, COFFEE, CHOCOLATE, CANCEL, CHANGE] (0, 0)
where
process SELL [MONEY, TEA, COFFEE, CHOCOLATE, CANCEL, CHANGE] (D, C:NAT) : noexit :=
   MONEY ?DOLLARS:NAT ?CENTS:NAT;
      (
      let D0:NAT=(D + DOLLARS), C0:NAT=(C + CENTS) in
         (
         [TOTAL (D0, C0) ge COST] ->
            (
            choice DRINK in [TEA, COFFEE, CHOCOLATE] []
               DRINK;
                  CHANGE !CHG_DOLLARS (D0, C0) !CHG_CENTS (D0, C0);
                     SELL [MONEY, TEA, COFFEE, CHOCOLATE, CANCEL, CHANGE] (0, 0)
            )
         []
         CANCEL;
            CHANGE !D0 !C0;
               SELL [MONEY, TEA, COFFEE, CHOCOLATE, CANCEL, CHANGE] (0, 0)
         []
         SELL [MONEY, TEA, COFFEE, CHOCOLATE, CANCEL, CHANGE] (D0, C0)
         )
      )
endproc
```

■

Exemple B-18
Pour décrire un distributeur qui délivre successivement trois sortes de boissons, on peut employer une permutation circulaire des paramètres portes :

```
   SELL [MONEY, TEA, COFFEE, CHOCOLATE]
where
process SELL [MONEY, DRINK_1, DRINK_2, DRINK_3] : noexit :=
   MONEY;
      DRINK_1;
         SELL [MONEY, DRINK_2, DRINK_3, DRINK_1]
endproc
```

■

Il est possible de spécifier en LOTOS la création et la destruction dynamique de processus concurrents ; on emploie pour cela la récursion en partie gauche ou droite d'un opérateur de composition parallèle.

B.16 Spécification LOTOS

Un programme (ou *spécification*) LOTOS est assimilable à la définition d'un processus qui englobe toutes les autres définitions de types et de processus. Par rapport à une définition de processus la syntaxe varie légèrement :

specification λ [`[`$G_0, \ldots\ G_m$`]`] [`(`$\widehat{X_0}$`:`$S_0, \ldots\ \widehat{X_n}$`:`$S_n$`)`] `:` *func*
   $type_1, \ldots\ type_p$
   behaviour B
   [**where** $block_0, \ldots\ block_q$]
endspec

Chaque non-terminal $type_i$ dénote une définition de type dont la visibilité s'étend à toute la spécification ; en particulier les sortes S_0, … S_n doivent être définies dans ces types. Le comportement B est le corps de la spécification. Le non-terminal *func* spécifie la fonctionnalité de B. Chaque non-terminal $block_i$ dénote une définition de type ou de processus.

L'identificateur λ joue le rôle d'un commentaire ; ce n'est pas un identificateur de processus, il ne peut donc pas être utilisé dans une instanciation.

B.17 Styles de programmation

Le langage LOTOS autorise plusieurs styles de spécification :

programmation logique et algébrique : on dispose pour cela, des types abstraits et des équations algébriques

programmation fonctionnelle : on peut modéliser les fonctions à décrire par des processus possédant des paramètres variables et retournant leurs résultats grâce à l'opérateur "**exit**", en se restreignant à l'emploi des opérateurs "**let**", "`->`", "`[]`", "`>>`" et l'instanciation, sans utiliser aucune porte[45] ni aucun rendez-vous

Exemple B-19
Le processus LOTOS suivant calcule le quotient et le reste de la division euclidienne de deux entiers :

[45] hormis la porte implicite δ

```
process DIV_MOD (X, Y:NAT) : exit (NAT, NAT) :=
   [X lt Y] ->
      exit (0, X)
   []
   [X ge Y] ->
      (
      DIV_MOD (X - Y, Y)
      >> accept Q, R:NAT in
         exit (Q + 1, R)
      )
endproc
```

■

programmation parallèle : il s'agit du style de programmation dual du précédent. En programmation fonctionnelle le contrôle est simplifié à l'extrême ; l'approche inverse consiste à n'avoir aucune structure de données, à se limiter à la synchronisation pure et à représenter tous les objets (piles, files, . . .) comme des réseaux de processus communicants dont la structure évolue de manière dynamique

programmation objet : nombre de langages effectuent une distinction entre les concepts d'encapsulation et de parallélisme ("**module/process**" en MODULA-2 [Wir83], "**package/task**" en ADA) alors que ces deux notions possèdent des caractéristiques voisines : un module change d'état quand les procédures qu'il exporte sont appelées ; un processus change d'état lorsque les rendez-vous qu'il propose sont acceptés.

Un objet peut être modélisé par un comportement LOTOS ; les primitives de manipulation de l'objet correspondent aux rendez-vous acceptés par le comportement. On peut ainsi imposer des restrictions sur l'ordre dans lequel les primitives de l'objet sont appelées et refuser un rendez-vous si une contrainte n'est pas respectée. Par le biais des interactions cachées on peut effectuer automatiquement certaines actions d'initialisation, de terminaison, . . .

Exemple B-20
L'exemple suivant contient la spécification en LOTOS d'une pile de nombres naturels. La structure de données qui implémente l'état de la pile est décrite par le type abstrait `STACK_DATA`. Le processus `STACK_CONTROL` constitue une interface qui accepte trois sortes de messages émis par l'utilisateur de la pile :

- `RESET` : initialiser ou ré-initialiser la pile
- `PUSH !`V : empiler la valeur V au sommet de la pile
- `POP ?`X`:NAT` : désempiler la valeur se trouvant au sommet et l'affecter à la variable X

```
process STACK [RESET, PUSH, POP] : noexit :=
   RESET;
      STACK_CONTROL [RESET, PUSH, POP] (VOID)
where
   process STACK_CONTROL [RESET, PUSH, POP] (S:STACK) : noexit :=
      RESET;
         STACK_CONTROL [RESET, PUSH, POP] (VOID)
      []
      [not (FULL (S))] ->
         PUSH ?X:NAT;
            STACK_CONTROL [RESET, PUSH, POP] (INSERT (X, S))
      []
      [not (EMPTY (S))] ->
         POP !(TOP (S));
            STACK_CONTROL [RESET, PUSH, POP] (REMOVE (S))
   endproc

   type STACK_DATA is NATURALNUMBER, BOOLEAN
      sorts STACK
      opns VOID : -> STACK
           TOP : STACK -> NAT
           INSERT : NAT, STACK -> STACK
           REMOVE : STACK -> STACK
           FULL : STACK -> BOOL
           EMPTY : STACK -> BOOL
   endtype
endproc
```

■

Annexe C

Application 1 : protocole du bit alterné

Je suis bi-alternatif
Alternativement positif
Je suis bi-bi-bi-bi-bi-bi-alternatif
Alternativement positif
Je suis alternativement bi-dégénéré, ça c'est super
Alternativement hyper-positif

GOGOL I^er^
poète, prophète, barbare

C.1 Description du service

Le protocole du bit alterné (*alternating bit protocol*) fait partie de la couche transport (4ème couche du modèle OSI). Il permet le transfert de données entre une paire d'entités pour lesquelles une connexion bi-directionnelle a été préalablement établie.

Pour simplifier le problème, on crée une disymétrie entre les deux entités : la première (T, comme *transmitter*) émet des messages à destination de la seconde (R, comme *receiver*).

Les messages sont modélisés par des numéros compris entre 1 et un entier maximal `N` ; ils sont spécifiés par le type abstrait suivant, qui ne précise pas leur nature exacte :

```
type MESSAGE is
   sorts MSG   (* type MSG = 1..N *)
endtype
```

Vu de la couche supérieure, le service fourni par le protocole du bit alterné est l'acheminement d'une série de messages de T vers R. La transmission est fiable : les messages ne peuvent pas être perdus ni dupliqués et ils sont reçus dans l'ordre où ils ont été émis. La spécification LOTOS suivante décrit ce comportement :

```
specification ALTERNATING_BIT_SERVICE [PUT, GET] : noexit behaviour
   SERVICE [PUT, GET]
where
   process SERVICE [PUT, GET] : noexit :=
      PUT ?M:MSG;  (* acquisition d'un message *)
         GET !M;   (* livraison du message *)
            SERVICE [PUT, GET]
   endproc
endspec
```

C.2 Description du protocole

Le fonctionnement "idéal" du protocole du bit alterné est le suivant : T envoie un message à R ; à la réception de ce message, R renvoie un acquittement à T.

La liaison entre T et R n'est pas fiable : il est possible que des messages ou des acquittements soient perdus. En cas de perte, le medium peut, de manière facultative, signaler cette perte au destinataire (T ou R) en envoyant une indication de perte.

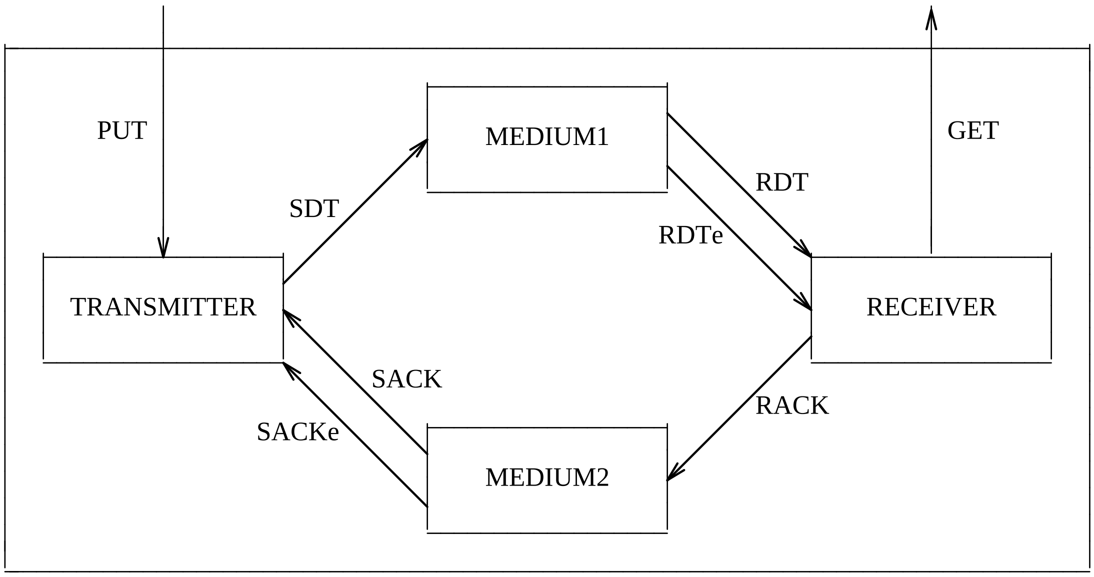
Pour détecter les pertes non signalées, les messages et les acquittements contiennent un bit de contrôle. Le bit de contrôle de chaque acquittement est égal au bit de contrôle du message qu'il acquitte. Les bits de contrôle de deux messages successivement émis ont des valeurs distinctes (la valeur du bit alterne à chaque émission).

Si l'entité T reçoit une indication de perte d'acquittement ou un acquittement avec un bit de contrôle erroné, elle réémet le dernier message envoyé.

Si l'entité R reçoit une indication de perte de message ou un message avec un bit de contrôle erroné, elle réémet le dernier acquittement envoyé.

C.3 Architecture du protocole

On choisit de décrire le protocole par quatre processus parallèles communicants :



processus	*signification*
`TRANSMITTER`	entité émettrice T
`RECEIVER`	entité réceptrice R
`MEDIUM1`	transmission des messages de T vers R
`MEDIUM2`	transmission des acquittements de R vers T

Le tableau suivant donne la liste des signaux utilisés. Les seuls signaux fournis par le service sont `PUT` et `GET` ; tous les autres désignent des signaux internes. Dans la suite M désigne un message (essentiellement un bloc de données) et B le bit de contrôle d'un message.

signal	*origine*	*destination*	*signification*
`PUT !M`	service	`TRANSMITTER`	émission d'un message
`SDT !M !B`	`TRANSMITTER`	`MEDIUM1`	envoi du message
`RDT !M !B`	`MEDIUM1`	`RECEIVER`	transmission du message
`RDTe`	`MEDIUM1`	`RECEIVER`	perte du message
`GET !M`	`RECEIVER`	service	réception du message
`RACK !B`	`RECEIVER`	`MEDIUM2`	renvoi d'un acquittement
`SACK !B`	`MEDIUM2`	`TRANSMITTER`	transmission de l'acquittement
`SACKe`	`MEDIUM2`	`TRANSMITTER`	perte de l'acquittement

L'émetteur et le récepteur fonctionnent de manière complètement asynchrone. Il en est de même pour les deux media. On peut donc spécifier l'architecture du protocole par le programme LOTOS ci-dessous, dans lequel les définitions des processus `TRANSMITTER`, `RECEIVER`, `MEDIUM1` et `MEDIUM2` ne sont pas explicitées :

```
specification ALTERNATING_BIT_PROTOCOL [PUT, GET] : noexit behaviour
   hide SDT, RDT, RDTe, RACK, SACK, SACKe in
      (
         (
         TRANSMITTER [PUT, SDT, SACK, SACKe] (0)
         |||
         RECEIVER [GET, RDT, RDTe, RACK] (0)
         )
      |[SDT, RDT, RDTe, RACK, SACK, SACKe]|
         (
         MEDIUM1 [SDT, RDT, RDTe]
         |||
         MEDIUM2 [RACK, SACK, SACKe]
         )
      )
where
   type BIT is
      sorts BIT
      opns 0 : -> BIT
           1 : -> BIT
           not : BIT -> BIT
   endtype
endspec
```

Remarque C-1
Le paramètre effectif `0` des processus `TRANSMITTER` et `RECEIVER` sert à initialiser la valeur du bit de contrôle ; par convention le bit de contrôle du premier message est égal à 0. ■

C.4 Spécification du medium des messages

En recevant un message M avec un bit de contrôle égal à B, le medium n$^{\text{o}}$ 1 peut réagir de trois façons différentes :

- transmettre correctement le message et son bit de contrôle. En aucun cas le medium ne peut changer la valeur du bit de contrôle
- perdre le message et envoyer une indication de perte à l'entité réceptrice
- perdre silencieusement le message

```
process MEDIUM1 [SDT, RDT, RDTe] : noexit :=
   SDT ?M:MSG ?B:BIT;   (* reception d'un message *)
      (
      RDT !M !B;        (* transmission correcte *)
         MEDIUM1 [SDT, RDT, RDTe]
      []
      RDTe;             (* perte avec indication *)
         MEDIUM1 [SDT, RDT, RDTe]
      []
      i;                (* perte silencieuse *)
         MEDIUM1 [SDT, RDT, RDTe]
      )
endproc
```

C.5 Spécification du medium des acquittements

Le fonctionnement du medium n$^{\text{o}}$ 2 est analogue à celui du medium n$^{\text{o}}$ 1. La seule différence réside dans les noms de signaux et dans le fait que les acquittements, contrairement aux messages, ne portent pas d'information autre que le bit de contrôle.

```
process MEDIUM2 [RACK, SACK, SACKe] : noexit :=
   RACK ?B:BIT;   (* reception d'un acquittement *)
      (
      SACK !B;    (* transmission correcte *)
         MEDIUM2 [RACK, SACK, SACKe]
      []
      SACKe;      (* perte avec indication *)
         MEDIUM2 [RACK, SACK, SACKe]
      []
      i;          (* perte silencieuse *)
         MEDIUM2 [RACK, SACK, SACKe]
      )
endproc
```

C.6 Spécification de l'émetteur

L'entité émettrice acquiert un message via `PUT` et le transmet au medium n$^{\text{o}}$ 1 après lui avoir ajouté la valeur courante B du bit de contrôle. Si elle reçoit en réponse un acquittement avec un bit de contrôle B la transmission a réussi, sinon il faut réémettre le message. Il y a 3 causes possibles de réémission :

- l'émetteur a reçu un acquittement ayant $(\neg B)$ comme bit de contrôle

- l'émetteur a reçu une indication de perte d'acquittement `SACKe`
- l'émetteur peut réémettre spontanément le message afin d'éviter le blocage dans le cas où le medium n° 1 (*resp.* n° 2) aurait perdu silencieusement un message (*resp.* un acquittement). En réalité, cette réémission n'a lieu que si une certaine contrainte de délai (*timeout*) est vérifiée (*cf.* [RSV86] et [RV87]) mais, LOTOS ne permettant pas d'exprimer le délai, on le modélise par un événement silencieux "**i**"

Remarque C-2
Pour décrire l'entité émettrice on a choisi une structure de contrôle paramétrée par des données (essentiellement la valeur B du bit de contrôle). D'autres solutions auraient été également viables, en particulier une modélisation sans données, avec une structure de contrôle complètement développée. Il s'agit de la dualité classique entre le contrôle et les données. ■

```
process TRANSMITTER [PUT, SDT, SACK, SACKe] (B:BIT) : noexit :=
   PUT ?M:MSG;  (* acquisition d'un message *)
      TRANSMIT [PUT, SDT, SACK, SACKe] (B, M)
where
   process TRANSMIT [PUT, SDT, SACK, SACKe] (B:BIT, M:MSG) : noexit :=
      SDT !M !B;          (* emission du message *)
         (
         SACK !B;         (* bit de controle correct *)
            TRANSMITTER [PUT, SDT, SACK, SACKe] (not (B))
         []
         SACK !(not (B));  (* bit de controle incorrect => reemission *)
            TRANSMIT [PUT, SDT, SACK, SACKe] (B, M)
         []
         SACKe;           (* indication de perte => reemission *)
            TRANSMIT [PUT, SDT, SACK, SACKe] (B, M)
         []
         i;               (* timeout => reemission *)
            TRANSMIT [PUT, SDT, SACK, SACKe] (B, M)
         )
   endproc
endproc
```

C.7 Spécification du récepteur

Remarque C-3
Le comportement de l'entité réceptrice est parfaitement symétrique de celui de l'entité paire, bien que la modélisation "contrôle+données" choisie puisse faire croire à la disymétrie. ■

Lorsqu'elle reçoit un message avec un bit de contrôle B correct, l'entité réceptrice délivre le message via `GET` et renvoie un acquittement avec un bit de contrôle égal à B. Dans les autres cas, elle renvoie un acquittement incorrect (ayant $(\neg B)$ comme bit de contrôle) ; ces cas sont au nombre de trois :

- le récepteur a reçu un message ayant $(\neg B)$ comme bit de contrôle
- le récepteur a reçu une indication de perte de message `RDTe`
- le récepteur peut émettre spontanément un acquittement invalide afin d'éviter le blocage dans le cas où le medium n° 1 (*resp.* n° 2) aurait perdu silencieusement un message (*resp.* un acquittement). Comme expliqué plus haut, il s'agit d'un *timeout* que l'on modélise par un événement silencieux "**i**"

```
process RECEIVER [GET, RDT, RDTe, RACK] (B:BIT) : noexit :=
   RDT ?M:MSG !B;          (* bit de controle correct *)
      GET !M;              (* livraison du message *)
         RACK !B;          (* envoi d'un acquittement correct *)
            RECEIVER [GET, RDT, RDTe, RACK] (not (B))
   []
   RDT ?M:MSG !(not (B));  (* bit de controle incorrect => *)
      RACK !(not (B));     (* envoi d'un acquittement incorrect *)
         RECEIVER [GET, RDT, RDTe, RACK] (B)
   []
   RDTe;                   (* indication de perte => *)
      RACK !(not (B));     (* envoi d'un acquittement incorrect *)
         RECEIVER [GET, RDT, RDTe, RACK] (B)
   []
   i;                      (* timeout => *)
      RACK !(not (B));     (* envoi d'un acquittement incorrect *)
         RECEIVER [GET, RDT, RDTe, RACK] (B)
endproc
```

C.8 Validation

A l'aide de la version 3.1 de CÆSAR on a produit, pour diverses valeurs de `N`, les réseaux et les graphes qui correspondent aux descriptions en LOTOS du service et du protocole.

Le réseau du service comporte 2 places, 2 transitions et 1 variable ; celui du protocole comporte 17 places, 21 transitions et 7 variables. Ces valeurs sont indépendantes de `N`.

Les deux tableaux suivants indiquent les résultats obtenus respectivement pour le service et le protocole. Dans chaque cas on indique la taille du graphe (nombre d'états et nombre d'arcs) ainsi que le temps total mis par CÆSAR pour produire ce graphe ; cette durée est exprimée sous la forme d'un couple *max–min* où :

- *max* est mesuré sur une station de travail SUN 3/50 avec 4Mo de mémoire principale, pour un seul utilisateur qui a ouvert 3 fenêtres sous X-windows version 10. On relève ainsi les performances de CÆSAR dans un environnement "normal"

- *min* est mesuré sur une station de travail SUN 3/60 avec 8Mo de mémoire principale, pour un seul utilisateur qui exécute seulement CÆSAR

Les durées *max* et *min* sont données sous la forme *minutes*:*secondes* ; il s'agit de temps utilisateur et non de temps virtuel. La dernière colonne du table contient les *débits*, c'est-à-dire le nombre d'états produits par seconde, calculés pour *max* et *min*.

N	états	arcs	temps	débit
5	11	35	0:27–0:22	0.4–0.5
10	21	120	0:28–0:18	0.8–1.2
15	31	255	0:30–0:17	1.0–1.8
20	41	440	0:30–0:17	1.4–2.4
25	51	675	0:34–0:19	1.5–2.7
30	61	960	0:32–0:17	1.9–3.5
35	71	1295	0:38–0:17	1.9–4.2
40	81	1680	0:29–0:15	2.8–5.4
45	91	2115	0:30–0:16	3.0–6
50	101	2600	0:31–0:19	3.3–5.3
70	141	5040	0:33–0:22	4.2–6.4

N	états	arcs	temps	débit
5	3 576	11 317	1:09–0:43	52–83
10	12 346	39 527	1:33–1:37	133–127
15	26 316	84 637	2:28–2:02	177–130
20	45 486	146 647	4:01–2:56	189–258
25	69 856	225 557	5:35–4:05	209–285
30	99 426	321 367	8:13–4:51	201–342
35	134 196	434 077	11:08–6:32	200–342
40	174 166	563 687	23:39–9:11	122–316
45	219 336	710 197	159:49–11:11	23–326
50	269 706	873 607	?–14:48	?–304
70	523 186	1 696 247	?–80:53	?–108

Pour `N` valant 5, 10 et 15, l'utilisation du logiciel ALDEBARAN [Fer88] a permis de montrer que le graphe du protocole était observationnellement équivalent à celui du service.

Pour des valeurs élevées de `N` la taille du graphe croît rapidement et avec elle le temps nécessaire pour comparer le protocole au service. Dans de telles situations, le recours aux logiques temporelles s'impose. Voici un exemple de propriétés, exprimées dans la logique R$\star$ICO, qui devraient être vérifiées par le graphe correspondant au protocole du bit alterné :

- il n'y a pas de blocage :

  not stop

- il est impossible d'avoir deux actions `PUT` successives :

  never `(PUT ?M1:MSG .` **i*** `. PUT ?M2:MSG)`

- il est impossible d'avoir deux actions `GET` successives :

  never `(GET ?M1:MSG .` **i*** `. GET ?M2:MSG)`

- il est impossible qu'une action `PUT !M1` soit suivie d'une action `GET !M2` si la valeur de `M2` est différente de celle de `M1` :

  all `M1:MSG` **in never** `(PUT !M1:MSG .` **i*** `. GET ?M2:MSG [not (M1 = M2)])`

- toute action `PUT !M` est inévitablement[46] suivie d'une action `GET !M` :

all `M:MSG` **in** `(PUT !M =>` **finev** `(`**i**`, GET !M))`

C.9 Notes bibliographiques

Le protocole du *bit alterné* a longtemps eu la faveur des concepteurs de systèmes de vérification automatique, principalement en raison de la petite taille de son graphe (quelques dizaines d'états). La version en LOTOS proposée ici permet, lorsque l'on donne à `N` une valeur élevée, d'obtenir des graphes de grande taille, démontrant ainsi les capacités réelles de CÆSAR.

La modélisation proposée s'inspire principalement de deux sources :

- [RSV86] et [RV87] : il s'agit d'une description du protocole, dans le formalisme ATP, qui prend en compte les contraintes de délai. Le medium entre les deux entités est défini sous la forme de deux processus identiques fonctionnant en parallèle. Les messages et les acquittements peuvent être perdus sans que le medium le signale

- [Lon87] : cette description représente le medium comme un processus unique. Les messages et les acquittements peuvent être perdus, mais le medium doit alors envoyer au destinataire une indication de perte de message ou d'acquittement

Par rapport à [RSV86], [RV87] et [Lon87], la transmission des messages (considérés comme des valeurs entières) est effectivement modélisée. Les deux cas d'erreur de transmission ont été modélisés : perte silencieuse et perte signalée au destinataire.

Enfin les contraintes temporelles n'ont pas été conservées parce qu'elles ne peuvent pas s'exprimer simplement en LOTOS. Ceci appelle quelques remarques :

- lorsqu'on remplace une contrainte de délai par un événement "**i**", le nouveau comportement est un sur-ensemble de l'ancien

- toutefois, en règle générale, les "bons algorithmes" ne dépendent pas des valeurs des délais et conservent leurs "bonnes propriétés" (*speed independance*)

- autrement dit l'ajustage des délais permet d'améliorer les performances du système mais il ne doit pas servir à rendre correct un algorithme qui serait faux pour des valeurs de délais quelconques

- toutefois, la modélisation d'un délai par un événement "**i**" peut modifier le comportement du système. Dans le cas du bit alterné, si l'on choisit judicieusement les valeurs des délais ([RSV86], [RV87]), à chaque instant, au plus un message ou acquittement circule sur l'ensemble des deux media (la liaison est *half-duplex*). Si les délais sont mal choisis ou modélisés de manière asynchrone, plusieurs messages ou acquittements peuvent transiter simultanément, dans les deux sens (la liaison devient *full-duplex*)

[46] sous l'hypothèse d'équité

Annexe D

Application 2 : convolution systolique

D.1 Produit de convolution

Soient n nombres $w_1, \ldots\ w_n$ appelés *poids* (*weights*). Soit (x_i) une suite de nombres. On appelle *produit de convolution* de la suite (x_i) par les poids $w_1, \ldots\ w_n$ la suite (y_i) définie par :

$$y_i = \sum_{j=1}^{n} w_j x_{i+j-1}$$

Remarque D-1
D'autres définitions du produit de convolution sont possibles, moyennant un renommage des coefficients $w_1, \ldots\ w_n$ qui sont en nombre fini. ■

D.2 Réseau systolique asynchrone

On cherche à construire un programme LOTOS calculant le produit de convolution. Le système possède une *porte d'entrée*, sur laquelle il reçoit successivement les valeurs de la suite (x_i), et une *porte de sortie*, sur laquelle il émet les valeurs (y_i) correspondantes.

Ce système doit être basé sur les principes systoliques : il est constitué par la juxtaposition de cellules identiques (autant de cellules que de poids w_i) exécutant chacune le même algorithme. Cet algorithme est cyclique et n'utilise que des opérations simples (addition, multiplication).

La plupart du temps, les algorithmes systoliques sont décrits pour un modèle synchrone utilisant une notion de temps global, déterminé par une horloge unique. A chaque top d'horloge, chaque cellule communique avec les cellules voisines.

Le modèle synchrone s'impose tout naturellement lorsque l'on envisage une implémentation en technologie VLSI ou autre. En revanche il ne convient pas pour d'autres types d'implémentations (réseaux de micro-processeurs, par exemple) ni pour les réseaux comprenant plusieurs sortes de cellules, dans lesquels la différence de vitesse entre les cellules interdit l'emploi d'une horloge commune.

Dans un modèle asynchrone, les cellules communiquent par rendez-vous. Chacune cellule ne peut pas émettre ou recevoir simultanément sur deux portes distinctes. En outre, en LOTOS, deux rendez-vous distincts n'ont jamais lieu simultanément.

Le comportement synchrone peut être considéré comme une abstraction du comportement asynchrone, obtenue en considérant que plusieurs rendez-vous ont lieu simultanément.

Il est possible de mettre sous forme asynchrone la plupart des algorithmes systoliques : la structure du réseau est conservée, ainsi que le comportement des cellules. En revanche la spécification asynchrone impose des contraintes supplémentaires sur l'ordonnancement des rendez-vous afin d'éviter les interblocages.

Remarque D-2
Pour permettre l'initialisation du système et le chargement des n premières valeurs $x_1, \dots x_n$, il faut établir une distinction entre le *comportement transitoire* d'une cellule à l'initialisation du système et le *comportement permanent* qui lui succède ; typiquement le régime transitoire prend fin avec la réception de la $n^{\text{ième}}$ valeur x_n. ■

Dans la suite on présente quatre architectures systoliques qui calculent la convolution. Ces quatre schemas sont connus sous les appellations suivantes : B1, F, W1 et W2.

Remarque D-3
On s'est restreint aux exemples dans lesquels chaque cellule est associée à un poids constant (*weights stay*) à l'exclusion des schémas dans lesquels les poids se déplacent : B2, R1 et R2. Ce choix n'est pas dicté par une limitation quelconque de LOTOS ou de CÆSAR. ■

D.3 Environnement

Le comportement du réseau systolique est paramétré par la suite des valeurs (x_i) qu'il reçoit en entrée : il ne s'agit pas d'un système fermé.

Or CÆSAR ne peut traiter que des systèmes fermés puisqu'il est généralement impossible de construire le graphe d'un comportement lorsque l'on ne connaît pas les valeurs qu'ont les variables de ce comportement. Cette restriction interdit la vérification du réseau pris isolément.

Une première solution consiste à connecter l'entrée du réseau à un générateur qui fournit une suite fixée de valeurs (x_i). Pour que le graphe correspondant au comportement de l'ensemble *réseau+générateur* soit fini, il faut que la suite (x_i) fournie par le générateur soit finie ou périodique à partir d'un certain rang.

Dans le cas présent, il serait toutefois préférable d'effectuer une vérification formelle du réseau paramétré et ne pas se contenter de prouver que le réseau fonctionne correctement lorsqu'on l'instancie par une suite (x_i) fixée.

Une seconde solution solution utilise une technique d'*évaluation symbolique*. Comme dans la première solution, l'entrée du réseau systolique est reliée à un générateur qui énumère une suite finie ou périodique (x_i).

La différence provient du fait que les éléments x_i et w_j ne sont plus des valeurs numériques mais des noms symboliques de variables. Les opérateurs d'addition et de multiplication opèrent alors sur des expressions symboliques et non plus sur des nombres.

Exemple D-1
Le résultat de "w_1" `*` "x_1" est l'expression symbolique "$w_1 x_1$" ; de même que le résultat de "$w_1 x_1$" `+` "$w_2 x_2$" est l'expression symbolique "$w_1 x_1 + w_2 x_2$". ■

Les expressions symboliques sont décrites par le type abstrait `EXP`. Les éléments des suites $(x_1, \dots x_m)$

et $(w_1, \ldots\ w_n)$ sont dénotés par des fonctions d'arité nulle `X1, ... Xm` et `W1, ... Wn`. L'addition et la multiplication sont représentées par les opérateurs binaires `+` et `*`. On note `0` l'élément neutre pour l'addition ; cette propriété suffit pour simplifier les expressions symboliques.

```
type EXP is
   sorts EXP
   opns 0 : -> EXP
        X1, ... Xm : -> EXP
        W1, ... Wn : -> EXP
        _+_, _*_ : EXP, EXP -> EXP
   eqns
      forall X:EXP ofsort EXP
         X + 0 = X;
         0 + X = X
endtype
```

On utilise aussi un type entier noté `NATURAL`, comportant une sorte `NAT`, les notations d'entiers de 0 à n et les opérations usuelles de soustraction et de comparaison :

```
type NATURAL is BOOLEAN
   sorts NAT
   opns 0, 1, ... n : -> NAT
        _-_ : NAT, NAT -> NAT
        _eq_, _gt_ : NAT, NAT -> BOOL
endtype
```

Le réseau systolique est dénoté par le processus `ARRAY [X, Y]` où `X` est la porte d'entrée et `Y` la porte de sortie. On note n le nombre de cellules et (w_j) leurs poids respectifs. Diverses définitions du processus `ARRAY` seront proposées dans les sections suivantes.

Le générateur fournit une suite finie (x_i) pour i variant entre 1 et un entier m. Il est décrit par le processus `GENERATOR`. On donne également la définition d'un processus, noté `ZERO`, qui sera utilisé par la suite.

```
specification SYSTOLIC_CONVOLUTION [Y] : noexit behaviour
   hide X in
      (
      GENERATOR [X]
      |[X]|
      ARRAY [X, Y] (W1, ... Wn)
      )
where
   process GENERATOR [X] : noexit :=
      X !X1;
         X !X2;
            ...
               X !Xm;
                  stop
   endproc

   process ZERO [Y] : noexit :=
      Y !(0 of EXP);
         ZERO [Y]
   endproc
endspec
```

Remarque D-4

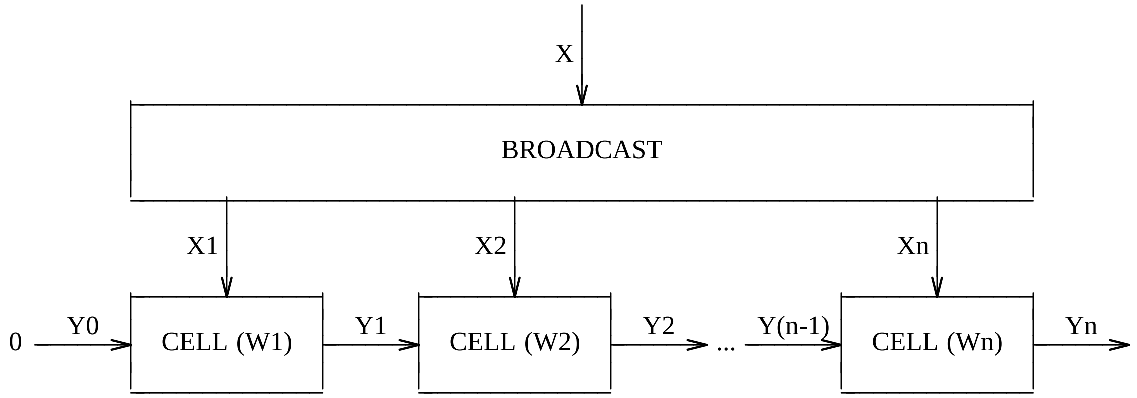
Par la suite, on donnera aussi les noms X1, ... Xn à des portes, sachant que LOTOS le permet et qu'il n'y a aucune confusion possible entre portes et opérations ■

D.4 Architecture B1

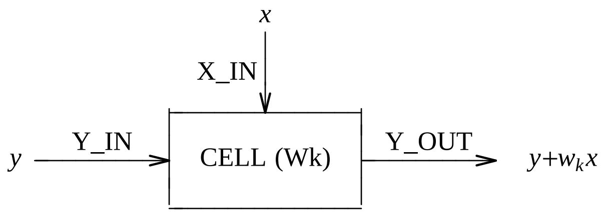
Le schema B1 (*broadcast*) se compose de n processus `CELL` disposés en pipe-line. Le flux de données entre les cellules est constitué de sommes partielles ; la $k^{\text{ième}}$ cellule transmet à la $(k+1)^{\text{ième}}$ des expressions de la forme :

$$y_i^k = \sum_{j=1}^{k} w_j x_{i+j-1}$$

La valeur courante de x_i est reçue par un processus `BROADCAST` et diffusée ensuite à toutes les cellules. Il ne s'agit pas à proprement parler d'une architecture systolique pure.



La $k^{\text{ième}}$ cellule a pour poids w_k ; elle possède deux portes d'entrée `X_IN` et `Y_IN` et une porte de sortie `Y_OUT`. En comportement permanent, elle lit une valeur x sur `X_IN` puis une valeur y sur `Y_IN` et émet $y + w_k x$ sur `Y_OUT`. En comportement transitoire elle lit (sans les traiter) $(k-1)$ valeurs sur la porte `X_IN`.



```
process ARRAY [X, Yn] (W1, ... Wn:EXP) : noexit :=
   hide X1, ... Xn in
      (
      BROADCAST [X, X1, ... Xn]
      |[X1, ... Xn]|
         (
         hide Y0, ... Y(n-1) in
            (
            ZERO [Y0]
            |[Y0]|
            CELL [X1, Y0, Y1] (W1, 1)
            |[Y1]|
            CELL [X2, Y1, Y2] (W2, 2)
            |[Y2]|
            ...
            |[Y(n-1)]|
            CELL [Xn, Y(n-1), Yn] (Wn, n)
            )
         )
      )
where
   process CELL [X_IN, Y_IN, Y_OUT] (W:EXP, K:NAT) : noexit :=
      [K gt 1] ->
         X_IN ?X:EXP;
            CELL [X_IN, Y_IN, Y_OUT] (W, K - 1)
      []
      [K eq 1] ->
         X_IN ?X:EXP;
            Y_IN ?Y:EXP;
               Y_OUT !(Y + (W * X));
                  CELL [X_IN, Y_IN, Y_OUT] (W, 1)
   endproc

   process BROADCAST [IN0, OUT1, ... OUTn] : noexit :=
      IN0 ?X:EXP;
         OUTn !X;
            OUT(n-1) !X;
               ...
                  OUT1 !X;
                     BROADCAST [IN0, OUT1, ... OUTn]
   endproc
endproc
```

Le tableau suivant aide à mieux comprendre le fonctionnement du réseau systolique. Chaque ligne correspond à un instant donné et les lignes sont rangées chronologiquement.

La colonne de gauche contient la séquence des événements qui ont lieu dans le système. Chaque événement est constitué d'un nom de porte et de la valeur échangée au moment du rendez-vous. Les événements visibles à l'extérieur du réseau (entrée d'une valeur x_i ou sortie d'une valeur y_i) sont marqués d'une étoile.

Les autres colonnes décrivent l'état courant de chaque processus. L'état d'attente pour émettre (*resp.* pour recevoir) une valeur sur une porte `G` est noté "`G !`" (*resp.* "`G ?`"). Lorsque l'état d'un processus n'est pas modifié par l'événement courant, on met un guillemet dans la case correspondante.

Le tableau décrit le régime transitoire et l'établissement du régime permanent. Les parties non hachurées mettent en évidence le premier cycle du comportement de chaque processus, une fois atteint le régime permanent.

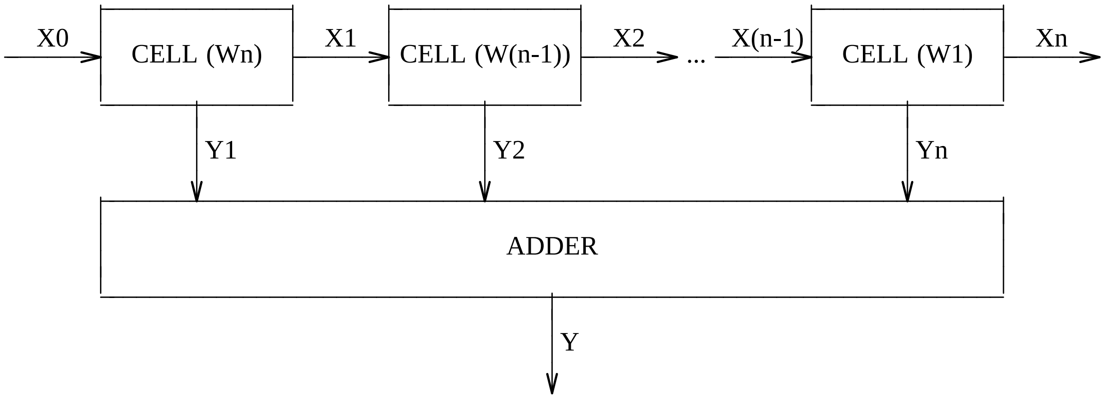
événement	*CELL (W1)*	*CELL (W2)*	*CELL (W3)*	*BROADCAST*
initialement :	X1 ? $(K=1)$	X2 ? $(K=2)$	X3 ? $(K=3)$	X ?
X $!x_1$ *	"	"	"	X3 !
X3 $!x_1$	"	"	X3 ? $(K=2)$	X2 !
X2 $!x_1$	"	X2 ? $(K=1)$	"	X1 !
X1 $!x_1$	Y0 ?	"	"	X ?
X $!x_2$ *	"	"	"	X3 !
X3 $!x_2$	"	"	X3 ? $(K=1)$	X2 !
X2 $!x_2$	"	Y1 ?	"	X1 !
Y0 $!0$	Y1 !	"	"	"
Y1 $!w_1x_1$	X1 ?	Y2 !	"	"
X1 $!x_2$	Y0 ?	"	"	X ?
X $!x_3$ *	"	"	"	X3 !
X3 $!x_3$	"	"	Y2 ?	X2 !
Y2 $!w_1x_1+w_2x_2$	"	X2 ?	Y3 !	"
Y3 $!w_1x_1+w_2x_2+w_3x_3$ *	"	"	X3 ?	"
X2 $!x_3$	"	Y1 ?	"	X1 !
Y0 $!0$	Y1 !	"	"	"
Y1 $!w_1x_2$	X1 ?	Y2 !	"	"
X1 $!x_3$	Y0 ?	"	"	X ?
X $!x_4$ *	"	"	"	X3 !
X3 $!x_4$	"	"	Y2 ?	X2 !
Y2 $!w_1x_2+w_2x_3$	"	X2 ?	Y3 !	"
Y3 $!w_1x_2+w_2x_3+w_3x_4$ *	"	"	X3 ?	"

D.5 Architecture F

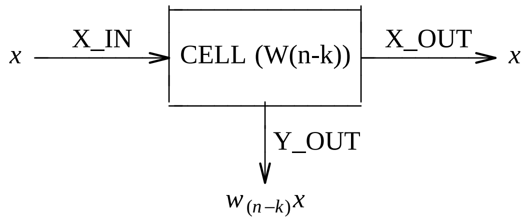
Le schema F (*fan-in*) se compose de n processus CELL disposés en pipe-line. Le flux de données entre les cellules est constitué des valeurs successives de x_i ; la $k^{\text{ième}}$ cellule transmet à la $(k+1)^{\text{ième}}$ des valeurs de la forme :

$$x_i^k = x_{i+n-k-1}$$

Chaque cellule effectue un produit partiel $w_{n-k}x_{i+n-k-1}$; ces valeurs sont recueillies par un processus ADDER qui en calcule la somme. Il ne s'agit pas à proprement parler d'une architecture systolique pure.



La $k^{\text{ième}}$ cellule a pour poids w_{n-k} ; elle possède une porte d'entrée `X_IN` et deux portes de sortie `X_OUT` et `Y_OUT`. En comportement permanent, elle lit une valeur x sur `X_IN` puis envoie successivement $w_{n-k}x$ sur `Y_OUT` et x sur `X_OUT`. En comportement transitoire elle lit (sans les traiter) $n-k$ valeurs sur la porte `X_IN`.




```
process ARRAY [X0, Y] (W1, ... Wn:EXP) : noexit :=
   hide Y1, ... Yn in
      (
      hide X1, ... Xn in
         (
         CELL [X0, Y1, X1] (Wn, n)
         |[X1]|
         CELL [X1, Y2, X2] (W(n-1), (n-1))
         |[X2]|
         ...
         |[X(n-1)]|
         CELL [X(n-1), Yn, Xn] (W1, 1)
         )
      |[Y1, ... Yn]|
      ADDER [Y1, ... Yn, Y]
      )
where
   process CELL [X_IN, Y_OUT, X_OUT] (W:EXP, K:NAT) : noexit :=
      [K gt 1] ->
         X_IN ?X:EXP;
            X_OUT !X;
               CELL [X_IN, Y_OUT, X_OUT] (W, K - 1)
      []
      [K eq 1] ->
         X_IN ?X:EXP;
            Y_OUT !(W * X);
               X_OUT !X;
                  CELL [X_IN, Y_OUT, X_OUT] (W, 1)
   endproc

   process ADDER [IN1, ... INn, OUT] : noexit :=
      IN1 ?Y1:EXP;
         IN2 ?Y2:EXP;
            ...
               INn ?Yn:EXP;
                  OUT !(Y1 + Y2 + ... Yn);
                     ADDER [IN1, ... INn, OUT]
   endproc
endproc
```

événement	*CELL (W3)*	*CELL (W2)*	*CELL (W1)*	*ADDER*
initialement :	X0 ? $(K = 3)$	X1 ? $(K = 2)$	X2 ? $(K = 1)$	Y1 ?
X0 !x_1 *	X1 !	"	"	"
X1 !x_1	X0 ? $(K = 2)$	X2 !	"	"
X2 !x_1	"	X1 ? $(K = 1)$	Y3 !	"
X0 !x_2 *	X1 !	"	"	"
X1 !x_2	X0 ? $(K = 1)$	Y2 !	"	"
X0 !x_3 *	Y1 !	"	"	"
Y1 !w_3x_3	X1 !	"	"	Y2 ?
Y2 !w_2x_2	"	X2 !	"	Y3 ?
Y3 !w_1x_1	"	"	X3 !	Y !
Y !$w_3x_3 + w_2x_2 + w_1x_1$ *	"	"	"	Y1 ?
X3 !x_1	"	"	X2 ?	"
X2 !x_2	"	X1 ?	Y3 !	"
X1 !x_3	X0 ?	Y2 !	"	"
X0 !x_4 *	Y1 !	"	"	"
Y1 !w_3x_4	X1 !	"	"	Y2 ?
Y2 !w_2x_3	"	X2 !	"	Y3 ?
Y3 !w_1x_2	"	"	X3 !	Y !
Y !$w_3x_4 + w_2x_3 + w_1x_2$ *	"	"	"	Y1 ?
X3 !x_2	"	"	X2 ?	"
X2 !x_3	"	X1 ?	"	"
X1 !x_4	X0 ?	"	"	"

D.6 Architecture W1

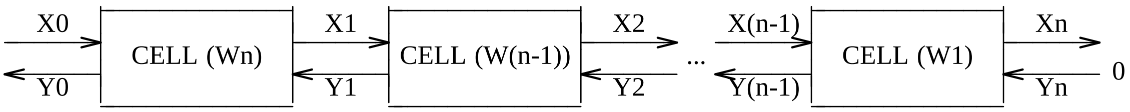
Le schema W1 (*pure systolic — weights stay*) se compose de n processus CELL disposés en double pipe-line. Deux flux de données circulent dans des directions opposées :

- les valeurs successives de x_i circulent dans un sens ; la $k^{\text{ième}}$ cellule transmet à la $(k+1)^{\text{ième}}$ des valeurs de la forme :

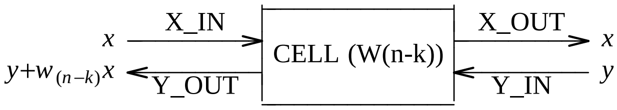
$$x_i^k = x_{i+n-k-1}$$

- les sommes partielles circulent dans le sens inverse ; la $k^{\text{ième}}$ cellule transmet à la $(k-1)^{\text{ième}}$ des expressions de la forme :

$$y_i^k = \sum_{j=k}^{n} w_j x_{i+j-1}$$



La $k^{\text{ième}}$ cellule a pour poids w_{n-k} ; elle possède deux portes d'entrée X_IN et Y_IN et deux portes de sortie X_OUT et Y_OUT. En comportement permanent, elle lit successivement une valeur x sur X_IN puis une valeur y sur Y_IN avant d'envoyer x sur X_OUT puis $y + w_{n-k}x$ sur Y_OUT. En comportement transitoire elle lit $n - k$ valeurs x sur la porte X_IN qu'elle retransmet immédiatement sur la porte X_OUT.



```
process ARRAY [X0, Y0] (W1, ... Wn:EXP) : noexit :=
   hide X1, ... Xn, Y1, ... Yn in
      (
      CELL [X0, Y0, X1, Y1] (Wn, n)
      |[X1, Y1]|
      CELL [X1, Y1, X2, Y2] (W(n-1), (n-1))
      |[X2, Y2]|
      ...
      |[X(n-1), Y(n-1)]|
      CELL [X(n-1), Y(n-1), Xn, Yn] (W1, 1)
      |[Yn]|
      ZERO [Yn]
      )
where
   process CELL [X_IN, Y_OUT, X_OUT, Y_IN] (W:EXP, K:NAT) : noexit :=
      [K gt 1] ->
         X_IN ?X:EXP;
            X_OUT !X;
               CELL [X_IN, Y_OUT, X_OUT, Y_IN] (W, K - 1)
      []
      [K eq 1] ->
         X_IN ?X:EXP;
            Y_IN ?Y:EXP;
               X_OUT !X;
                  Y_OUT !(Y + (W * X));
                     CELL [X_IN, Y_OUT, X_OUT, Y_IN] (W, 1)
   endproc
endproc
```

événement		*CELL (W3)*	*CELL (W2)*	*CELL (W1)*
initialement :		X0 ? $(K=3)$	X1 ? $(K=2)$	X2 ? $(K=1)$
X0 !x_1	*	X1 !	"	"
X1 !x_1		X0 ? $(K=2)$	X2 !	"
X2 !x_1		"	X1 ? $(K=1)$	Y3 ?
X0 !x_2	*	X1 !	"	"
X1 !x_2		X0 ? $(K=1)$	Y2 ?	"
X0 !x_3	*	Y1 ?	"	"
Y3 !0		"	"	X3 !
X3 !x_1		"	"	Y2 !
Y2 !w_1x_1		"	X2 !	X2 ?
X2 !x_2		"	Y1 !	Y3 ?
Y1 !$w_1x_1+w_2x_2$		X1 !	X1 ?	"
X1 !x_3		Y0 !	Y2 ?	"
Y0 !$w_1x_1+w_2x_1+w_3x_3$	*	X0 ?	"	"
X0 !x_4	*	Y1 ?	"	"
Y3 !0		"	"	X3 !
X3 !x_2		"	"	Y2 !
Y2 !w_1x_2		"	X2 !	X2 ?
X2 !x_3		"	Y1 !	Y3 ?
Y1 !$w_1x_2+w_2x_3$		X1 !	X1 ?	"
X1 !x_4		Y0 !	Y2 ?	"
Y0 !$w_1x_2+w_2x_3+w_3x_4$	*	X0 ?	"	"

Remarque D-5
Dans le modèle synchrone, la suite de valeurs qu'il faut fournir en entrée du réseau n'est pas la suite

(x_i) mais la suite (x'_i) définie par :

$$\begin{cases} x'_{2i} & = x_i \\ x'_{2i+1} & = 0 \end{cases}$$

La suite (x'_i) est deux fois plus lente que la suite (x_i). Ce n'est pas le cas dans le modèle asynchrone : il suffit d'alimenter le réseau avec la suite (x_i). ■

D.7 Architecture W2

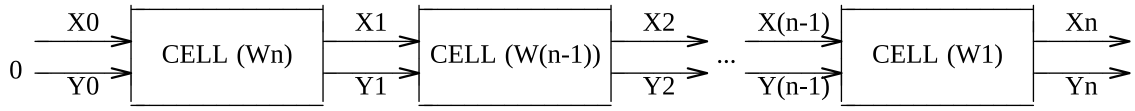
Le schema W2 (*pure systolic — weights stay*) se compose de n processus `CELL` disposés en double pipe-line. Deux flux de données circulent dans la même direction :

- les valeurs successives de x_i : la $k^{\text{ième}}$ cellule transmet à la $(k+1)^{\text{ième}}$ des valeurs de la forme :

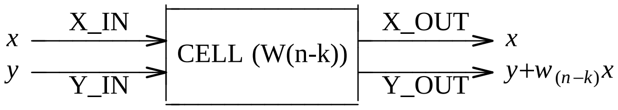
$$x_i^k = x_{i+n-k-1}$$

- les sommes partielles : la $k^{\text{ième}}$ cellule transmet à la $(k+1)^{\text{ième}}$ des expressions de la forme :

$$y_i^k = \sum_{j=k}^{n} w_j x_{i+n-k-1}$$



La $k^{\text{ième}}$ cellule a pour poids w_{n-k} ; elle possède deux portes d'entrée `X_IN` et `Y_IN` et deux portes de sortie `X_OUT` et `Y_OUT`. En comportement permanent, elle lit successivement une valeur x sur `X_IN` puis une valeur y sur `Y_IN` avant d'envoyer $y + w_{n-k}x$ sur `Y_OUT` puis x sur `X_OUT`. En comportement transitoire elle lit $n - k$ valeurs x sur la porte `X_IN` qu'elle retransmet immédiatement sur la porte `X_OUT`.



```
process ARRAY [X0, Yn] (W1, ... Wn:EXP) : noexit :=
   hide X1, ... Xn, Y0, ... Y(n-1) in
      (
      ZERO [Y0]
      |[Y0]|
      CELL [X0, Y0, X1, Y1] (Wn, n)
      |[X1, Y1]|
      CELL [X1, Y1, X2, Y2] (W(n-1), (n-1))
      |[X2, Y2]|
      ...
      |[X(n-1), Y(n-1)]|
      CELL [X(n-1), Y(n-1), Xn, Yn] (W1, 1)
      )
where
   process CELL [X_IN, Y_IN, X_OUT, Y_OUT] (W:EXP, K:NAT) : noexit :=
      [K gt 1] ->
         X_IN ?X:EXP;
            X_OUT !X;
               CELL [X_IN, Y_IN, X_OUT, Y_OUT] (W, K - 1)
      []
      [K eq 1] ->
         X_IN ?X:EXP;
            Y_IN ?Y:EXP;
               Y_OUT !(Y + (W * X));
                  X_OUT !X;
                     CELL [X_IN, Y_IN, X_OUT, Y_OUT] (W, 1)
   endproc
endproc
```

événement		*CELL (W3)*	*CELL (W2)*	*CELL (W1)*
initialement :		X0 ? $(K = 3)$	X1 ? $(K = 2)$	X2 ?
X0 $!x_1$	*	X1 !	"	"
X1 $!x_1$		X0 ? $(K = 2)$	X2 ! $(K = 2)$	"
X2 $!x_1$		"	X1 ?	Y2 ?
X0 $!x_2$	*	X1 ! $(K = 2)$	"	"
X1 $!x_2$		X0 ?	Y1 ?	"
X0 $!x_3$	*	Y0 ?	"	"
Y0 $!0$		Y1 !	"	"
Y1 $!w_3x_3$		X1 !	Y2 !	"
Y2 $!w_3x_3 + w_2x_2$		"	X2 !	Y3 !
Y3 $!w_3x_3 + w_2x_2 + w_1x_1$	*	"	"	X3 !
X3 $!x_1$		"	"	X2 ?
X2 $!x_2$		"	X1 ?	Y2 ?
X1 $!x_3$		X0 ?	Y1 ?	"
X0 $!x_4$	*	Y0 ?	"	"
Y0 $!0$		Y1 !	"	"
Y1 $!w_3x_4$		X1 !	Y2 !	"
Y2 $!w_3x_4 + w_2x_3$		"	X2 !	Y3 !
Y3 $!w_3x_4 + w_2x_3 + w_1x_2$	*	"	"	X3 !
X3 $!x_2$		"	"	X2 ?
X2 $!x_3$		"	X1 ?	"
X1 $!x_4$		X0 ?	"	"

Remarque D-6
Dans le modèle synchrone, le flux des (y_i) est deux fois plus rapide que le flux des (x_i). Ce n'est pas le cas dans le modèle asynchrone : les deux flux ont la même “vitesse”. Plus précisément, si l'on note

$\Delta(e_1, e_2)$ le nombre d'événements ayant lieu entre les deux événements e_1 et e_2, alors :

$$(\exists c)\ \ (\forall i)\ \ \Delta(\text{“}\texttt{X0 !}x_i\text{”}, \text{“}\texttt{X0 !}x_{i+1}\text{”}) = \Delta(\text{“}\texttt{Y3 !}y_i\text{”}, \text{“}\texttt{Y3 !}y_{i+1}\text{”}) = c$$

■

D.8 Validation

L'utilisation de CÆSAR a permis de mettre au point et de valider les descriptions en LOTOS des différents réseaux systoliques. Dans chaque cas il a fallu donner des valeurs particulières au nombre n de cellules et au nombre m de valeurs x_i fournies par le générateur. En revanche, les valeurs x_i ont été manipulées sous forme d'expressions symboliques[47] afin de prouver la correction du réseau pour toute suite d'entrée (x_i) de longueur m.

A l'aide de CÆSAR les graphes correspondants aux réseaux systoliques ont été construits, ce qui a permis la détection de plusieurs erreurs : blocages, inversion des coefficients w_i, ... On constate que les graphes ainsi obtenus comportent approximativement autant d'arcs que d'états, ce qui se justifie par le fait que les différentes parties du réseau sont fortement synchronisées : à chaque instant, le nombre de rendez-vous possibles avoisine 1. Ce degré de non-déterminisme n'est pas le même pour les quatre schémas systoliques, ce qui explique les différences importantes entre les tailles des graphes.

Le tableau suivant regroupe les résultats expérimentaux obtenus. L'interprétation des temps et des débits est la même que celle donnée pour l'exemple du bit alterné (§ C.8, p. 244).

réseau	n	m	places	transitions	variables	états	arcs	temps	débit
B1	3	9	54	46	29	101 451	151 146	286:50–10:47	5–157
F	7	19	65	69	34	438	669	5:23–3:51	1.3–1.9
W1	3	6	33	37	14	64 085	87 775	88:44–4:54	12–218
W2	7	19	78	97	34	355	463	5:30–3:23	1.1–1.7

Ces graphes ont été minimisés par le logiciel ALDEBARAN [Fer88] selon la relation d'équivalence observationnelle ; dans tous les cas on a ainsi pu vérifier que le graphe minimal est une chaîne de $m - n + 1$ arcs dont le $i^{\text{ème}}$ a pour label :

$$\texttt{Y !}w_1 x_i + w_2 x_{i+1} + \ldots + w_n x_{i+n-1}$$

D.9 Notes bibliographiques

Le produit de convolution est caractéristique d'un grand nombre d'applications qui admettent une solution systolique : filtrage, reconnaissance de formes, corrélation, interpolation, évaluation polynômiale, transformation de Fourier discrète, addition et division polynômiale.

Les architectures B1, F, W1 et W2 sont tirées de [Kun82] qui recense et compare les différents schémas systoliques connus. Plusieurs tentatives ont été faites pour décrire des réseaux systoliques dans des langages parallèles et pour valider cette spécification. Les langages employés sont généralement synchrones : [HP86] est basé sur LUSTRE, [Gri88] utilise une variante synchrone de CSP.

[47] implémentées par des chaînes de caractères

Bibliographie

[ABG+88] Mark Ardis, Victor Basili, Susan Gerhart, Donald Good, David Gries, Richard Kemmerer, Nancy Leveson, David Musser, Peter Neumann, and Friedrich von Henke. Editorial Process Verification (ACM Forum). *Communications of the ACM*, 32(3):287–288, mars 1988.

[AF88] Sukhvinder S. Aujla and Matthew Fletcher. The Boyer-Moore Theorem Prover and LOTOS. In Kenneth J. Turner, editor, *Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88 (Stirling, Scotland)*, pages 169–183, Amsterdam, septembre 1988. North-Holland.

[AHU83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Computer science and information processing. Addison-Wesley, Reading, Massachusetts, 1983.

[Ail86] Georges Ailloud. Verification in ECRINS of LOTOS programs. In Ed Brinksma, editor, *Towards practical verification of LOTOS specifications — ESPRIT/SEDOS/C2/N89.2 — Second year project report*, Enschede, novembre 1986. Universiteit Twente. ESPRIT/SEDOS/C2/N48.1.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

[Bar88] Christian Bard. *CÆSAR.ADT 1.0 Reference Manual*. Laboratoire de Génie Informatique — Institut IMAG, Grenoble, août 1988.

[BB88] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–29, janvier 1988.

[BC88] Tommaso Bolognesi and Maurizion Caneve. SQUIGGLES: A Tool for the Analysis of LOTOS Specifications. In Kenneth J. Turner, editor, *Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88 (Stirling, Scotland)*, pages 201–216, Amsterdam, septembre 1988. North-Holland.

[BCG87] Gérard Berry, Philippe Couronné, and Georges Gonthier. Programmation synchrone des systèmes réactifs : le langage ESTÉREL. *Technique et Science Informatiques*, 6(4):305–316, 1987.

[BD88] Stanislas Budkowski and Piotr Dembinski. An Introduction to ESTELLE: A Specification Language for Distributed Systems. *Computer Networks and ISDN Systems*, 14(1):3–24, janvier 1988.

[BGLN85] Stanislas Budkowski, T. Gilot, L. Lumbrosso, and Elie Najm. General Presentation of SCAN — A Distributed Systems, Modelling and Verification Tool. In Michel Diaz, editor, *IFIP Workshop on Protocol Specification, Testing and Verification (Moissac, France)*, pages 103–118, Amsterdam, juin 1985. IFIP, North-Holland.

[BH88] Pascal Bouchon and Jean-Michel Houdouin. Analyseur LOTOS pour CÆSAR. Rapport de projet ENSIMAG, Laboratoire de Génie Informatique — Institut IMAG, Grenoble, juin 1988.

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the ACM*, 31(3):560–599, juillet 1984.

[BLM88] Christine Baumann, Fabienne Lagnier, and Florence Maraninchi. *Interface graphique de XESAR, un outil pour la validation de protocoles.* Laboratoire de Génie Informatique — Institut IMAG, Grenoble, janvier 1988.

[BR85] S. D. Brookes and A. W. Roscoe. *An Improved Failure Model for CSP.* In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Proceedings NSF-SERC Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, page ? Springer-Verlag, Berlin, 1985.

[Bra83] G. W. Brams. *Réseaux de Petri : théorie et pratique*, volume 1 (théorie et analyse) et 2 (modélisation et applications). Masson, Paris, mars 1983.

[Bri88] Ed Brinksma. *Of the design of Extended LOTOS.* Thèse de Doctorat, Universiteit Twente (Enschede), nov 1988.

[BS86] Ed Brinksma and Guiseppe Scollo. Formal Notions of Implementation and Conformance in LOTOS. Memorandum INF-86- 13, Universiteit Twente, Enschede, novembre 1986.

[BS87] Gérard Berry and Ravi Sethi. From Regular Expressions to Deterministic Automata. *Theoretical Computer Science*, 48:117–126, 1987.

[CCI88] CCITT. Specification and Description Language. Recommendation Z.100, International Consultative Committee for Telephony and Telegraphy, Genève, mars 1988.

[CES85] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specifications. TR 85–31, Department of Computer Sciences, University of Texas, Austin (Texas), novembre 1985.

[CHPP87] Paul Caspi, Nicolas Halbwachs, John A. Plaice, and Daniel Pilaud. LUSTRE : A Declarative Language for Programming Synchronous Systems. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages, München*, pages 178–188, 1987.

[Dio89] Christophe Diot. Efficient Techniques for Lowering Purity Test Scores : A Survey. In Jacques Menestrot, editor, *Proceedings of the 4th European Annual Symposium on Immoral Performance Evaluation, Mont-St-Martin (France)*, pages 169–196. Cortex Foundation Press, octobre 1989.

[EFH83] H. Ehrig, W. Fey, and H. Hansen. ACT ONE: An Algebraic Language with two Levels of Semantics. Bericht-Nr 83–03, Technische Universität Berlin, 1983.

[Eij88] P. H. J. van Eijk. *Software Tools for the Specification Language LOTOS.* Doctoral Dissertation, Universiteit Twente, 1988.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1 — Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1985.

[FA88] David Freestone and Sukhindver S. Aujla. Specifying ROSE in LOTOS. In Kenneth J. Turner, editor, *Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88 (Stirling, Scotland)*, pages 231–245, Amsterdam, septembre 1988. North-Holland.

[Fer88] Jean-Claude Fernandez. *ALDEBARAN : un système de vérification par réduction de processus communicants.* Thèse de Doctorat, Université Joseph Fourier (Grenoble), mai 1988.

[FRV85] Jean-Claude Fernandez, Jean-Luc Richier, and Jacques Voiron. Verification of Protocol Specifications using the CESAR System. In Michel Diaz, editor, *Proceedings of the 5th International Workshop on Protocol Specification, Testing and Verification*, pages 71–90, Amsterdam, jun 1985. IFIP, North-Holland.

[FSS83] Jean-Claude Fernandez, J. P. Schwartz, and Joseph Sifakis. *An Example of Specification and Verification in CESAR.* In G. Goos and J. Hartmanis, editors, *The analysis of concurrent systems*, volume 207 of *Lecture Notes in Computer Science*, pages 199–210. Springer Verlag, Berlin, septembre 1983.

[Gar89a] Hubert Garavel. *CÆSAR Reference Manual.* Laboratoire de Génie Informatique — Institut IMAG, Grenoble, avril 1989.

[Gar89b] Hubert Garavel. Compilation of LOTOS Abstract Data Types. In Son T. Vuong, editor, *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*, Amsterdam, décembre 1989. North-Holland.

[GHHL88] R. Guillemot, R. Haj-Hussein, and L. Logrippo. Executing Large LOTOS Specifications. In *Proceedings of the 8th International Workshop on Protocol Specification, Testing and Verification*, Amsterdam, 1988. IFIP, North-Holland.

[Gin83] Michel Ginguay. *Dictionnaire d'informatique (anglais—francais).* Masson, Paris, 7th edition, juillet 1983.

[GM84] U. Goltz and A. Mycroft. *On the Relationship of CCS and Petri Nets.* In Jan Paredaens, editor, *Proceedings of the 11th International Colloquium in Automata, Languages and Programming (ICALP 84) Antwerp, Belgium*, volume 172 of *Lecture Notes in Computer Science*, pages 196–208. Springer-Verlag, Berlin, juillet 1984.

[Gra84] Suzanne Graf. *Logiques du temps arborescent pour la spécification et la preuve de programmes.* Thèse de Doctorat, Institut Polytechnique de Grenoble, février 1984.

[Gri88] E. Pascal Gribomont. Proving Systolic Arrays. In Max Dauchet and Maurice Nivat, editors, *CAAP'88 13th Colloquium on Trees in Algebra and Programming, Nancy, France*, pages 185–199, Berlin, mars 1988. Springer Verlag.

[GRRV89] Suzanne Graf, Jean-Luc Richier, Carlos Rodriguez, and Jacques Voiron. *What are the Limits of Model Checking Methods for the Verification of Real Life Protocols?* In Joseph Sifakis, editor, *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems (Grenoble, France)*, volume 407 of *Lecture Notes in Computer Science*, pages 275–285. Springer-Verlag, Berlin, juin 1989.

[GS86a] Suzanne Graf and Joseph Sifakis. A Logic for the Description of Non-deterministic Programs and their Properties. *Information and Control*, 68(1–3):254–270, January–March 1986.

[GS86b] Suzanne Graf and Joseph Sifakis. Readiness Semantics for Processes with Silent Actions. Rapport technique SPECTRE C3, Laboratoire de Génie Informatique — Institut IMAG, Grenoble, novembre 1986.

[Hen85] M. C. B. Hennessy. Acceptance Trees. *Journal of the ACM*, 32(4):896–928, octobre 1985.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 583, octobre 1969.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, août 1978.

[HP86] Nicolas Halbwachs and Daniel Pilaud. Use of a Real-Time Declarative Language for Systolic Array Design and Simulation. In Will Moore, Andrew McCabe, and Roddy Urquhart, editors, *Proceedings of the International Workshop on Systolic Arrays, Oxford*, pages 81–90, Bristol and Boston, juillet 1986. Adam Hilger.

[Hul88] Wilfried H. P. van Hulzen. LOTTE — A LOTOS Tool Environment. In Kenneth J. Turner, editor, *Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88 (Stirling, Scotland)*, pages 61–65, Amsterdam, septembre 1988. North-Holland.

[ISO84] ISO. Basic Reference Model. International Standard 7498, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, 1984.

[ISO87] ISO. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Draft International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, juillet 1987.

[ISO88a] ISO. ESTELLE — A Formal Description Technique Based on an Extended State Transition Model. International Standard 9074, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, septembre 1988.

[ISO88b] ISO. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, septembre 1988.

[Joh88] Stuart G. Johnston. SPIDER — Service and Protocole Interactive Development Environment. In Kenneth J. Turner, editor, *Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88 (Stirling, Scotland)*, pages 67–71, Amsterdam, septembre 1988. North-Holland.

[JW78] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report*. Springer-Verlag, New York, 2nd edition, 1978.

[Knu68] Donald E. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. See also *ibidem* 5(1):95–96, 1971.

[Knu73] Donald E. Knuth. *The Art of Computer Programming — Volume III : Sorting and Searching*. Computer Science and Information Processing. Addison-Wesley, Reading, Massachusetts, 1973.

[Koz82] Dexter Kozen. *Results on the Propositional μ-Calculus*. In G. Goos and J. Hartmanis, editors, *Proceedings of the 9th International Colloquium in Automata, Languages and Programming (ICALP 82) Aarhus, Denmark*, volume 140 of *Lecture Notes in Computer Science*, pages 348–359. Springer-Verlag, Berlin, juillet 1982.

[Kun82] H. T. Kung. Why systolic architectures ? *Computer*, 15(1):37–46, janvier 1982.

[Led87] G. J. Leduc. The Intertwining of Data Types and Processes in LOTOS. In Harry Rudin and Colin H. West, editors, *Proceedings of the 7th International Symposium on Protocol Specification, Testing and Verification (Zurich)*, Amsterdam, mai 1987. IFIP, North-Holland.

[LMV87] Valérie Lecompte, Eric Madeleine, and Didier Vergamini. AUTO: un système de vérification de processus parallèles et communicants. Rapport technique 83, I.N.R.I.A., Sophia-Antipolis, France, mars 1987.

[Lon87] Brigitte Lonc. *Techniques formelles de vérification des systèmes distribués : application aux protocoles de communication des systèmes ouverts*. Thèse de Doctorat, Conservatoire National des Arts et Métiers (Paris), juin 1987.

[LS88] Jeroen van de Lagemaat and Giuseppe Scollo. On the Use of LOTOS for the Formal Description of a Transport Protocol. In Kenneth J. Turner, editor, *Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88 (Stirling, Scotland)*, pages 247–261, Amsterdam, septembre 1988. North-Holland.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.

[ML88] Saturnino Marchena and Gonzalo Leon. Transformation from LOTOS Specs to Galileo Nets. In Kenneth J. Turner, editor, *Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88 (Stirling, Scotland)*, pages 217–230, Amsterdam, septembre 1988. North-Holland.

[ndM88] J. A. Ma nas and T. de Miguel. From LOTOS to C. In Kenneth J. Turner, editor, *Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88 (Stirling, Scotland)*, pages 79–84, Amsterdam, septembre 1988. North-Holland.

[NH84] R. De Nicola and M. C. B. Hennessy. Testing Equivalences for Processes. *Theoretical Computer Science*, 34:83–133, 1984.

[Par81] David Park. *Concurrency and Automata on Infinite Sequences*. In Peter Deussen, editor, *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, Berlin, mars 1981.

[PG88] Marc Phalippou and Roland Groz. Using ESTELLE for Verification: An Experience with the T.70 Teletex Transport Protocol. In Kenneth J. Turner, editor, *Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88 (Stirling, Scotland)*, pages 185–199, Amsterdam, septembre 1988. North-Holland.

[Plo81] G. Plotkin. A Structural Approach to Operational Semantics. DAIMI FN 19, Århus University Computer Science Department, Århus, Denmark, 1981.

[Pnu86] Amir Pnueli. Specification and Development of Reactive Systems. In H. J. Kugler, editor, *Information Processing 86 — Proceedings of the IFIP 10th World Computer Congress*, pages 845–858, Amsterdam, septembre 1986. IFIP, North-Holland.

[QPF88] Juan Quemada, Santiago Pavón, and Angel Fernández. Transforming LOTOS Specifications with LOLA : The Parametrized Expansion. In Kenneth J. Turner, editor, *Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88 (Stirling, Scotland)*, pages 45–54, Amsterdam, septembre 1988. North-Holland.

[QS83] Jean-Pierre Queille and Joseph Sifakis. Fairness and Related Properties in Transition Systems — A Temporal Logic to Deal with Fairness. *Acta Informatica*, 19:195–220, 1983.

[Que82] Jean-Pierre Queille. *Le système CESAR : description, spécification et analyse des spécifications réparties*. Thèse de docteur-ingénieur, Université de Grenoble, juin 1982.

[Ras88] Anne Rasse. CLEO : interprétation de la non-correction de programmes sur un modèle. Rapport technique SPECTRE C10, Laboratoire de Génie Informatique — Institut IMAG, Grenoble, juin 1988.

[Rod88] Carlos Rodriguez. *Spécification et validation de systèmes en XESAR*. Thèse de Doctorat, Institut National Polytechnique de Grenoble, mai 1988.

[RRSV87a] Jean-Luc Richier, Carlos Rodriguez, Joseph Sifakis, and Jacques Voiron. Verification in XESAR of the Sliding Window Protocol. In Harry Rudin and Colin H. West, editors, *Proceedings of the 7th International Symposium on Protocol Specification, Testing and Verification (Zurich)*. IFIP, North-Holland, mai 1987.

[RRSV87b] Jean-Luc Richier, Carlos Rodriguez, Joseph Sifakis, and Jacques Voiron. *XESAR: A Tool for Protocol Validation. User's Guide*. Laboratoire de Génie Informatique — Institut IMAG, Grenoble, septembre 1987.

[RSV86] Jean-Luc Richier, Joseph Sifakis, and Jacques Voiron. ATP — An Algebra for Timed Processes. Rapport technique SPECTRE C1, Laboratoire de Génie Informatique — Institut IMAG, Grenoble, décembre 1986.

[RV87] Jean-Luc Richier and Jacques Voiron. Spécification et analyse d'un protocole de communication à l'aide du système XESAR. *BIGRE+GLOBULE*, 55:137–148, juillet 1987.

[SAC88] Marten van Sinderen, Ibrahim Ajubi, and Fausto Caneschi. The Application of LOTOS for the Formal Description of the ISO Session Layer. In Kenneth J. Turner, editor, *Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88 (Stirling, Scotland)*, pages 263–277, Amsterdam, septembre 1988. North-Holland.

[Sch83] Jean-Philippe Schwartz. *QUASAR, une réalisation du système CESAR*. Thèse de docteur-ingénieur, Université de Grenoble, novembre 1983.

[Sch88a] Philippe Schnoebelen. Refined Compilation of Pattern-Matching for Functional Languages. *Science of Computer Programming*, 11:133–159, 1988.

[Sch88b] Philippe Schnoebelen. Refined Compilation of Pattern-Matching for Functional Languages. RR 715-I-71, Laboratoire d'Informatique Fondamentale et d'Intelligence Artificielle — Institut IMAG, Grenoble, avril 1988.

[Sif86] Joseph Sifakis. A Response to Amir Pnueli. In H. J. Kugler, editor, *Information Processing 86 — Proceedings of the IFIP 10th World Computer Congress*, Amsterdam, septembre 1986. IFIP, North-Holland.

[ST87] R. Saracco and P. A. J. Tilanus. CCITT SDL: Overview of the Language and its Applications. *Computer Networks and ISDN Systems*, 13(2):65–74, 1987.

[Tar72] Robert E. Tarjan. Depth First Search and Linear Graph Algorithm. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[Ten76] R. D. Tennent. The Denotational Semantics of Programming Languages. *Communications of the ACM*, 19(8):437–453, août 1976.

[Tre87] Jan Tretmans. HIPPO Handout. In *ESPRIT/SEDOS/C3/WP/54/T — Third year project report*, Enschede, décembre 1987. Universiteit Twente.

[Vis88] Chris A. Vissers. FDTs for Open Systems — An ISO Perspective on FDTs. In Kenneth J. Turner, editor, *Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88 (Stirling, Scotland)*, Amsterdam, septembre 1988. North-Holland. invited paper.

[Wir83] Niklaus Wirth. *Programming in Modula-2.* Springer-Verlag, New York, 2nd edition, 1983.

[Xuo84] N. H. Xuong. *Eléments de combinatoire pour l'informatique. Tome II : graphes — théorie, algorithmes et applications.* Université de Grenoble, 1984.