

Sémantique des programmes récursifs-parallèles et méthodes pour leur analyse

Olga Kouchnarenko

▶ **To cite this version:**

Olga Kouchnarenko. Sémantique des programmes récursifs-parallèles et méthodes pour leur analyse. Autre [cs.OH]. Université Joseph-Fourier - Grenoble I, 1997. Français. NNT : . tel-00004949

HAL Id: tel-00004949
https://theses.hal.science/tel-00004949

Submitted on 20 Feb 2004

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

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

THESE

présentée par

Olga Kouchnarenko

pour obtenir le titre de Docteur
de l'Université Joseph Fourier - Grenoble I
et
de l'Université d'Etat de Iaroslavl
(Arrêtés ministériels du 5 juillet 1984 et du 30 mars 1992)

Spécialité : INFORMATIQUE

Sémantique des programmes récursifs-parallèles et méthodes pour leur analyse

Date de soutenance: le 24 février 1997

Composition du jury:

M. Joseph SIFAKIS	Président
Mme Olga BANDMAN	Rapporteur
M. Luc BOUGE	Rapporteur
M. Philippe SCHNOEBELEN	Examinateur
M. Valéry SOKOLOV	Examinateur
M. Jacques VOIRON	Examinateur
M. Philippe JORRAND	Examinateur

Thèse préparée au sein du **L**aboratoire **LEIBNIZ** de l'**I**nstitut **IMAG**

Remerciements

Je tiens tout d'abord à remercier Joseph Sifakis, qui a bien voulu nous faire l'honneur de présider le jury.

Je tiens ensuite à remercier Olga Bandman et Luc Bougé, qui ont accepté le rôle difficile de rapporteur, et ce malgré des délais relativement courts, apportant ainsi à cette thèse la garantie de leur grande compétence scientifique. Je remercie Luc Bougé et Olga Bandman de leurs très précieux commentaires qui m'ont beaucoup servi à améliorer la qualité de cette thèse, qu'ils soient assurés de toute ma gratitude.

Je remercie particulièrement Jacques Voiron et Philippe Jorrand qui ont eu la gentillesse, en acceptant, pour la première fois, de faire partie du jury d'une thèse en co-tutelle, d'apporter à ce travail le soutien de leur renommée. Je remercie Philippe Jorrand de m'avoir accueillie au sein du laboratoire LEIBNIZ où j'ai passé une excellente année: j'ai beaucoup apprécié l'ambiance qui règne dans ce laboratoire et j'en garderai un très bon souvenir.

Je suis très reconnaissante à Valéry Sokolov. Je le remercie d'avoir encadré cette thèse, de m'avoir guidé. Il a su offrir, sans jamais l'imposer, tout ce qui pouvait manquer à un chercheur débutant: une large connaissance de la littérature, une vue d'ensemble du sujet, l'insertion dans la communauté internationale. Je remercie mon chef pour toutes ces années! Qu'il soit assuré de toute ma reconnaissance.

La possibilité d'effectuer les recherches présentées ici dans le cadre de la thèse en co-tutelle a été pour moi très bénéfique. Philippe Schnoebelen, mon directeur de recherche français, m'a accueillie dans un cadre très sympathique, m'a entouré avec amitié, compétence, et une très grande disponibilité. Qu'il trouve ici l'expression de l'estime que je lui porte ainsi que de ma reconnaissance pour les nombreuses discussions que nous avons eu. Je me souviendrai toujours de son soutien et de ses encouragements dans les moments difficiles. Je lui dois bien plus que ce que je peux présenter ici.

Cette thèse doit beaucoup à tous mes amis et collègues, russes et français, anciens et récents. Je ne peux en citer ici que quelques uns: Ludmila, Lilia, Nina, Thierry le Grand, Catherine, Cristophe, Denis, Thierry, Andrée, Jacky, Danièle, Jean-Claude... Je dois énormément à Gilles qui a toujours été là lorsque j'avais besoin de conseils d'un expert...

Je remercie de tout coeur une famille "de ma connaissance", Madeleine et

Alain Cayot, qui sont devenus une véritable famille d'adoption. Merci pour leur générosité, leur compréhension et leur soutien.

Enfin, une pensée chaleureuse à tous ceux que j'aime...

Table des matières

Index des notations utilisées **11**

1 Introduction et motivations **13**
1.1 Parallélisme 13
1.2 Motivations et historique 14
1.2.1 RP: une approche du parallélisme 14
1.2.2 Le langage RPC 16
1.2.3 Analyse et diagnostic des programmes RPC 16
1.3 Les objectifs généraux de la thèse et sa structure 18

I Cadre général **23**

2 Le langage RPC **27**
2.1 Introduction 27
2.2 Présentation de (processus) RPC 29
2.2.1 Exemple introductif 29
2.2.2 Particularités de RPC 30

3 Modèles comportementaux pour le parallélisme **35**
3.1 Préliminaires 35
3.2 Les mots 36
3.3 Les multi-ensembles 37
3.4 Systèmes de transitions (étiquetés) 38
3.5 Non-déterminisme des systèmes de transitions 40
3.6 Chemins et traces 41
3.7 Equivalence de traces 42

4 Equivalences de bisimulation **45**
4.1 Equivalence de bisimulation (forte) 46
4.2 Décidabilité de la bisimulation 48

4.3 Equivalences projectives 48
4.4 Bisimulation modulo τ 50
4.5 Une autre bisimulation modulo τ 52
4.6 Traces modulo τ 53

5 Algèbres de Processus **55**
5.1 Déclarations de processus 55
5.2 Comportement des processus 57
5.3 Déclarations gardées 59
5.4 Propriétés algébriques des opérateurs de PA 60
5.5 Sous-classes de processus et leurs formes normales 64

II Modèle formel pour RPC 69

6 Sémantique comportementale de RPC **71**
6.1 Schémas de programmes récursifs-parallèles 72
6.2 Sémantique d'états hiérarchiques 75
6.3 Comportement des états hiérarchiques 77
6.4 Etude du comportement des schémas 80
6.5 Etats hiérarchiques normés 84
6.6 Conclusion 89

7 Etude d'expressivité du modèle **91**
7.1 Langages des schémas RPPS 92
7.2 De PA à RPPS 93
7.3 Motifs RPPS et leur correction 96
7.4 Codage des processus PA 101
7.5 De Δ_τ à Δ 104
7.6 De RPPS à PA 108
7.7 Comparaison des classes de langages 114

8 Schémas RPPS bien structurés **117**
8.1 Systèmes de transitions bien structurés 118
8.2 Compatibilité vers le bas 121
8.3 Compatibilité vers le haut 126
8.4 Conclusion 134

III Des modèles d'implémentation 137

9 Une réalisation distribuée 141
9.1 Une sémantique distribuée 141
9.2 Correspondances entre $\mathcal{M}_G$ et $\mathcal{S}_G$ 146

10 Implémentation avec parallélisme borné 153
10.1 Une sémantique vectorielle 153

11 Schémas RPPS interprétés 161
11.1 Propriétés de programmes 161
11.2 Modèles interprétés 161
11.3 Conclusion 166

Bibliographie 169

Liste des figures

1.1 Style de programmation récursif-parallèle 15

2.1 Un programme qui calcule une somme $\sum_{l=i}^{j} F(l)$ 30
2.2 Un programme RPC calculant une somme $\sum_{l=i}^{j} F(l)$ 32

3.1 Un système de transitions étiqueté 40
3.2 Deux systèmes de transitions ayant mêmes traces 43

4.1 Deux systèmes bisimilaires 47
4.2 Deux systèmes τ-bisimilaires 51

5.1 Le système de transitions pour $\{X \stackrel{déf}{=} \alpha.(X.Y) + \gamma.\mathbf{0};\ Y \stackrel{déf}{=} \beta.\mathbf{0}\}$ 58
5.2 Un système à branchement infini pour $\{X \stackrel{déf}{=} X.\alpha + \beta\}$ . . . 59

6.1 Un programme RPC et le schéma associé 73
6.2 L'état hiérarchique ξ comme un arbre 76

7.1 Motif RPPS pour le choix non-déterministe 94
7.2 Motif RPPS pour la composition séquentielle 94
7.3 Motif RPPS pour la composition parallèle 95
7.4 Noeud `pcall` et composition parallèle 98
7.5 Schéma associé à la déclaration de l'exemple 7.2 103
7.6 Un noeud `pcall` et ses suivants 109
7.7 Exemple de décomposition d'un chemin 113
7.8 Comparaison des classes de langages (modulo les actions silencieuses) 115

8.1 Compatibilité 120
8.2 Plongement entre états hiérarchiques: $\xi_1 \preceq \xi$ 122
8.3 Plongement entre états hiérarchiques: $\xi_2 \preceq \xi$ 122
8.4 Compatibilité 123
8.5 $\perp$-plongement entre états hiérarchiques: $\xi \preceq_{\perp} \xi'$ 127

8.6 Compatibilité vers le haut pour $\preceq_\perp$ 129
8.7 Un arbre d'atteignabilité fini 132
8.8 Un petit exemple explicatif 134

9.1 Un état distribué 143

10.1 Un état vectoriel 154

11.1 Comparaison des modèles formels 164

Index des notations utilisées

$\mathcal{A}$ b.f., 41
A^*, 36
$\mathcal{A}_1 \sim_{Tr} \mathcal{A}_2$, 43
$\mathcal{A}_1 \sim_{Tr_\tau} \mathcal{A}_2$, 54
$\mathcal{A}_1 \underline{\leftrightarrow}_\tau \mathcal{A}_2$, 51
$\mathcal{A}_1 \underline{\leftrightarrow}_d \mathcal{A}_2$, 52
Act_ϵ, 50
Act_τ, 50
A^∞, 36
A^ω, 36
$Bloc$, 143
BPA, 65
BPP, 65
$\Delta \underline{\leftrightarrow} \Delta'$, 61
Δ_τ, 95
$\mathcal{EXP}$, 56
ϵ, 36
$Ex(\mathcal{A})$, 41
$Ex(q)$, 41
φ, 75
fng, 66
G, 74
GNF, 65
Id_X, 35
I_q, 110
$isnil(E)$, 58
L, 74
Λ, 74
λ, 50
Λ_τ, 74
L(PA), 91
$\mid M \mid$, 37
M_G, 75
$\mathcal{M}_G$, 77
$\mathcal{N}(\xi)$, 85
ω, 142
ω_{init}, 143
p `div`, 52
$p \underline{\leftrightarrow}_\tau p'$, 51
$p \underline{\leftrightarrow}_d p'$, 52
$p \xrightarrow{\alpha} q$, 39
$p \xrightarrow{\alpha}$, 39
$p \xrightarrow{w} p'$, 39
$p \rightarrow q$, 39
$p \rightarrow$, 39
PA, 66
$\mathcal{P}_G$, 155
π, 153
π_0, 155
Π_G, 155
$Post(\xi)$, 123
$q \downarrow$, 87
$q \stackrel{\alpha}{\Rightarrow} q'$, 50
$q \stackrel{\epsilon}{\Rightarrow} q'$, 50
$q_1 \sim_{Tr} q_2$, 43
$q_1 \sim_{Tr_\tau} q_2$, 54
$q_1 \equiv q_2$, 49
$q_1 \equiv_n q_2$, 49
Q_q^{suiv}, 109
$Q_{\texttt{wait}}$, 109
R^{-1}, 35
$R : \mathcal{A}_1 \underline{\leftrightarrow} \mathcal{A}_2$, 47
RCCS, 64
RdP, 91
$R \circ R'$, 35
$R : p \underline{\leftrightarrow} q$, 47

RPC, 31
$RPPS_{\Lambda_\tau}$, 75
RPPS, 72
$RT(\xi)$, 131
$RW(q)$, 98
$\uparrow(S)$, 118
$\mathcal{S}_G$, 143
Σ, 143
$\mid \sigma \mid$, 42
σ, 142
σ_{dep}, 147
σ_0, 143
$min_{\leq}(S)$, 119
ST, 39
$sub(\Delta)$, 102
τ, 50
$tr(\sigma)$, 42
$tr_\tau(\sigma)$, 53
$Tr(\mathcal{A})$, 42
$Tr(q)$, 42
$Tr_\tau(\mathcal{A})$, 54
$Tr_\tau(q)$, 54
$\mathcal{U}(X)$, 38
$\Upsilon(q)$, 41
Var, 55
$Var(\Delta)$, 57
$w \ll w'$, 37
$\mid w \mid$, 36
$w \cdot w'$, 37
$\mid X \mid$, 35
ξ, 75
$\xi\downarrow$, 85
$\xi\uparrow$, 85
$\xi \preceq \xi'$, 121
$X \subset Y$, 35
ζ, 75

Chapitre 1

Introduction et motivations

1.1 Parallélisme

Les besoins grandissants en puissance de calcul rendent nécessaire la recherche d'architectures et de techniques multipliant les capacités actuelles de traitement des informations par ordinateurs. L'idée du parallélisme n'est pas nouvelle dans ce domaine; elle s'impose par son évidence. Il est inutile d'attendre qu'une tâche soit terminée pour en commencer une autre, si celle-ci ne dépend pas de la première et si rien n'empêche de la commencer plus tôt. Contrairement à l'approche séquentielle dans laquelle les événements ont lieu successivement, le parallélisme autorise l'exécution simultanée de plusieurs actions. Historiquement, on peut dire que les systèmes parallèles sont nés de la volonté de mettre dans une seule machine plusieurs processeurs (qui sont a priori semblables) et de les faire travailler ensemble (donc avec une certaine idée de mettre ces processeurs dans un même boîtier).

Toutefois, le parallélisme ajoute une dimension supplémentaire à la difficulté de l'activité de programmation. De nombreux domaines de l'informatique ont été invoqués dans le processus d'élaboration des systèmes parallèles. On a eu et on a besoin de la recherche sur les architectures, les systèmes d'exploitation, les algorithmes parallèles. Il faut trouver comment attribuer une tâche à chaque processeur d'un système parallèle, l'ensemble de ces tâches formera alors une tâche globale. C'est la *programmation parallèle*. Le développement de matériels et de logiciels exigé par le parallélisme doit être basé sur des méthodes formelles car la taille des systèmes informatiques s'est accrue, leur structure et leur fonctionnement étant devenus de plus en plus complexes.

En ce qui concerne la programmation parallèle, pour pouvoir raisonner

formellement sur un programme, pour donner une preuve de sa correction ou de sa terminaison, pour évaluer sa complexité, énoncer des principes généraux, comparer des programmes, etc., on doit représenter les programmes et leur comportement par des objets mathématiques. Il faut développer des modèles, des concepts, des notations, des théories adaptés à la programmation parallèle.

La *théorie du parallélisme* s'est donc attachée à définir un cadre mathématique pour décrire les systèmes parallèles et à en étudier les comportements à l'aide de modèles formels.

1.2 Motivations et historique

Entre les années soixante (avec des formalismes tels que ceux de Karp & Miller) et aujourd'hui, des mathématiciens et des informaticiens ont cherché à raisonner sur les programmes parallèles, à étudier des principes spécifiques propres à la programmation parallèle.

Parmi ces principes, ceux qui développent une idée du parallélisme permettant la récursivité présentent un grand intérêt. Mais leur mise en oeuvre pose de nombreux problèmes fondamentaux.

1.2.1 RP: une approche du parallélisme

Pour aider à la conception et à la compréhension de systèmes complexes, on utilise des méthodes de conception modulaires. La conception modulaire consiste à construire le programme en assemblant des sous-programmes et en décrivant la manière dont ils interagissent, par exemple en s'exécutant en parallèle, en communiquant par messages, etc.

Une analyse effectuée par des spécialistes de l'Institut des Problèmes de la Technique de Calcul (IPTC) de l'Académie des Sciences de Russie a montré que, pour une classe assez large de problèmes qui utilisent des algorithmes classiques d'algèbre linéaire et d'analyse numérique (par exemple, il s'agit du logiciel LINPACK), il est facile de découper récursivement le calcul en parties à peu près égales qui peuvent être effectuées de façon indépendante [VKT90, AG85] (cf. figure 1.1).

Ce développement de processus de calcul fait apparaître des sous-programmes qui doivent être effectués sans répartition de données. La collection des résultats des sous-programmes sur le plus haut niveau d'un problème (niveau 1 dans la figure 1.1) indique que tous les processus parallèles sont accomplis.

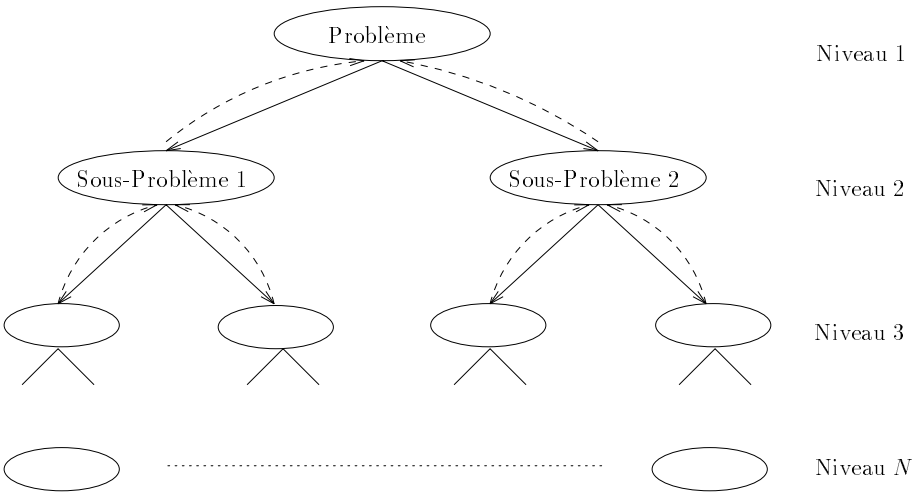


Figure 1.1 : Style de programmation récursif-parallèle

Ce style de programmation est dit *récursif-parallèle* (RP-style de programmation). Ses avantages sont indiqués dans [VEKM94, Pan89]:

- l'invariance d'une RP-description d'un algorithme par rapport aux structures d'un système de calcul et au nombre m de modules de ses processeurs;

- la possibilité de décrire le parallélisme qui dépend de données à traiter et, comme conséquence, qui apparaît seulement pendant une exécution d'un problème;

- théoriquement, l'ordre de croissance au cours du temps du nombre de processus parallèles et de leur distribution, est de $\log_2 m$, où m est le nombre de processeurs d'un RP-système.

Un matériel supportant cette RP-approche du parallélisme est celui d'une architecture multiprocesseurs à mémoire commune globale d'accès direct et à mémoire distribuée locale où le contrôle des m modules de processeurs utilise des unités assurant les connexions inter-modules. En particulier, l'IPTC élabore un système d'ordinateurs récursif-parallèle adoptant cette architecture [MVVK88].

1.2.2 Le langage RPC

Pour que le RP-style de programmation soit implémenté il est nécessaire de changer le mécanisme d'invocation de procédures de telle façon que le matériel et le logiciel du système assurent la transmission de processus parallèles entre des modules et la collection des résultats. C'est ce système d'ordinateurs que l'IPTC est en train d'élaborer.

La programmation de ces systèmes repose sur des langages parallèles spécialisés, de haut-niveau. Il est évident que ces langages doivent posséder des instructions spéciales qui permettent d'organiser un processus de calcul en RP-style [Pan89]. Au minimum, ce sont des opérateurs de description et d'invocation de processus parallèles, ainsi que de leur synchronisation.

L'un de ces langages est RPC, langage récursif-parallèle impératif supportant le parallélisme et la récursivité, et ayant une discipline parallèle particulière, actuellement utilisé à l'IPTC où un compilateur RPC a été développé pour une machine parallèle [VEKM94]. Nous donnerons au chapitre 2 une description plus détaillée du langage (en fait, le fragment "RPC pur"). Nous aborderons dans les chapitres 9, 10 et 11 de cette thèse des problèmes liés à la modélisation de l'implémentation de RPC.

1.2.3 Analyse et diagnostic des programmes RPC

Dans le domaine de la programmation parallèle, le problème de la correction de programme prend une importance et une difficulté nettement accrues du fait de la complexité considérable de l'organisation des calculs dans un système multiprocesseur. Quant au système récursif-parallèle qui est un cas particulier de systèmes multiprocesseurs, là aussi, l'exécution efficace des calculs dépend de la structuration du programme récursif-parallèle, dans la mesure où la répartition du calcul et les exigences de synchronisation influent sur le rendement réel d'un algorithme parallèle.

La conception de systèmes complexes et la garantie de leur fiabilité incitent au développement d'outils formels pour s'assurer qu'un programme donné se comporte bien comme on le souhaite. Du fait de la complexité des programmes parallèles, il serait utile de mettre au point des systèmes automatisés pour l'analyse et le diagnostic (un *outil d'analyse*) des programmes récursifs-parallèles.

Pour ce faire, on représente les programmes (et leur comportement) par des objets mathématiques, il devient alors possible d'énoncer formellement ce qu'est la "correction" d'un programme et de démontrer en un sens mathé-

matique qu'un programme donné est "correct".

Chronologiquement, le premier problème que nous avons abordé était celui de l'analyse du flot de contrôle dans des programmes écrits en langage RPC [SK95].

Ainsi nous avons vu l'outil d'analyse comme un sous-système du compilateur qui lui-même n'utilise qu'une description formalisée des processus de calcul à mettre en oeuvre. Il est intéressant que cette description se fasse à un certain niveau d'abstraction, sans faire référence aux détails d'implémentation.

Dans un premier temps, nous avons modélisé un flot de contrôle d'un RP-programme dans le formalisme des organigrammes. Ce formalisme est assez proche des constructions langagières de RPC, de sorte que la modélisation est facile à définir et à comprendre. Nous avons défini, dans un deuxième temps, une traduction des organigrammes dans les réseaux de Petri places/transitions, qui semblaient un bon formalisme sur lequel bâtir de tels outils. Par ailleurs, ils sont largement utilisés pour modéliser des processus parallèles, et leurs propriétés peuvent être étudiées par des méthodes mathématiques.

Dans cette problématique, les travaux suivants ont d'ores et déjà été réalisés:

1. Analyse et formalisation (en liaison avec des utilisateurs de RPC) des critères de correction pertinents.

2. Isolation d'un ensemble réduit de constructions RPC de base, suffisamment expressives.

3. Définition et étude des réseaux de Petri *primitifs* qui correspondent à la sémantique et aux constructions de base du langage RPC.

Un module logiciel correspondant à cette étape a été déjà construit. Il permet, lors de la compilation, de générer un réseau P/T de Petri pour chaque procédure d'un RP-programme à partir de son texte en utilisant son organigramme et des réseaux de Petri primitifs, de visualiser le modèle de réseau, de simuler et de contrôler le fonctionnement du programme, d'analyser son comportement. Une telle analyse des réseaux P/T de Petri générés permet de faire un diagnostic d'un RP-programme qui indique, en particulier, si des instructions sont atteignables, si des instructions de transition et/ou celles de choix peuvent influencer (de façon indésirable) la discipline d'invocation ou de synchronisation de procédures parallèles.

Quand la non-correction est établie, ce module logiciel donne un diagnostic et propose soit de poursuivre la compilation, soit de modifier le programme.

En arrivant à Grenoble, nous avons constaté que l'outil présente des problèmes:

1. il ne s'appuie pas sur une sémantique formelle de RPC;
2. il ne tient pas compte des données;
3. il ne tient pas compte de la discipline de synchronisation.

Il est possible de justifier ces deux dernières limitations, qui rendent l'outil correct, mais l'absence de sémantique formelle ne permet pas d'énoncer de propriété de correction.

Au départ, notre projet était de décrire formellement ce que fait l'outil actuel. En cours de route, l'ancien outil est apparu moins intéressant et nous avons été conduits à nous poser des questions théoriques d'analyse du nouveau modèle. Nous avons décidé de donner un modèle formel pour RPC, modèle sur lequel nous pourrions construire une nouvelle génération d'outils qui n'aurait plus ces problèmes. Aujourd'hui l'équipe des chercheurs du département d'informatique théorique de l'Université d'Etat à Iaroslavl a décidé de faire évoluer l'outil actuel en se basant sur notre modèle.

1.3 Les objectifs généraux de la thèse et sa structure

Cette thèse s'inscrit dans le cadre des travaux consacrés au développement des modèles sémantiques destinés aux langages de programmation concurrents. Une particularité de notre étude réside dans la considération explicite d'une récursivité au niveau de programmes parallèles. Notre but est de présenter un modèle abstrait du langage RPC et d'en étudier les propriétés.

Dans cette thèse, nous construisons un domaine sémantique basé sur des systèmes de transitions [Kel76] modulo (diverses) bisimulation. Nous utilisons des systèmes de transitions parce qu'ils offrent un cadre technique simple, pratique et général pour la modélisation des comportements parallèles. Nous utilisons une famille d'équivalences sémantiques du temps arborescent

basée sur la bisimulation parce qu'elles sont puissantes, faciles à manipuler, et présentent un intérêt théorique fondamental [Mil89, BK89].

Dans la première partie de la thèse, nous précisons notre cadre général et nous donnons quelques rappels: dans le chapitre 2, nous décrivons des particularités du langage qui nous intéresse; les trois chapitres suivants présentent les notions de base qui nous servent dans la suite.

Les travaux présentés dans la deuxième partie sont issus de recherches sur une sémantique comportementale de programmes parallèles écrits en RPC. Pour l'essentiel, nous reprenons ici, en les développant, les résultats de [KS96a].

Tout d'abord, nous introduisons (chapitre 6) une notion de *schéma de RP-programme* (RPPS) permettant de représenter la structure d'un RP-programme donné. Notre notion de schémas de RP-programmes sert à distinguer précisément la structure de contrôle d'un programme de ses autres aspects en s'abstrayant des données et de l'effet des instructions de base. Notre sémantique d'états hiérarchiques ne capture que partiellement le comportement réel d'un RP-programme. Néanmoins, cette simplification n'empêche pas d'analyser de façon pertinente le flot de contrôle dans un programme écrit en RPC.

Pour ces schémas nous proposons une sémantique dite d'*états hiérarchiques*, dont nous étudions en détail les propriétés de base.

Dans le chapitre 7, nous nous intéressons à l'expressivité du modèle: nous examinons la classe des langages de traces engendré modulo les actions silencieuses L(RPPS), nous montrons qu'il est équivalent à celui engendré par *Processes Algebra* (PA) de Bergstra et Klop [BK89].

Un de nos objectifs est de fournir des méthodes pour l'analyse des programmes récursifs-parallèles.

Les systèmes de transition finis sont la base de nombreuses méthodes et outils d'analyse des programmes concurrents (cf. [IP91] pour une comparaison des principaux outils).

Toutefois de nombreux modèles ne sont pas d'états finis. Dans ce cadre, la plupart des problèmes de vérification deviennent indécidables car de nombreuses techniques pour la vérification de systèmes parallèles procèdent par un parcours exhaustif de l'espace d'états. Par conséquent, ces techniques sont constitutivement incapables de considérer des systèmes dont les ensembles d'états sont infinis.

Récemment, certaines méthodes ont été développées pour vaincre cette

limitation, du moins pour des classes restreintes de tels systèmes. Il y a lieu de citer ici par exemple [CH93, Mol96, ACJY96, Esp96].

Dans notre recherche, nous avons choisi un cadre des systèmes de transitions bien structurés dont l'utilisation pour l'analyse de systèmes à nombre d'états infini est apparue dans [Fin90] et plus tard a été développée dans [ACJY96].

Notre modèle donne naissance à des systèmes de transitions potentiellement infinis à cause de la récursivité, mais cependant il est possible de le munir d'une structure de système de transitions bien structuré. Ce fait permet de montrer la décidabilité de plusieurs problèmes classiques.

Le chapitre 8 montre comment les schémas de RPPS peuvent être munis d'une sémantique formelle en terme de systèmes de transitions bien structurés (au sens de [Fin90, ACJY96]). Cette structure est différente (donc, elle est intéressante) des exemples déjà connus de systèmes de transitions bien structurés (tels que hybrid automata, data-independent systems, relational automata, lossy channel systems et Réseaux de Petri) et, en un certain sens, elle est moins simple qu'eux. Les résultats principaux de cette partie permettent d'établir la décidabilité de nombreuses questions, par exemple d'atteignabilité (d'accessibilité), de terminaison, etc.

Dans la dernière partie de cette thèse, nous abordons les problèmes liés à la formalisation de la stratégie d'implémentation de RPC sur machine parallèle de [VEKM94]. (Ces travaux ont déjà été présentés partiellement dans [KS96b].)

Le modèle RPPS est basé sur le langage "RPC pur", mais il est possible d'appliquer la même démarche de formalisation au travail d'implémentation actuelle de RPC avec des raisonnements en terme de distribution de processus, de files d'attente, etc.

Ici notre souci est de relier notre modèle formel et l'implémentation effective des programmes RPC telle qu'elle a été réalisée à l'IPTC. Nous envisageons deux modèles pour l'implémentation: une réalisation *distribuée* avec parallélisme non-borné, et une réalisation, dite *vectorielle*, à base de files d'attente pour un modèle d'implémentation avec parallélisme borné.

Les trois modèles (d'états hiérarchiques, distribués, vectoriels) correspondent à des visions différentes sur le parallélisme et la synchronisation du langage récursif-parallèle. En fait, la sémantique d'états hiérarchiques présentée dans la première partie correspond au niveau d'abstraction le plus élevé, et celle d'états vectoriels au niveau le plus bas, proche de la stratégie d'implémentation. Nous montrons comment il est possible de relier formelle-

ment les trois modèles, et ainsi d'énoncer en quel sens précis l'implémentation de RPC est correcte. Le critère que nous utilisons est une notion de τ-*simulation* (au sens Milner) préservant à la fois les propriétés de *sûreté* et la *terminaison*.

D'autre part, nous montrons comment raffiner ces trois modèles en incorporant l'interprétation des actions de base. Les modèles obtenus correspondent là aussi à des niveaux d'abstraction décroissants. Ce cadre permet de comprendre pourquoi toute propriété de sûreté ou de terminaison démontrée pour notre modèle d'états hiérarchiques est vraie de la réalisation effective. Ainsi les résultats obtenus dans la deuxième partie de cette thèse peuvent être appliqués aux programmes RPC réels tels qu'ils sont exécutés sur la machine parallèle de l'IPTC.

Première partie

Cadre général

Modèles formels

L'utilisation des modèles formels est une nécessité par exemple lorsqu'on désire prouver la validité de programmes et leur conformité vis-à-vis de leurs spécifications. Ce besoin apparaît dans le cas séquentiel, où il existe des techniques de preuve de programmes satisfaisantes, et s'accroît dans le cas concurrent en raison de l'apparition de phénomènes complexes dûs à l'interaction entre les parties concurrentes. Il n'est pas aisé de trouver de bonnes représentations pour certaines constructions de langages évolués, exprimant des aspects particuliers liés à la concurrence, comme les communications et(ou) les synchronisations.

Dans le cas des programmes parallèles, il existe de très nombreux modèles [FF84]. Les différents modèles formels de programmes et de processus parallèles ont été élaborés avec des objectifs parfois différents. Certains servaient comme instruments pour la recherche de propriétés fondamentales des calculs parallèles. Pour d'autres, la destination principale était de justifier de nouvelles constructions de programmation introduites dans des langages de programmation parallèle et de vérifier des méthodes nouvelles d'organisation des programmes parallèles. D'autres encore étaient utilisés pour le développement de méthodes d'analyse structurelle et sémantique des programmes parallèles, pour l'élaboration de systèmes automatiques de vérification, de synthèse et d'optimisation de programmes. C'est tout à fait naturel qu'une telle variété d'objectifs conduise à la variété de modèles.

Modèles comportementaux

Une famille particulière est celle des modèles *comportementaux*, qui s'intéressent à la description du comportement du programme modélisé plus qu'à sa structure. Des exemples classiques sont les traces de Hoare [Hoa85], les arbres de synchronisation de Milner [Mil80], les structures d'événements de [NPW81], etc.

D'autres modèles (parmi lesquels les réseaux de Petri [Pet62], les systèmes de transitions [Kel76], les algèbres de processus [Mil89, BK86, Hoa85, BW90]) peuvent être lus à différents niveaux d'abstraction. En particulier, il est

fréquent de les munir d'une *équivalence comportementale*, par exemple la bisimulation, qui en fait des modèles comportementaux.

En pratique, les *systèmes de transitions* sont un des modèles du parallélisme les plus utilisés pour décrire des comportements concurrents. Il existe de nombreuses études et réalisations qui donnent une sémantique à un langage parallèle avec les systèmes de transitions pour modèle, y compris CCS de Milner [Mil89].

Si nous nous en tenons à des sémantiques comportementales, où la définition du programme est donnée en fonction des actions qu'il peut effectuer, nous pouvons considérer deux grandes familles d'approches (voir par exemple [Pnu85, Gla90, WN92] pour une présentation détaillée de ces approches et les motivations associées).

La première est dite du *temps linéaire*: le comportement du système est considéré comme une suite d'actions, et la sémantique est donnée en terme d'ensembles de suites qui peuvent être générées (par exemple sémantique de traces [Hoa85]). La seconde est celle du *temps arborescent*: les systèmes ne sont plus identifiés seulement en terme de séquences d'actions qu'ils peuvent effectuer, mais aussi en fonction du moment où sont effectués les choix entre plusieurs comportements possibles. Cette deuxième famille recouvre des sémantiques plus fines que celle du temps linéaire, et notamment les équivalences de *bisimulation*, qui demandent aux systèmes de pouvoir se simuler, mais aussi de pouvoir effectuer les mêmes choix aux mêmes moments.

Comme nous l'avons mentionné dans l'introduction de cette thèse, nous suivons cette deuxième approche et construisons un domaine sémantique basé sur des systèmes de transitions modulo (diverses) bisimulation.

Cette partie de la thèse est organisée de la façon suivante. Le chapitre 2 présente une description des particularités du langage de programmation parallèle RPC. Les chapitres 3, 4 et 5 introduisent les différentes notions théoriques (principalement liées aux systèmes de transitions) que nous utiliserons. Nous mettons l'accent sur les concepts utiles pour cette thèse mais n'hésitons pas à déborder du cadre fixé pour présenter quelques points d'intérêt.

Chapitre 2

Le langage RPC

2.1 Introduction

Comme nous l'avons mentionné dans l'introduction, le parallélisme est défini comme la faculté d'exécuter différentes actions simultanément, en les synchronisant éventuellement.

En général, il existe deux possibilités pour déterminer des activités (potentiellement) parallèles d'un problème à exécuter par un système multiprocesseur:

1. soit c'est un programmeur qui peut indiquer des actions parallèles au moyen d'instructions spéciales d'un langage de programmation;

2. soit c'est un système qui peut inférer des opportunités pour le parallélisme sur lui-même.

Dans le premier cas, l'écriture de programmes parallèles devient plus complexe car il faut trouver une façon d'attribuer une tâche à chaque processeur d'un système composé et il faut savoir "ramasser" les résultats dont on a besoin. C'est cette première approche qui est à la base de recherches à l'IPTC destinées à implémenter le RP-style.

Pour pouvoir écrire des programmes parallèles on a besoin de langages parallèles spécialisés, de haut-niveau. Quoique les langages de la programmation parallèle diffèrent considérablement l'un de l'autre, ils doivent présenter trois grands traits en communs:

- la capacité à exprimer une exécution parallèle,

- la possibilité d'une synchronisation d'activités parallèles, et

- la possibilité pour que des activités parallèles puissent communiquer.

Quant à l'exécution parallèle, l'état actuel de la question offre quatre principales possibilités dans la littérature pour la mettre en oeuvre.

- La première et la plus simple d'entre elles, utilise des *coroutines*. Une coroutine est une composante élémentaire d'un programme ainsi qu'une procédure (en anglais: routine). Une coroutine diffère d'une procédure: l'invocation d'une coroutine déclenche un comportement "en parallèle" et donc le contrôle passe immédiatement à l'instruction qui suit l'invocation, ceci tandis que la coroutine s'exécute en parallèle. Les coroutines utilisent la mémoire partagée pour des calculs et se synchronisent par la mémoire commune. Les coroutines sont présentées dans des langages tels que *Simula* et *Modula-2* [Wir78].

- En second lieu, les primitives *fork* et *quit* utilisées dans les premières implémentations des langages parallèles, permettent d'interrompre un processus en cours d'exécution et d'en faire des copies ou bien quasi-copies (dans son état actuel). L'effet est qu'à partir du fork deux processeurs exécutent le même programme en même temps: chacun a son propre flot de contrôle. Puisque chacun des deux peut effectuer un fork de nouveau, cette technique de programmation est connue comme un *multithreading*. Ces primitives furent introduites dans [DH66]. Cet article a introduit aussi la primitive *join* dont on a besoin pour réunir deux activités (c.-à-d. attendre quand un processus se termine pour qu'un autre puisse continuer). Le multithreading est coûteux à réaliser, sa difficulté principale est de faire des copies de la mémoire.

  Des variantes de fork, quit et join sont utilisées dans de nombreux systèmes d'exploitation; par exemple dans UNIX [RT74], où ils sont nommés *fork*, *exit* et *wait*. Des primitives similaires sont inclues dans PL/I et plus récemment, dans *Modula-3* [CDG+89].

- Une solution du problème de copies de la mémoire du multithreading a été proposée par E. Dijkstra: elle consiste à assurer qu'après avoir effectué le fork les deux processeurs exécutent des blocs complètement différents du programme, sans communiquer entre eux. La construction *Cobegin ... Coend* (ou *Parbegin ... Parend*) introduite par Dijkstra est utilisée dans *Communicating Sequential Processes* [Hoa78], *Edison*, *Occam* [May83] et *Argus* [LHG86].

- Finalement, la *déclaration explicite de processus* est une façon d'exprimer une activité parallèle dans la syntaxe d'un langage de la programmation parallèle. Typiquement, les processus utilisent la mémoire partagée; ils ont souvent les noms (ou les adresses). La déclaration

explicite de processus peut être trouvée dans le *Concurrent Pascal* et *Modula*.

[FF84] mentionne que les activités parallèles communiquent en utilisant des *données partagées* ou des *messages*. Une communication par des données partagées est liée à une synchronisation dont on trouve deux formes dans les programmes parallèles: l'*exclusion mutuelle* et la *synchronisation de conditions*. On parle d'exclusion mutuelle quand on exige que les sections critiques d'instructions ne soient pas exécutées en même temps. Il s'agit d'une synchronisation de conditions quand on exige qu'une activité parallèle attende jusqu'à ce qu'une condition donnée soit vraie. Une communication par des messages peut être synchrone ou asynchrone; ces deux formes diffèrent de sorte que dans la transmission synchrone, l'émetteur et le récepteur du message exécutent simultanément l'action de communication.

La pratique de la programmation parallèle montre que, dans un grand nombre de cas, un langage de programmation reflète un concept qui réunit une exécution parallèle, une synchronisation et une communication.

En particulier, le RP-style [Pan89] de programmation décrit dans le chapitre 1, est reflété par un des langages de programmation parallèle **RPC** que nous décrivons dans la section suivante.

2.2 Présentation de (processus) RPC

Le langage RPC (pour **Recursive Parallel C**) de [VEKM94] est un langage parallèle impératif supportant le parallélisme et la récursivité et ayant une discipline parallèle particulière. Cette section donne les particularités du langage "RPC pur".

2.2.1 Exemple introductif

Tout d'abord, nous allons donner un exemple d'un programme récursif-parallèle $Sum(i,j)$ qui calcule une somme $\sum_{l=i}^{j} F(l)$ de résultats d'une fonction $F()$ et range le résultat à l'adresse s.

Pour calculer une somme $\sum_{i}^{j} F(l)$ de résultats d'une fonction $F()$, on peut par exemple découper le travail à effectuer en deux et calculer (en parallèle) la somme des valeurs de $F()$ de i à $\lfloor (i+j)/2 \rfloor$ et la somme de $\lfloor (i+j)/2 \rfloor + 1$ à j.

C'est ce que font les deux invocations

```
pcall Sum(i,(i+j)/2,*s1);
pcall Sum((i+j)/2+1,j,*s2);
```

```
Parallel Sum(int i,j,float *s)
{
  if (i==j)
   then *s=F(i);
   else
   {
    float *s1,*s2;
    pcall Sum(i,(i+j)/2,*s1);
    pcall Sum((i+j)/2 + 1,j,*s2);
    wait();
    *s=*s1+*s2;
   }
}
```

Figure 2.1 : Un programme qui calcule une somme $\sum_{l=i}^{j} F(l)$

Ensuite, il faut additionner les deux sommes obtenues. Il est évident qu'on doit attendre la terminaison de chacune des deux parties avant de faire cette addition. D'où l'instruction `wait()`; avant l'addition. On voit aisément que chacune des deux moitiés peut être séparée de nouveau en deux et ainsi de suite.

Il est très commode et naturel de présenter l'idée de cet algorithme comme une procédure récursive qui utilise une forme de parallélisme (`pcall` pour un `call` parallèle) et de synchronisation (`wait` pour attendre des résultats indispensables).

Il convient de noter que notre exemple présente un programme où on effectue autant d'invocations de la procédure récursive, qu'on a de valeurs $F()$ à additionner. C'est seulement après ce découpage du travail que la récursion commence sa "marche arrière" et que des invocations "filles" fournissent leurs résultats à des invocations "parentales". Là, on a besoin d'une forme de synchronisation permettant d'additionner les résultats des invocations "filles". Cet assemblage continue jusqu'au moment où on obtient le résultat final.

2.2.2 Particularités de RPC

Comme nous l'avons mentionné dans la section précédente, le langage RPC permet au programmeur d'indiquer des actions parallèles au moyen d'instructions spéciales. Le **parallélisme d'une exécution** est exprimé en

utilisant un concept de *coroutines* ainsi qu'une discipline de type "fork/join". Ainsi un programme RPC est un ensemble de procédures parallèles qui peuvent s'invoquer récursivement (voir l'exemple d'un programme de la figure 2.1). L'invocation (via `pcall`) d'une procédure récursive-parallèle donne lieu au déclenchement d'un processus *fils* concurrent, placé hiérarchiquement sous le contrôle du processus *père* qui l'a invoqué.

Les spécificités du langage tiennent essentiellement à son choix de primitives pour la **synchronisation**. Les processus fils s'exécutent librement mais leur père peut demander, via une instruction spéciale de synchronisation, le `wait`, d'attendre la terminaison de tous ses fils. La terminaison d'une invocation donnée peut être obtenue par l'instruction `end`. A contrario, les processus fils ne peuvent pas savoir si leur père est terminé. Notons aussi que l'utilisation de l'instruction `wait` n'est pas obligatoire: le processus père peut aussi choisir de terminer et laisser ses fils en marche.

En ce qui concerne la gestion de la mémoire, le système de l'IPTC est celui d'une architecture multiprocesseur à mémoire commune globale d'accès direct et à mémoire distribuée locale. Comme les coroutines, les procédures parallèles de RPC utilisent la mémoire partagée pour exécuter des calculs: une allocation d'une procédure parallèle sur un des processeurs du système permet d'utiliser la mémoire distribuée (locale de ce processeur) pour les variables locales de cette procédure. Comme les coroutines, les procédures parallèles de RPC se synchronisent par la mémoire commune où les variables globales se trouvent.

Sans vouloir entrer dans les détails (pour lesquels nous renvoyons à [Pan89, VEKM94]), nous disons que ce langage de programmation est une extension du langage C qui est étendu par appels de macros spéciales du RP-système incluant une discipline de parallélisme de type "fork/join". RPC fournit une possibilité de décrire des processus (parallèles) en les construisant hiérarchiquement à partir de processus plus simples.

Le langage RPC satisfait les exigences suivantes:

- les idées présentées ci-dessus sont réalisées en ajoutant des macros à une implémentation de C de sorte qu'un programme écrit dans RPC peut être traduit par un compilateur standard du langage C dans le code parallèle à exécuter par un RP-système ainsi que dans le code séquentiel à exécuter par un ordinateur traditionnel;

- c'est un moyen de recherche sur le parallélisme des programmes (ou de

leurs modèles) et sur l'efficacité du fonctionnement du système récursif-parallèle exécutant ces programmes (ou leurs modèles).

```
struct BP_Sum
{    int i,j,k;
      float s;
}


Parallel(Sum,bp)
PARAM (BP_Sum,*bp);
{  NEW_PARAM (BP_Sum,bp1);
   NEW_PARAM (BP_Sum,bp2);
   int m;
   if ((bp→i)-(bp→j)<(bp→k))
     then
     {
      bp→s=0;
      for (m=bp→i;m<bp→j;m++)
      bp→s+=F(m);
     }
     else
     {
      bp1.i=bp→i; bp1.k=bp→k;
      bp1.j=(bp→i + bp→j)/2;
      bp2.i=bp1.j+1;
      bp2.j=bp→j; bp2.k=bp→k;
      P_Call(Sum,&bp1);
      P_Call(Sum,&bp2);
      Wait();
      bp→s=bp1.s+bp2.s;
     }
}
```

Figure 2.2 : Un programme RPC calculant une somme $\sum_{l=i}^{j} F(l)$

Le langage RPC comme sous-ensemble du langage C a certaines restrictions qui sont liées, d'abord, à la nécessité de prendre en compte deux types de mémoire ce qui doit être reflété dans un RP-programme, ensuite à des conventions spéciales sur une transmission de blocs de paramètres entre des

procédures parallèles. La figure 2.2 présente le programme écrit en RPC effectuant le même calcul que le programme de la figure 2.1. Dans la figure 2.2, la procédure `Sum` est décrite comme parallèle; son bloc de paramètres `bp` est une structure de type `BP_Sum`. La procédure `Sum` décrit et forme les blocs de paramètres `bp1` et `bp2` de type `BP_Sum` pour deux invocations parallèles.

Parmi les restrictions sur le langage C on remarque les suivantes (qui sont motivées par l'utilisation d'un compilateur standard C):

- ce sont seulement des procédures qui peuvent être décrites parallèles;
- une procédure parallèle doit être invoquée par une instruction spéciale RPC;
- les paramètres des procédures parallèles doivent être décrits au moyen d'une structure (qui peut être vide), et une instruction d'invocation utilise un pointeur sur un bloc de paramètres;
- une procédure parallèle peut être invoquée en mode séquentiel et exécutée sur le même processeur, grâce à une instruction spéciale RPC.

Quant à la **communication**, qui est un des traits d'un langage parallèle soulignés dans la section précédente, le langage RPC tel qu'il a été implémenté autorise l'accès à la mémoire commune au moyen des macros spéciales d'accès qui déclenchent des processus parallèles pouvant être synchronisés au moyen d'une instruction `wait`. Ce fait permet par exemple de lire/écrire des données en parallèle avec un traitement d'une autre partie des données.

Un point important est à noter: évidemment, la façon de partager un travail en beaucoup de parties très petites (voir figure 2.1) n'est pas toujours bonne du point de vue d'une organisation des communications mais ce problème ne fait pas partie de notre recherche, et en outre il existe des moyens [VEKM94] pour organiser un calcul de façon efficace.

Nous avons présenté des particularités du langage RPC, et notre description ne va pas au-delà de ce qui sera nécessaire au chapitre 6. On pourra consulter [Pan89, VKT90, VEKM94] pour plus de détails sur le langage RPC, et [MVVK88, VEKM94] sur l'implémentation actuelle de RPC et sur le système multiprocesseur de l'IPTC où un compilateur RPC a été développé.

Chapitre 3

Modèles comportementaux pour le parallélisme

3.1 Préliminaires

Nous précisons simplement ici les quelques notations de base que nous utiliserons sur les ensembles et les relations.

Notation 3.1.1. *Soient X et Y deux ensembles, nous notons $X \subseteq Y$ si X est un sous-ensemble de Y et $X \subset Y$ si X est un sous-ensemble propre de Y.*

$|X|$ est le cardinal *de X.*

Nous notons par $X \cup Y$ l'union de X et Y et par $X \cap Y$ l'intersection de X et Y. Nous notons par $\mathbb{N}$ l'ensemble des entiers naturels.

Notation 3.1.2. *(Relations)*
Une relation R entre X et Y est un sous-ensemble de $X \times Y$. Nous utiliserons les notations suivantes:

- $Id_X \stackrel{\text{déf}}{=} \{(x,x) | x \in X\}$ *est la relation d'identité sur X;*
- $R^{-1} \stackrel{\text{déf}}{=} \{(y,x) | (x,y) \in R\}$ *est l'inverse de R; $R^{-1} \subseteq Y \times X$;*
- $R \circ R' \stackrel{\text{déf}}{=} \{(x,z) | \exists\, y : (x,y) \in R' \text{ et } (y,z) \in R\}$ *est la composition des relations R et R';*

Pour une relation sur X (i.e. entre X et X), on note

- R^k *, pour $k \in \mathbb{N}$, la k-ième puissance de R, définie récursivement par*

$$R^0 \stackrel{\text{déf}}{=} Id_X,$$
$$R^{k+1} \stackrel{\text{déf}}{=} R^k \circ R;$$

- $R^+ \stackrel{\text{déf}}{=} R^1 \cup R^2 \cup \ldots$, *la fermeture transitive de* R*;*
- $R^* \stackrel{\text{déf}}{=} Id_X \cup R^+$, *la fermeture réflexive et transitive de* R*.*

Nous allons maintenant rappeler une série de définitions et de propriétés classiques concernant les relations d'ordre.

Définition 3.1.3. *Un* préordre $(X, \leq)$ *est constitué par un ensemble* X *et une relation réflexive et transitive sur l'ensemble* X*. Un* ordre *est un préordre antisymétrique.*

Le cas particulier des *beaux préordres* sera très utilisé:

Définition 3.1.4. *(Beau préordre)*
Un préordre $\leq$ *sur un ensemble* X *est* beau *ssi de toute suite infinie d'éléments de* X*, on peut extraire une sous-suite infinie croissante c.-à-d.*

$$(\forall x_0, x_1, x_2, \ldots) \quad (\exists\ i_0 < i_1 < i_2 < \ldots)\ t.q.\ (x_{i_0} \leq x_{i_1} \leq x_{i_2} \leq \ldots).$$

Il convient de noter qu'un beau préordre est bien fondé, c.-à-d. qu'il n'existe pas de suites infinies strictement décroissantes. En général la réciproque n'est pas vraie. Par exemple l'ordre de divisibilité sur $\mathbb{N}$ est bien fondé mais n'est pas un bel ordre (cf. la suite des nombres premiers).

3.2 Les mots

Les mots (séquences de lettres) jouent un rôle fondamental en informatique. Nous considérerons des mots finis et infinis [Tho90].

On considère un ensemble $A = \{a, b, \ldots\}$, appelé *alphabet* (c.-à.-d., un ensemble fini), dont les éléments s'appellent des *lettres*. Un *mot fini* sur A est une suite finie $w = a_1 \ldots a_n$ de lettres de A. Le mot vide est noté ϵ. La longueur d'un mot w est notée $|w|$ et $|\epsilon| = 0$. Un *mot infini* sur A est une suite infinie $w = a_1 \ldots a_n \ldots$. La longueur d'un mot infini w est $|w| \stackrel{déf}{=} \omega$.

Notation 3.2.1. *On note* A^* *l'ensemble des mots finis sur* A*, et* A^ω *l'ensemble des mots infinis.* $A^\infty \stackrel{\text{déf}}{=} A^* \cup A^\omega$.

La définition suivante généralise la notion classique de concaténation de mots finis au cas infini.

Définition 3.2.2. *La concaténation $w \cdot w'$ de deux mots $w = a_1 a_2 \ldots$ et $w' = b_1 b_2 \ldots$ de A^∞ est définie par*

$$w \cdot w' \stackrel{\text{déf}}{=} \begin{cases} a_1 \ldots a_{|w|} b_1 \ldots, & \text{si } w \text{ est fini;} \\ w & \text{sinon.} \end{cases}$$

Nous rappelons la notion du plongement entre les mots finis, défini comme suit:

Définition 3.2.3. *(Plongement entre les mots finis)*
Soient $w = a_1 a_2 \ldots a_n$ et $w' = b_1 b_2 \ldots b_m$ deux mots de A^*, on dit que w se plonge dans *w' (on note $w \ll w'$) ssi $\exists j_1, \ldots, j_i, \ldots, j_n$ $(1 \leq j_1 < \ldots < j_i < \ldots < j_n \leq m)$ tels que $a_i = b_{j_i}$ pour tout $i = 1, \ldots, n$.*

Autrement dit, w peut être obtenu en effaçant certaines lettres de w'. Par exemple $abcd \ll abracadabra$. Le plongement est une relation réflexive, transitive et antisymétrique, i.e. un ordre, sur A^*.

Nous rappelons le résultat fondamental de Higman

Théorème 3.2.4. *[Hig52]*
Le plongement sur les mots d'un alphabet A fini est un bel ordre.

Nous utiliserons son extension aux arbres, le théorème de Kruskal [Kru60], dans le chapitre 8.

3.3 Les multi-ensembles

Dans cette section, nous allons rappeler la définition et certaines propriétés des multi-ensembles. L'intuition est que, dans un multi-ensemble, un même élément peut apparaître plusieurs fois.

Définition 3.3.1. *(Multi-ensembles)*
Soit X un ensemble, un multi-ensemble sur X est une application M de X dans l'ensemble $\mathbb{N}$ des entiers naturels, $M : X \to \mathbb{N}$. Pour tout $x \in X$ nous appellerons $M(x)$ la multiplicité *de x dans M, et l'on dit que $x \in M$ quand $M(x) > 0$.*

Nous dirons qu'un multi-ensemble est fini *ssi pour tout $x \in X$, $M(x) = 0$ sauf pour un nombre fini d'entre eux.*

Notation 3.3.2. *La cardinalité d'un multi-ensemble fini est définie par*

$$|M| \stackrel{\text{déf}}{=} \sum_{x \in M} M(x).$$

Le multi-ensemble de cardinalité nulle est noté de la même manière que l'ensemble vide: $\emptyset$.

Nous notons $\mathcal{U}(X)$ *l'ensemble de tous les multi-ensembles finis sur* X.

Quand $M(x) \leq 1$ pour tout $x \in M$, M est un ensemble classique. Nous étendons aux multi-ensembles certaines des opérations classiques sur les ensembles dont nous aurons besoin par la suite:

$$\begin{array}{ll} \forall M, M' \in \mathcal{U}(X), \forall x \in X, & (M + M')(x) \stackrel{déf}{=} M(x) + M'(x) \\ & (M - M')(x) \stackrel{déf}{=} max(0, M(x) - M'(x)) \\ & M \subseteq M' \text{ ssi } \forall x \in X,\ M(x) \leq M'(x) \end{array}$$

Exemple 3.1.
Soient $M = \{1, 1, 3\}$ et $M' = \{1, 2, 2, 3\}$ deux multi-ensembles sur $\mathbb{N}$, alors on a:

- $M + M' = \{1, 1, 1, 2, 2, 3, 3\}$
- $M - M' = \{1\}$
- $M \not\subseteq M'$ et $M' \not\subseteq M$

3.4 Systèmes de transitions (étiquetés)

Les *systèmes de transitions étiquetés* [Kel76] constituent un modèle de base permettant de donner une sémantique opérationnelle à une variété de langages [Plo81]. Les concepts utilisés sont très simple: une notion d'état, et une notion de changement d'état (les transitions).

Nous allons introduire la notion de systèmes de transitions par

Définition 3.4.1. *(Système de transitions (étiqueté))*
Un système de transitions (étiqueté) *est une structure*

$$\mathcal{A} = (St_{\mathcal{A}}; Act_{\mathcal{A}}; \rightarrow_{\mathcal{A}}), \text{ où}$$

- $St_{\mathcal{A}}$ *est un ensemble* $\{q_0, q_1, \ldots\}$ *d'*états,
- $Act_{\mathcal{A}} = \{\alpha, \beta, \gamma, \ldots\}$ *est un ensemble de* noms d'actions,
- $\rightarrow_{\mathcal{A}} \subseteq St_{\mathcal{A}} \times Act_{\mathcal{A}} \times St_{\mathcal{A}}$ *est la* relation de transition *entre les états de* $\mathcal{A}$.

Un système de transitions est fini *si $St_{\mathcal{A}}$ et $Act_{\mathcal{A}}$ sont finis.*

Lorsque le système de transitions $\mathcal{A}$ est sous-entendu, on notera directement St, Act et $\rightarrow$ au lieu de $St_{\mathcal{A}}$, $Act_{\mathcal{A}}$ et $\rightarrow_{\mathcal{A}}$. Nous noterons en abrégé ST pour "système de transitions".

Dans la suite **ST** désigne la classe des ST (étiquetés sur Act).

En pratique on enrichit souvent la structure des systèmes de transitions étiquetés avec des décorations supplémentaires: par exemple état initial, relation d'indépendance entre les événements, durée des événements, etc. ce qui donne lieu à des notions de systèmes de transitions *enracinés*, *asynchrones* [Shi85], *temporisés* [GRS95], etc.

Par la suite, on va utiliser les notations suivantes:

Notation 3.4.2.

- $p \xrightarrow{\alpha} q \stackrel{\text{déf}}{=} (p, \alpha, q) \in \rightarrow$
- $p \xrightarrow{\alpha} \stackrel{\text{déf}}{=} \exists\, q :\ p \xrightarrow{\alpha} q$
- $p \rightarrow q \stackrel{\text{déf}}{=} \exists\, \alpha :\ p \xrightarrow{\alpha} q$
- $p \rightarrow \stackrel{\text{déf}}{=} \exists\, \alpha :\ p \xrightarrow{\alpha}$

Une *transition* $q \xrightarrow{\alpha}_{\mathcal{A}} q'$ signifie que l'on peut passer de l'état q à l'état q' en réaction[1] à l'événement α. L'état q' est appelé un *α-successeur*, ou *successeur*, de q. Un état sans successeur est dit *état terminal*. Cette terminologie exprime que, dans un tel état, le système ne peut plus évoluer.

Les transitions peuvent être enchaînées. Pour $w \in Act^*$, de la forme $a_1 \ldots a_n$, on note $q_0 \xrightarrow{w} q_n$ l'existence d'une suite $q_{i-1} \xrightarrow{a_i} q_i$ (en particulier on a toujours $q \xrightarrow{\epsilon} q$) et on étend à cette relation la notation "$q \xrightarrow{w}$".

Définition 3.4.3. *(Atteignabilité)*
Un état q de $\mathcal{A}$ sera dit atteignable à partir de q' *s'il existe un mot w (éventuellement vide) de Act^* tel que $q' \xrightarrow{w} q$. Il est* atteignable *(simplement) si il est atteignable à partir d'un état initial. On note $Att(\mathcal{A})(\subseteq St_{\mathcal{A}})$ l'ensemble des états atteignables de $\mathcal{A}$.*

[1] ou "en donnant lieu à", ou "par le biais de", ... suivant l'interprétation donnée aux événements

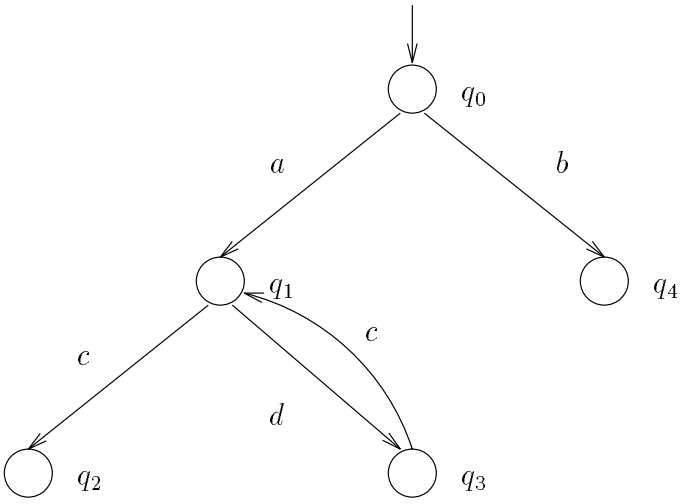


Figure 3.1 : Un système de transitions étiqueté

Exemple 3.2. Exemple d'un système de transitions étiqueté.

La figure 3.1 donne l'exemple d'un système de transitions étiqueté $\mathcal{A}$. Dans cette exemple, $St = \{q_0, q_1, q_2, q_3, q_4\}$ et $Act = \{a, b, c, d\}$. La relation $\rightarrow_{\mathcal{A}}$ est décrite par des arcs orientés entre les états, les étiquettes des transitions sont indiquées à côté des arcs qui les représentent, et formellement,

$$\rightarrow_{\mathcal{A}} = \{(q_0, a, q_1), (q_0, b, q_4), (q_1, c, q_2), (q_1, d, q_3), (q_3, c, q_1)\}.$$

L'état initial de $\mathcal{A}$ est l'état q_0, ce que nous représenterons graphiquement par une flèche incidente vers cet état.

Nous avons explicitement considéré les systèmes de transitions qui permettent de modéliser les systèmes concurrents par un formalisme séparant actions et états. De cette manière, la structure et le comportement des systèmes sont décrits dans le même formalisme.

3.5 Non-déterminisme des systèmes de transitions

La définition 3.4.1 ne fait aucune restriction sur les transitions possibles. En particulier, les systèmes de transitions sont intuitivement interprétés comme des automates *non-déterministes*, où la seule manière de représenter le parallélisme est le *non-déterminisme séquentiel*. De fait, un système essentiellement déterministe mais parallèle sera modélisé comme un ST non-déterministe.

Les systèmes de transitions sont des modèles dits d'*entrelacement*, ou *interleaving*: le parallélisme entre deux événements est représenté par un non-déterminisme séquentiel. Ainsi, le comportement "a et b en parallèle" sera représenté par un choix non-déterministe entre les séquences "a suivi de b" et "b suivi de a".

En pratique, comme par exemple dans le langage CCS avec récursivité gardée, des nombreux programmes ne donnent lieu qu'à un type restreint de non-déterminisme: le *branchement fini*, qui a des propriétés mathématiques très intéressantes comme nous le verrons dans les sections 6.2 et 8.3.

Définition 3.5.1. *(Système de transitions à branchement fini)*
$\mathcal{A} \in \mathbf{ST}$ *est à* branchement fini, *noté "*$\mathcal{A}$ b.f.*" en abrégé, si pour tout* $q \in St$, *l'ensemble* $\{q' \in St \mid q \xrightarrow{\alpha} q'\}$ *est fini.*

Informellement, pour de tels systèmes, le nombre de choix possibles pour effectuer une action dans chaque état est fini.

Si $\mathcal{A} \in \mathbf{ST}$ n'est pas à b.f. nous disons alors qu'il est à branchement infini.

3.6 Chemins et traces

Nous introduisons à présent la notion d'*exécution* dans un système de transitions. Une exécution correspond à un calcul du ST. Puisque les systèmes sont en général non-déterministes, plusieurs exécutions sont possibles à partir d'un même état. De plus, puisque le système n'a pas forcément pour vocation de se terminer un jour, les exécutions peuvent être infinies.

Soient $\mathcal{A} \in \mathbf{ST}$ et $q \in St_{\mathcal{A}}$.

Définition 3.6.1. *(Chemins et exécutions)*
Un chemin *dans* $\mathcal{A}$ *partant de* q *est une séquence de transitions de la forme* $q \xrightarrow{\alpha_1}_{\mathcal{A}} q_1 \xrightarrow{\alpha_2}_{\mathcal{A}} q_2 \ldots$.

Une exécution *partant de* q *est un chemin maximal partant de* q. *Il peut être fini (et s'achever dans un état terminal), ou infini.*

Nous notons $\Upsilon_{\mathcal{A}}(q)$ *(ou plus simplement* $\Upsilon(q)$*) l'ensemble de tous les chemins dans* $\mathcal{A}$ *qui partent de l'état* q, *et* $Ex_{\mathcal{A}}(q)$ *(ou plus simplement* $Ex(q)$*) l'ensemble des exécutions qui partent de* q.
$Ex(\mathcal{A})$ *désigne* $Ex(q_0)$, *où* q_0 *est un état initial.*

Nous utilisons les symboles $\sigma, \sigma_1, \ldots$ *pour désigner des chemins dans* $\mathcal{A}$, *et* $\pi, \pi_1, \ldots$ *pour désigner les exécutions.*

Les opérations sur les chemins héritent des notations utilisées pour les mots (cf. page 36). Etant donné $\sigma = q_0 \stackrel{\alpha_1}{\rightarrow}_{\mathcal{A}} q_1 \stackrel{\alpha_2}{\rightarrow}_{\mathcal{A}} q_2 \ldots$ un chemin dans $\mathcal{A}$, $|\sigma|$ dénote sa longueur, i.e. le nombre de transitions de σ; on pose $|\sigma| = \omega$ si σ est infini.

Pour chaque calcul effectué par le système, on extrait une observation externe constituée de la suite des actions qui ont eu lieu. Ce mot (éventuellement infini) est appelée une *trace* (du chemin) et se définit formellement par:

Définition 3.6.2. *(Trace)*
*Soit $\mathcal{A} \in$ **ST**. Etant donné un chemin $\sigma = q_0 \stackrel{\alpha_1}{\rightarrow} q_1 \stackrel{\alpha_2}{\rightarrow} q_2 \ldots$ dans $\mathcal{A}$, on définit sa* trace, *notée $tr(\sigma)$, par*

$$tr(\sigma) \stackrel{\text{déf}}{=} \alpha_1 \alpha_2 \ldots$$

La trace d'une exécution est une trace complète *du ST. Pour tout $q \in St_{\mathcal{A}}$, nous définissons l'ensemble des traces complètes dans $\mathcal{A}$ partant de q par*

$$Tr_{\mathcal{A}}(q) \stackrel{\text{déf}}{=} \{tr(\pi) \,|\, \pi \in Ex(q)\}$$

*Pour un ST $\mathcal{A} \in$ **ST** avec état initial q_0, on définit l'ensemble des traces complètes de $\mathcal{A}$ comme les traces des exécutions partant de q_0:*

$$Tr(\mathcal{A}) \stackrel{\text{déf}}{=} Tr_{\mathcal{A}}(q_0)$$

Lorsqu'il n'y a pas d'ambiguïté, on note $Tr(q)$ au lieu de $Tr_{\mathcal{A}}(q)$.

Une trace complète est donc un mot de Act^{∞}. Il s'agit de l'analogue, pour les systèmes de transitions, de la notion bien connue de langage reconnu (ou engendré) pour les automates. Dans la suite de cette thèse, on écrira souvent "trace" au lieu de "trace complète".

3.7 Equivalence de traces

Cette section introduit l'*équivalence de traces* de [Hoa85] qui est une notion fondamentale et qui nous servira au cours d'une étude d'expressivité de notre modèle. Prenons un système $\mathcal{A}$ qui évolue par changement d'états, et que l'on peut interrompre à tout instant. L'observation d'un tel système consiste simplement en une séquence d'actions qu'il peut effectuer. L'équivalence de traces identifie les systèmes qui admettent le même ensemble d'observations. Formellement,

Définition 3.7.1. *Deux états q_1, q_2 ont les mêmes traces, noté $q_1 \sim_{Tr} q_2$, si $Tr(q_1) = Tr(q_2)$.*

Deux ST $\mathcal{A}_1, \mathcal{A}_2$ ont les mêmes traces, noté $\mathcal{A}_1 \sim_{Tr} \mathcal{A}_2$, si les exécutions partant de leurs états initiaux ont les mêmes traces, i.e. $Tr(\mathcal{A}_1) = Tr(\mathcal{A}_2)$.

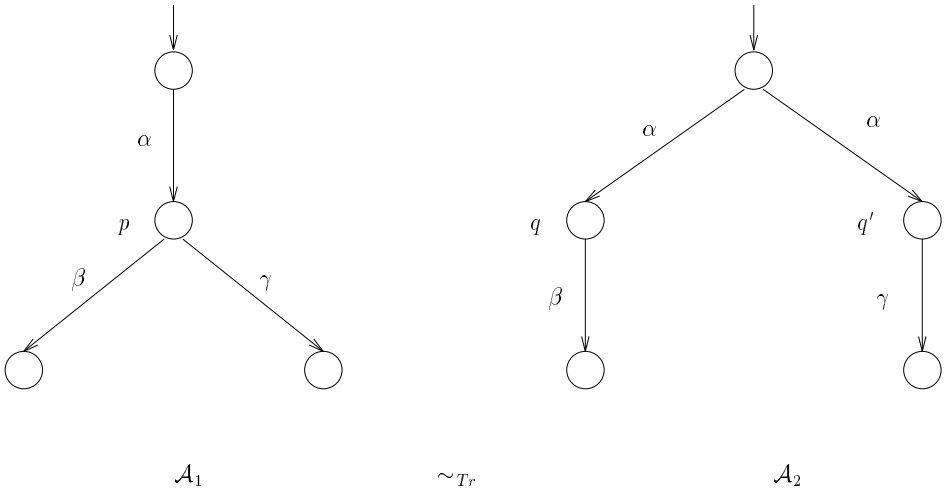


Figure 3.2 : Deux systèmes de transitions ayant mêmes traces

Il est clair que $\sim_{Tr}$ induit une équivalence sur **ST**.

Par exemple, les ST de la figure 3.2 admettent le même ensemble de traces $\{\alpha\beta, \alpha\gamma\}$. Néanmoins, il est clair que ces deux systèmes ne font pas les mêmes choix aux mêmes moments: les deux systèmes peuvent choisir entre effectuer $\alpha\beta$ ou $\alpha\gamma$, mais $\mathcal{A}_1$ choisit entre β et γ après avoir effectué α, tandis que $\mathcal{A}_2$ choisit avant.

Chapitre 4

Equivalences de bisimulation

Pour comparer les processus, nous avons besoin de notions d'équivalence plus fortes que l'équivalence de traces (i.e. de langages) et qui prennent en considération par exemple les notions comportementales de *deadlock, livelock, branchement, causalité*. Il existe beaucoup d'équivalences comportementales dans la littérature (cf. par exemple [Gla90]). En particulier, l'équivalence de *bisimulation forte*, introduite par Milner et Park (cf. [Mil80, Par81]), a été largement étudiée et elle forme maintenant une base fondamentale pour l'étude de langages de type CCS [Mil89].

C'est cette famille d' équivalences que nous allons utiliser dans cette thèse.

Nous avons choisi la bisimulation pour plusieurs raisons:

- c'est une équivalence comportementale fondamentale;
- elle préserve les comportements arborescents et donc la plupart des propriétés dynamiques;
- une équivalence respecte le temps arborescent ssi elle est plus fine ou égale à la bisimulation ([Gla94]);
- c'est une congruence pour un grand nombre d'opérateurs (composition parallèle, séquentielle, etc.);
- elle fait abstraction de l'identité des transitions pour ne considérer que leur effet visible;
- il en existe de nombreuses variantes, par exemple pour les actions invisibles (ou silencieuses);
- elle est plus facile à manipuler que des équivalences du temps linéaire;

- elle a des liens forts avec les logiques temporelles [HM85];
- il existe une littérature abondante (par exemple [BK89]) et c'est maintenant une notion bien cernée.

Dans ce chapitre, nous introduisons les notions fondamentales dont nous aurons besoin par la suite.

Après avoir présenté la bisimulation et certaines des ses propriétés, nous rappelons également que la bisimulation et l'équivalence projective de Milner coïncident sur la classe des systèmes à branchement fini. Nous utiliserons la deuxième comme un moyen technique pour démontrer certains résultats.

Enfin, nous rappelons brièvement les variantes de l'équivalence de bisimulation proposées dans la littérature dans le cas où des actions silencieuses sont considérées.

4.1 Equivalence de bisimulation (forte)

Les systèmes de transitions présentés dans le chapitre précédent, s'ils permettent de représenter un système parallèle, n'offrent pas assez d'abstraction quand on ne s'intéresse qu'aux aspects comportementaux. Ainsi, deux systèmes de transitions non structurellement équivalents peuvent représenter des programmes ayant des comportements jugés équivalents. Une équivalence comportementale fondamentale sur ces modèles est la bisimulation forte [Mil80, Par81].

L'équivalence de *bisimulation forte* est reconnue comme une équivalence sémantique de base: elle se contente d'identifier des systèmes dont les comportements ont la même structure arborescente. Son nom vient de ce qu'elle définit la capacité de deux systèmes $\mathcal{A}_1$ et $\mathcal{A}_2$ à s'imiter mutuellement, c'est-à-dire que toute action effectuée par $\mathcal{A}_1$ peut être imitée par une action identique effectuée par $\mathcal{A}_2$, et réciproquement. D'autre part, les états atteints après ces actions doivent être bisimilaires, c'est-à-dire permettre aux systèmes d'avoir à nouveau des comportements bisimilaires.

On considère deux systèmes de transitions $\mathcal{A}_1$ et $\mathcal{A}_2$.

Définition 4.1.1. *(Propriété de transfert)*
Une relation $R \subseteq St_{\mathcal{A}_1} \times St_{\mathcal{A}_2}$ entre les états de $\mathcal{A}_1$ et $\mathcal{A}_2$ vérifie la propriété de transfert*, ssi pour tous pRq*

1. *pour toute transition $p \stackrel{\alpha}{\rightarrow}_{\mathcal{A}_1} p'$ il existe un état q' de $\mathcal{A}_2$ tel que $q \stackrel{\alpha}{\rightarrow}_{\mathcal{A}_2} q'$ et $p'Rq'$,*

2. *réciproquement, pour toute transition* $q \xrightarrow{\alpha}_{\mathcal{A}_2} q'$, *il existe un état* p' *de* $\mathcal{A}_1$ *tel que* $p \xrightarrow{\alpha}_{\mathcal{A}_1} p'$ *et* $p' R q'$.

Définition 4.1.2. *(Bisimulation (forte))*
Une relation $R \subseteq St_{\mathcal{A}_1} \times St_{\mathcal{A}_2}$ *ayant la propriété de transfert est une* bisimulation (forte) *entre* $\mathcal{A}_1$ *et* $\mathcal{A}_2$.

Si alors p *et* q *sont deux états tels que* pRq, *on écrit* $R : p \underline{\leftrightarrow} q$. *On note* $R : \mathcal{A}_1 \underline{\leftrightarrow} \mathcal{A}_2$ *quand* R *relie les états initiaux de* $\mathcal{A}_1$ *et* $\mathcal{A}_2$.

On note $p \underline{\leftrightarrow} q$ (resp. $\mathcal{A}_1 \underline{\leftrightarrow} \mathcal{A}_2$) quand $R : p \underline{\leftrightarrow} q$ (resp. ...) pour une bisimulation R, et on dit que p et q (resp. ...) sont bisimilaires.

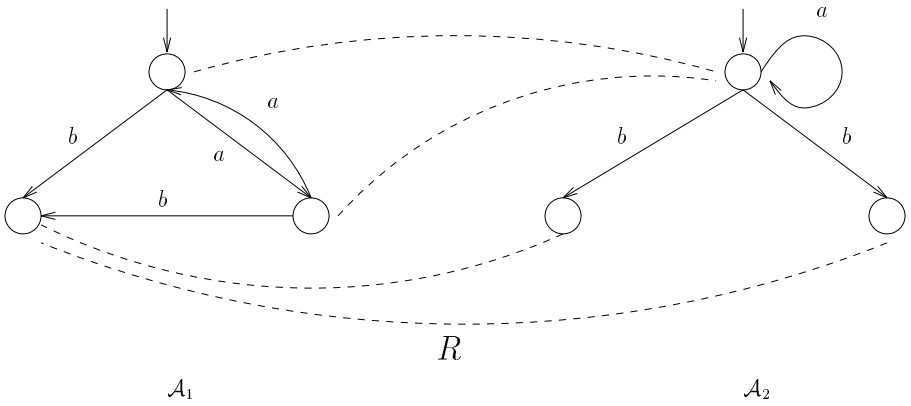


Figure 4.1 : Deux systèmes bisimilaires

Sur l'exemple de la figure 4.1, on a représenté par des lignes pointillées la relation R établissant la bisimilarité entre $\mathcal{A}_1$ et $\mathcal{A}_2$. Il est facile de vérifier que R a la propriété de transfert en considérant toutes les paires $p\,R\,q$.

La définition 4.1.2 nous assure des propriétés suivantes (cf. [Mil89]):

Proposition 4.1.3. *Pour tous systèmes de transitions* $\mathcal{A}, \mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3 \in \mathbf{ST}$,

1. $Id_{St_{\mathcal{A}}}$ *est une bisimulation entre* $\mathcal{A}$ *et lui-même;*

2. *si* R *est une bisimulation entre* $\mathcal{A}_1$ *et* $\mathcal{A}_2$, *alors* R^{-1} *est une bisimulation entre* $\mathcal{A}_2$ *et* $\mathcal{A}_1$;

3. *si* R, R' *sont deux bisimulations resp. entre* $\mathcal{A}_1$ *et* $\mathcal{A}_2$, *et entre* $\mathcal{A}_2$ *et* $\mathcal{A}_3$, *alors* $R' \circ R$ *est une bisimulation entre* $\mathcal{A}_1$ *et* $\mathcal{A}_3$;

4. *si pour tout* $i \in I$ (I *ensemble quelconque*) R_i *est une bisimulation entre* $\mathcal{A}_1$ *et* $\mathcal{A}_2$, *alors* $\bigcup_{i \in I} R_i$: *est une bisimulation entre* $\mathcal{A}_1$ *et* $\mathcal{A}_2$.

Il est facile de voir que les propriétés de 1 à 3 font de $\underline{\leftrightarrow}$ une relation d'équivalence entre les états (et entre les systèmes de transitions). 4 assure l'existence d'une plus grande bisimulation entre $\mathcal{A}_1$ et $\mathcal{A}_2$. En particulier, on peut définir la plus grande bisimulation entre $\mathcal{A}_1$ et $\mathcal{A}_2$ par:

$$\underline{\leftrightarrow} \stackrel{déf}{=} \bigcup \{R \subseteq St_{\mathcal{A}_1} \times St_{\mathcal{A}_2} \mid R \text{ a la propriété de transfert}\}$$

La propriété de transfert se généralise des transitions aux chemins. Le corollaire est

Proposition 4.1.4. *Pour tous* $\mathcal{A}_1, \mathcal{A}_2 \in$**ST**,

$$\mathcal{A}_1 \underline{\leftrightarrow} \mathcal{A}_2 \textit{ implique } \mathcal{A}_1 \sim_{Tr} \mathcal{A}_2$$

La réciproque est évidemment fausse: par exemple les systèmes de la figure 3.2 ne sont pas bisimilaires. (Voir [Gla90] pour plus de détails).

4.2 Décidabilité de la bisimulation

Certainement, la propriété d'être bisimilaires, la *bisimilarité*, est décidable sur les systèmes de transitions d'états finis. Pour ces systèmes on peut dénombrer les relations binaires sur l'ensemble d'états (fini) et chercher une bisimulation parmi eux contenant la paire (p, q) que nous voudrions comparer. Il est décidable qu'une relation R finie donnée a la propriété de transfert. Si on trouve une bisimulation contenant (p, q), on conclut que $p \underline{\leftrightarrow} q$, et réciproquement, si $p \underline{\leftrightarrow} q$, nous allons inévitablement trouver une bisimulation contenant p et q.

Si le système de transitions que nous considérons n'est pas d'états finis, la procédure naïve décrite ci-dessus ne suffit plus parce qu'il existe alors un nombre infini de relations binaires, elles-mêmes pouvant contenir un nombre infini de paires. En général le problème est alors indécidable, et seulement pour certains cas particuliers [CH93], il existe des méthodes de décision de la bisimilarité.

4.3 Equivalences projectives

Les équivalences projectives sont elles aussi basées sur un principe de bisimulation entre les systèmes. Simplement, la définition est inductive au lieu de reposer sur une notion de point fixe. Initialement, c'est la définition que Milner utilisa pour CCS dans [Mil80], avant que Park propose une définition beaucoup plus pratique [Par81] qu'on adopte aujourd'hui. Pour

nous, l'intérêt de l'équivalence projective ne réside pas dans de subtiles différences sémantiques [BBK87b]: c'est simplement un outil technique utile dans certaines preuves de bisimilarité.

Les définitions et les résultats sont extraits de [Mil80, Mil90].

Définition 4.3.1. *Soit $\mathcal{A} = (St; Act; \rightarrow)$ un système de transitions étiqueté. Pour $n \in \mathbb{N}$, on définit les relations $\equiv_n \subseteq St \times St$ par:*

- $q_1 \equiv_0 q_2$ *pour tous* $q_1, q_2 \in St$,
- $q_1 \equiv_{n+1} q_2$ *ssi*
  1. *pour tout* $q_1 \stackrel{\alpha}{\rightarrow} q_1'$, *il existe* $q_2 \stackrel{\alpha}{\rightarrow} q_2'$ *tel que* $q_1' \equiv_n q_2'$,
  2. *et réciproquement, pour tout* $q_2 \stackrel{\alpha}{\rightarrow} q_2'$, *il existe* $q_1 \stackrel{\alpha}{\rightarrow} q_1'$ *tel que* $q_1' \equiv_n q_2'$.

On définit $\equiv$ par

- $q_1 \equiv q_2$ *ssi* $q_1 \equiv_n q_2$ *pour tout* $n \in \mathbb{N}$.

Les relations $\equiv_n$ sont des équivalences entre les états. Comme d'habitude, on note $\mathcal{A}_1 \equiv_n \mathcal{A}_2$ (resp. $\mathcal{A}_1 \equiv \mathcal{A}_2$) quand les états initiaux de $\mathcal{A}_1$ et $\mathcal{A}_2$ sont reliés par $\equiv_n$ (resp. $\equiv$).

Intuitivement, l'équivalence $\equiv$ de la définition 4.3.1 distingue deux états q_1 et q_2 s'ils peuvent être distingués à un niveau n fini, i.e. si q_1 peut effectuer une transitions $q_1 \stackrel{\alpha}{\rightarrow} q_1'$ et si q_1' est distingué au niveau $n-1$ de tous les α-successeurs de q_2. En particulier, $\equiv_1$ distingue des états à partir desquels il n'est pas possible d'exécuter les mêmes actions.

Le nom "équivalence projective" vient du fait que deux systèmes sont $\equiv_n$ ssi leurs arbres de comportements tronqués à la profondeur n (leurs "projections") sont bisimilaires [BK89].

La bisimulation forte $\underline{\leftrightarrow}$ est en général plus fine que sa variante projective $\equiv$. Les résultats qui relient ces équivalences sont classiques (cf. [Mil89]):

Proposition 4.3.2. *Pour tous $\mathcal{A}_1, \mathcal{A}_2 \in \mathbf{ST}$,*

1. *on a*
$$\mathcal{A}_1 \underline{\leftrightarrow} \mathcal{A}_2 \quad \Rightarrow \quad \mathcal{A}_1 \equiv \mathcal{A}_2$$
2. *si de plus $\mathcal{A}_1$ ou $\mathcal{A}_2$ est à b.f.,*
$$\mathcal{A}_1 \underline{\leftrightarrow} \mathcal{A}_2 \quad - \quad \mathcal{A}_1 \equiv \mathcal{A}_2$$

[BBK87b] montre qu'un "ou" suffit pour le point 2.

4.4 Bisimulation modulo τ

Dans les systèmes de transitions que nous avons considérés jusqu'à présent, on considérait implicitement que toutes les actions étaient visibles. D'où nos définitions d'équivalence de traces, ou de bisimulation. Néanmoins dans de nombreuses situations, il est nécessaire de lire plus finement le modèle.

Dans cette section, on va introduire le concept classique d'action "invisible de l'extérieur" (ou encore d'action "silencieuse"). Ces actions sont un outil classique pour obtenir de façon méthodique certains comportements. Typiquement, elles apparaissent lors de la mise en parallèle de deux transitions avec synchronisation; les transitions synchronisées deviennent invisibles pour un observateur extérieur.

De fait, cette notion vient des algèbres de processus telles que CCS. Il faut alors adapter la définition de la bisimulation, entre autres, de façon à tenir compte des actions invisibles. Une idée est d'autoriser le système à évoluer de manière invisible (interne) lorsqu'il doit reproduire une action bisimilaire. C'est sur ce principe qu'est basée la τ-bisimulation de [Mil81].

Pour cela nous ajoutons à l'alphabet d'action Act un élément particulier τ pour dénoter les actions invisibles[1]. Nous notons Act_τ l'ensemble $Act \cup \{\tau\}$ et nous considérons dans la suite la classe des ST sur l'ensemble d'actions Act_τ.

Nous considérons la chaîne vide ϵ et nous notons Act_ϵ l'ensemble $Act \cup \{\epsilon\}$. Les éléments typiques de Act_ϵ seront notés $\lambda, \lambda', \ldots$. Notons que $\tau \notin Act_\epsilon$.

Soit $\mathcal{A} \in \mathbf{ST}$. Pour tout $\alpha \in Act_\tau$ nous notons $q \stackrel{\alpha}{\Rightarrow} q'$ lorsqu'il existe $n, m \geq 0$ tels que $q \stackrel{\tau^n \alpha \tau^m}{\longrightarrow} q'$. Intuitivement, une transition $\stackrel{\alpha}{\Rightarrow}$ permet d'absorber un nombre fini de τ avant et/ou après l'exécution de l'action α; nous écrivons $q \stackrel{\epsilon}{\Rightarrow} q'$ lorsque $q \stackrel{\tau^n}{\rightarrow} q'$ pour $n \geq 0$.

Définition 4.4.1. *(τ-bisimulation)*
Etant donnés $\mathcal{A}_1, \mathcal{A}_2 \in \mathbf{ST}$, une relation $R \subseteq St_{\mathcal{A}_1} \times St_{\mathcal{A}_2}$ est une τ-bisimulation *entre $\mathcal{A}_1$ et $\mathcal{A}_2$, ssi pour tous pRq, pour tout $\lambda \in Act_\epsilon$,*

1. *pour toute transition $p \stackrel{\lambda}{\Rightarrow}_{\mathcal{A}_1} p'$, il existe un état q' de $\mathcal{A}_2$ tel que $q \stackrel{\lambda}{\Rightarrow}_{\mathcal{A}_2} q'$ et $p' \, R \, q'$;*

2. *réciproquement, pour toute transition $q \stackrel{\lambda}{\Rightarrow}_{\mathcal{A}_2} q'$, il existe un état p' de $\mathcal{A}_1$ tel que $p \stackrel{\lambda}{\Rightarrow}_{\mathcal{A}_1} p'$ et $p' \, R \, q'$.*

[1] Il est clair qu'il n'est pas nécessaire d'introduire une constante pour chaque action invisible, voir [Mil81]

Nous disons d'une relation R ayant les propriétés 1 et 2 de la définition 4.4.1 qu'elle a la *propriété de τ-transfert* et que c'est une *τ-bisimulation* entre $\mathcal{A}_1$ et $\mathcal{A}_2$.

Par la suite, nous noterons "$R : p \underline{\leftrightarrow}_\tau p'$", "$p \underline{\leftrightarrow}_\tau p'$" et "$\mathcal{A}_1 \underline{\leftrightarrow}_\tau \mathcal{A}_2$" avec la signification qu'on attend, et nous parlerons d'états (ou de systèmes) τ-bisimilaires.

La définition de $\underline{\leftrightarrow}_\tau$ est équivalente à celle qui exige simplement

- d'une action τ qu'elle soit simulée par zéro ou plusieurs actions τ et
- d'une action $\alpha \neq \tau$ qu'elle soit simulée par une suite de transitions $\stackrel{\tau^n \alpha \tau^m}{\longrightarrow}$ avec $n, m \geq 0$

alors que pour la bisimulation forte chaque action $\alpha \in Act_\tau$ doit être simulée par exactement une action α même s'il s'agit d'un τ. Ainsi

$$\mathcal{A}_1 \underline{\leftrightarrow} \mathcal{A}_2 \text{ implique } \mathcal{A}_1 \underline{\leftrightarrow}_\tau \mathcal{A}_2$$

Remarquons que si $\mathcal{A}_{1,2}$ ne contiennent pas de τ-transition, ils sont τ-bisimilaires ssi ils sont bisimilaires.

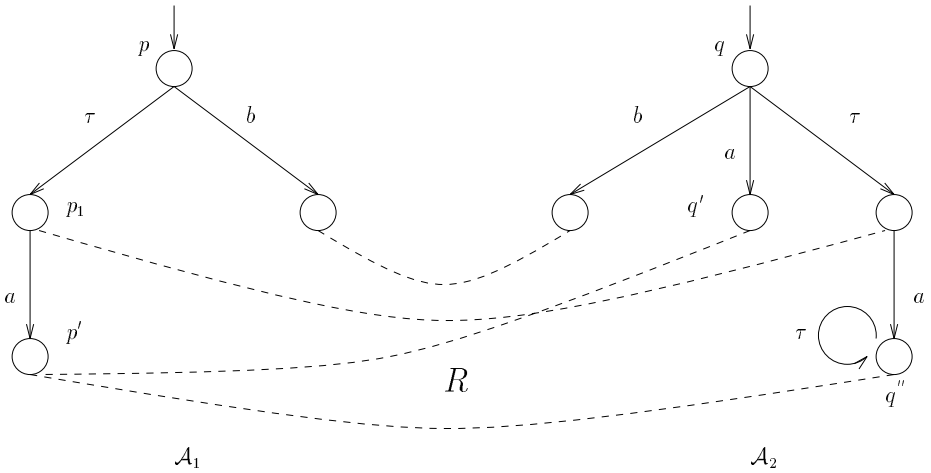


Figure 4.2 : Deux systèmes τ-bisimilaires

Exemple 4.1. Exemple de τ-bisimulation.

La figure 4.2 nous montre deux systèmes τ-bisimilaires ainsi qu'une τ-bisimulation R entre eux. Vérifions p.ex. que la transition $q \stackrel{a}{\rightarrow} q'$ de $\mathcal{A}_2$ peut être imitée dans $\mathcal{A}_1$. Clairement, cette transition est imitée dans $\mathcal{A}_1$ par la suite de transitions $p \stackrel{\tau}{\rightarrow} p_1 \stackrel{a}{\rightarrow} p'$, et on a bien $p' R q'$.

Une des limitations de la τ-bisimulation est le fait qu'elle ne traite pas la divergence (suite infinie de transitions invisibles) d'un état: par exemple, un état terminal et un état de divergence sont τ-bisimilaires (cf. p' et q'' dans la figure 4.2).

4.5 Une autre bisimulation modulo τ

Nous allons introduire une variante de $\underline{\leftrightarrow}_\tau$ dont nous aurons besoin par la suite. Ici on prendra en compte la divergence d'un état, définie formellement comme suit:

Définition 4.5.1. *(Divergence)*
Un état $p \in St$ d'un système de transitions $\mathcal{A}$ diverge *ssi il existe une suite infinie*

$$p \xrightarrow{\tau} p' \xrightarrow{\tau} p'' \xrightarrow{\tau} \cdots \xrightarrow{\tau} \cdots$$

de τ-transitions partant de p.

Notation 4.5.2. *On écrira p* `div` *quand p diverge.*

On peut alors définir la d-bisimulation (comme dans [GG89]) qui nous servira au chapitre 7.

Définition 4.5.3. *(d-bisimulation)*
Une relation $R \subseteq St \times St$ est une d-bisimulation *ssi*

1. *R a* la propriété de τ-transfert *(de la définition 4.4.1),*
2. *$p\,R\,q$ implique (p* `div` *ssi q* `div` *).*

On introduit alors les notations "$p \underline{\leftrightarrow}_d p'$", "$\mathcal{A}_1 \underline{\leftrightarrow}_d \mathcal{A}_2$", ...comme d'habitude.

La d-bisimulation que nous venons de définir est un cas particulier de τ-bisimulation, donc $\underline{\leftrightarrow}_d$ est clairement plus fine que $\underline{\leftrightarrow}_\tau$:

$$p \underline{\leftrightarrow}_d q \Rightarrow p \underline{\leftrightarrow}_\tau q$$

D'autre part, on voit aisément que

$$p \underline{\leftrightarrow} q \Rightarrow p \underline{\leftrightarrow}_d q$$

car $\underline{\leftrightarrow}$ préserve la divergence.

Après avoir présenté la d-bisimulation, nous donnerons certaines de ses propriétés. La première nous dit que si un état p est d-bisimilaire à un état terminal, alors depuis p il est possible d'atteindre un état terminal en effectuant une suite finie (éventuellement vide) de τ-transitions.

Proposition 4.5.4. *Etant donné un état p d'un système $\mathcal{A}$, si $q \in St$ est un état terminal et $p \underline{\leftrightarrow}_d q$, alors $\exists q' \in St : p \stackrel{\epsilon}{\Longrightarrow} q'$ & q' est un état terminal.*

Preuve. Deux cas peuvent se présenter suivant que

1. p est un état terminal, alors on prend $q' \stackrel{déf}{=} p$;
2. p n'est pas terminal, alors $\exists p \stackrel{\alpha}{\rightarrow} p^1$ et cette transition est imitable depuis q. Mais q est terminal et donc $\alpha = \tau$ et $p^1 \underline{\leftrightarrow}_d q$ car $q \stackrel{\epsilon}{\Rightarrow} q$ est la seule possibilité. On reprend le même raisonnement avec p^1 et, s'il n'est pas terminal, on peut construire inductivement une suite

   $$p \stackrel{\tau}{\rightarrow} p^1 \stackrel{\tau}{\rightarrow} \cdots \stackrel{\tau}{\rightarrow} p^n$$

   où pour chaque $i = 0, \ldots, n$: $p^i \underline{\leftrightarrow}_d q$. Cette suite ne peut pas être prolongée infiniment car sinon p `div` (or q ne diverge pas). Ainsi un des p^n est terminal, et $q' = p^n$ convient.

□

(En fait, on voit que, si $p \underline{\leftrightarrow}_d q$, p ne peut faire qu'un nombre fini de τ puis finalement terminer.)

D'autres propriétés de d-bisimulation sont des lemmes techniques 7.3.1 et 7.3.2 qui seront présentés dans la section 7.3 et serviront par la suite.

4.6 Traces modulo τ

Pour conclure ce chapitre, on va relier la d-bisimulation avec l'équivalence de traces *modulo* τ. Dans ce but on introduit une notion de *trace modulo* τ. Pour un chemin σ, $tr_\tau(\sigma)$ est obtenu à partir de $tr(\sigma)$ en retirant toutes les suites finies composées de τ (les suites infinies de τ sont donc conservées).

Exemple 4.2.
En retirant toutes les suites finies composées de τ on obtient

$$\begin{array}{lcl} abc\ldots & \rightarrow & abc\ldots \\ a\tau\tau b\tau\tau c\ldots & \rightarrow & abc\ldots \\ \tau\tau a\tau\tau\tau b\tau\tau\ldots\tau\ldots & \rightarrow & ab\tau^\omega \ (\text{ou} ab\tau\tau\ldots\tau\ldots) \end{array}$$

Ainsi, les seules actions τ restantes sont éventuellement un suffixe infini de τ (dénotant une divergence) de sorte que

$$tr_\tau(\sigma) \in Act^* \cup Act^*.\tau^\omega \cup Act^\omega$$

Pour tout $q \in St_{\mathcal{A}}$, nous définissons l'ensemble des *traces (complètes) modulo* τ de q par

$$Tr_\tau(q) \stackrel{déf}{=} \{tr_\tau(\pi) \,|\, \pi \in Ex(q)\}$$

et de façon similaire

$$Tr_\tau(\mathcal{A}) \stackrel{déf}{=} \{tr_\tau(\pi) \,|\, \pi \in Ex(\mathcal{A})\}$$

L'équivalence de traces modulo τ identifie les systèmes qui admettent le même ensemble de traces modulo τ. Formellement,

Définition 4.6.1. *Deux états,* q_1, q_2 *ont les mêmes traces modulo* τ*, noté* $q_1 \sim_{Tr_\tau} q_2$*, si* $Tr_\tau(q_1) = Tr_\tau(q_2)$.

Deux ST, $\mathcal{A}_1, \mathcal{A}_2$ *ont les mêmes traces modulo* τ*, noté* $\mathcal{A}_1 \sim_{Tr_\tau} \mathcal{A}_2$*, si* $Tr_\tau(\mathcal{A}_1) = Tr_\tau(\mathcal{A}_2)$.

A partir des définitions 4.5.3 et 4.6.1 il est facile de vérifier que

Proposition 4.6.2. *Pour tous* $q_1, q_2 \in St$

$$q_1 \underline{\leftrightarrows}_d q_2 \text{ implique } Tr_\tau(q_1) = Tr_\tau(q_2)$$

Bien sûr, la réciproque n'est en général pas vraie.

Chapitre 5

Algèbres de Processus

Les nombreux exemples de l'utilisation d'approches algébriques pour étudier des *calculs de processus* sont apparus pendant les quinze dernières années [Mil80, Hoa85, Hen88, Mil89, BB90].

Le concept du raisonnement de composition joue un rôle fondamental dans les algèbres telles que CCS [Mil89], ACP [BB90] et CSP [Hoa85]: on y décrit un processus en indiquant comment il est construit à partir de processus plus petits.

Dans ce chapitre, nous regroupons et résumons l'ensemble des bases théoriques concernant les algèbres de processus dont nous aurons besoin par la suite. Les définitions et les résultats sont la plupart du temps classiques et nous les donnons sans preuves, en indiquant au fur et à mesure les références correspondantes.

Nous allons définir l'algèbre PA (Process Algebra) [BK89] puis en isoler des fragments pertinents: RCCS (Regular CCS), BPA (Basic Process Algebra) et BPP (Basic Parallel Processes). Ceci nous permettra, dans le chapitre 7 de cette thèse, de situer notre modèle par rapport à ces algèbres classiques.

5.1 Déclarations de processus

Cette section donne la *syntaxe* de l'algèbre PA.

On considère un ensemble infini $Act = \{\alpha, \beta, \gamma, \ldots\}$ d'actions atomiques. Soient τ un élément distingué tel que $\tau \notin Act$ et $Act_\tau = Act \cup \{\tau\}$. Finalement on suppose que $Var = \{X, Y, Z, \ldots\}$ est un ensemble fini de *variables de processus*.

La définition suivante donne une syntaxe abstraite d'expressions de processus.

Définition 5.1.1. *Les* expressions de processus $E, E', \ldots$ *sont données par la grammaire suivante:*

$$\begin{array}{rcll} E, E' & ::= & \mathbf{0} & \textit{le processus inactif,} \\ & | & X & \textit{une variable de processus } (X \in Var), \\ & | & \alpha.E & \textit{un processus préfixé } (\alpha \in Act_\tau), \\ & | & E + E' & \textit{une sommation,} \\ & | & E \| E' & \textit{une composition parallèle,} \\ & | & E.E' & \textit{une composition séquentielle,} \end{array}$$

et on note $\mathcal{EXP}_{PA}$ (ou plus simplement $\mathcal{EXP}$) l'ensemble de toutes les expressions de processus.

Nous allons donner une intuition du sens de ces constructions (bien connues, cf. par exemple [BBK87c]).

- **0** représente un processus qui n'est pas actif, i.e. aucune action ne peut être effectuée.

- Pour deux processus donnés E et E', l'opérateur de sommation rend $E + E'$, un processus qui se comporte soit comme E, soit comme E'.

- Pour deux processus donnés E et E', la composition séquentielle rend $E.E'$, un processus dont le comportement est celui de E jusqu'à sa terminaison, suivi alors de E'.

- Pour deux processus donnés E et E', la composition parallèle rend $E \| E'$, un processus qui est capable d'accomplir les actions de E et E' en parallèle, i.e. arbitrairement entrelacées.

- Pour $\alpha \in Act_\tau$, le préfixage "$\alpha._$" est un opérateur unaire qui pour un processus E donné rend $\alpha.E$, un processus qui peut d'abord accomplir α et ensuite continuer comme E.

Ainsi le préfixage est un cas particulier de la composition séquentielle. Le processus inactif peut être vu comme une somme vide (ainsi que comme une mise en parallèle vide). Ces intuitions seront justifiées formellement plus tard.

Notation 5.1.2. *Le fait que $+$ et $\|$ sont associatifs et commutatifs*[1] *nous permet d'écrire $E_1 + E_2 + E_3$ et $E_1\|E_2\|E_3$ sans plus de parenthèses. De même, on écrira souvent αE pour $\alpha.E$ et α pour $\alpha.\mathbf{0}$.*

La priorité des opérateurs nous permet d'écrire "$\alpha\beta+\alpha\beta'\|\gamma$" pour $(\alpha\beta)+((\alpha'\beta')\|\gamma)$.

Une somme vide $\sum_{i\in\emptyset} E_i$, de même qu'une mise en parallèle vide, représentent $\mathbf{0}$.

Un *processus* est une expression de processus dont les variables sont définies par une famille finie d'équations récursives (de processus). Formellement:

Définition 5.1.3. *Une* déclaration de processus *est une famille Δ d'équations récursives de processus*

$$\Delta = \{X_i \stackrel{\text{déf}}{=} E_i \mid 1 \leq i \leq n\},$$

où n est un nombre fini, où les variables X_i sont toutes distinctes et où les expressions E_i n'utilisent que des variables appartenant à $Var(\Delta) \stackrel{\text{déf}}{=} \{X_1, X_2, \ldots, X_n\}$.

La variable X_1 s'appelle la *variable de tête* et sert d'état initial quand rien d'autre n'est précisé.

En général, une déclaration de processus reste sous-entendue. Elle est le contexte qui permet de donner un sens aux expressions de processus qui utilisent ses variables.

5.2 Comportement des processus

Comme c'est devenu standard dans le cadre des algèbres de processus, on définit le comportement de nos processus au moyen de la *sémantique opérationnelle structurelle* (donnée comme un système de transitions étiquetée).

Chaque déclaration Δ de processus détermine un système de transitions étiqueté: les états en sont les expressions de processus construites sur $Var(\Delta)$, l'ensemble d'étiquettes est donné par Act_τ et les relations de transitions sont données comme les plus petites relations dérivées des règles d'inférence suivantes:

[1] ce que nous justifierons dans la section 5.4.

Définition 5.2.1. *(Sémantique opérationnelle de PA)*

$$\frac{}{\alpha E \xrightarrow{\alpha} E} \qquad \frac{E \xrightarrow{\alpha} E'}{X \xrightarrow{\alpha} E'} \; si\ X \stackrel{\text{déf}}{=} E \in \Delta$$

$$\frac{E \xrightarrow{\alpha} E'}{E + F \xrightarrow{\alpha} E'} \qquad \frac{E \xrightarrow{\alpha} E'}{E \| F \xrightarrow{\alpha} E' \| F} \qquad \frac{E \xrightarrow{\alpha} E'}{E.F \xrightarrow{\alpha} E'.F}$$

$$\frac{E \xrightarrow{\alpha} E'}{F + E \xrightarrow{\alpha} E'} \qquad \frac{F \xrightarrow{\alpha} F'}{E \| F \xrightarrow{\alpha} E \| F'} \qquad \frac{isnil(E),\ F \xrightarrow{\alpha} F'}{E.F \xrightarrow{\alpha} F'}$$

Les règles pour la composition séquentielle utilisent un prédicat syntaxique *isnil* défini par cas sur la structure d'une expression E.

Définition 5.2.2. *Le prédicat isnil est la plus grande solution de:*

- $isnil(\mathbf{0}) \stackrel{\text{déf}}{=} true,$
- $isnil(\alpha.E) \stackrel{\text{déf}}{=} false,$
- $isnil(E + F) \stackrel{\text{déf}}{=} isnil(E.F) \stackrel{\text{déf}}{=} isnil(E \| F) \stackrel{\text{déf}}{=} isnil(E) \wedge isnil(F),$
- $isnil(X) \stackrel{\text{déf}}{=} isnil(E)$, *où* $X \stackrel{\text{déf}}{=} E \in \Delta.$

La relation de transition ainsi définie dépend évidemment de Δ, de sorte que l'on notera parfois explicitement $E \xrightarrow{\alpha}_{\Delta} F$ quand il est nécessaire de préciser la déclaration sous-jacente.

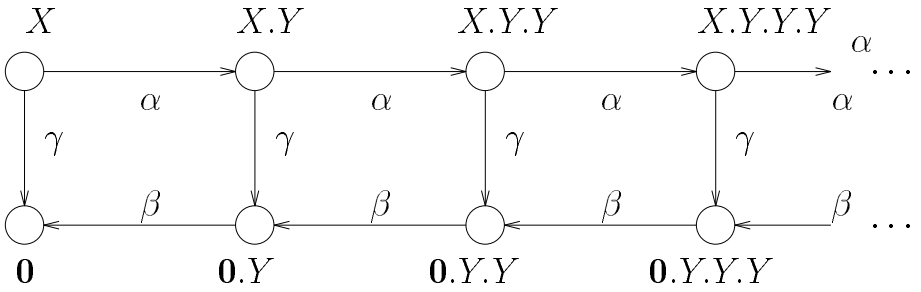


Figure 5.1 : Le système de transitions pour $\{X \stackrel{déf}{=} \alpha.(X.Y) + \gamma.\mathbf{0};\ Y \stackrel{déf}{=} \beta.\mathbf{0}\}$

Exemple 5.1.

Soit Δ la déclaration $\{X \stackrel{déf}{=} \alpha.(X.Y) + \gamma.\mathbf{0};\ Y \stackrel{déf}{=} \beta.\mathbf{0}\}$. Les règles de transitions génèrent un système de transitions (dont une partie est) représentée dans la figure 5.1. Notons que l'ensemble des états atteignables depuis X n'est pas fini.

Après avoir interprété les processus dans le cadre des systèmes de transitions étiquetés, il est possible d'étudier la bisimilarité entre des processus en les considérant comme états du ST correspondant.

Par exemple, il est possible de vérifier que le prédicat $isnil(E)$ a bien le sens attendu:

Proposition 5.2.3. [Chr93] $E \underline{\leftrightarrow} \mathbf{0}$ *si et seulement si* $isnil(E)$.

Il est connu que la bisimulation est une relation de congruence par rapport à tous les opérateurs présentés ci-dessus.

5.3 Déclarations gardées

Les variables de processus et les équations de la forme $X_i \stackrel{déf}{=} E_i (\in \Delta)$ permettent la récursion et par conséquent, la possibilité de décrire les comportements infinis. En particulier, il est possible de décrire des systèmes à branchement infini.

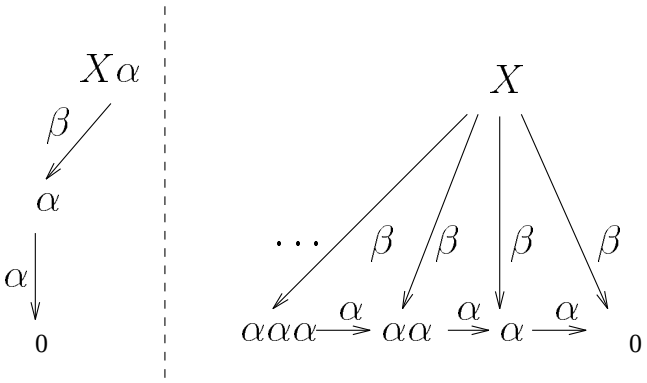


Figure 5.2 : Un système à branchement infini pour $\{X \stackrel{déf}{=} X.\alpha + \beta\}$

Par exemple, la déclaration $\{X \stackrel{déf}{=} X.\alpha + \beta\}$ génère un système de transitions dont une partie est représentée dans la figure 5.2.

C'est la raison pour laquelle habituellement l'attention de chercheurs est restreinte aux déclarations de processus gardés.

Définition 5.3.1. *Une expression de processus E est dite* gardée *ssi chaque occurrence d'une variable est sous la portée d'un opérateur de préfixage. Une déclaration*

$$\Delta = \{X_i \stackrel{\text{déf}}{=} E_i \mid 1 \leq i \leq n\}$$

est gardée *ssi chaque E_i est gardée.*

Ainsi $\Delta = \{X \stackrel{déf}{=} Y + \alpha X;\ \ Y \stackrel{déf}{=} \beta(X + Y)\}$ n'est pas gardée, mais $\Delta' = \{X \stackrel{déf}{=} \alpha(X + Y.X.Y);\ \ Y \stackrel{déf}{=} \beta(X + Y)\}$ l'est.

Proposition 5.3.2. *Le système de transitions associé à une déclaration Δ gardée est à branchement fini.*

Une autre propriété fondamentale est que les déclarations gardées admettent une solution unique (modulo bisimulation).

5.4 Propriétés algébriques des opérateurs de PA

Nous avons déjà eu l'occasion de mentionner que la somme non-déterministe était "associative et commutative". Par là nous entendions par exemple que $E + F$ et $F + E$, même s'ils ne donnaient pas lieu à des systèmes de transitions identiques, engendraient néanmoins le même comportement au sens où $E + F$ et $F + E$ sont bisimilaires.

Plus généralement, les opérateurs de PA vérifient les propriétés de base suivantes:

Proposition 5.4.1. *Pour tous processus E, F, G de PA*

- $E + F \underline{\leftrightarrow} F + E$,
- $E + (F + G) \underline{\leftrightarrow} (E + F) + G$,
- $E + \mathbf{0} \underline{\leftrightarrow} E + E \underline{\leftrightarrow} E$,
- $E \| F \underline{\leftrightarrow} F \| E$,
- $E \| (F \| G) \underline{\leftrightarrow} (E \| F) \| G$,
- $E \| \mathbf{0} \underline{\leftrightarrow} E$,
- $E.(F.G) \underline{\leftrightarrow} (E.F).G$,
- $E.\mathbf{0} \underline{\leftrightarrow} \mathbf{0}.E \underline{\leftrightarrow} E$.

dont la preuve est standard.

Ces équivalences nous permettent de simplifier les écritures des processus parce que, de plus, la bisimulation est substitutive, c.-à-d. qu'elle est une congruence pour tous les opérateurs de PA:

Proposition 5.4.2. *Pour tous processus E, E', F, F', si $E \underline{\leftrightarrow} E'$ et $F \underline{\leftrightarrow} F'$ alors*

- $\alpha E \underline{\leftrightarrow} \alpha E'$,
- $E + F \underline{\leftrightarrow} E' + F'$,
- $E.F \underline{\leftrightarrow} E'.F'$,
- $E \| F \underline{\leftrightarrow} E' \| F'$,

et

- $(\Delta \cup \{X \stackrel{\text{déf}}{=} E\}) \underline{\leftrightarrow} (\Delta \cup \{X \stackrel{\text{déf}}{=} E'\})$.

Ici aussi la preuve est classique (voir la preuve de la proposition 5.4.3 pour le cas de la d-bisimulation). La notation "$\Delta \underline{\leftrightarrow} \Delta'$" sera très utilisée pour dire que deux déclarations sont équivalentes (ici au sens de la bisimulation). Elle signifie que chaque expression E sur les variables communes à Δ et Δ' donne lieu à des comportements bisimilaires dans les systèmes de transitions associés à Δ et Δ'.

Dans les situations où nous devons faire abstraction des actions invisibles, nous utilisons la d-bisimulation définie page 52. Il est important qu'elle soit elle aussi substitutive, ce qui est le cas. En fait, la définition 4.5.3 adapte la définition classique de τ-bisimulation de façon justement à ce que $\underline{\leftrightarrow}_d$ soit substitutive par rapport à la composition séquentielle (ce qui n'est pas le cas de $\underline{\leftrightarrow}_\tau$).

Proposition 5.4.3. *Pour tous processus E, E', F, F',*
si $E \underline{\leftrightarrow}_d E'$ et $F \underline{\leftrightarrow}_d F'$, alors

- $\alpha E \underline{\leftrightarrow}_d \alpha E'$,
- $\alpha E + \beta F \underline{\leftrightarrow}_d \alpha E' + \beta F'$,
- $E.F \underline{\leftrightarrow}_d E'.F'$,
- $E \| F \underline{\leftrightarrow}_d E' \| F'$,

et

- $(\Delta \cup \{X \stackrel{\text{déf}}{=} E\}) \underline{\leftrightarrow}_d (\Delta \cup \{X \stackrel{\text{déf}}{=} E'\})$.

Nous allons donner la preuve des deux principales propriétés, i.e. la congruence pour la composition parallèle et pour la composition séquentielle.

Preuve. (Congruence pour la composition parallèle)
On va montrer que la relation $S \subseteq \mathcal{EXP} \times \mathcal{EXP}$, donnée par

$$S \stackrel{déf}{=} \{(E_1 \| F_1, E_2 \| F_2) \mid E_1 \underline{\leftrightarrow}_d E_2, F_1 \underline{\leftrightarrow}_d F_2\}$$

est une d-bisimulation. Considérons donc une paire $(E_1 \| F_1,\ E_2 \| F_2) \in S$.

1. Il faut d'abord vérifier que $E_1 \| F_1$ `div` ssi $E_2 \| F_2$ `div`. Supposons donc $E_1 \| F_1$ `div`. Alors E_1 `div` ou F_1 `div` . Par hypothèse (c.-à-d. puisque $E_1 \underline{\leftrightarrow}_d E_2$, $F_1 \underline{\leftrightarrow}_d F_2$) E_2 `div` ou F_2 `div`, et donc $E_2 \| F_2$ `div`. La réciproque se traite identiquement.

2. Il reste maintenant à vérifier la propriété de transfert. Supposons donc $E_1 \| F_1 \stackrel{\alpha}{\rightarrow} G_1$. Pour commencer on suppose $\alpha \neq \tau$. Selon le type des règles de dérivation pour la composition parallèle, on a les deux cas suivants:

   - **Cas 1:** $E_1 \stackrel{\alpha}{\rightarrow} E_1'$ et $G_1 = E_1' \| F_1$.
     Alors puisque $E_1 \underline{\leftrightarrow}_d E_2$ il existe $E_2 \stackrel{\alpha}{\Rightarrow} E_2'$ t.q. $E_1' \underline{\leftrightarrow}_d E_2'$. Par application des règles pour la composition parallèle sur chacune des transitions de la séquence $E_2 \stackrel{\alpha}{\Rightarrow} E_2'$, on déduit l'existence d'une séquence $E_2 \| F_2 \stackrel{\alpha}{\Rightarrow} E_2' \| F_2$ et il est évident que $(G_1, E_2' \| F_2) \in S$.
   - **Cas 2:** $F_1 \stackrel{\alpha}{\rightarrow} F_1'$ et $G_1 = E_1 \| F_1'$.
     On raisonne comme pour le Cas 1.

   Si maintenant $\alpha = \tau$, le même raisonnement convient, cette fois avec une séquence $E_2 \stackrel{\epsilon}{\Rightarrow} E_2'$ qui peut être vide.

3. Nous venons de vérifier une direction de la propriété de transfert, et la direction réciproque se vérifie exactement de la même façon, ce qui complète la démonstration de ce que S est une d-bisimulation.

□

Preuve. (Congruence pour la composition séquentielle)
On suit le même plan que pour la preuve précédente et on va montrer que la relation $S \subseteq \mathcal{EXP} \times \mathcal{EXP}$, donnée par

$$S \stackrel{déf}{=} \underline{\leftrightarrow}_d \cup \{(E_1.F_1, E_2.F_2) \mid E_1 \underline{\leftrightarrow}_d E_2, F_1 \underline{\leftrightarrow}_d F_2\}$$

est une d-bisimulation. Considérons donc une paire tirée de S, de la forme $(E_1.F_1, E_2.F_2)$ (clairement la propriété de transfert est évidente pour les autres paires de S).

1. Il faut d'abord vérifier que $E_1.F_1$ `div` ssi $E_2.F_2$ `div`. Supposons donc $E_1.F_1$ `div` alors il y a deux possibilités: soit (1) E_1 `div`, soit (2) F_1 `div` et il existe $E_1 \stackrel{\epsilon}{\Rightarrow} E_1'$ tel que $isnil(E_1')$. Par hypothèse (c.-à-d. puisque $E_1 \underline{\leftrightarrow}_d E_2$, $F_1 \underline{\leftrightarrow}_d F_2$) on a E_2 `div` dans le cas (1), et F_2 `div` dans le cas (2). Dans le cas (2) $E_1 \underline{\leftrightarrow}_d E_2$ et la proposition 5.2.3 nous donnent l'existence d'un $E_2 \stackrel{\epsilon}{\Rightarrow} E_2'$ avec $isnil(E_2')$. Finalement, on a bien $E_2.F_2$ `div`. La réciproque se traite identiquement.

2. Il reste maintenant à vérifier la propriété de transfert. Supposons donc $E_1.F_1 \stackrel{\alpha}{\rightarrow} G_1$. Pour commencer on suppose $\alpha \neq \tau$. Selon le type des règles de dérivation pour la composition séquentielle, on a les deux cas suivants:

   - **Cas 1:** $E_1 \stackrel{\alpha}{\rightarrow} E_1'$ et $G_1 = E_1'.F_1$.
     Alors puisque $E_1 \underline{\leftrightarrow}_d E_2$ il existe $E_2 \stackrel{\alpha}{\Rightarrow} E_2'$ t.q. $E_1' \underline{\leftrightarrow}_d E_2'$. Par application des règles pour la composition séquentielle sur chacune des transitions de la séquence $E_2 \stackrel{\alpha}{\Rightarrow} E_2'$, on déduit l'existence d'une séquence $E_2.F_2 \stackrel{\alpha}{\Rightarrow} E_2'.F_2$ et il est évident que $(G_1, E_2'.F_2) \in S$.
   - **Cas 2:** $isnil(E_1)$ et $F_1 \stackrel{\alpha}{\rightarrow} G_1$.
     Alors il existe $E_2 \stackrel{\epsilon}{\Rightarrow} E_2'$ t.q. $isnil(E_2')$ (proposition 5.2.3) et $F_2 \stackrel{\alpha}{\Rightarrow} G_2$ t.q. $G_1 \underline{\leftrightarrow}_d G_2$. Les règles de la composition séquentielle nous permettent d'enchaîner ces deux séquences pour obtenir $E_2.F_2 \stackrel{\alpha}{\Rightarrow} G_2$ et on a clairement $(G_1, G_2) \in S$.

   Si maintenant $\alpha = \tau$, le même raisonnement convient, cette fois en autorisant une séquence $E_2 \stackrel{\epsilon}{\Rightarrow} E_2'$ ou $F_2 \stackrel{\epsilon}{\Rightarrow} G_2$ vide.

3. Nous venons de vérifier une direction de la propriété de transfert, et la direction réciproque se vérifie exactement de la même façon, ce qui complète la démonstration de ce que S est une d-bisimulation.

□

Il convient de noter que la d-bisimulation n'est pas une congruence pour la somme non-déterministe ($+$).

Exemple 5.2. La d-bisimulation n'est pas une congruence pour ($+$).
Il est facile de voir que

$$\alpha.\mathbf{0} \underline{\leftrightarrow}_d \tau.\alpha.\mathbf{0}$$
$$\beta.\mathbf{0} \underline{\leftrightarrow}_d \beta.\mathbf{0}$$

mais $\alpha.\mathbf{0} + \beta.\mathbf{0} \not\underline{\leftrightarrow}_d \tau.\alpha.\mathbf{0} + \beta.\mathbf{0}$

Le problème est classique [Mil89, BK88] et sa solution est connue: définir une notion de "*rooted-d-bisimulation*". Néanmoins cette solution est techniquement peu agréable à manipuler. Par ailleurs nous n'aurons finalement pas besoin de propriété de congruence pour la somme puisque nos sommes sont toutes préfixées de sorte que la proposition 5.4.3 permet de les traiter. C'est pourquoi nous nous contenterons dans la suite de notre notion de *d*-bisimulation.

5.5 Sous-classes de processus et leurs formes normales

Maintenant nous allons isoler plusieurs sous-classes de processus qui vont attirer notre attention dans le chapitre 7. Dans cette section on va considérer quatre classes de processus qui sont communément connues comme *RCCS* (Regular CCS), *BPA* (Basic Process Algebra), *BPP* (Basic Parallel Processes) et *PA* (Process Algebra).

La première classe de processus est RCCS, Regular CCS:

Définition 5.5.1. *Les expressions de processus de RCCS sont les expressions construites sur Var en utilisant seulement* $\mathbf{0}$*, les variables, le préfixage et la somme. Une déclaration $\Delta = \{X_i \stackrel{\text{déf}}{=} E_i \ : \ i = 1, 2, \ldots, n\}$ définit des processus de RCCS ssi chaque E_i est une expression de RCCS gardée pour $i = 1, 2, \ldots, n$.*

Les processus de RCCS correspondent aux automates finis de la théorie de langages formels et ont une théorie bien établie. Du fait de la finitude de l'ensemble des états atteignables, la plupart des équivalences sémantiques sont décidables sur RCCS.

On va utiliser une notion de *forme normale* pour RCCS, car il est démontré que chaque déclaration Δ (gardée) de processus de RCCS peut être effectivement transformée en une déclaration Δ' en forme normale de telle façon qu'une bisimulation est préservée, c.-à-d. $\Delta \underline{\leftrightarrow} \Delta'$.

Définition 5.5.2. *Une déclaration gardée $\Delta = \{X_i \stackrel{\text{déf}}{=} E_i \ : \ i = 1, 2, \ldots, n\}$ de processus de RCCS est en* forme normale *si chaque expression E_i est de la forme*

$$E_i = \sum_{j=1}^{n_i} \alpha_{i_j} X_{i_j}$$

Dans ce travail nous prenons les définitions suivantes de [Chr93] pour BPA et BPP:

Définition 5.5.3. *Les expressions de processus de BPA sont les expressions construites en utilisant seulement* $\mathbf{0}$*, les variables, le préfixage, la somme et la composition séquentielle. Une déclaration* $\Delta = \{X_i \stackrel{\text{déf}}{=} E_i \ : \ i = 1, 2, \ldots, n\}$ *définit des processus de BPA ssi chaque* E_i *est une expression de BPA gardée pour* $i = 1, 2, \ldots, n$*.*

Cette classe de processus, introduite dans [BBK87a], correspond aux systèmes de transitions associés aux grammaires hors-contexte. Elle a été intensivement étudiée par plusieurs chercheurs. Dans [CHS92] il est démontré que la bisimulation est décidable pour la classe entière de processus de BPA.

Si on remplace l'opérateur de la composition séquentielle par celui de la composition parallèle, on obtient des processus de BPP:

Définition 5.5.4. *Les expressions de processus de BPP sont les expressions construites en utilisant seulement* $\mathbf{0}$*, les variables, le préfixage, la somme et la composition parallèle. Une déclaration* $\Delta = \{X_i \stackrel{\text{déf}}{=} E_i \ : \ i = 1, 2, \ldots, n\}$ *définit des processus de BPP ssi chaque* E_i *est une expression de BPP gardée pour* $i = 1, 2, \ldots, n$*.*

Les déclarations de BPP correspondent aux réseaux de Petri places/transitions sans synchronisation, i.e. où la précondition d'une transition est réduite à une seule place. [CHM93] montre que la bisimilarité est décidable pour les processus BPP.

Par exemple la déclaration de l'exemple 5.1 définit un processus de BPP mais pas de BPA.

Il est possible d'aller plus loin dans la normalisation des déclarations. Toute déclaration gardée Δ de BPA (resp. de BPP) peut être effectivement transformée en une déclaration Δ' de BPA (resp. BPP) en *forme normale de Greibach* (GNF) de degré 3 (cf. [BBK87c] et [Chr93]) telle que $\Delta \underline{\leftrightarrow} \Delta'$.

Définition 5.5.5. *Une déclaration gardée* $\Delta = \{X_i \stackrel{\text{déf}}{=} E_i : i = 1, \ldots, n\}$ *de BPA (resp. BPP) est en forme normale de Greibach de degré* k *ssi chaque équation est de la forme*

$$X_i \stackrel{\text{déf}}{=} \sum_{j=1}^{n_i} \alpha_{i_j} T_{i_j}$$

où les T_{i_j} *sont des compositions séquentielles* $X_{i_j}^1 . X_{i_j}^2 . \ldots . X_{i_j}^s$ *(resp. des compositions parallèles* $X_{i_j}^1 \| X_{i_j}^2 \| \cdots \| X_{i_j}^s$*) de taille* $s < k$*.*

(Notons que RCCS correspond à la forme normale de Greibach de degré 2.)

Ce qu'on appelle PA (pour *Process Algebra*) en suivant [BK89] correspond en fait au langage général (avec compositions parallèle et séquentielle) que nous avons introduit dans la définition 5.1.1 modulo l'hypothèse de déclaration gardée:

Définition 5.5.6. *Une déclaration* $\Delta = \{X_i \stackrel{\text{déf}}{=} E_i \ : \ i = 1, 2, \ldots, n\}$ *d'expressions de processus gardées définit des processus de PA.*

Par rapport à un langage comme CCS [Mil89], PA contient la composition séquentielle (qui ne figure pas dans CCS) mais n'a ni la synchronisation, ni le renommage, ni surtout la restriction qui permet de forcer des synchronisations.

Une proposition importante 5.5.8 montre que toute déclaration gardée Δ d'expressions de processus de PA peut être effectivement présentée en *forme normale gardée (fng)* de degré 3 en préservant la bisimilarité. Puisque la preuve de ce résultat est assez compliquée, fait introduire beaucoup de notions et notations de l'algèbre de processus et reprend finalement des idées des preuves de [BBK87c] pour BPA et de [Chr93] pour BPP, on a choisi ne pas la donner dans la thèse.

Définition 5.5.7. *Une expression E de PA est en* forme normale gardée *(fng) ssi E est de la forme*

$$E = \sum_{i=1}^{n} \alpha_i.T_i$$

où $\alpha \in \Lambda$ et où chaque T_i est un terme sans préfixe ni somme respectant la grammaire suivante

$$\begin{array}{rcl} T & ::= & X \ (\in Var(\Delta)) \\ & | & T \| T' \\ & | & T.T' \end{array}$$

Une déclaration gardée $\Delta = \{X_i \stackrel{\text{déf}}{=} E_i \,|\, 1 \leq i \leq n\}$ de PA est en forme normale gardée *ssi chaque E_i est en fng.*

Exemple 5.3. Forme normale gardée d'une déclaration.

Cet exemple contient une déclaration gardée Δ de PA qui est en fng:

$$\begin{array}{rcl} X_1 & \stackrel{déf}{=} & \alpha.(X_1.(X_1 \| X_2)) + \beta.(X_1 \| (X_1 \| X_2)).X_2 \\ X_2 & \stackrel{déf}{=} & \beta \end{array}$$

et une déclaration gardée Δ' de PA qui ne l'est pas:

$$\begin{array}{lll} X_1 & \stackrel{déf}{=} & \alpha.(X_1.(X_1 \| \alpha.X_2)) + \beta.(X_1 \| (X_1 + X_2)).X_2 \\ X_2 & \stackrel{déf}{=} & \beta \end{array}$$

Proposition 5.5.8. *Toute déclaration Δ gardée de PA peut être transformée en une déclaration Δ' de PA en fng du degré 3 telle que*

$$\Delta \underline{\leftrightarrow} \Delta'$$

Ce résultat nous permettra de ne considérer que certaines formes de déclarations PA lors de traductions de PA dans un autre formalisme, et ceci sans perte de généralité.

Deuxième partie

Un modèle formel pour RPC

Chapitre 6

Sémantique comportementale de RPC

Nous disposons maintenant de suffisamment d'éléments pour construire dans ce chapitre un modèle formel pour RPC. Ce modèle nous servira de domaine sémantique dans lequel le langage RPC pourra être interprété.

Il existe plusieurs façons différentes de donner une sémantique à un langage de programmation. Un des cadres les plus connus est celui de la *sémantique dénotationnelle*, reposant principalement sur les travaux de Scott et Strachey [Sto77, Mos90]. Les premiers travaux pour donner une sémantique à un langage parallèle ont repris le cadre de la sémantique dénotationnelle. Mais la simple introduction du non-déterminisme (inhérent au parallélisme) dans un langage "à la Pascal" nécessite la construction de domaines très complexes qui sont lourds à manipuler.

Quand Milner a décrit son calcul CCS [Mil80], il a préféré introduire une notion de programmes parallèles "ayant le même comportement". Une sémantique des programmes parallèles est alors composée d'une sémantique opérationnelle qui donne le comportement d'un programme, et d'une équivalence entre programmes, qui doit être une congruence par rapport aux combinateurs de programmes.

Dans ce chapitre, nous allons suivre cette approche pour donner une sémantique au langage RPC. La construction combine une algèbre de systèmes de transitions avec une équivalence entre systèmes.

Dans l'introduction de cette thèse, on a expliqué comment notre recherche est motivée par le souci de donner un modèle formel pour RPC, modèle sur lequel on pourrait construire une nouvelle génération d'outils pour l'analyse du flot de contrôle dans des programmes récursifs-parallèles et pour leur diag-

nostic. C'est ce souci donc qui justifie notre niveau d'abstraction quand on considère des programmes RPC non interprétés.

Nous allons rappeler brièvement les particularités de RPC qui nous intéressent (cf. chapitre 2). Un programme RPC est un ensemble de procédures parallèles qui peuvent s'invoquer récursivement. L'invocation (via `pcall`) d'une procédure donne lieu au déclenchement d'un processus fils concurrent, placé hiérarchiquement sous le contrôle du processus père qui l'a invoqué. Les processus fils s'exécutent librement mais leur père peut demander (via une instruction spéciale de synchronisation, le `wait`) d'attendre la terminaison de tous ses fils.

La terminaison d'une invocation donnée peut être obtenue par l'instruction `end`. Un point important est à rappeler: l'asymétrie apportée par le fait que les processus fils ne peuvent pas savoir si leur père est terminé. Notons que l'utilisation de l'instruction `wait` n'est pas obligatoire: le processus père peut aussi choisir de terminer et laisser ses fils en marche.

6.1 Schémas de programmes récursifs-parallèles

Lors de la réalisation de l'outil pour l'analyse et la vérification des programmes récursifs-parallèles, le premier problème abordé était celui de la modélisation du flot de contrôle dans des programmes écrits en RPC. Le formalisme des schémas de programmes a été choisi parce qu'il est assez proche des constructions langagières, de sorte que la modélisation est facile à définir et à comprendre.

Il est bien connu que les schémas de programmes [KM69, Kel73] sont un des modèles mathématiques fondamentaux des programmes dans lesquels leur structure et des corrélations des composantes sont reflétées.

Les schémas des programmes servent à distinguer précisément la structure de contrôle d'un programme de ses autres aspects. Il existe beaucoup de types de schémas de programmes dont une présentation plus détaillée peut être trouvée dans [Kot91]. Dans notre recherche nous allons définir des *schémas de programmes récursifs-parallèles* (RPPS) basés sur les approches de Keller [Kel73] et Podlovchenko [Pod91] en s'abstrayant des données et de l'effet des instructions de base (qui resteront sans interprétation). Notre approche diffère de celles de [Kel73] et [Pod91] de sorte que la récursivité et le parallélisme sont considérés ensemble: le parallélisme n'est possible qu'au niveau de procédures qui peuvent s'invoquer récursivement.

Avant de donner des définitions formelles on considère un exemple.

```
program main
       a1;
 l1:   pcall subr1;
       a2;
       if b1 then {
       goto l1;
       } else {
       wait;
       a3;
       end;
       }

procedure subr1
       if b2 then {
       a4;
       end;
       } else {
       pcall subr1;
       a5;
       wait;
       end;
       }
```

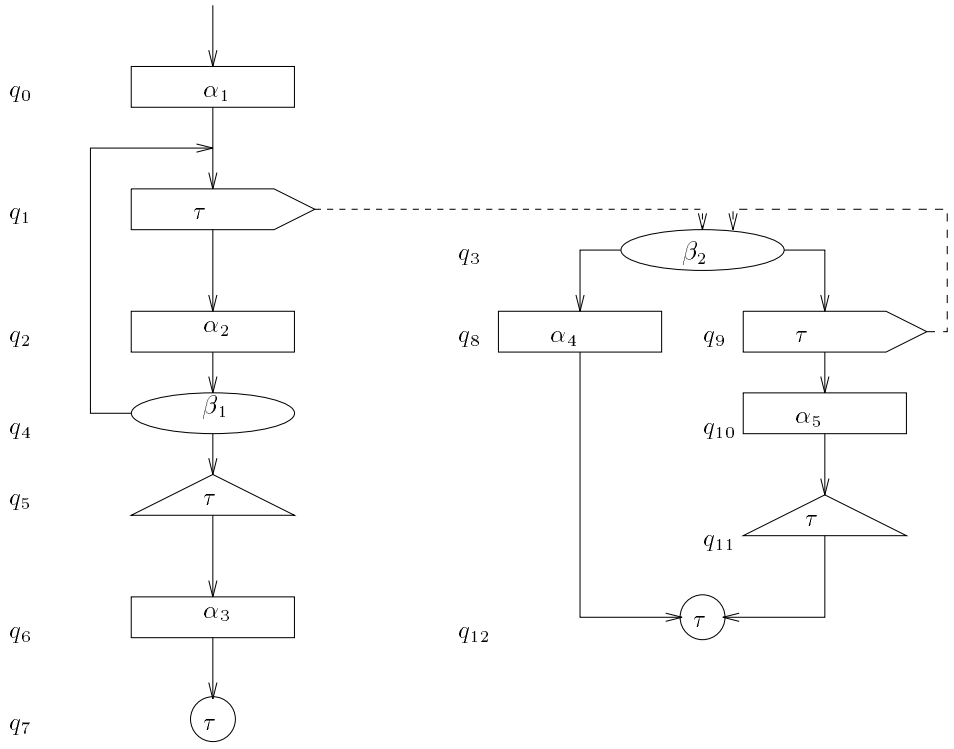


Figure 6.1 : Un programme RPC et le schéma associé

Exemple 6.1. Un programme RPC et le schéma associé.

Considérons le programme de la figure 6.1 et le schéma qui lui est associé.

Les schémas de RPPS sont présentés sous forme de graphes. Notre exemple montre comment un schéma est associé de façon évidente à un programme récursif-parallèle (ici composé d'une procédure principale et d'une coroutine).

Un schéma a différents types de noeuds. On va utiliser des notations graphiques suivantes pour les schémas RPPS: un noeud correspondant à une instruction d'action a la forme d'un rectangle; un noeud correspondant à une instruction de choix a celle d'une ellipse; un noeud correspondant à une instruction d'invocation celle d'un pentagone; un noeud correspondant à une instruction de synchronisation celle d'un triangle; un noeud correspondant à l'instruction `end` celle d'un cercle. On inscrit des étiquettes dans les noeuds, correspondant aux actions qu'ils schématisent.

On considère une *base* Λ: c'est un alphabet $\{a_1, a_2, \ldots\}$ contenant des noms d'actions abstraites au lieu des actions de base habituelles des langages impératifs.

On va supposer que la base Λ est fixée. On va utiliser le nom spécial τ pour dénoter des actions silencieuses correspondantes aux calculs internes (invocation, synchronisation, terminaison) et on va noter $\Lambda_\tau = \Lambda \cup \{\tau\}$.

Définition 6.1.1. *Un* schéma de programme *(sur la base Λ_τ) est un graphe fini enraciné $G = (Q_G, q_0, \mapsto_G, L_G)$, où*

- *$Q_G = \{q_0, q, q_1, \ldots, q_n, \}$ est un ensemble fini de* sommets *(ou* noeuds*);*
  chaque noeud est d'un des cinq types: d'action, de choix, d'invocation, de synchronisation, `end`*-noeud;*
- *q_0 est le noeud* initial*;*
- *$\mapsto_G : Q_G \to Q_G^*$ est une fonction telle que pour chaque $q \in Q_G$ elle permet de trouver des noeuds suivants de q;*
- *$L_G : Q_G \to \Lambda_\tau \times Q_G^*$ est un* étiquetage *des noeuds du graphe avec des noms de Λ_τ, tel que en plus L_G indique le noeud invoqué pour chaque noeud d'invocation.*

Quand $\mapsto_G$ relie un noeud q à un autre noeud q', on dit que q' est un suivant de q. Les noeuds `end` n'ont pas de suivant. Les noeuds `pcall` sont reliés à deux noeuds, mais un seul est le "suivant" (l'autre est le noeud "invoqué" et c'est L_G qui contient l'information sur des noeuds invoqués). Ainsi, $\mapsto_G$ donne le ou les arcs reliant un noeud q à d'autres noeuds (ses suivants), tandis que L_G donne le symbole porté par un noeud du graphe,

et indique quel est le noeud invoqué par un `pcall` (cf. les pointillés vers les noeuds invoqués dans la figure 6.1).

Lorsque le schéma G est sous-entendu, on notera directement $Q, \mapsto$ et L au lieu de $Q_G, \mapsto_G$ et L_G.

Puisque les instructions de base restent sans interprétation, les branchements if-then-else de RPC donnent lieu à du non-déterminisme dans les schémas, représenté en autorisant la présence de plusieurs suivants pour un même noeud.

On note $RPPS_{\Lambda_\tau}$ la classe de tous les schémas sur la base Λ_τ.

6.2 Sémantique d'états hiérarchiques

Les schémas de programmes RPC présentés ci-dessus permettent de représenter la structure d'un programme. L'étape suivante de formalisation nous permet de donner un sens à nos programmes, une *sémantique comportementale abstraite*. Maintenant, la première chose à faire c'est d'introduire une notion d'*état*.

Pour un schéma de $RPPS_{\Lambda_\tau}$, la notion fondamentale est celle d'*état hiérarchique*. Formellement,

Définition 6.2.1. *L'ensemble des* états hiérarchiques *de* $G \in RPPS_{\Lambda_\tau}$ *est le plus petit ensemble* $M_G = \{\xi, \zeta, \varphi, \ldots\}$ *tel que:*

- *si* $q_1, \ldots, q_n$ *sont des noeuds de* Q_G *et*
- *si* $\xi_1, \ldots, \xi_n \in M_G$,

alors le multi-ensemble $\xi \stackrel{\text{déf}}{=} \{(q_1, \xi_1), \ldots, (q_n, \xi_n)\}$ *appartient à* M_G.

En particulier, $\emptyset \in M_G$.

Intuitivement, un état hiérarchique ξ peut être considéré comme un marquage d'un réseau de Petri (avec des jetons dans les noeuds $q, q', \ldots$) avec une information supplémentaire sur la structure (de type "arbre" ou "forêt") "père-fils" entre les jetons.

L'état hiérarchique $\{(q_1, \xi_1), \ldots, (q_n, \xi_n)\}$ correspond à l'activation de n processus indépendants. Chacun de ces processus, p.ex. (q_i, ξ_i), a une composante père (ici dans l'état (de contrôle) q_i) et une famille de composantes filles. Cette famille est représentée inductivement sous la forme ξ_i d'un état hiérarchique.

Exemple 6.2. Un état hiérarchique.

Notre schéma de l'exemple 6.1 produit, parmi d'autres, un état hiérarchique

$$\xi = \{(q_5, \underbrace{\{(q_8, \emptyset), (q_9, \emptyset), (q_9, \emptyset)\}}_{\xi'})\}$$

Cet état a une composante père (qui est dans l'état de contrôle q_5) et une famille de trois composantes filles $\{(q_8, \emptyset), (q_9, \emptyset), (q_9, \emptyset)\}$ qui sont indépendantes. De nouveau, cette famille représente un état hiérarchique ξ'.

Cet état hiérarchique ξ peut être considéré comme un marquage d'un réseau de Petri avec un jeton dans chacun des noeuds q_5, q_8 et avec deux jetons dans le noeud q_9 dans la figure 6.1 avec une information supplémentaire sur la structure de dépendance entre les jetons.

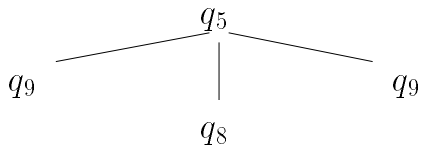


Figure 6.2 : L'état hiérarchique ξ comme un arbre

Notons que la structure de multi-ensemble ne prend pas en compte une notion d'ordre entre les différents fils d'un même noeud père. Mathématiquement, les éléments de M_G sont des multi-ensembles emboîtés. En résumé, l'ensemble M_G de tous les états hiérarchiques (de G) est la plus petite solution de l'équation de domaine

$$M_G = \mathcal{U}(Q_G \times M_G)$$

où on rappelle que $\mathcal{U}(Q_G \times M_G)$ désigne l'ensemble de tous les multi-ensembles finis sur $Q_G \times M_G$.

Notation 6.2.2. *1. On va écrire M au lieu de M_G.*

2. *On écrira q, ξ plutôt que $\{(q, \xi)\}$ et en général on ne mettra les parenthèses et les accolades que quand cela évite des ambiguïtés.*

3. *On va noter $\xi[q]$ un état hiérarchique ξ, où une occurrence du noeud q est isolée, ce qui permettra de noter $\xi[q']$ l'état hiérarchique obtenu en remplaçant dans ξ cette occurrence de q par q'.*

Exemple 6.3.
Un état hiérarchique de l'exemple 6.2

$$\xi = \{(q_5, \underbrace{\{(q_8, \emptyset), (q_9, \emptyset), (q_9, \emptyset)\}}_{\xi'})\}$$

peut être noté par $\xi[q_8]$, où une occurrence du noeud q_8 est isolée. Quand on remplace dans ξ cette occurrence de q_8 par q_{12}, cela permet de noter $\xi[q_{12}]$ l'état hiérarchique obtenu par ce remplacement.

Puisque les états sont des multi-ensembles, on va utiliser par la suite certaines des notions classiques (par exemple $\xi + \xi', \xi \subseteq \xi'$).

On fera de nombreux raisonnements basés sur la structure inductive des états. Pour cela on définit la taille $|\xi|$ d'un état hiérarchique ξ par:

1. $|\{(q_1, \xi_1), \ldots, (q_n, \xi_n)\}| \stackrel{déf}{=} |(q_1, \xi_1)| + \ldots + |(q_n, \xi_n)|$;
2. $|(q, \xi)| \stackrel{déf}{=} 1 + |\xi|$;
3. $|\emptyset| \stackrel{déf}{=} 0$.

Exemple 6.4.
Pour un état hiérarchique de l'exemple 6.2

$$\xi = \{(q_5, \underbrace{\{(q_8, \emptyset), (q_9, \emptyset), (q_9, \emptyset)\}}_{\xi'})\}$$

on a $|\xi| = 1 + |\xi'| = 1 + |(q_8, \emptyset)| + |(q_9, \emptyset)| + |(q_9, \emptyset)| = 4$.

6.3 Comportement des états hiérarchiques

On considère un schéma donné $G = (Q, q_0, \mapsto, L) \in RPPS_{\Lambda_\tau}$ et on définit une notion formelle de comportement de schémas sous la forme d'un système de transitions étiqueté $\mathcal{M}_G$, suivant en cela une approche maintenant classique, initiée par Plotkin [Plo81]. Les étiquettes sont prises dans Λ_τ.

Définition 6.3.1. *Le système de transitions étiqueté*

$$\mathcal{M}_G \stackrel{\text{déf}}{=} (M_G, \Lambda_\tau, \longmapsto, \xi_0)$$

a comme état initial

$$\xi_0 \stackrel{\text{déf}}{=} (q_0, \emptyset)$$

et comme relation de transitions $\longmapsto \subseteq M_G \times \Lambda_\tau \times M_G$ *définie comme la plus petite famille de triplets* (ξ, α, ξ') *qui obéit aux règles suivantes:*

ACT: *si $q \in Q$ est un noeud d'action (ou de choix) étiqueté par $\alpha \in \Lambda$ et q' est un suivant de q, alors*

$$q, \xi \stackrel{\alpha}{\longmapsto} q', \xi.$$

CALL: *si $q \in Q$ est un noeud d'invocation, de suivant q' et de noeud invoqué q'', alors*

$$q, \xi \stackrel{\tau}{\longmapsto} q', (\xi + (q'', \emptyset)).$$

WAIT: *si $q \in Q$ est un noeud* `wait`*, de suivant q', alors*

$$q, \emptyset \stackrel{\tau}{\longmapsto} q', \emptyset.$$

END: *si $q \in Q$ est un* `end`*-noeud, alors*

$$q, \xi \stackrel{\tau}{\longmapsto} \xi.$$

PAR_1:

$$\frac{\xi \stackrel{\alpha}{\longmapsto} \xi'}{q, \xi \stackrel{\alpha}{\longmapsto} q, \xi'}$$

PAR_2:

$$\frac{\xi \stackrel{\alpha}{\longmapsto} \xi'}{\xi + \xi'' \stackrel{\alpha}{\longmapsto} \xi' + \xi''}$$

Les règles ACT, CALL, WAIT et END décrivent comment un jeton passe d'un noeud de M_G à un des noeuds suivants en gardant le contact avec ses fils.

La règle CALL formule comment les invocations-"filles" sont créées: alors que le père passe dans le noeud suivant, une nouvelle invocation est ajoutée à l'ensemble des fils.

La règle WAIT déclare qu'on peut accomplir une instruction `wait` dans un état q et passer à un noeud suivant q' s'il n'y a pas (ou plus) de processus fils en cours d'exécution (ce qui veut dire qu'ils sont tous terminés, cf. la proposition 6.4.2).

La règle END explique ce que deviennent les "fils" quand le "père" termine et c'est cette règle qui permet d'atteindre l'état vide $\emptyset$.

Les règles $\mathrm{PAR}_1, \mathrm{PAR}_2$ formulent que toute activité $\xi \stackrel{\alpha}{\longmapsto} \xi'$ de ξ reste possible à l'identique quand un "père" est présent ou bien quand des "frères"

sont ajoutés.

On peut expliquer le sens des règles de la définition 6.3.1 en utilisant la métaphore du marquage d'un réseau de Petri.

La règle ACT décrit un déplacement d'un jeton d'un noeud q dans un des ses noeuds suivants q' et en même temps, il garde l'information sur tous les jetons dépendants de lui directement.

Selon la règle CALL un jeton passe d'un noeud q dans le noeud suivant en gardant l'ensemble des jetons dépendants de lui. Dans le même mouvement, un jeton correspondant à une nouvelle invocation (dans un noeud invoqué q'') est ajouté à cet ensemble.

En suivant la règle WAIT un jeton passe d'un noeud q dans un noeud suivant q' s'il n'y a nulle part des jetons dépendant de lui.

La règle END nous permet d'enlever un jeton d'un noeud q et c'est cette règle qui permet d'atteindre l'ensemble vide de jetons.

Un point important est à noter: la relation de dépendance parmi les jetons est créée par les invocations de coroutines, et reste maintenue (elle suit les jetons) lors du déplacement des jetons, jusqu'à leur terminaison.

Exemple 6.5. Un comportement.

Notre schéma de l'exemple 6.1 admet, parmi d'autres, une exécution débutant par

$$q_0, \emptyset \xmapsto{\alpha_1} q_1, \emptyset \xmapsto{\tau} q_2, (q_3, \emptyset) \xmapsto{\beta_2} q_2, (q_9, \emptyset) \xmapsto{\alpha_2} q_4, (q_9, \emptyset) \xmapsto{\beta_1} q_5, (q_9, \emptyset)$$
$$\xmapsto{\tau} q_5, (q_{10}, (q_3, \emptyset))$$

où le `wait`-noeud q_5 avec l'étiquette τ ne peut pas être passé tant que les invocations-"filles" ne sont pas terminées.

Ainsi, les actions atomiques d'un programme RPC seront les étiquettes visibles d'un système de transitions, tandis que le contrôle du parallélisme: invocation de coroutine, synchronisation par `wait`, terminaison d'une coroutine par `end`, est représenté par des transitions internes invisibles.

Puisque nous n'avons pas interprété les actions de base dans G, notre notion de comportement ne capture que partiellement le comportement réel d'un programme P auquel on voudrait associer le schéma G. La différence consiste en ce que:

- $\mathcal{M}_G$ ne prend pas en compte l'existence de variables ni de leurs valeurs dans un état-mémoire donné dans sa définition d'un état, et
- $\mathcal{M}_G$ considère des instructions du type "`if ... then` c_1 `else` c_2" comme non-déterminées "c_1 `ou` c_2".

Cette simplification n'empêche pas d'analyser de façon pertinente le comportement d'un programme (réel) P par l'examen de $\mathcal{M}_G$: cf. chapitres 9, 10, 11.

6.4 Etude du comportement des schémas

Certaines conséquences de la définition 6.3.1 peuvent être formulées immédiatement comme propriétés de base.

Proposition 6.4.1. *$\mathcal{M}_G$ est un système de transitions à branchement fini.*

Preuve. Soit G et ξ fixés. On montre, par induction sur ξ, qu'il n'existe qu'un nombre fini de transitions issues de ξ.

Puisque le schéma G est fini[1], les règles ACT, CALL, WAIT et END ne permettent de dériver qu'un nombre fini de transitions partant de ξ. Les règles $\mathrm{PAR}_{1,2}$ permettent de dériver des transitions partant de ξ à partir de transitions partant d'états ξ' de taille plus petite que celle de ξ. Puisque il y a un nombre fini de ξ', puisque une transition de la forme $\xi' \stackrel{\alpha}{\longmapsto} \zeta'$ ne peut, au maximum, donner lieu qu'à une transition $\xi \stackrel{\alpha}{\longmapsto} \zeta$ par une des règles PAR, l'hypothèse d'induction nous permet de borner le nombre de transitions issues de ξ.

□

Maintenant nous allons aborder le lemme suivant qui formule une des propriétés de base: il existe un seul état terminal. L'intérêt de cette propriété n'est pas l'unicité mais le fait qu'on décrive les états terminaux de façon "syntaxique" (cf. la règle WAIT): on sait relier Ø avec un comportement. Formellement,

Proposition 6.4.2.

$$\xi \longmapsto \quad ssi \quad \xi \neq \emptyset$$

Preuve.

1. "⇒" On suppose qu'il existe une transition $\xi \longmapsto \xi'$ et on raisonne par induction sur la structure de la dérivation de $\xi \longmapsto \xi'$.

   (a) La dernière règle appliquée est ACT et donc ξ est de la forme q, ζ et $\xi \neq \emptyset$.

[1] en fait, il est suffisant de considérer des schémas où chaque noeud a un nombre fini de suivants et d'invoqués.

(b) Pour les règles CALL, WAIT, END le raisonnement est le même.

(c) Pour la règle PAR_1 on a ξ de la forme q, ζ et $\zeta \longmapsto$. C'est évident que $\xi \neq \emptyset$.

(d) La preuve pour la règle PAR_2 utilise l'hypothèse d'induction. On a ξ de la forme $\xi_1 + \xi_2$ et $\xi_i \longmapsto \xi_i'$ ($i = 1$ ou 2). Par hypothèse d'induction $\xi_i \neq \emptyset$ et donc $\xi \neq \emptyset$.

2. La direction "$\Leftarrow$" se montre par induction sur la taille de ξ. Il existe deux cas à considérer:

(a) si

i. $\xi = q, \xi_1$ avec ξ_1 non-vide ou

ii. $\xi = \xi_1 + \xi_2$ avec ξ_1 et ξ_2 non-vides,

alors l'hypothèse d'induction nous donne $\xi_1 \longmapsto$ et on en déduit $\xi \longmapsto$ par la règle PAR_1 (ou respectivement PAR_2).

(b) sinon $\xi = q, \emptyset$ et une des règles ACT, CALL, WAIT, END s'applique forcément à ξ.

□

Avant de formuler la propriété suivante, on va l'expliquer informellement: si q n'est pas un noeud `wait`, alors le comportement de q, ξ reste possible en présence de fils supplémentaires. Cette situation est reflétée dans l'exemple 6.5 où une exécution présentée contient

$$q_2, (q_9, \emptyset) \stackrel{\alpha_2}{\longmapsto} q_4, (q_9, \emptyset) \stackrel{\beta_1}{\longmapsto} q_5, (q_9, \emptyset)$$

Le lemme suivant n'est qu'une traduction formelle de ce commentaire.

Lemme 6.4.3.

$$\begin{array}{ll} \textit{Si} & q, \xi \stackrel{\alpha}{\longmapsto} q', \xi' \\ & q \textit{ n'est pas un } \texttt{wait}\textit{-noeud}, \end{array}$$

$$\textit{alors } q, \xi + \xi'' \stackrel{\alpha}{\longmapsto} q', \xi' + \xi''$$

Preuve. Par induction sur la dérivation de $q, \xi \stackrel{\alpha}{\longmapsto} q', \xi'$.

1. Le résultat est clairement vrai pour les règles ACT, CALL, END et vrai à défaut d'application pour WAIT.

2. Si maintenant on suppose que PAR_2 est la dernière règle appliquée il faut que q, ξ soit une union $\zeta + \zeta'$. Alors ζ ou ζ' est vide, et l'hypothèse d'induction nous donne immédiatement le résultat.

3. Si enfin la dernière règle appliquée est PAR_1, alors $q = q'$ et on a $\xi \stackrel{\alpha}{\longmapsto} \xi'$ comme prémisse. Par application de la règle PAR_2 on peut obtenir $\xi + \xi'' \stackrel{\alpha}{\longmapsto} \xi' + \xi''$ et ensuite, avec PAR_1 on a $q, \xi + \xi'' \stackrel{\alpha}{\longmapsto} q, \xi' + \xi''$. Puisque $q = q'$ on obtient le résultat recherché.

□

Maintenant il nous semble très important de souligner les liens entre nos états hiérarchiques (père, fils, etc.) et les notions habituelles de compositions parallèles, séquentielles, etc. de PA. Même si les points de départ de notre modèle d'états hiérarchiques et de formalisme de PA sont différents, il peut être plus naturel de comprendre notre modèle à un autre niveau car il est possible de relier l'état $\emptyset$ de M_G au terme $\mathbf{0}$ de PA, la somme d'états de M_G à la composition parallèle de PA et la composition hiérarchique d'états de M_G à la composition séquentielle de PA. En plus de l'éclairage ainsi apporté, ces propriétés nous seront utiles pour l'étude de l'expressivité de notre modèle (chapitre 7).

On suppose une déclaration Δ de PA et on note $\mathcal{A}$ le système de transition ainsi engendré.

Lemme 6.4.4. $\emptyset \underline{\leftrightarrow} \mathbf{0}$

Preuve. Immédiate comme corollaire de la proposition 6.4.2.

□

Lemme 6.4.5. $\forall\, \xi_1, \xi_2 \in M_G,\ \forall\, E_1, E_2 \in St_{\mathcal{A}}$:

$$si \quad \begin{array}{l} \xi_1 \underline{\leftrightarrow} E_1, \\ \xi_2 \underline{\leftrightarrow} E_2, \end{array} \quad alors\ \xi_1 + \xi_2 \underline{\leftrightarrow} E_1 \| E_2$$

Preuve. On va montrer que S est une bisimulation, où

$$S \stackrel{déf}{=} \{(\varphi_1 + \varphi_2, F_1 \| F_2,) :\ \varphi_1 \underline{\leftrightarrow} F_1, \varphi_2 \underline{\leftrightarrow} F_2\} \quad \varphi_1, \varphi_2 \in M_G,\ F_1, F_2 \in St_{\mathcal{A}}.$$

Pour cela, il faut vérifier la propriété de transfert et on considère donc une paire $(\varphi_1 + \varphi_2, F_1 \| F_2) \in S$.

- Soit une transition $\varphi_1 + \varphi_2 \stackrel{\alpha}{\longmapsto} \zeta$. Au vu des règles de transitions de la définition 6.3.1 et en prenant en considération que $\varphi_1 + \varphi_2 = \varphi_2 + \varphi_1$ pour les multi-ensembles φ_1, φ_2 on a les deux cas suivants:

- **Cas 1:** $\varphi_1 \stackrel{\alpha}{\longmapsto} \varphi_1'$ et $\zeta = \varphi_1' + \varphi_2$. Puisque $\varphi_1 \underline{\leftrightarrow} F_1$ on a une transition $F_1 \stackrel{\alpha}{\rightarrow} F_1'$ dans $\mathcal{A}$ t.q. $\varphi_1' \underline{\leftrightarrow} F_1'$. Par application de la règle pour la composition parallèle, on déduit $F_1 \| F_2 \stackrel{\alpha}{\rightarrow} F_1' \| F_2$. Clairement, $(\zeta, F_1' \| F_2) \in S$.
- **Cas 2:** $\varphi_2 \stackrel{\alpha}{\longmapsto} \varphi_2'$ et $\zeta = \varphi_1 + \varphi_2'$. Un raisonnement symétrique convient.

- L'autre direction de la propriété de transfert se traite de façon semblable.

□

Lemme 6.4.6. *Soit $q \in Q_G$ un* `wait`*-noeud. $\forall\, \xi_1 \in M_G$, $\forall\, E_1, E_2 \in St_{\mathcal{A}}$:*

$$si \quad \begin{array}{l}(q, \emptyset) \underline{\leftrightarrow} E_2, \\ \xi_1 \underline{\leftrightarrow} E_1,\end{array} \quad alors\ (q, \xi_1) \underline{\leftrightarrow} E_1.E_2$$

Preuve. On va montrer que S est une bisimulation, où

$$S \stackrel{déf}{=} \{((p, \varphi_1), F_1.F_2) :\ \varphi_1 \underline{\leftrightarrow} F_1, (p, \emptyset) \underline{\leftrightarrow} F_2\} \cup \underline{\leftrightarrow}$$

pour $F_1, F_2, F_3 \in St_{\mathcal{A}}$, p un `wait`-noeud de Q, et $\varphi_1 \in M_G$.

Il faut vérifier la propriété de transfert. On considère donc une paire dans S de la forme $((p, \varphi_1), F_1.F_2) \in S$ (clairement, les autres paires ne posent pas de problème).

Soit une transition $F_1.F_2 \stackrel{\alpha}{\rightarrow} G$. Selon le type des règles de dérivation pour composition séquentielle on a deux cas:

- **Cas 1:** $F_1 \stackrel{\alpha}{\rightarrow} F_1'$ et $G = F_1'.F_2$. Puisque $F_1 \underline{\leftrightarrow} \varphi_1$, par définition il existe une transition $\varphi_1 \stackrel{\alpha}{\longmapsto} \varphi_1'$ dans $\mathcal{M}_G$ t.q. $F_1' \underline{\leftrightarrow} \varphi_1'$. Par application de PAR_1 on est amené à $p, \varphi_1 \stackrel{\alpha}{\longmapsto} \zeta\ (= p, \varphi_1')$. On voit aisément que $(\zeta, G) \in S$.

- **Cas 2:** $F_2 \stackrel{\alpha}{\rightarrow} F_2'$, $isnil(F_1)$ et $G = F_2'$. Par hypothèse du lemme, on a $F_1 \underline{\leftrightarrow} \varphi_1$, d'où il vient $\varphi_1 \underline{\leftrightarrow} \mathbf{0}$ et, par la proposition 6.4.2, $\varphi_1 = \emptyset$. En outre, puisque $(p, \emptyset) \underline{\leftrightarrow} F_2$, il existe une transition $p, \emptyset \stackrel{\alpha}{\longmapsto} \varphi_2$ dans $\mathcal{M}_G$ t.q. $F_2' \underline{\leftrightarrow}_d \varphi_2$. En résumé, $(p, \varphi_1 =)\ p, \emptyset \stackrel{\alpha}{\longmapsto} \zeta = \varphi_2$ et on voit que $(\zeta, G) \in S$.

Soit maintenant une transition $p, \varphi_1 \stackrel{\alpha}{\longmapsto} \zeta$. Selon le type des règles de dérivation de $\mathcal{M}_G$ on a deux cas possibles.

- **Cas 1:** la règle PAR_1, alors $\varphi_1 \stackrel{\alpha}{\longmapsto} \varphi_1'$ et $\zeta = p, \varphi_1'$. Puisque $\varphi_1 \underline{\leftrightarrow} F_1$ il existe une transition $F_1 \stackrel{\alpha}{\rightarrow} F_1'$ dans $\mathcal{A}$ t.q. $\varphi_1' \underline{\leftrightarrow} F_1'$ et donc, par application des règles pour la composition séquentielle on a

$$F_1.F_2 \stackrel{\alpha}{\rightarrow} G = F_1'.F_2$$

  On voit sans peine que $(\zeta, G) \in S$.

- **Cas 2:** la règle WAIT, alors $p, \emptyset \stackrel{\alpha}{\longmapsto} \varphi_2$. Puisque p est un `wait`-noeud, on a $\varphi_1 = \emptyset$ et $\zeta = \varphi_2$. Par hypothèse du lemme $p, \emptyset \underline{\leftrightarrow} F_2$ et donc, il existe une transition $F_2 \stackrel{\alpha}{\rightarrow} F_2'$ t.q. $\varphi_2 \underline{\leftrightarrow} F_2'$. En outre, $F_1 \underline{\leftrightarrow} \emptyset$ implique $F_1 \underline{\leftrightarrow} \mathbf{0}$ (par le lemme 6.4.4) et donc on a $isnil(F_1)$ (par la proposition 5.2.3). Dans cette situation on obtient $F_1.F_2 \stackrel{\alpha}{\rightarrow} G = F_2'$. On voit sans peine que $(\zeta, G) \in S$.

□

Ces différentes propriétés permettent de saisir intuitivement les liens (et les différences) entre notre modèle d'états hiérarchiques et les combinateurs de PA.

D'abord, nous notons que les états hiérarchiques sont plus concrets, ils ont une cohérence interne. Ensuite, une question importante est la différence du point de vue donnée par les primitives de synchronisation. Comme notre modèle est inspiré de toute une famille de langages récursifs-parallèles, nous avons tenu compte de ces primitives, ce qui implique de traiter explicitement des problèmes dépendant de la synchronisation. Il est connu que la synchronisation n'est pas incorporée à PA.

Nous avons déjà mentionné que l'approche de PA est très intéressante sur le plan algébrique, mais elle n'est pas plus ni moins complète et souffre des ses limitations théoriques. Un des inconvénients de notre modèle est de ne pas avoir une structure algébrique riche, mais les propositions que nous venons de donner permettent de relier partiellement la structure des états hiérarchiques avec les combinateurs usuels de PA.

6.5 Etats hiérarchiques normés

Nous avons déjà mentionné dans l'introduction de cette thèse que pour de nombreux modèles de parallélisme qui ne sont pas d'états finis la plupart des problèmes de vérification deviennent indécidables. Nous avons aussi mentionné l'approche décrite dans [Fin90, ACJY96] pour l'analyse de tels systèmes.

Il nous serait très utile de disposer d'une structure de système de transitions bien structuré pour l'analyse de nos schémas. Ce sera le thème du chapitre 8. En préparation nous allons

- en premier lieu, introduire une notion d'état hiérarchique qui *peut terminer* (elle fera une partie d'un autre formalisme (cf. la section 8.3)) et
- en second lieu, considérer de près le problème de savoir si (un état d') un schéma G de RPPS peut terminer.

La définition suivante est justifiée par la proposition 6.4.2:

Définition 6.5.1. *Un état hiérarchique ξ peut terminer, ce qu'on note $\xi \downarrow$, ssi il existe une séquence finie $\mu = \alpha_1 \dots \alpha_n \in \Lambda_\tau^*$ tel que $\xi \stackrel{\alpha_1}{\longmapsto} \cdots \stackrel{\alpha_n}{\longmapsto} \emptyset$.*

Par dualité, on note $\xi \uparrow$ le fait qu'aucun comportement issu de ξ ne termine.

En suivant [Chr93] on dit qu'un état qui peut terminer est *normé*. La *norme* d'un état $\xi \in M_G$ t.q. $\xi \downarrow$ est le plus petit nombre de pas requis pour la terminaison:

Définition 6.5.2. *Pour tout $\xi \in M_G$ t.q. $\xi \downarrow$, la* norme *de ξ, noté $\mathcal{N}(\xi)$, est*

$$\mathcal{N}(\xi) \stackrel{\text{déf}}{=} min\{|\mu| \,:\, \xi \stackrel{\mu}{\longmapsto} \emptyset \ , \mu \in \Lambda_\tau^*\}.$$

Si $\xi \uparrow$, alors $\mathcal{N}(\xi) \stackrel{\text{déf}}{=} \infty$.

Une des propriétés importantes de la norme de l'état hiérarchique que nous allons utiliser est celle d'être *additive*. Ce fait est exprimé par la proposition suivante:

Proposition 6.5.3. *Soient $\xi, \xi_1, \xi_2 \in M_G$. On a*

1.

$$\mathcal{N}(\xi_1 + \xi_2) = \mathcal{N}(\xi_1) + \mathcal{N}(\xi_2),$$

2.

$$\mathcal{N}(\{(q, \xi)\}) = \mathcal{N}(\{(q, \emptyset)\}) + \mathcal{N}(\xi).$$

Preuve.

1. Grâce au lemme 6.4.5, $\mathcal{N}(\xi_1 + \xi_2) = \mathcal{N}(\xi_1) + \mathcal{N}(\xi_2)$ peut être prouvé en reprenant la preuve de [Chr93] pour l'opérateur de composition parallèle de PA.

2. La preuve pour $\mathcal{N}(\{(q,\xi)\}) = \mathcal{N}(\{(q,\emptyset)\}) + \mathcal{N}(\xi)$ est similaire.

   Soit un comportement de q,ξ, on peut le décomposer en une combinaison d'un comportement de $q,\emptyset$ et d'un comportement de ξ, de sorte qu'on obtient $\mathcal{N}(\{(q,\xi)\}) \geq \mathcal{N}(\{(q,\emptyset)\}) + \mathcal{N}(\xi)$.

   On montre maintenant qu'il existe un comportement t.q.

   $$\mathcal{N}(\{(q,\xi)\}) \leq \mathcal{N}(\{(q,\emptyset)\}) + \mathcal{N}(\xi)$$

   Si $\mathcal{N}(\{(q,\emptyset)\}) = \infty$ ou $\mathcal{N}(\xi) = \infty$, alors l'assertion du lemme est vraie.

   Supposons que $\mathcal{N}(\{(q,\emptyset)\}) = n$ et $\mathcal{N}(\xi) = m$ c.-à-d. $(q,\emptyset) \longmapsto^n \emptyset$ et $\xi \longmapsto^m \emptyset$. On voit aisément qu'il existe un comportement

   $$(q,\xi) \longmapsto^m (q,\emptyset) \longmapsto^n \emptyset$$

   et cela complète la preuve.

□

Avant de nous attaquer à la décision de "$\xi \downarrow$?", nous donnons quelques lemmes qui permettront de décomposer le problème et de l'exprimer en termes de noeuds du graphe G.

D'abord, les définitions 6.5.1, 6.5.2 et la proposition 6.5.3 nous assurent les propriétés suivantes:

Lemme 6.5.4. *Soient $\xi, \xi' \in M_G$ et $q \in Q_G$.*

1. $\xi \uparrow$ *ssi* $\forall\, \xi'\ (\xi + \xi') \uparrow$
2. $\xi \uparrow$ *implique* $\forall\, q \in Q_G : (q,\xi) \uparrow$
3. $(\xi + \xi') \uparrow$ *ssi* $\xi \uparrow$ *ou* $\xi' \uparrow$
4. $(q,\xi) \uparrow$ *ssi* $\xi \uparrow$ *ou* $(q,\emptyset) \uparrow$
5. $\xi + (q,\xi') \uparrow$ *ssi* $\xi + (q,\emptyset) \uparrow$ *ou* $\xi' \uparrow$

Nous pouvons décider si $\xi \downarrow$ en deux étapes. Premièrement, nous réduisons le fait d'être normé pour un état hiérarchique arbitraire de M_G au fait d'être normé pour un noeud de G au moyen des égalités suivantes (qui sont la conséquence du lemme 6.5.4):

$$\begin{aligned}(\xi + \xi') \downarrow &= \xi \downarrow \text{ et } \xi' \downarrow \\ \emptyset \downarrow &= true \\ (q, \xi) \downarrow &= \xi \downarrow \text{ et } (q, \emptyset) \downarrow\end{aligned}$$

Remarquons qu'on adopte ici un point de vue suivant lequel "$\downarrow$" est une fonction booléenne. En écrivant $q \downarrow$ au lieu de $(q, \emptyset) \downarrow$, nous pouvons formuler les égalités suivantes, qui sont basées sur la structure de G:

$$q \downarrow = true \text{ si } q \text{ est un noeud } \texttt{end}, \tag{6.1}$$

$$q \downarrow = q' \downarrow \text{ si } q' \text{ est le suivant d'un } \texttt{wait}\text{-noeud } q, \tag{6.2}$$

$$q \downarrow = \bigvee\{q' \downarrow,\ q' \text{ un suivant d'un noeud d'action ou de test } q\}, \tag{6.3}$$

$$q \downarrow = q' \downarrow \text{ et } q'' \downarrow \text{ si } q', q'' \text{ sont resp. le suivant et le noeud invoqué d'un noeud } \texttt{pcall}\ q. \tag{6.4}$$

On va démontrer que le prédicat "$\downarrow$" est la plus petite solution de l'ensemble d'équations (6.1-6.4) (lues en prenant $\downarrow$ comme inconnue), où "plus petite" considère l'ordre induit sur les applications de Q dans l'ensemble $\{false, true\}$, avec $false \leq true$.

Pour cela, deux points sont à mettre en évidence:

1. évidemment, "$\downarrow$" est bien une solution de (6.1-6.4),

2. $\downarrow$ est plus petit que n'importe quelle solution.

Soit donc $\searrow$ une solution de l'ensemble d'équations (6.1-6.4).

Lemme 6.5.5. *Si $q \downarrow$, alors $q \searrow$*

Preuve. On démontre par induction sur la norme de $q, \emptyset$. Considérons les différents cas possibles:

1. Si q est un noeud `end`, alors forcément $q, \emptyset \searrow$, cf. (6.1).

2. Si q est un noeud `wait`, alors il existe une exécution finie partant de $q, \emptyset$, forcément de la forme $q, \emptyset \stackrel{\tau}{\longmapsto} q', \emptyset \stackrel{\tau}{\longmapsto} \ldots \emptyset$. Par hypothèse d'induction $q' \downarrow$ implique $q' \searrow$ et donc $q \searrow$ puisque $\searrow$ vérifie les équations (6.2).

3. Si q est un noeud d'action ou de test tel que $q, \emptyset \stackrel{\alpha}{\longmapsto} \xi$, alors $\xi = q', \emptyset$ pour un q' suivant de q.

$$\mathcal{N}(q, \emptyset) = 1 + \min_{q' \text{ suivant de } q} \mathcal{N}(q', \emptyset)$$

Donc q a un suivant q' de norme moindre que lui. L'hypothèse d'induction nous donne $q' \searrow$ et par application de (6.3) on obtient $q \searrow$.

4. Si q est un `pcall`-noeud tel que $q, \emptyset \stackrel{\tau}{\longmapsto} \xi$, alors $\xi = q', (q'', \emptyset)$ en notant q' et q'' son suivant et son noeud invoqué. Par la proposition 6.5.3 on a

$$\mathcal{N}(q, \emptyset) = 1 + \mathcal{N}(q', \emptyset) + \mathcal{N}(q'', \emptyset)$$

Ainsi q' et q'' ont une norme moindre. Par hypothèse d'induction on a alors $q' \searrow$ et $q'' \searrow$ et donc $q \searrow$ grâce à (6.4).

□

Maintenant, le prédicat "$\downarrow$" peut être calculé par les algorithmes habituels de point fixe:

Lemme 6.5.6. *On peut décider si $q \downarrow$ grâce à l'algorithme suivant.*

Algorithme 6.5.7. *1.* **initialisation:** *faire $mark[q] := True$ pour tous les $q \in Q_G$,*

2. répéter jusqu'à **stabilisation:**
si

$$\exists\, q \text{ tel que } \begin{array}{ll} & mark[q] == True \;\&\; q \not\longmapsto \\ ou & \\ & q \longmapsto q' \qquad\quad \&\; mark[q'] == False \\ ou & \\ & q, \emptyset \stackrel{\tau}{\longmapsto} q', (q'', \emptyset) \;\&\; mark[q'] == mark[q''] == False \end{array}$$

alors $mark[q] := False$

Théorème 6.5.8. *(Terminaison et Correction)*
L'algorithme termine et à la fin $mark[q] == False$ ssi $(q, \emptyset) \downarrow$.

Corollaire 6.5.9. *On peut décider si $\xi \downarrow$.*

Remarquons que le corollaire 6.5.9 dit que la propriété "ξ est normé" est décidable. Il ne s'agit pas du problème classique de l'arrêt, qui sera considéré dans la section 8.3, mais plutôt d'un cas particulier de problème d'atteignabilité.

6.6 Conclusion

Les schémas de RPPS et leur sémantique d'états hiérarchiques sont un modèle formel du parallélisme, inspiré par le langage récursif-parallèle RPC. Par la façon dont les états hiérarchiques évoquent des marquages du schéma fini G, ce modèle fait penser aux réseaux de Petri. Néanmoins, il ne peut pas être considéré comme un type spécial de réseaux de haut niveau au sens p.ex. de [Smi96].

Il existe d'autres modèles du parallélisme. A notre connaissance, le modèle RPPS que nous avons introduit n'est pas clairement apparenté à d'autres modèles du parallélisme:

- Les résultats que nous présenterons au chapitre 7 semblent indiquer un lien entre les restrictions à la synchronisation que fait le langage RPC, et le fragment PA des algèbres de processus. Néanmoins, le point de vue des algèbres de processus a ses propres avantages et inconvénients. Un avantage de RPPS est que notre notion d'états hiérarchiques fait abstraction de la syntaxe qui est exigée dans PA et ainsi donne une vision nette d'une forme normale pour des états. Ainsi, les méthodes que nous développons au chapitre 8, basée sur cette structure des états hiérarchiques, ne semble pas devoir s'appliquer aux algèbres de processus.

- Dans ce sens, les états hiérarchiques font davantage penser aux marquages de réseaux de Petri. Cependant, les schémas RPPS ne peuvent pas être vus comme un type spécial de réseaux de haut niveau [Smi96]. Dans les réseaux prédicats/transitions, ou bien dans les réseaux colorés, c'est la nature des jetons qui est changée. Dans RPPS, les "jetons" ne sont pas plus riches. Plutôt, ils sont emboîtés avec des relations précises de dépendance qui sont héritées quand le jeton se déplace.

- D'autres modèles permettant la récursivité dans le parallélisme existent, et sont même faciles à construire p.ex. dans un cadre de sémantique dénotationnelle [Mos90], ... En général, ces modèles ne cherchent pas à contrôler la puissance d'expressivité et perdent alors la décidabilité de nombreuses propriétés.

- Peut-être les systèmes de réécriture de termes sont le formalisme le plus naturel dans lequel nous pourrions présenter notre proposition de modèle RPPS. Dans un tel cadre, le côté arborescent des états hiérarchiques serait plus banal. Mais les systèmes de réécriture de termes sont un formalisme beaucoup plus général que RPPS (dès lors, ils ont moins de

propriétés décidables) et ne sont pas particulièrement orientés vers la description de calculs parallèles.

Chapitre 7

Etude d'expressivité du modèle

Dans ce chapitre nous nous intéressons aux questions d'expressivité du modèle RPPS.

Les schémas de $RPPS_{\Lambda_\tau}$ ont un pouvoir d'expression limité, mais néanmoins non trivial. Par exemple, ils autorisent qu'une composante parallèle attende la terminaison de toutes ses procédures-filles, ce qui n'est pas possible dans les réseaux de Petri (RdP). Mais, par ailleurs, ils ne permettent pas la forme générale de synchronisation de composantes parallèles qu'on trouve avec les RdP.

Afin de situer notre modèle parmi d'autres qui jouent un rôle fondamental dans la littérature, nous étudierons le pouvoir d'expression en termes de langages engendrés. Outre les schémas RPPS, on considérera PA, BPA, BPP et RdP comme des mécanismes générant des langages. Notre étude s'appuiera sur des travaux antérieurs de Baker [Bak72], Hack [Hac75], Peterson [Pet74], lesquels ont été développés par Starke [Sta78], Jantzen [Jan86] pour RdP et par Christensen [Chr93] pour BPP.

Premièrement, nous définirons la classe des langages générés par les schémas de RPP; nous notons cette classe L(RPPS).

Pour annoncer les principaux résultats de ce chapitre, nous comparerons L(RPPS) à la classe des langages engendrés par les processus de PA, ce qu'on note L(PA).

Notre démarche sera la suivante.

- D'abord nous montrerons comment on peut coder une déclaration gardée de PA dans les schémas de RPPS et nous démontrerons que ce codage est correct au sens où il préserve les langages modulo τ.

- Ensuite, à partir d'un schéma de RPPS nous construirons une déclaration de PA correspondante, elle aussi préservant le langage modulo τ.

Ainsi, nous arriverons à la conclusion que les schémas RPPS ont le même pouvoir d'expression que l'algèbre PA en termes de langages engendrés modulo les actions silencieuses, c.-à-d. en adoptant un point de vue "temps linéaire". Ensuite, par l'intermédiaire de L(PA) nous comparerons L(RPPS) avec L(BPP), L(BPA) et finalement, avec L(RdP), la classe des langages générés par les réseaux de Petri étiquetés.

D'autres points de vue sur la comparaison des pouvoirs d'expression sont possibles. D'un point de vue "temps arborescent" la situation est plus compliquée. Nous constaterons que notre codage permettrait, moyennant certains détails techniques, de préserver le comportement arborescent des processus PA quand on les code en RPPS. Dans la direction opposée, il ne nous paraît pas possible de coder en PA le comportement arborescent de nos schémas RPPS (nous n'avons pas de preuve formelle de cette conjecture.)

7.1 Langages des schémas RPPS

Pour commencer ce chapitre, nous allons définir le langage généré par un schéma, à partir d'un état initial donné. Cette notion est une variante de la notion de trace introduite dans la section 3.6.

Définition 7.1.1. *Soit $\mathcal{M}_G = (M_G, \Lambda_\tau, \longmapsto)$ le système de transitions étiqueté associé à un schéma G, soit $\xi \in M_G$. Le* langage généré par G à partir de ξ *est l'ensemble de traces de ξ dans $\mathcal{M}_G$: $Tr_{\mathcal{M}_G}(\xi)$. On dit que deux états ξ_1 et ξ_2 sont* équivalents en termes de langages *ssi $Tr(\xi_1) = Tr(\xi_2)$.*

La même définition peut être facilement adaptée pour *le langage modulo* τ généré par G à partir de ξ: c'est l'ensemble de traces modulo τ de ξ, noté $Tr_\tau(\xi)$. On dit que deux états ξ_1 et ξ_2 sont *équivalents en termes de langages modulo* τ ssi $Tr_\tau(\xi_1) = Tr_\tau(\xi_2)$.

Rappelons que notre notion de trace (définition 3.6.2) considère des traces complètes, ou traces de chemins maximaux (d'exécutions) qui peuvent être infinis. Donc, $Tr(\xi)$ ne prend pas seulement en considération les comportements conduisant de l'état hiérarchique ξ à l'état final $\emptyset$ comme dans la théorie classique des automates.

Exemple 7.1. Langages de traces.

Pour le système de transitions étiqueté associé au schéma G de la figure 6.1 et pour l'état hiérarchique $\xi_1 = \{q_5, (q_{10}, (q_8, \emptyset))\}$:

$$\begin{array}{lcl} Tr(\xi_1) & = & \{\alpha_4\alpha_5\tau^4\alpha_3\tau, \alpha_5\alpha_4\tau^4\alpha_3\tau\} \\ Tr_\tau(\xi_1) & = & \{\alpha_4\alpha_5\alpha_3, \alpha_5\alpha_4\alpha_3\} \end{array}$$

Parmi les comportements généré par G à partir de l'état hiérarchique $\xi_2 = \{q_5, (q_9, \emptyset)\}$ $(\in M_G)$ on a celui qui est infini:

$$q_5, (q_9, \emptyset) \stackrel{\tau}{\longmapsto} q_5, (q_{10}, (q_3, \emptyset)) \stackrel{\alpha_5}{\longmapsto} q_5, (q_{11}, (q_3, \emptyset)) \stackrel{\beta_2}{\longmapsto}$$

$$q_5, (q_{11}, (q_9, \emptyset)) \stackrel{\tau}{\longmapsto} q_5, (q_{11}, (q_{10}, (q_3, \emptyset))) \stackrel{\beta_2}{\longmapsto} q_5, (q_{11}, (q_{10}, (q_9, \emptyset))) \stackrel{\tau}{\longmapsto} \ldots$$

Ainsi, $\tau\alpha_5(\beta_2\tau)^\omega \in Tr(\xi_2)$ et $\alpha_5(\beta_2)^\omega \in Tr_\tau(\xi_2)$.

La classe de langages qui nous intéresse d'un bout de ce chapitre à l'autre est formellement définie comme suit:

Définition 7.1.2. *(Classe L(RPPS))*
La classe L(RPPS) est composée des langages générés par les schémas de RPPS. C'est-à-dire, L est un langage de L(RPPS) s'il existe un schéma G de RPPS tel que $L = Tr_\tau(\xi_0)$, où ξ_0 est l'état initial de G.

7.2 De PA à RPPS

Nous considérons une déclaration Δ de PA, en fng, et souhaitons la traduire en un schéma RPPS. Δ est de la forme

$$\Delta = \{X_i \stackrel{déf}{=} \sum_{j=1}^{n_i} \alpha_{i_j} T_{i_j} \mid i = 1, \ldots, n\}$$

Nous serons amenés au résultat de correction de traduction en utilisant les idées et les démarches suivantes.

- Les exemples de schémas vus auparavant (cf. chapitre 6) montrent qu'on peut exprimer en RPPS des notions de choix non-déterministe, de mise en parallèle et de composition séquentielle. Nous formaliserons ces constructions au moyen de *motifs* génériques qui sont présentés dans cette section. C'est la section 7.3 qui est consacrée à la preuve de correction de cette modélisation.

La première présentation est celle du motif de la figure 7.1 qui sert à modéliser le **choix non-déterministe** de PA. On note que notre construction utilise un noeud q étiqueté par τ et c'est la position des actions sur les noeuds dans le formalisme d'un schéma de RPPS qui nous oblige à mettre cette étiquette τ auxiliaire.

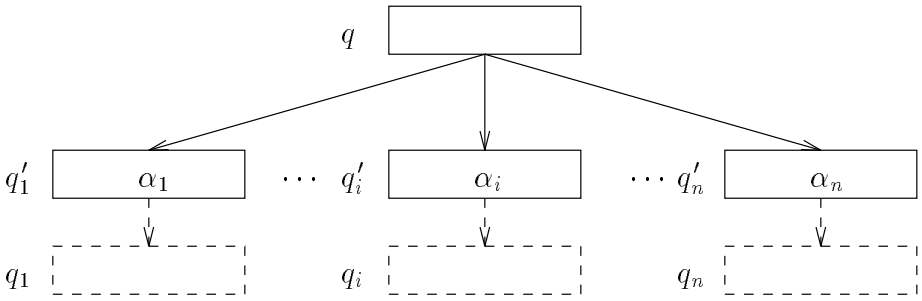


Figure 7.1 : Motif RPPS pour le choix non-déterministe

Quant à la **composition séquentielle**, il est facile de voir que, si on met un `wait`-noeud comme le suivant d'un noeud d'invocation d'une procédure parallèle, la procédure parentale ne peut continuer son exécution qu'une fois que la coroutine invoquée ait terminé. Cette remarque est exprimée par le motif de la figure 7.2 qui utilise, lui aussi, deux noeuds q (d'invocation) et q' (de synchronisation) étiquetés par τ.

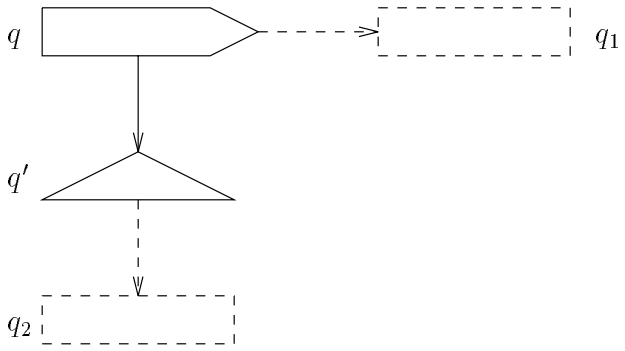


Figure 7.2 : Motif RPPS pour la composition séquentielle

Avant de présenter un motif qui implémente la **composition parallèle**, on remarque que l'exécution d'une instruction d'invocation, le `pcall`, de RPPS peut mettre en parallèle le suivant et l'invoqué. On considère un schéma G contenant le motif de la figure 7.3. Si le suivant q_{n-1} ne peut pas atteindre une instruction de synchronisation `wait`,

alors il s'exécute simultanément avec l'invoqué q'_n. Cette idée est utilisée pour introduire un motif générique pour la mise en parallèle (cf. la figure 7.3.) De nouveau, notre construction demande des noeuds (`pcall` et `end`) étiquetés par τ.

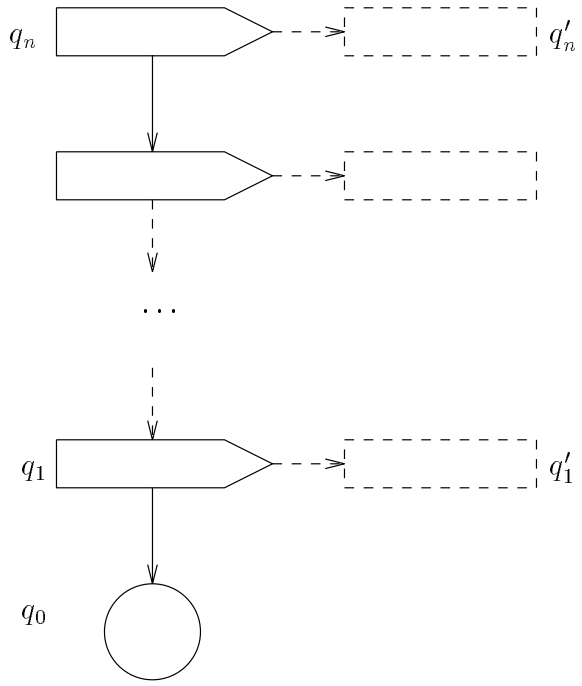


Figure 7.3 : Motif RPPS pour la composition parallèle

- Comme PA utilise seulement le choix non-déterministe, la mise en parallèle et la composition séquentielle, notre ensemble de motifs est suffisant.
- Mais (comme on a déjà remarqué) ces motifs génériques utilisent quelques noeuds auxiliaires pour la construction et en plus la position des actions sur les noeuds demande quelques τ auxiliaires.
- On doit donc raisonner modulo τ.
- Techniquement, on va traduire une déclaration Δ en Δ_τ obtenue simplement en insérant des préfixages par τ

$$\Delta_\tau = \{X_i \stackrel{déf}{=} \sum_{j=1}^{n_i} \tau.\alpha_{ij}.T_{ij} \mid i = 1, \ldots, n\}$$

Ainsi, nous obtenons une déclaration que nos motifs nous permettent de traiter, donnant lieu à un schéma G_{Δ_τ}. Finalement, on obtiendra (cf. les sections 7.4 et 7.5)

$$\Delta \quad \sim_{Tr_\tau} \quad \Delta_\tau \quad \sim_{Tr_\tau} \quad G_{\Delta_\tau}$$

qui nous permettra d'affirmer la correction du codage

$$\Delta \quad \sim_{Tr_\tau} \quad G_{\Delta_\tau}$$

(cf. la définition 4.6.1, page 54 pour $\sim_{Tr_\tau}$).

7.3 Motifs RPPS et leur correction

Comme nous l'avons déjà remarqué, le formalisme RPPS et les particularités de la construction des motifs génériques nous obligent à utiliser des noeuds auxiliaires portant des étiquettes τ. C'est la raison pour laquelle nous allons tout d'abord, généraliser les lemmes 6.4.5 et 6.4.6 au cas de la d-bisimulation. Ce sont deux lemmes dont nous avons besoin pour énoncer la correction du codage.

Lemme 7.3.1. $\forall\, E_1, E_2 \in \mathcal{EXP} \;\; \forall\, \xi_1, \xi_2 \in M_G \;\; :$

$$si \quad \begin{array}{l} E_1 \underline{\leftrightarrow}_d \xi_1 \\ E_2 \underline{\leftrightarrow}_d \xi_2, \end{array} \quad alors \; E_1 \| E_2 \underline{\leftrightarrow}_d \xi_1 + \xi_2.$$

Preuve. Immédiat d'après les propositions 5.4.3 et 6.4.5.

□

Lemme 7.3.2. *Soit $q \in Q$* `wait`*-noeud de G.*
$\forall\, E_1, E_2 \in \mathcal{EXP} \;\; \forall\, \xi_1 \in M_G \;\; :$

$$si \quad \begin{array}{l} E_1 \underline{\leftrightarrow}_d \xi_1 \\ E_2 \underline{\leftrightarrow}_d (q, \emptyset), \end{array} \quad alors \; E_1.E_2 \underline{\leftrightarrow}_d (q, \xi_1).$$

Preuve. La démonstration découle des propositions 5.4.3 et 6.4.6.

□

Dans la section précédente, nous avons donné les motifs pour le choix non-déterministe, la composition séquentielle et la mise en parallèle, ainsi qu'une intuition de leur correction. Nous allons maintenant nous consacrer aux énoncés et preuves formels de cette correction.

Remarque 7.3.3. *Ces traductions entre PA et RPPS utilisent le symbole "+" à la fois pour dénoter le choix non-déterministe de PA et l'union d'états hiérarchiques dans RPPS.*

Parfois, nous sommes conduits à mélanger les deux formalismes. Ainsi le + apparaissant dans une somme d'états hiérarchiques préfixés "$\alpha.\xi + \alpha'.\xi'$" représente-t-il le choix non-déterministe.

Choix non-déterministe.

Le motif de la figure 7.1 implémente correctement la notion de choix non-déterministe réalisé par l'opérateur + de PA. Formellement, on a le lemme suivant:

Lemme 7.3.4. *Dans la figure 7.1*

$$\begin{array}{l} si\ q_i, \emptyset \underline{\leftrightarrow}_d E_i \ \ \forall\, i = 1, \ldots, n, \\ alors\ q, \emptyset \underline{\leftrightarrow}_d \sum_{i=1}^{n} \tau.\alpha_i.E_i. \end{array}$$

Preuve. Un état hiérarchique $q, \emptyset$ peut faire une des τ-transitions $q, \emptyset \stackrel{\tau}{\longmapsto} q_i', \emptyset$. C'est imitable par $\sum_{i=1}^{n} \tau.\alpha_i.E_i \stackrel{\tau}{\longmapsto} \alpha_i.E_i$ (d'après la règle de transitions de PA pour la sommation). Ensuite, pour un état hiérarchique $q_i', \emptyset$ nous avons la seule transition possible $q_i', \emptyset \stackrel{\alpha_i}{\longmapsto} q_i, \emptyset$, où q_i' est le noeud d'action étiqueté par α_i (cf. la figure 7.1). Suivant la règle de transitions de PA pour la composition séquentielle on a $\alpha_i.E_i \stackrel{\alpha_i}{\longmapsto} E_i$. Par hypothèse du lemme $q_i, \emptyset \underline{\leftrightarrow}_d E_i$ et donc on a vérifié la propriété de transfert dans le sens direct. L'autre direction se traite de la même façon.

□

Le lecteur comprendra qu'il n'est pas possible d'obtenir un résultat tel que le lemme précédent à partir d'un terme $\sum_{i=1}^{n} \alpha_i.E_i$ sans les τ-actions que nous avons ajoutées. Ceci est dû au fait que les étiquettes du modèle RPPS ne servent pas de garde lors d'un choix non-déterministe comme elles le font en CCS, CSP, etc. Dans un noeud q donné, l'action visible est fixée et rien d'extérieur ne peut influencer le choix non-déterministe entre les noeuds suivants. Notre choix non-déterministe correspond à l'opérateur $\oplus$ de [DH87].

Composition séquentielle.

Nous avons déjà vu dans la section précédente que, si on met un `wait`-noeud comme le suivant d'un noeud d'invocation d'une procédure parallèle, la procédure parentale peut continuer son exécution seulement à condition que la coroutine invoquée ait terminé. Cette remarque est exprimée par le

motif de la figure 7.2 qui implémente correctement la composition séquentielle réalisée par l'opérateur (.) de PA.

Formellement:

Lemme 7.3.5. *Pour un schéma G contenant le motif de la figure 7.2*

$$\begin{array}{l} si\ q_i, \emptyset \underline{\leftrightarrow}_d E_i \ \ \forall\, i = 1, 2, \\ alors\ q, \emptyset \underline{\leftrightarrow}_d E_1.E_2 \end{array}$$

Preuve. L'état hiérarchique $q, \emptyset$ ne peut faire que la seule transition $q, \emptyset \stackrel{\tau}{\longmapsto} q', (q_1, \emptyset)$ et donc, $q, \emptyset \underline{\leftrightarrow}_d q', (q_1, \emptyset)$. Par le lemme 7.3.2 on voit sans peine que $q', (q_1, \emptyset) \underline{\leftrightarrow}_d E_1.E_2$ car pour le `wait`-noeud q' nous avons $(q', \emptyset) \underline{\leftrightarrow}_d (q_2, \emptyset)$.

□

Composition parallèle.

Avant de présenter un motif qui implémente la composition parallèle de PA, nous étudions de près comment l'exécution d'une instruction d'invocation de RPP, le `pcall`, peut mettre en parallèle le suivant et l'invoqué.

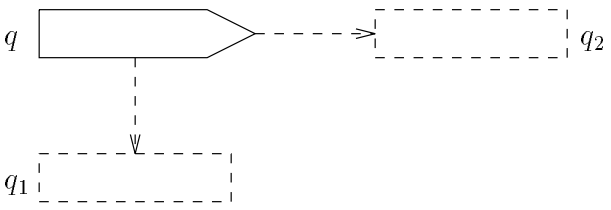


Figure 7.4 : Noeud `pcall` et composition parallèle

On considère un schéma G contenant le motif de la figure 7.4. Si le suivant q_1 ne peut pas atteindre une instruction de synchronisation `wait`, il s'exécute simultanément avec l'invoqué q_2; ce qu'exprime le lemme 7.3.7 qui suit (après la définition indispensable).

On dit qu'un noeud $q \in Q_G$ peut atteindre un noeud de synchronisation `wait`, si et seulement si un de ses suivants peut atteindre le `wait` ou si un suivant d'un de ses suivants peut atteindre le `wait` et ainsi de suite. Formellement,

Définition 7.3.6. *(Un* `wait` *atteignable)*
Un noeud q peut atteindre un `wait`, *on note* $RW(q)\,(= True)$, *si*

1. *soit q est un* `wait`*-noeud,*
2. *soit $RW(q')$ pour un q' suivant de q.*

Cette définition est récursive. On considère la plus petite solution.

Avec $RW(q)$ sous la main on est prêt pour énoncer le résultat suivant.

Lemme 7.3.7. *Si $\neg RW(q)$, alors $q, \xi \underline{\leftrightarrow} (q, \emptyset) + \xi$.*

Preuve. On va montrer que $\forall\, n :\ q, \xi \underline{\leftrightarrow}_n (q, \emptyset) + \xi$.

1. Pour $n = 0$ c'est vrai selon la définition 4.3.1.
2. On suppose que c'est vrai jusqu'à $n-1$. On va démontrer pour n. On va démontrer la propriété de transfert de $\underline{\leftrightarrow}_n$ dans le sens direct. Soit donc, $q, \xi \stackrel{\alpha}{\longmapsto} \zeta$ pour un $\alpha \in \Lambda_\tau$. On raisonne par cas suivant la règle de la définition 6.3.1 qui justifie cette transition:
   - **Cas 1.** $\xi \stackrel{\alpha}{\longmapsto} \xi'$ et $\zeta = q, \xi'$. Dans ce cas $(q, \emptyset) + \xi \stackrel{\alpha}{\longmapsto} \phi = (q, \emptyset) + \xi'$ (par règle PAR_2) et par hypothèse d'induction $\zeta \underline{\leftrightarrow}_{n-1} \phi$.
   - **Cas 2.** $q \longmapsto$. Trois sous-cas peuvent se présenter suivant la règle utilisée.
     (a) **Sous-cas 2.1.** Pour la règle CALL, on a $q \longmapsto q', \xi''$ et alors $\zeta = q', (\xi + \xi'')$ Notons que $\neg RW(q')$ (par hypothèse). Dans ce sous-cas $(q, \emptyset) + \xi \stackrel{\tau}{\longmapsto} \phi = (q', \xi'') + \xi$ (par la règle PAR_2). Mais par hypothèse d'induction (et par l'associativité des équivalences $\underline{\leftrightarrow}_i$) on a
     $$\phi \underline{\leftrightarrow}_{n-1} (q', \emptyset) + \xi + \xi''.$$
     (b) **Sous-cas 2.2.** Si c'est la règle END par laquelle on a dérivé, alors on a $\zeta = \xi$ et $q, \emptyset \stackrel{\tau}{\longmapsto} \emptyset$. Dans ce sous-cas $(q, \emptyset) + \xi \stackrel{\tau}{\longmapsto} \phi = \xi$ et $\zeta \underline{\leftrightarrow}_{n-1} \phi$.
     (c) **Sous-cas 2.3.** Si on a dérivé par la règle ACT, alors on a $\zeta = q', \xi$ et par ailleurs, $q, \emptyset \stackrel{a}{\longmapsto} q', \emptyset$. Dans ce sous-cas $(q, \emptyset) + \xi \stackrel{\tau}{\longmapsto} \phi = (q', \emptyset) + \xi$. Par hypothèse d'induction $\zeta \underline{\leftrightarrow}_{n-1} \phi$.

Ainsi, on a vérifié la première direction de la propriété de transfert de la définition 4.3.1; la preuve pour l'autre direction suit des arguments symétriques, et donc, nous avons démontré que

$$q, \xi \underline{\leftrightarrow}_n (q, \emptyset) + \xi.$$

Dès lors, l'assertion du lemme est immédiate du fait que le système de transitions considéré est à branchement fini et donc tel que

$$\underline{\leftrightarrow} \;=\; \bigcap_{n=0,1,\ldots} \equiv_n$$

d'après la définition 4.3.1 et les résultats classiques de [BBK87a, Mil90].

□

On vient de démontrer que si un noeud q ne peut pas atteindre un `wait`-noeud ($\neg RW(q)$), la procédure parentale $q, \emptyset$ s'exécute dans l'état q, ξ en parallèle avec ses fils: $q, \xi \underline{\leftrightarrow} (q, \emptyset) + \xi$.

En utilisant ce résultat nous pouvons maintenant énoncer la correction d'un motif implémentant la composition parallèle de PA.

On considère un schéma G contenant le motif de la figure 7.3. On a

Lemme 7.3.8. *Pour le motif de la figure 7.3*

$$q_n, \emptyset \underline{\leftrightarrow}_d \{(q'_1, \emptyset), \ldots, (q'_n, \emptyset)\}.$$

Preuve. Par induction sur le nombre n de composantes parallèles en utilisant le fait que $\neg RW(q_i)\ \ \forall\, i = 0, \ldots, n$ et que $q_0, \emptyset \underline{\leftrightarrow} \emptyset$.

1. Pour $n = 0$ on est dans un état hiérarchique $q_n, \emptyset$, où q_n est un noeud `end`. La seule transition à faire est $q_n, \emptyset \stackrel{\tau}{\longmapsto} \emptyset$ et donc, $q_n, \emptyset \underline{\leftrightarrow}_d \emptyset$.

2. On suppose que notre assertion est vraie jusqu'à $n-1$. On va démontrer qu'elle est vraie pour n.

   - On considère d'abord la direction "$\Leftarrow$" pour démontrer que $q_n, \emptyset \underline{\leftrightarrow}_d \{(q'_1, \emptyset), \ldots, (q'_n, \emptyset)\}$.
     Soit $\{(q'_1, \emptyset), \ldots, (q'_n, \emptyset)\} \underline{\leftrightarrow}_d q_n, \emptyset$. Pour chaque transition de l'état hiérarchique $\{(q'_1, \emptyset), \ldots, (q'_n, \emptyset)\}$ on voit aisément (par construction du motif 7.3 d'un schéma G) qu'il existe une suite de n τ-transitions telle que $q_n, \emptyset (\stackrel{\tau}{\longmapsto})^n \{(q'_1, \emptyset), \ldots, (q'_n, \emptyset)\}$ et donc, à partir de ce moment, il est possible de faire la même transition $\longmapsto$ et obtenir exactement les mêmes états qui sont, évidemment, d-bisimilaires.
   - Maintenant on considère la direction "$\Rightarrow$" pour démontrer la propriété de d-transfert pour$\{(q'_1, \emptyset), \ldots, (q'_{n-1}, \emptyset), (q'_n, \emptyset)\}$ et$(q_n, \emptyset)$.

Pour un état hiérarchique $(q_n, \emptyset)$ la seule transition possible est $(q_n, \emptyset) \stackrel{\tau}{\longmapsto} (q_{n-1}, (q'_n, \emptyset))$ et donc, $(q_n, \emptyset) \underline{\leftrightarrow}_d (q_{n-1}, (q'_n, \emptyset))$. Puisque

$q_{n-1}, \emptyset \underline{\leftrightarrow}_d \{(q'_1, \emptyset), \ldots, (q'_{n-1}, \emptyset)\}$ (par hypothèse d'induction),
$(q'_n, \emptyset) \underline{\leftrightarrow}_d (q'_n, \emptyset)$,

on a (par la proposition 5.4.3)

$$(q_{n-1}, \emptyset) \| (q'_n, \emptyset) \underline{\leftrightarrow}_d \{(q'_1, \emptyset), \ldots, (q'_{n-1}, \emptyset)\} \| (q'_n, \emptyset).$$

Outre cela, d'après la conséquence du lemme 6.4.5 on voit que

$$\{(q'_1, \emptyset), \ldots, (q'_{n-1}, \emptyset)\} \| \{(q'_n, \emptyset)\}$$

et

$$\{(q'_1, \emptyset), \ldots, (q'_{n-1}, \emptyset)\} + \{(q'_n, \emptyset)\}$$

sont d-bisimilaires. Il est facile de voir que

$$(q_{n-1}, \emptyset) \| (q'_n, \emptyset) \underline{\leftrightarrow}_d \{(q'_1, \emptyset), \ldots, (q'_{n-1}, \emptyset), (q'_n, \emptyset)\}.$$

(par construction du motif, on a $\neg RW(q_{n-1})$). Par le lemme 7.3.7 on a

$$(q_{n-1}, \emptyset) \| (q'_n, \emptyset) \underline{\leftrightarrow}_d (q_{n-1}, (q'_n, \emptyset)),$$

alors on obtient $(q_n, \emptyset) \underline{\leftrightarrow}_d \{(q'_1, \emptyset), \ldots, (q'_{n-1}, \emptyset), (q'_n, \emptyset)\}$.

□

7.4 Codage des processus PA

En utilisant les motifs présentés par les figures 7.1, 7.2 et 7.3 nous pouvons coder une déclaration Δ de PA en fng dans un schéma de RPPS.

Donc, soit

$$\Delta \; = \; \{X_i \stackrel{déf}{=} \sum_{j=1}^{n_i} \alpha_{i_j}.T_{i_j} \,|\, 1 \leq i \leq n\}$$

une déclaration de PA en fng. Comme on a déjà remarqué, on doit raisonner modulo τ. Techniquement, on considère une déclaration Δ_τ obtenue à partir de Δ simplement en insérant des préfixages par τ

$$\Delta_\tau = \{X_i \stackrel{déf}{=} \sum_{j=1}^{n_i} \tau.\alpha_{i_j}.T_{i_j} \,|\, i = 1, \ldots, n\}$$

Ainsi, nous obtenons une déclaration que nos motifs nous permettront de traiter au moyen d'un schéma G_{Δ_τ}.

Afin de coder la déclaration Δ_τ dans un schéma de RPPS on introduit l'ensemble des sous-expressions de Δ_τ par

$$sub(\Delta_\tau) = \{X_1, X_2, \ldots, X_n, T_{1_1}, T_{1_2}, \ldots, \}.$$

Cet ensemble contient les variables $X_1, X_2, \ldots, X_n$, les expressions T_{i_j} elles-mêmes, et leurs sous-expressions. Notons qu'il ne contient pas les expressions en partie droite de Δ_τ, dont les T_{i_j} ne sont que des sous-termes (cf. l'exemple 7.2). Evidemment, cet ensemble est fini.

A partir de la déclaration $\Delta_\tau = \{X_i \stackrel{déf}{=} \sum_{j=1}^{n_i} \tau.\alpha_{i_j}.T_{i_j} \mid 1 \leq i \leq n\}$ en fng on définit le schéma G_{Δ_τ} en utilisant un noeud pour chaque sous-expression de $sub(\Delta_\tau)$, c.-à-d. un noeud q_{X_i} pour chaque variable $X_i \in Var(\Delta)$, un noeud $q_{T_{i_j}}$ pour chaque T_{i_j} et ainsi de suite. Ces noeuds sont reliés entre eux par des noeuds supplémentaires en utilisant les motifs présentés dans la section 7.3 de façon à recréer les motifs, bien sûr en fonction des opérateurs de PA qui relient les sous-expressions. C'est possible car dans la déclaration Δ_τ les seules occurrences de $+$ sont de la forme particulière

$$X_i \stackrel{déf}{=} \sum_{j=1}^{n_i} \tau.\alpha_{i_j}.T_{i_j}$$

Un point important est à noter: les motifs introduisent des noeuds auxiliaires, de sorte que G_{Δ_τ} peut avoir plus de noeuds que les seuls q_E pour $E \in sub(\Delta_\tau)$.

Nous n'allons pas définir plus formellement la construction de G_{Δ_τ}. On donnera plutôt un exemple qui permettra au lecteur de voir comment elle fonctionne.

Exemple 7.2. Une déclaration Δ et le schéma G_{Δ_τ} de RPPS associé.

On considère la déclaration suivante Δ de PA en fng:

$$\begin{array}{lcll} X_1 & = & \alpha_1 & +\alpha_2.(X_1 \| X_2) \\ X_2 & = & \alpha_2.(X_2.X_3) & +\alpha_3 \\ X_3 & = & \alpha_3 & \end{array}$$

Alors Δ_τ est donné par

$$\begin{array}{lcll} X_1 & = & \tau.\alpha_1 & +\tau.\alpha_2.(X_1 \| X_2) \\ X_2 & = & \tau.\alpha_2.(X_2.X_3) & +\tau.\alpha_3 \\ X_3 & = & \tau.\alpha_3 & \end{array}$$

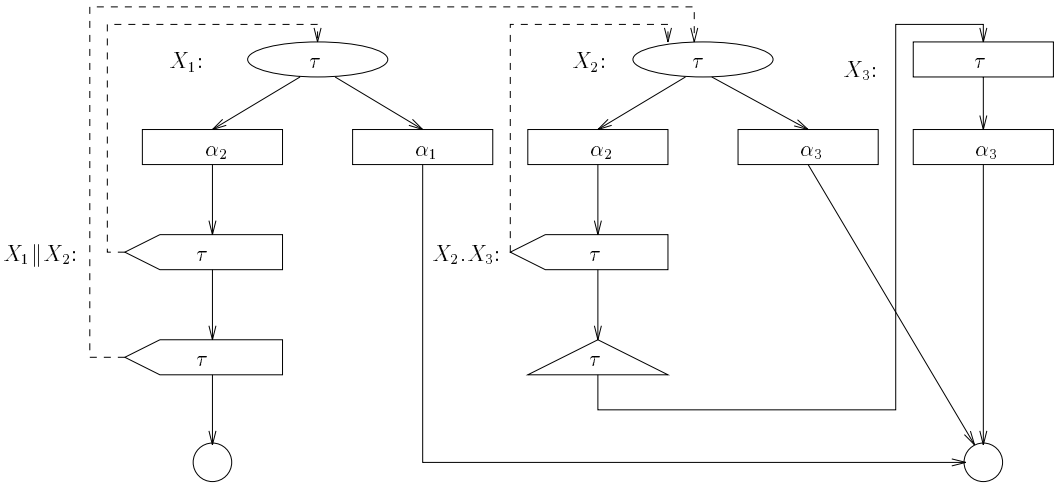


Figure 7.5 : Schéma associé à la déclaration de l'exemple 7.2

Ainsi l'ensemble des sous-expressions de Δ_τ est

$$sub(\Delta_\tau) = \{X_1, X_2, X_3, X_1 \| X_2, X_2.X_3, \mathbf{0}\}.$$

Le codage de cette déclaration dans RPPS est donné par la figure 7.5.

On peut alors énoncer la correction de ce codage:

Théorème 7.4.1.

$$\Delta_\tau \sim_{Tr_\tau} G_{\Delta_\tau}$$

Preuve. On va donner l'idée de la preuve.

- La façon dont nous construisons le schéma G_{Δ_τ} en assemblant les motifs des figures 7.1, 7.2 et 7.3 nous assure que le comportement des états hiérarchiques $(q_{X_i}, \emptyset)$ est bien une solution (au sens de $\underline{\leftrightarrow}_d$) de Δ_τ vue comme un système d'équations. C'est en fait ce qu'énoncent les lemmes 7.3.4, 7.3.5 et 7.3.8 de correction des motifs.

- La déclaration Δ_τ est gardée et par conséquent, elle a une solution unique modulo bisimulation (cf. page 60). Malheureusement, cette propriété n'est pas vraie modulo d-bisimulation: par exemple, pour $\Delta_\tau = \{X \stackrel{déf}{=} \tau.\tau.X\}$ on a deux solutions ($\mathbf{0}$ et X) qui ne sont pas d-bisimilaires.

- Néanmoins, pour les motifs de la section 7.3, on pourrait donner des énoncés de correction en terme de bisimulation forte:

1. Pour le motif de la figure 7.3 on a

$$q_n, \emptyset \underline{\leftrightarrow} \tau.(q'_n, \emptyset \| \tau.(q'_{n-1}, \emptyset \| \tau.(\ldots (q'_1, \emptyset \| \tau.\mathbf{0})))).$$

2. Pour le motif de la figure 7.2 on a

$$q, \emptyset \underline{\leftrightarrow} \tau.(q_1, \emptyset).\tau.(q_2, \emptyset).$$

3. Pour le motif de la figure 7.1, on a

$$q, \emptyset \underline{\leftrightarrow} \sum_{i=1}^{n} \tau.\alpha_i.(q_i, \emptyset).$$

Ainsi, on pourrait considérer une autre déclaration Δ_τ^τ (où on n'insère qu'un nombre fini de τ dans Δ_τ, et toujours par des préfixages) telle que le comportement de notre schéma G_{Δ_τ} soit solution de Δ_τ^τ en terme de bisimulation forte. L'unicité des solutions donne alors:

$$\Delta_\tau^\tau \underline{\leftrightarrow} G_{\Delta_\tau}$$

- Mais, par ailleurs, il est clair que le passage de Δ_τ à Δ_τ^τ préserve la d-bisimulation: les τ nouvellement introduits ne créent pas de divergence et ne modifient pas les gardes des choix non-déterministes (NB: contrairement au passage de Δ à Δ_τ).

- On en déduit

$$\Delta_\tau \underline{\leftrightarrow}_d \Delta_\tau^\tau \underline{\leftrightarrow} G_{\Delta_\tau}$$

ce qui prouve le théorème.

□

7.5 De Δ_τ à Δ

Après le théorème 7.4.1 reliant Δ_τ et G_{Δ_τ}, il nous reste à démontrer que

$$\Delta \quad \sim_{Tr_\tau} \quad \Delta_\tau$$

C'est le but de cette section.

On considère donc une déclaration Δ en fng

$$\Delta = \{X_i \stackrel{déf}{=} \sum_{j=1}^{n_i} \alpha_{i_j}.T_{i_j} \mid i = 1, \ldots, n\}$$

avec donc

$$\Delta_\tau = \{X_i \stackrel{déf}{=} \sum_{j=1}^{n_i} \tau.\alpha_{i_j}.T_{i_j} \mid i = 1, \ldots, n\}$$

Le lemme suivant nous permettra de retrouver le comportement de la déclaration Δ dans celui de Δ_τ:

Lemme 7.5.1.

$$\forall\, E, F \in \mathcal{EXP} \;:\; si\;\; E \stackrel{\alpha}{\rightarrow}_\Delta F,\;\; alors\; E(\stackrel{\tau}{\rightarrow}_{\Delta_\tau})^* \stackrel{\alpha}{\rightarrow}_{\Delta_\tau} F$$

Preuve. Par induction sur la dérivation de $E \stackrel{\alpha}{\rightarrow}_\Delta F$. On considère la forme de E:

1. Pour $E = \alpha.E'$ alors $F = E'$ et la seule transition possible dans Δ et Δ_τ est $E \stackrel{\alpha}{\rightarrow} F$.

2. Si $E = X$ et $X \stackrel{\alpha}{\rightarrow}_\Delta F$, alors il existe forcément la prémisse $E \stackrel{\alpha}{\rightarrow}_\Delta F$ avec $X \stackrel{déf}{=} E \in \Delta$ et avec E de la forme $\sum_i E_i$ donc, il existe une prémisse $E_i \stackrel{\alpha}{\rightarrow}_\Delta F$. Dans Δ_τ on a:

   $E_i(\stackrel{\tau}{\rightarrow}_{\Delta_\tau})^* \stackrel{\alpha}{\rightarrow}_{\Delta_\tau} F$ (par hypothèse d'induction), et donc,
   $\tau.E_i(\stackrel{\tau}{\rightarrow}_{\Delta_\tau})^* \stackrel{\alpha}{\rightarrow}_{\Delta_\tau} F$ ensuite, on en déduit que
   $\sum_i \tau.E_i(\stackrel{\tau}{\rightarrow}_{\Delta_\tau})^* \stackrel{\alpha}{\rightarrow}_{\Delta_\tau} F$ et donc,
   $X(\stackrel{\tau}{\rightarrow}_{\Delta_\tau})^* \stackrel{\alpha}{\rightarrow}_{\Delta_\tau} F$ avec $X \stackrel{déf}{=} \sum_i \tau.E_i \in \Delta_\tau$

3. Si $(E =)\; E_1 \| E_2 \stackrel{\alpha}{\rightarrow}_\Delta F$, alors on a $E_1 \stackrel{\alpha}{\rightarrow}_\Delta F_1$ (ou bien, $E_2 \stackrel{\alpha}{\rightarrow}_\Delta F_2$) t.q. $F = F_1 \| E_2$ (resp. $F = E_1 \| F_2$). Par hypothèse d'induction nous avons $E_1(\stackrel{\tau}{\rightarrow}_{\Delta_\tau})^* \stackrel{\alpha}{\rightarrow}_{\Delta_\tau} F_1$ (ou bien, $E_2(\stackrel{\tau}{\rightarrow}_{\Delta_\tau})^* \stackrel{\alpha}{\rightarrow}_{\Delta_\tau} F_2$), d'où on peut dériver

   $$\underbrace{E_1 \| E_2}_{E}(\stackrel{\tau}{\rightarrow}_{\Delta_\tau})^* \stackrel{\alpha}{\rightarrow}_{\Delta_\tau} \underbrace{F_1 \| E_2}_{F} \;\text{(resp., }\; \underbrace{E_1 \| E_2}_{E}(\stackrel{\tau}{\rightarrow}_{\Delta_\tau})^* \stackrel{\alpha}{\rightarrow}_{\Delta_\tau} \underbrace{E_1 \| F_2}_{F}).$$

4. Pour $E = E_1.E_2$ et $E = E_1 + E_2$ la preuve est par analogie avec le cas précédent (en utilisant les règles correspondantes).

L'induction est complète.

□

Exemple 7.3.

On considère la déclaration Δ de l'exemple 7.2 et une expression

$$E = \alpha_2.(X_1 \| X_2)$$

Si

$$(E =)\ \alpha_2.(X_1 \| X_2) \stackrel{\alpha_2}{\longmapsto}_{\Delta} (X_1 \| X_2) \stackrel{\alpha_2}{\longmapsto}_{\Delta}$$
$$((X_1 \| X_2) \| X_2) \stackrel{\alpha_2}{\longmapsto}_{\Delta} ((X_1 \| X_2) \| (X_2.X_3))\ \ (= F)$$

alors pour Δ_τ on a:

$$(E =)\ \alpha_2.(X_1 \| X_2) \stackrel{\alpha_2}{\longmapsto}_{\Delta_\tau} (X_1 \| X_2) \stackrel{\tau}{\longmapsto}_{\Delta_\tau} \stackrel{\alpha_2}{\longmapsto}_{\Delta_\tau}$$
$$((X_1 \| X_2) \| X_2) \stackrel{\tau}{\longmapsto}_{\Delta_\tau} \stackrel{\alpha_2}{\longmapsto}_{\Delta_\tau} ((X_1 \| X_2) \| (X_2.X_3))\ \ (= F)$$

Le lemme 7.5.1 donne immédiatement

Corollaire 7.5.2. $Tr_\tau(\Delta) \subseteq Tr_\tau(\Delta_\tau)$.

Nous allons maintenant étudier l'inclusion dans l'autre direction.

Compte tenu du fait que la seule différence entre Δ et Δ_τ consiste en ce que Δ_τ est obtenue de Δ en insérant des préfixages par τ, on voit que si $E \stackrel{\alpha}{\longmapsto}_{\Delta_\tau} F$ alors il y a deux possibilités:

1. soit il existe la même transition dans Δ et

$$E \stackrel{\alpha}{\longmapsto}_{\Delta} F$$

2. soit il s'agit d'une action τ ajoutée par la transformation Δ en Δ_τ et c'est une variable X qui effectue cette τ-transition et qui devient un des $\alpha_i.E_i$ conformément à la définition $X \stackrel{déf}{=} \sum_i \tau.\alpha_i.E_i\ \in \Delta_\tau$ de X. Quant à Δ, il n'y a pas de telle τ-transition. Néanmoins, ce cas peut être considéré comme un remplacement dans E d'une occurrence non gardée de cette variable X par le même $\alpha_i.E_i$ qui est présent dans la définition $X \stackrel{déf}{=} \sum_i \alpha_i.E_i\ \in \Delta$ de X.

Exemple 7.4. Un remplacement.
On considère la déclaration

$$\Delta = \{X \stackrel{déf}{=} \beta.Y + \alpha.X;\ \ Y \stackrel{déf}{=} \beta.(X + Y)\},$$

la déclaration

$$\Delta_\tau = \{X \stackrel{déf}{=} \tau.\beta.Y + \tau.\alpha.X;\ \ Y \stackrel{déf}{=} \tau.\beta.(X + Y)\}$$

et une expression $E = X \| (\alpha.Y)$.

Si $(E =)\ X\|(\alpha.Y) \stackrel{\alpha}{\longmapsto}_{\Delta_\tau} X\|Y$, il existe $(E =)\ X\|(\alpha.Y) \stackrel{\alpha}{\longmapsto}_{\Delta} X\|Y$. Ensuite, si $\Delta_\tau \ni Y \stackrel{\tau}{\longmapsto}_{\Delta_\tau} \beta.(X+Y)$, alors par la règle pour la mise en parallèle de PA on a $X\|Y \stackrel{\tau}{\longmapsto}_{\Delta_\tau} X\|\beta.(X+Y)$. On voit qu'il n'y a pas de cette τ-transition dans Δ, mais $X\|\beta.(X+Y)$ peut être obtenu de $X\|Y$ en remplaçant l'occurrence non gardée de Y par $\beta.(X+Y)$ présent dans la définition de $Y \in \Delta$.

Pour démontrer le résultat dont l'intuition vient d'être donnée, on introduit une notion d'une "restriction d'une expression". Cette restriction dépend d'une déclaration qui sert de contexte. Formellement,

Définition 7.5.3. *On dit que F est une* Δ-restriction *de E, notée $E \lhd F$, si $E \lhd F$ est la plus petite relation définie par les règles suivantes:*

1. $\mathbf{0} \lhd \mathbf{0}$
2. $X \lhd X$
3. $\alpha.E \lhd \alpha.E$
4. $X \stackrel{\text{déf}}{=} \sum E_i \ \in \Delta \quad \Rightarrow \quad X \lhd \sum_{i\in J\subseteq\{1,\ldots,n\}} E_i$
5. (a) $E_1 \lhd E_2 \quad \Rightarrow \quad E_1 + E_3 \lhd E_2$
   (b) $E_1 \lhd E_2 \quad \Rightarrow \quad E_3 + E_1 \lhd E_2$
6. (a) $E_1 \lhd F_1 \,\&\, E_2 \lhd F_2 \quad \Rightarrow \quad E_1\|E_2 \lhd F_1\|F_2$
   (b) $E_1 \lhd F_1 \,\&\, E_2 \lhd F_2 \quad \Rightarrow \quad E_1 + E_2 \lhd F_1 + F_2$
7. $E_1 \lhd F_1 \quad \Rightarrow \quad E_1.E_2 \lhd F_1.F_2$
8. $E_2 \lhd F_2 \,\&\, isnil(E_1) \quad \Rightarrow \quad E_1.E_2 \lhd E_1.F_2$

Exemple 7.5. Des restrictions.

On considère la déclaration $\Delta = \{X \stackrel{déf}{=} \beta.Y + \alpha.X;\ \ Y \stackrel{déf}{=} \beta.(X+Y)\}$ et l'expression $E = X\|(\alpha.Y)$.

L'expression $H = X\|(\alpha.\beta.(X+Y))$ n'est pas une Δ-restriction de E car on a remplacé une occurrence gardée de Y, mais $F = (\beta.Y)\|(\alpha.Y)$ l'est.

Lemme 7.5.4. $\forall\, E, F \in \mathcal{EXP}_{\Delta_\tau}$
si $E \stackrel{\alpha}{\rightarrow}_{\Delta_\tau} F$, alors

- *soit $E \stackrel{\alpha}{\rightarrow}_{\Delta} F$,*
- *soit $\alpha = \tau$ et F est une Δ-restriction de E.*

Preuve. On raisonne par induction sur la dérivation de $E \stackrel{\alpha}{\rightarrow}_{\Delta} F$.

L'induction est évidente si la dernière règle utilisée est celle du préfixage, une des règles de la mise en parallèle, ou la première règle de la composition séquentielle. Le cas de la deuxième règle de la composition séquentielle nécessite de remarquer que $isnil(E)$ a la même valeur dans Δ et Δ_τ. Les règles pour le choix non-déterministe nécessitent de remarquer que si E' est une Δ-restriction de E_1, alors c'est une Δ-restriction de $E_1 + E_2$.

□

Il est alors possible de construire une d-simulation entre Δ_τ et Δ, établissant le

Corollaire 7.5.5. $Tr_\tau(\Delta_\tau) \subseteq Tr_\tau(\Delta)$

d'où (par combinaison avec le corollaire 7.5.2)

Théorème 7.5.6.

$$Tr_\tau(\Delta_\tau) = Tr_\tau(\Delta)$$

C'est ce théorème qui autorise l'utilisation de la transformation de Δ en Δ_τ utilisée dans ce chapitre. Il ne reste dès lors plus qu'à conclure:

Corollaire 7.5.7.

$$Tr_\tau(\Delta) = Tr_\tau(G_{\Delta_\tau})$$

Preuve. En combinant les théorèmes 7.4.1 et 7.5.6.

□

7.6 De RPPS à PA

Nous pouvons maintenant aborder le problème du codage des schémas RPPS dans l'algèbre PA.

L'inclusion L(RPPS) $\subseteq$ L(PA) est plus difficile à démontrer que l'inclusion inverse. L'idée sous-jacente de la preuve peut être illustrée sur un exemple.

Exemple 7.6. Un noeud `pcall` et son voisinage.

On considère le noeud q_0 dans la figure 7.6. Ici $\xi_0 = q_0, \emptyset$ va engendrer une invocation-fille $\xi_1 = q_1, \emptyset$. Cette invocation ξ_1 va s'exécuter "en parallèle" avec une continuation de ξ_0 jusqu'au moment où la continuation rencontre un noeud `wait`. A partir de ce point, la relation entre la continuation de ξ_0 et ξ_1 sera

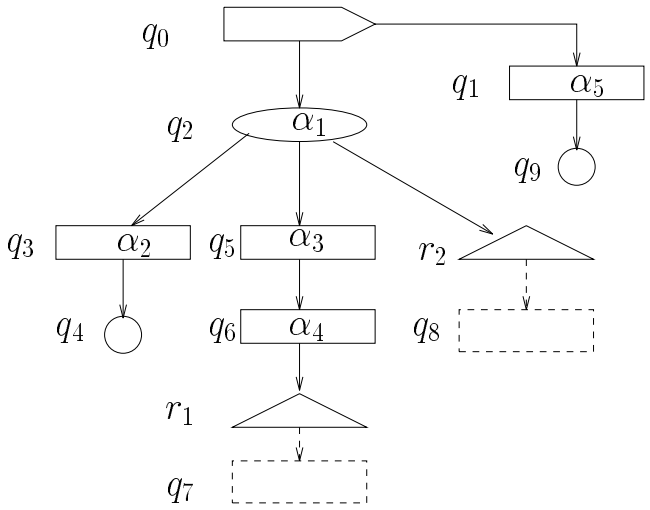


Figure 7.6 : Un noeud `pcall` et ses suivants

une composition strictement séquentielle. Ainsi, un codage convenable du schéma (fragmentaire) de la figure 7.6 serait

$$\begin{aligned} X_{q_0} &= (\alpha_1.\alpha_2.\mathbf{0} \| X_{q_1}) \\ &\quad + (\alpha_1.\alpha_3.\alpha_4.\mathbf{0} \| X_{q_1}) \,.\, X_{q_7} \\ &\quad + (\alpha_1.\mathbf{0} \| X_{q_1}) \,.\, X_{q_8} \\ X_{q_1} &= \ldots \end{aligned}$$

où on combine composition parallèle et composition séquentielle.

Le codage est plus compliqué quand les noeuds `wait` sont à l'intérieur de boucles de suivants.

Maintenant, nous allons définir formellement comment construire la déclaration Δ_G pour un schéma G donné.

Considérons Q_G et L_G d'un schéma $G \in RPPS_{\Lambda_\tau}$. On distingue les noeuds `wait` des autres:

$$Q_G = Q_{\texttt{wait}} \cup Q_{reste} = \{r_1, \ldots, r_m\} \cup Q_{reste}.$$

Nous rappelons et généralisons une notation définie en section 6.2 et souvent utilisée dans les prochains résultats. Etant donné un noeud $q \in Q_G$, nous noterons l'ensemble des suivants de q par Q_q^{suiv}. Si pour un noeud q, l'ensemble des suivants est un singleton $\{q'\}$, nous écrirons simplement $Q_q^{suiv} = q' = suiv(q)$ (c'est en particulier le cas pour les noeuds `wait`).

Pour chaque noeud $q \in Q$ nous allons former un ensemble $I_q \subseteq Q_{\texttt{wait}}$ de noeuds `wait` (qui apparaîtront dans l'équation définissant X_q). Cet ensemble est donné par: pour chaque $r \in Q_{\texttt{wait}}$, $r \in I_q$ s'il existe un chemin de suivants partant de q et atteignant r ceci sans passer par d'autres `wait`-noeuds. (Le chemin peut être de longueur nulle, et en particulier $I_r = \{r\}$ pour $r \in Q_{\texttt{wait}}$.) On dit que q *peut éviter les* `wait`*-noeuds* s'il existe un chemin maximal de "suivants" partant de q et ne visitant aucun $r \in Q_{\texttt{wait}}$.

Exemple 7.7.
Dans la figure 7.6, on a

$$\begin{array}{ll} I_{q_0} = \{r_1, r_2\} & q_0 \text{ peut éviter les } \texttt{wait}\text{-noeuds;} \\ I_{r_1} = \{r_1\} & r_1 \text{ ne peut pas éviter le } \texttt{wait}\text{-noeud.} \end{array}$$

Remarquons que pour tout q, l'ensemble I_q est facilement calculable.

A partir de l'ensemble I_q, on construit une déclaration de PA pour la variable X_q:

$$X_q \stackrel{déf}{=} \begin{cases} X_q^{nw} + \sum_{r \in I_q} X_q^r . X_{suiv(r)} & \text{si } q \text{ peut éviter les } \texttt{wait}\text{-noeuds} \\ \sum_{r \in I_q} X_q^r . X_{suiv(r)} & \text{sinon} \end{cases}$$

Notons que si q ne peut pas éviter les `wait`-noeuds, $I_q \neq \emptyset$.

Les déclarations des X_q utilisent des variables auxiliaires X_q^{nw} et X_q^r que nous devons définir en PA; pour chaque $r \in I_q$ on définit une variable X_q^r suivant le type du noeud $q \in Q$:

1. q est un noeud `end`: impossible car alors I_q est vide;

2. q est un noeud `wait`: alors $q = r$ et on ajoute

$$X_q^r \stackrel{déf}{=} \tau.\mathbf{0}$$

3. q est un noeud d'action ou de test, étiqueté par $\alpha \in \Lambda$ avec $Q_q^{suiv} = \{q_1, \ldots, q_n\}$: alors on ajoute

$$X_q^r \stackrel{déf}{=} \sum_{i=1}^{n} \sum_{r \in I_{q_i}} \alpha . X_{q_i}^r$$

   Remarquons que cette somme n'est pas vide car sinon q n'atteint pas r.

4. q est un noeud `pcall`, de noeud suivant q' et de noeud invoqué q'': alors on pose
$$X_q^r \stackrel{déf}{=} \tau.(X_{q'}^r \| X_{q''})$$

Exemple 7.8.
Dans la figure 7.6, on a
$$\begin{array}{lll} X_{q_0}^{r_2} & \stackrel{déf}{=} & \tau.(X_{q_2}^{r_2} \| X_{q_1}) \\ X_{q_2}^{r_2} & \stackrel{déf}{=} & \alpha_1.X_{r_2}^{r_2} \\ X_{r_2}^{r_2} & \stackrel{déf}{=} & \tau.\mathbf{0} \\ \ldots & & \end{array}$$

Pour définir les variables X_q^{nw} on procède de nouveau par cas sur le type du noeud q:

1. q est un noeud `end`: alors
$$X_q^{nw} \stackrel{déf}{=} \tau.\mathbf{0}$$

2. q est un noeud `wait`: ce cas est impossible

3. q est un noeud d'action ou de test, étiqueté par $\alpha \in \Lambda$ avec $Q_q^{suiv} = \{q_1, \ldots, q_n\}$: on pose
$$Q^{nw} \stackrel{déf}{=} \{q' \in Q_q^{suiv} \mid \ q' \text{ peut éviter les } \texttt{wait}\}$$
et on ajoute
$$X_q^{nw} \stackrel{déf}{=} \sum_{q' \in Q^{nw}} \alpha.X_{q'}^{nw}$$

4. q est un noeud `pcall` de noeud suivant $suiv(q) = q'$ et d'invoqué q'', alors on pose
$$X_q^{nw} \stackrel{déf}{=} \tau.(X_{q'}^{nw} \| X_{q''})$$

Exemple 7.9.
Dans la figure 7.6, on a
$$\begin{array}{lll} X_{q_1} & \stackrel{déf}{=} & X_{q_1}^{nw} \\ X_{q_1}^{nw} & \stackrel{déf}{=} & \alpha_5.X_{q_9}^{nw} \\ X_{q_9}^{nw} & \stackrel{déf}{=} & \tau.\mathbf{0} \end{array}$$

Finalement, à un schéma G, on associe une déclaration Δ_G dont les variables sont

$$\begin{array}{rcl} Var(\Delta_G) & \stackrel{déf}{=} & \{X_q | \, q \in Q\} \\ & \cup & \{X_q^r | \, r \in I_q, \, q \in Q\} \\ & \cup & \{X_q^{nw} | \, q \text{ peut éviter les } \texttt{wait}, \, q \in Q\} \end{array}$$

et chaque variable de $Var(\Delta_G)$ est définie par une équation dont la construction a été décrite ci-dessus.

Nous venons de mettre en évidence les détails techniques de construction de la déclaration Δ_G de PA à partir d'un schéma $G \in RPPS_{\Lambda_\tau}$. Cette étude détaillée va nous être immédiatement utile pour énoncer le résultat suivant, concernant la correction du codage:

Théorème 7.6.1.

$$G \sim_{Tr_\tau} \Delta_G$$

Preuve. Nous considérons deux systèmes de transitions étiquetés:

$$\mathcal{M}_G = (M_G, \longmapsto, \Lambda_\tau) \quad \text{associé à } G;$$

$$\mathcal{D}_{\Delta_G} = (\mathcal{EXP}_{\Delta_G}, \rightarrow_{\Delta_G}, \Lambda_\tau) \quad \text{associé à } \Delta_G.$$

1. Avec les variables que nous avons introduites, il est possible d'associer une expressions E_ξ à chaque état hiérarchique ξ. Comme pour la déclaration de X_q, la définition pour $E_{q,\xi}$ considère les deux cas suivant que q peut éviter ou non les noeuds `wait`:

$$E_{q,\xi} \stackrel{déf}{=} \sum_{r \in I_q} (X_q^r \| E_\xi).X_{suiv(r)} \quad (+X_q^{nw} \text{ si } q \text{ peut éviter les } \texttt{wait})$$

et bien sûr

$$E_{\xi+\zeta} \stackrel{déf}{=} E_\xi \| E_\zeta$$

Intuitivement, l'expression E_ξ a le même comportement que l'état ξ sauf que $X_q \in Var(\Delta_G)$ doit choisir immédiatement le noeud `wait` qu'elle va rencontrer (ou choisir de ne pas en rencontrer), alors que l'état hiérarchique $\{(q, \emptyset)\}$ peut (et doit) choisir plus tard, en fonction de la forme de G.

Formellement, on peut montrer qu'il y a une simulation entre E_ξ et ξ (la preuve, fastidieuse, est omise). D'où on déduit $Tr(E_\xi) \subseteq Tr(\xi)$. Et comme $E_{q_0} = X_{q_0}$

$$Tr(\mathcal{D}_{\Delta_G}) \subseteq Tr(\mathcal{M}_G)$$

2. L'inclusion inverse $Tr(\mathcal{M}_G) \subseteq Tr(\mathcal{D}_{\Delta_G})$ est plus difficile à démontrer car ici on ne peut pas établir une relation de simulation: le choix dans le schéma G et celui dans la déclaration associée Δ_G s'effectuent dans les moments différents.

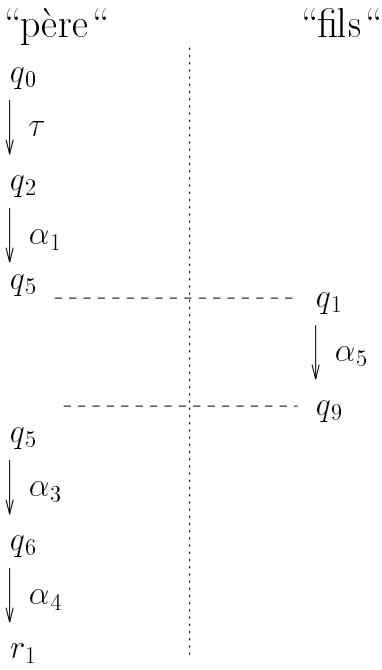


Figure 7.7 : Exemple de décomposition d'un chemin

L'idée est de considérer un chemin partant de ξ dans le système de transitions $\mathcal{M}_G$, chemin qui correspond à un comportement (peut-être inachevé) de $\mathcal{M}_G$. On peut voir ce chemin comme obtenu, en combinant par entrelacement des chemins correspondant aux différentes composantes de ξ, et en fait il est possible de le décomposer de façon à extraire, pour chaque composante de ξ, la partie du chemin qui peut lui être attribuée.

Exemple 7.10.

La figure 7.7 contient un exemple d'un chemin déplié qui est fourni par le schéma G de la figure 7.6.

Si on considère un chemin de suivants de

$$(\xi_0 =)\ q_0 \stackrel{\tau}{\longmapsto} q_2 \stackrel{\alpha_1}{\longmapsto} q_5 \stackrel{\alpha_3}{\longmapsto} q_6 \stackrel{\alpha_4}{\longmapsto} r_1 \stackrel{\tau}{\longmapsto} q_7,$$

alors, en atteignant le `wait`-noeud r_1, on sait quel choix était fait dans l'équation $X_{q_0} \stackrel{déf}{=} E_{\xi_0} = \sum \ldots$ (ξ_0 est simulé par E_{ξ_0}) de la déclaration Δ_G et comme conséquence, on sait construire l'expression $E_{\xi[r_1]}$ qui simule un état hiérarchique $\xi[r_1]$ donc, on sait trouver le même chemin dans le ST $\mathcal{D}_{\Delta_G}$.

Nous pouvons faire les mêmes démarches pour chaque "couche" de décomposition.

□

7.7 Comparaison des classes de langages

Nous avons montré, dans ce chapitre, comment notre modèle de schémas de RPPS et celui de l'algèbre PA sont étroitement (mais imparfaitement) liés. Nous avons montré que, modulo les actions silencieuses, la classe L(RPPS) des langages de traces engendrés par les schémas finis de RPPS est égale à la classe des langages engendrés par PA.

Dans cette optique "temps linéaire", la discipline `pcall`/`wait` peut donc être exprimée comme une combinaison du parallélisme disjoint sans synchronisation entre processus et de la composition séquentielle.

Maintenant, nous aimerions utiliser le résultat mentionné ci-dessus en faisant la comparaison L(RPPS) avec trois classes de langages qui sont communément connues: L(BPP), L(BPA) et L(RdP).

Classiquement, les réseaux de Petri (RdP) peuvent être lus comme un exemple de systèmes de transitions. Si, en plus, des transitions de RdP portent des étiquettes, y compris τ, on peut alors considérer L(RdP), la classe des langages générés par les réseaux de Petri étiquetés.

Notre résultat L(RPPS)=L(PA) nous permet de réutiliser les résultats connus sur PA. Ces résultats sont donnés p.ex. dans [Chr93]. Il nous est seulement nécessaire de les compléter avec le

Théorème 7.7.1. *L(RdP) n'est pas inclus dans L(PA).*

Preuve. Considérons le langage

$$L \stackrel{déf}{=} \{a^n.b^m.c^m \mid m \leq n\}$$

Clairement, $L \in$ L(RdP) et $L \notin$ L(BPA). Mais on peut montrer que si L est engendré par une déclaration Δ de PA, il est engendré par une déclaration

Δ' de BPA. En effet, dans Δ, toutes les mises en parallèle de la forme $E\|F$ s'appliquent à des termes E et F qui ne peuvent faire que des actions de même étiquette (p.ex. E et F ne peuvent faire que des a), et peuvent donc être remplacées par des compositions séquentielles $E.F$.

□

Nous pouvons alors énoncer que

- L(BPA) et L(BPP) sont strictement inclus dans L(RPPS),
- les classes L(RdP) et L(RPPS) sont incomparables, leur intersection contient L(BPP).

Nous pouvons conclure ce chapitre en résumant comment les principaux modèles que nous avons mentionnés se comparent entre eux. La figure 7.8 rassemble les résultats de ce chapitre.

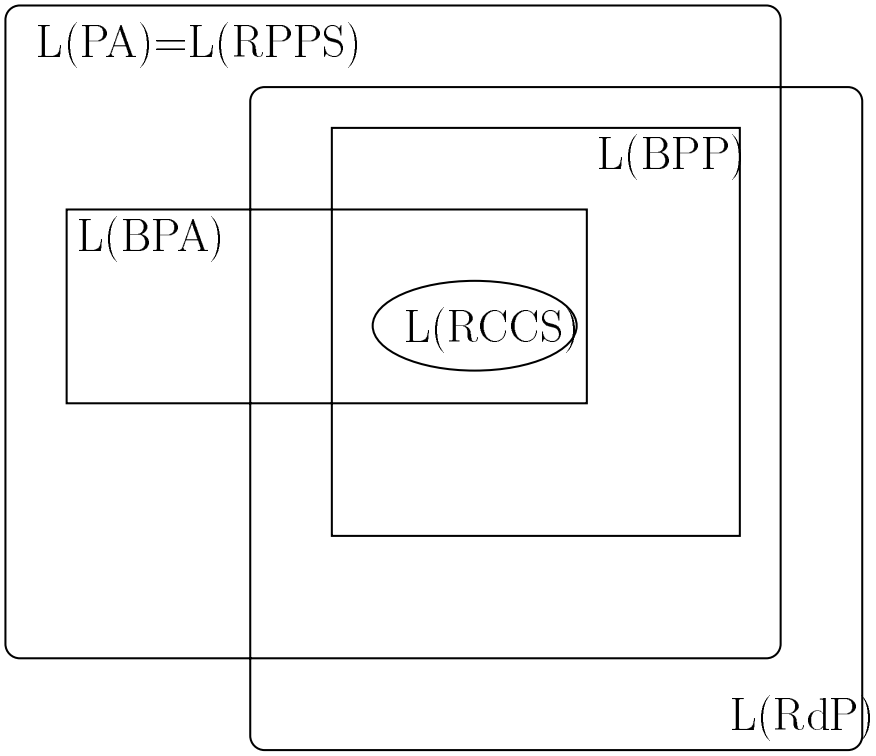


Figure 7.8 : Comparaison des classes de langages (modulo les actions silencieuses)

Chapitre 8

Schémas RPPS bien structurés

Nous nous sommes jusqu'à présent intéressés à des problèmes concernant la définition de notre modèle et sa comparaison avec d'autres modèles formels. Nous allons maintenant nous pencher sur les méthodes permettant d'analyser les programmes représentés dans notre modèle.

Comme nous l'avons déjà mentionné, les schémas RPPS (programmes RPC non interprétés) donnent naissance à des systèmes de transitions potentiellement infinis à cause de la récursivité. La plupart des techniques pour la vérification de systèmes parallèles procède par un parcours exhaustif de l'espace d'états. Par conséquent, ces techniques sont intrinsèquement incapables de considérer des systèmes dont les ensembles d'états sont infinis.

Récemment, certaines méthodes ont été développées pour vaincre cette limitation, du moins pour des classes restreintes de tels systèmes. Il y a lieu de citer ici par exemple [CH93, Mol96, ACJY96, Esp96].

Dans notre recherche, nous avons choisi un cadre de systèmes de transitions bien structurés dont l'utilisation pour l'analyse de systèmes à nombre d'états infini est apparue dans [Fin90] et plus tard a été développée dans [ACJY96].

Selon les définitions proposées notamment par Finkel, la principale propriété des systèmes "bien structurés" est l'existence d'un beau préordre entre les états qui est compatible avec la relation de transition.

Dans ce chapitre, basé sur [KS96a], on montre comment les schémas RPPS peuvent être munis d'une sémantique formelle ayant la propriété d'être "bien structurée".

Nous voudrions justifier notre choix des systèmes de transitions "bien structurés" en soulignant que c'est la généralité de l'approche qui est très

importante. En effet, on trouve de plus en plus d'exemples des systèmes de transitions "bien structurés" tels que les *hybrid automata* [Hen95], les *data-independent systems* [JP93, Wol86], les *relational automata* [Cer94], les *lossy channel systems* [AJ93, AK95] et les réseaux de Petri (avec un beau préordre pris comme une inclusion entre des marquages [Fin90]).

Notons cependant que notre point de vue sur la propriété d'un beau préordre entre les états d'être compatible avec la relation de transition, est très différent des exemples déjà connus de systèmes de transitions bien structurés. Ainsi, notre étude élargit le domaine d'application des méthodes de [Fin90, ACJY96].

Ce chapitre est organisé de la façon suivante. La première section rappelle la notion de système de transitions bien structuré et donne deux conséquences importantes de l'existence d'un beau préordre. La section 8.2 explique comment les schémas de RPPS peuvent être vus comme des systèmes de transitions bien structurés, d'où nous tirons la décidabilité de certains problèmes d'atteignabilité. La section 8.3 donne une autre vue "bien structurée" des schémas de RPPS, d'où nous tirons de nouveaux résultats de décidabilité (par exemple le problème de l'arrêt).

8.1 Systèmes de transitions bien structurés

On a déjà mentionné dans l'introduction de ce chapitre que quand on passe des systèmes d'états finis à des systèmes dont les ensembles d'états sont infinis, beaucoup de problèmes de vérification deviennent indécidables. Néanmoins, un cadre général pour la compréhension de certains résultats est apparu: les systèmes de transitions bien structurés.

Les systèmes de transitions bien structurés ont obtenu leur nom de leur utilisation d'un beau préordre, cf. définition 3.1.4, (en anglais: "well-ordering") entre les états. Nous allons donner deux conséquences importantes (le lemme 8.1.1 et la proposition 8.1.2) d'un beau préordre qui nous seront utiles par la suite.

Soit $(X, \leq)$ un ensemble muni d'un beau préordre. Un *idéal* (dans X) est un sous-ensemble $I \subseteq X$ fermé vers le haut par $\leq$, i.e. tel que $q_2 \geq q_1 \in I$ implique $q_2 \in I$.

Un idéal peut être obtenu par fermeture vers le haut d'un ensemble. Pour un $S \subseteq X$, la *fermeture vers le haut de* S, notée $\uparrow(S)$, est

$$\uparrow(S) \stackrel{déf}{=} \{x \in X \mid y \leq x \text{ pour un } y \in S\}$$

Ainsi, si I est un idéal t.q. $I = \uparrow(S)$, nous disons que S est une *base* pour I.

Pour une partie $S \subseteq X$ pour un $(X, \leq)$ bien fondé, nous définissons l'*ensemble des éléments minimaux dans* S, noté $min_{\leq}(S)$, comme

$$min_{\leq}(S) \stackrel{déf}{=} \{x \in S \mid \nexists\, y \in S,\, y \leq x\}$$

Pour tout idéal I, $min(I)$ est une base pour I.

Lemme 8.1.1. *Toute séquence infinie $I_0 \subseteq I_1 \subseteq I_2 \subseteq \ldots$ d'idéals croissants est stationnaire, i.e. il existe un $k \in \mathbb{N}$ tel que $I_k = I_{k+1} = I_{k+2} = \ldots$.*

Preuve. Par l'absurde, si la suite n'est pas stationnaire nous pouvons extraire une suite strictement croissante. Il suit de là qu'il existe une séquence $q_0, q_1, q_2, \ldots$ telle que $q_k \in I_k$ et $q_k \notin I_j$ pour tout $j < k$. Cela veut dire que $q_j \not\leq q_k$ pour $j < k$, autrement $q_k \in I_j$ parce que I_j est un idéal. Cette séquence $q_0, q_1, q_2, \ldots$ viole alors l'hypothèse du beau préordre.

□

Le lemme 8.1.1 peut être utilisé pour démontrer la terminaison des algorithmes basés sur les méthodes de saturation où on calcule une séquence de plus grands ensembles fermés vers le haut (cf. section 8.2). De tels algorithmes ont à manipuler des ensembles fermés vers le haut qui sont en général infinis. On en a une représentation finie via une base finie et c'est une deuxième conséquence importante de la propriété de beau préordre qui peut être formulée immédiatement:

Proposition 8.1.2. *Tout idéal $I \subseteq X$ d'un beau préordre peut être représenté par une base finie $I_0 = \{x_1, \ldots, x_n\}$ d'éléments minimaux t.q. $I = \uparrow(I_0)$.*

Les bases finies nous permettent de répondre à de nombreuses questions sur les ensembles infinis qu'elles représentent:

Proposition 8.1.3. *Si $\leq$ est décidable, alors le problème de savoir, étant donnés deux ensembles finis I_0 et J_0, si $\uparrow(I_0) \subseteq \uparrow(J_0)$, est décidable.*

On rappelle que $\leq$ est décidable s'il existe un algorithme qui répond à toutes les questions "$x \leq y$?" pour $x, y \in X$.

Dans la littérature [Fin90, ACJY96, KS96a], on peut trouver plusieurs propositions différentes pour la notion de système de transitions bien structuré. Mais toutes les propositions partagent les mêmes éléments de base: système de transitions étiqueté muni d'un beau préordre qui est décidable et compatible avec la relation de transitions. Un *système de transitions bien structuré* (un *STBS*) est une structure $(\mathcal{A}, \leq)$, où

- $\mathcal{A} = (St, Act, \rightarrow)$ est un système de transitions étiqueté dans le sens habituel, avec $\rightarrow \subseteq St \times Act \times St$;
- $\leq$ est un beau préordre sur St;
- $\leq$ est décidable;
- $\leq$ est *compatible* avec la relation de transition $\rightarrow$, i.e. pour tous $q_1 \leq q_1'$ et toutes transitions $q_1 \xrightarrow{\alpha} q_2$, il existe une transition $q_1' \xrightarrow{\alpha} q_2'$ avec $q_2 \leq q_2'$.

La figure 8.1 donne une présentation par diagramme de la compatibilité où nous quantifions l'universalité par des lignes pleines et l'existence par des pointillés.

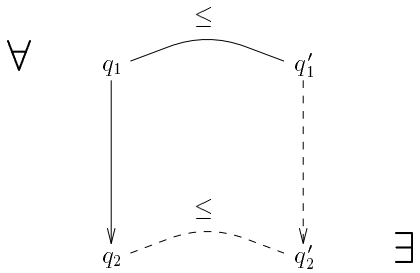


Figure 8.1 : Compatibilité

Avec ces éléments de base, certaines propriétés des systèmes de transitions bien structurés sont décidables. Par exemple, [ACJY96] montre la décidabilité de

- la simulation avec un système d'états finis;
- l'atteignabilité d'un ensemble d'états à partir d'un état donné;
- les propriétés d'inévitabilité.

Des exemples de systèmes de transitions "bien structurés" incluent les réseaux de Petri (avec un beau préordre pris comme une inclusion entre des marquages [Fin90]), les grammaires sans contexte (où une transition est un pas de dérivation $\alpha \rightarrow \beta$ entre des mots de $(T \cup N)^*$), les processus BPA, les lossy channel systems (avec un beau préordre pris comme un emboîtement de mots entre des contenus de canaux).

Dans notre cadre des schémas RPPS, il est possible de munir le système de transition $\mathcal{M}_G$ d'un beau préordre le rendant bien structuré. C'est ce que

nous faisons dans la section 8.3 avec l'ordre $\preceq_\perp$. Mais cet ordre est techniquement assez complexe, de sorte qu'il nous a paru préférable de présenter d'abord un autre ordre, plus simple, $\preceq$. En dehors des aspects pédagogiques, l'intérêt de $\preceq$ est qu'il est compatible "vers le bas" avec les transitions de $\mathcal{M}_G$, ce qui en fait nous conduit à une autre notion de système bien structuré, pour laquelle nous pourrons énoncer des résultats de décidabilité.

8.2 Compatibilité vers le bas

$\mathcal{M}_G$, le système de transitions associé à un schéma $G \in RPPS_{\Lambda_\tau}$, peut être muni d'une structure de STBS par la notion suivante de plongement entre états hiérarchiques.

Définition 8.2.1. *(Plongement entre états hiérarchiques)*
Un état hiérarchique $\xi = \{(q_1, \xi_1), \ldots, (q_n, \xi_n)\}$ se plonge dans un état hiérarchique $\xi' = \{(q'_1, \xi'_1), \ldots, (q'_m, \xi'_m)\}$, on note $\xi \preceq \xi'$, si

- *soit $\exists\, j\ (1 \leq j \leq m)$: $\xi \preceq \xi'_j$,*
- *soit $\exists\, j_1, \ldots, j_n$ dans $\{1, \ldots, m\}$ (tous différents), tels que pour $i = 1, \ldots, n$ on a*

$$(q_i, \xi_i) \preceq (q'_{j_i}, \xi'_{j_i}),\ \textit{défini comme} \begin{cases} q_i = q'_{j_i} \textit{ et } \xi_i \preceq \xi'_{j_i}, \textit{ ou} \\ \{(q_i, \xi_i)\} \preceq \xi'_{j_i} \end{cases}$$

Cette définition récursive est par induction structurelle sur les états hiérarchiques. Elle donne un ordre partiel bien fondé avec Ø comme élément minimal.

Pratiquement, $\xi \preceq \xi'$ quand ξ peut être obtenu à partir de ξ' en enlevant certains noeuds dans ξ' (et en greffant convenablement les branches qui restent); ce que nous présente l'exemple de la figure 8.2 où on a $\xi_1 \preceq \xi$ et celui de la figure 8.3 où $\xi_2 \preceq \xi$ pour deux cas de la définition 8.2.1.

Cette notion de plongement entre des arbres est utilisée dans d'autres domaines, par exemple dans les systèmes de réécriture [DJ90]. La différence consiste en ce que dans notre définition, les sous-arbres ne sont pas ordonnés entre eux du point de vue des termes générés.

Il est facile de démontrer que $\preceq$ est un ordre (relation réflexive, antisymétrique et transitive): il faut procéder par cas sur la définition 8.2.1 en utilisant l'induction sur la structure d'état hiérarchique. Ce qui est plus important est la propriété de bel ordre. Il s'agit du théorème de Kruskal [Kru60] adapté à notre cadre:

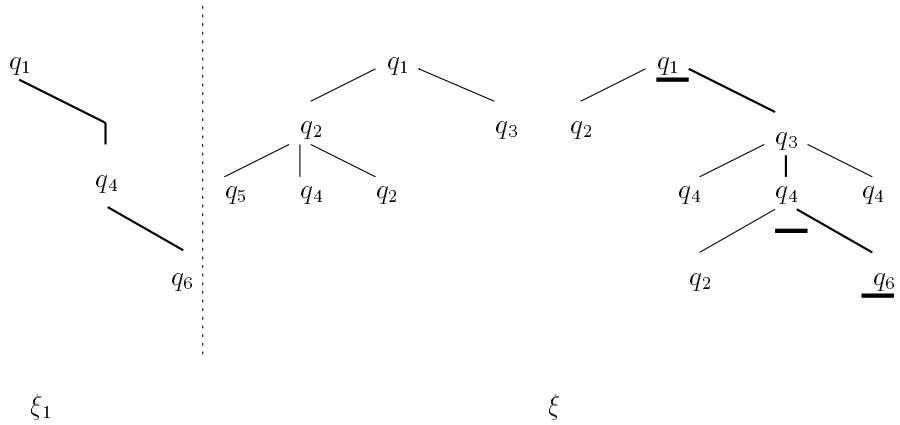


Figure 8.2 : Plongement entre états hiérarchiques: $\xi_1 \preceq \xi$

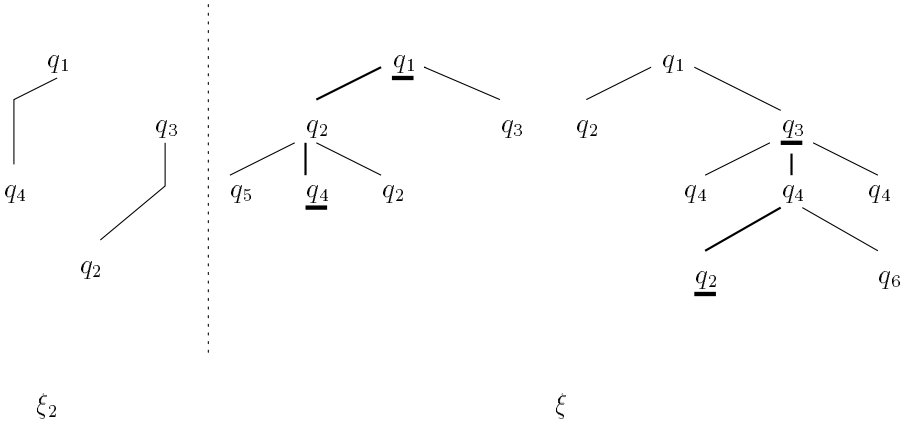


Figure 8.3 : Plongement entre états hiérarchiques: $\xi_2 \preceq \xi$

Théorème 8.2.2. [Kru60] $(M_G, \preceq)$ *est un bel ordre.*

Dans notre modèle, cet ordre est très important parce que les transitions sont compatibles (dans un certain sens) avec le plongement:

Lemme 8.2.3. *(Compatibilité vers le bas pour $\preceq$)*
Pour tous $\xi_1 \preceq \xi_1'$ et toutes transitions $\xi_1' \stackrel{\alpha}{\longmapsto} \xi_2'$,

- *soit $\xi_1 \preceq \xi_2'$,*
- *soit il existe une transition $\xi_1 \stackrel{\alpha}{\longmapsto} \xi_2$ avec $\xi_2 \preceq \xi_2'$.*

La figure 8.4 donne une présentation par diagramme de la compatibilité vers le bas.

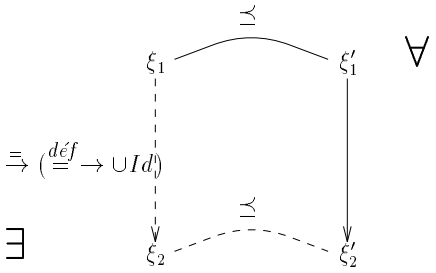


Figure 8.4 : Compatibilité

Preuve. Si $\xi_1' \stackrel{\alpha}{\longmapsto} \xi_2'$, alors ξ_1' est de la forme $\xi_1'[q]$ (la notation de la section 6.2) et ξ_2' est $\xi_1'[\zeta]$ avec une transition valide $q \stackrel{\alpha}{\longmapsto} \zeta$. Maintenant, si $\xi_1 \preceq \xi_1'$, alors il existe deux cas à considérer:

1. soit $\xi_1 \preceq \xi_1'[\emptyset]$ (i.e. le plongement de ξ_1 n'a pas besoin de l'occurrence de la composante q dans ξ_1') et donc, $\xi_1 \preceq \xi_2'$;
2. soit ξ_1 est de la forme $\xi_1[q]$ avec $\xi_1[.] \preceq \xi_1'[.]$. Il existe alors une transition $\xi_1 \stackrel{\alpha}{\longmapsto} \xi_2 = \xi_1[\zeta]$ occasionnant $\xi_2 \preceq \xi_2'$ d'après la définition 8.2.1.

□

On écrit $Post(\xi) \stackrel{déf}{=} \{\xi' \mid \xi \longmapsto \xi'\}$ pour l'ensemble de tous les successeurs immédiats d'un état ξ. Cette notation peut être étendue pour les ensembles $Post^m(\xi)$ (respectivement, $Post^*(\xi)$) des successeurs d'un ξ obtenus par m transitions (respectivement, un nombre fini quelconque de transitions). Une conséquence de la compatibilité vers le bas peut être énoncée comme

Proposition 8.2.4.

$$\uparrow(S) \subseteq \uparrow(S') \;\; \textit{implique} \;\; \begin{cases} \uparrow(S \cup Post(S)) \subseteq \uparrow(S' \cup Post(S')) \\ \uparrow(Post^*(S)) \subseteq \uparrow(Post^*(S')) \end{cases} \tag{8.1}$$

où $S, S' \subseteq M_G$ sont des ensembles d'états quelconques et où $\uparrow(S)$ est la fermeture vers le haut *en utilisant l'ordre de plongement.*

Preuve. Puisque

$$\uparrow(S \cup Post(S)) = \uparrow(S) \cup \uparrow(Post(S)) \subseteq \uparrow(S) \cup Post(\uparrow(S))$$
$$\uparrow(S' \cup Post(S')) = \uparrow(S') \cup \uparrow(Post(S')) \subseteq \uparrow(S') \cup Post(\uparrow(S'))$$

il nous reste à montrer que $Post(\uparrow(S)) \subseteq Post(\uparrow(S'))$. Cette dernière condition se vérifie à partir de la compatibilité vers le bas et du fait que $\uparrow_{\preceq}(S) \subseteq \uparrow_{\preceq}(S')$.

Cette démarche peut être étendue pour un nombre fini quelconque de transitions.

□

Nous pouvons alors énoncer

Théorème 8.2.5. *Pour un état hiérarchique ξ donné, on peut évaluer (une base finie de) la fermeture vers le haut de $Post^*(\xi)$, l'ensemble de tous les états atteignables de ξ.*

Preuve. Soit ξ donné. On définit une séquence d'ensembles d'états par

$$S_0 \stackrel{déf}{=} \{\xi\}, \quad S_{i+1} \stackrel{déf}{=} S_i \cup Post(S_i)$$

On obtient $S_i = \bigcup_{k \leq i} Post^k(\xi)$. Il est clair que tous les S_i peuvent être calculés parce que la relation de transition est à branchement fini et $Post(\zeta)$ peut être calculé pour chaque ζ. Le fait que $S_i \subseteq S_{i+1}$ implique $\uparrow(S_0) \subseteq \uparrow(S_1) \subseteq \cdots$. Mais tous les $\uparrow(S_i)$ sont fermés vers le haut, par conséquent, il existe un rang k t.q. $\uparrow(S_k) = \uparrow(S_{k+1}) = \uparrow(S_{k+2}) = \ldots$ (cf. 8.1.1) (même si peut-être $S_k \neq S_{k+1} \neq \ldots$). De fait, dès que $\uparrow(S_k) = \uparrow(S_{k+1})$ pour un k, la proposition 8.2.4 entraîne $\uparrow(S_k) = \uparrow(S_{k'})$ pour tout $k' \geq k$. Or nous pouvons effectivement calculer un tel k: il s'agit simplement de détecter quand deux ensembles finis ont la même fermeture vers le haut, ce qui est facile lorsque $\preceq$ est décidable (cf. proposition 8.1.3).
Ensuite il suffit de voir que $\uparrow(S_k) = \uparrow(Post^*(\xi))$, mais c'est évident car $S_i = Post^0(\xi) \cup \cdots \cup Post^i(\xi)$. Finalement, S_k est une base finie de $\uparrow(Post^*(\xi))$.

□

Le corollaire est

Théorème 8.2.6. *Le problème de savoir si, étant donné un état ξ et un ensemble $I \subseteq M(G)$, fermé vers le haut, tous les états atteignables à partir de ξ sont dans I, est décidable.*

Preuve. Soit ξ donné. Puisque I est fermé vers le haut, $Post^*(\xi) \subseteq I$ ssi $\uparrow (Post^*(\xi)) \subseteq I$. Avec la proposition 8.1.3, cette dernière condition est facile à vérifier dès que nous avons une base finie pour $\uparrow (Post^*(\xi))$ (on suppose que I est donné par une base finie).

□

(Evidemment, des implémentations plus élaborées du résultat de décidabilité donné par le théorème 8.2.5 sont possibles.)

Ce théorème général peut être utilisé pour des problèmes concrets. On parle de méthodes de saturation quand on a des méthodes dont la terminaison repose sur le lemme 8.1.1. A titre d'exemple, l'algorithme pour un problème d'atteignabilité d'états de contrôle et celui pour un problème de simulation d'un système de transitions d'états finis par un système de transitions bien structuré dans [ACJY96] utilisent des méthodes de saturation.

Dans l'introduction de cette thèse, nous avons mentionné que le modèle de schémas RPPS que nous proposons est destiné à analyser le flot de contrôle dans des programmes récursifs-parallèles. Nous avons déjà montré que les RPPS donnent naissance à des systèmes de transitions potentiellement infinis à cause de la récursivité.

Du fait de l'infinité potentielle de l'ensemble d'états hiérarchiques, il serait utile de considérer le problème d'atteignabilité d'états de contrôle. Un tel état de contrôle peut être représenté par un noeud q d'un schéma G de RPPS. En termes de questions que se pose un utilisateur de RPC, il serait intéressant de savoir si une procédure parallèle peut être invoquée ou bien si une instruction (y compris une dernière instruction à exécuter) peut être atteinte à partir d'une autre instruction (y compris une première instruction à exécuter).

Maintenant, ce problème (d'atteignabilité d'états de contrôle) est formulé comme suit:

Atteignabilité d'états de contrôle. Etant donnés un état ξ et un noeud q, est-ce possible d'atteindre à partir de ξ un état contenant q?

Corollaire 8.2.7. *Le problème de savoir si, étant donnés un état ξ et un noeud q, on peut atteindre à partir de ξ un état contenant q, est décidable.*

Ce point est très important car ce résultat de décidabilité du problème d'atteignabilité d'états de contrôle obtenu sur une abstraction non interprétée

des schémas, peut intervenir naturellement dans l'analyse pour produire une information pertinente sur des programmes réels.

On peut conclure cette section en résumant que le principal avantage de notre modèle de schémas RPPS est la simplification apportée aux méthodes d'analyse de programmes récursifs-parallèles.

8.3 Compatibilité vers le haut

Dans la littérature, c'est la compatibilité vers le haut qui est utilisée [Fin90, ACJY96].

Dans notre cadre des schémas RPPS, il est possible de proposer un beau préordre compatible vers le haut. On a besoin, pour ce faire, d'un préordre plus sophistiqué que le plongement $\preceq$ de la section précédente qui prenne en compte une notion de *terminaison possible*, comme introduite dans la définition 6.5.1.

Nous rappelons qu'un état hiérarchique ξ peut terminer, noté $\xi \downarrow$, ssi il existe une séquence finie $\mu = \alpha_1 \dots \alpha_n \in \Lambda_\tau^*$ telle que $\xi \stackrel{\alpha_1}{\longmapsto} \cdots \stackrel{\alpha_n}{\longmapsto} \emptyset$.

En conservant les notations usuelles, nous allons définir un nouveau type d'ordre, appelé *plongement avec norme*.

Définition 8.3.1. *(Plongement avec norme)*
Un état hiérarchique $\xi = \{(q_1, \xi_1), \dots, (q_n, \xi_n)\}$ *est* $\perp$*-plongé dans un état hiérarchique* $\xi' = \{(q'_1, \xi'_1), \dots, (q'_m, \xi'_m)\}$, *on note* $\xi \preceq_\perp \xi'$, *si*

1. $\xi \downarrow$ *ssi* $\xi' \downarrow$ *et*
2. - *soit* $\exists j \ (1 \leq j \leq m) \ : \ \xi \preceq_\perp \xi'_j$,
   - *soit* $\exists j_1, \dots, j_n$ *dans* $\{1, \dots, m\}$ *(2 à 2 différents), tels que* $\forall i = 1, \dots, n$

$$(q_i, \xi_i) \preceq_\perp (q'_{j_i}, \xi'_{j_i}), \ \textit{défini comme} \ \begin{cases} q_i = q'_{j_i} \ \textit{et} \ \xi \preceq_\perp \xi'_{j_i} \ \textit{ou} \\ (q_i, \xi_i) \preceq_\perp \xi'_{j_i} \end{cases}$$

Cette définition est semblable à la définition 8.2.1 mais elle demande, en plus, que chaque plongement respecte la possibilité de terminer, et ceci à chaque étape de la définition inductive.

Exemple 8.1. Plongement avec norme.

On obtient un ordre partiel bien fondé, mais $\emptyset$ n'est plus le seul élément minimal. Il est facile de démontrer que

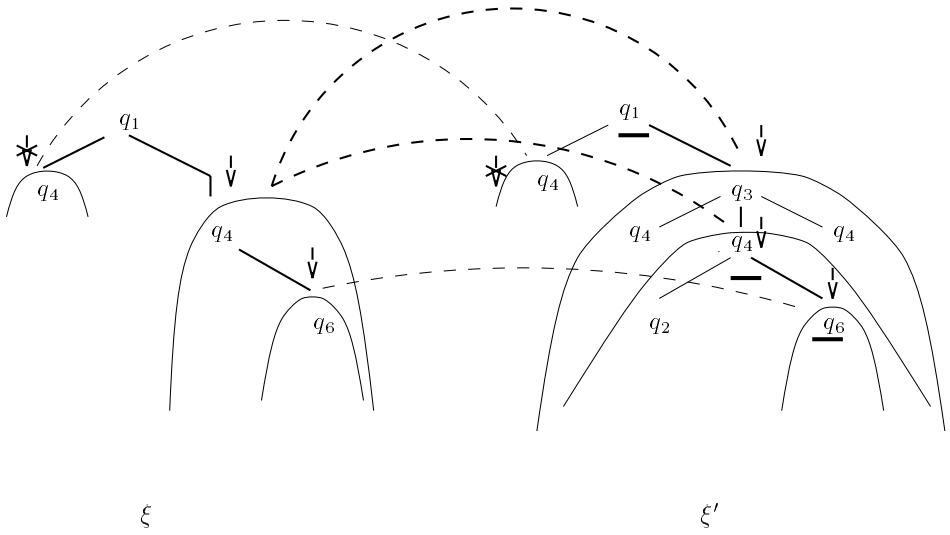


Figure 8.5 : $\perp$-plongement entre états hiérarchiques: $\xi \preceq_\perp \xi'$

Proposition 8.3.2.

$$\forall \xi \in M_G : \ \emptyset \preceq_\perp \xi \ ssi \ \xi \downarrow.$$

Une propriété utile est donnée par le

Lemme 8.3.3. *Si* $\xi \preceq_\perp \xi'$ *et* ($\xi' \uparrow$ *ou* $\xi'' \downarrow$), *alors* $\xi \preceq_\perp \xi' + \xi''$.

Preuve. On considère deux cas suivant la condition du lemme:

1. $\xi' \uparrow$.
   Le lemme 6.5.4 nous donne alors $\xi' + \xi'' \uparrow$, mais ($\xi' \uparrow$) et ($\xi \preceq_\perp \xi'$), alors $\xi \uparrow$, d'où on déduit immédiatement $\xi \preceq_\perp \xi' + \xi''$.

2. $\xi'' \downarrow$.

$$\xi \preceq_\perp \xi', \text{ alors } \begin{cases} \xi \uparrow \text{ et } \xi' \uparrow \text{ , ou} \\ \xi \downarrow \text{ et } \xi' \downarrow \end{cases}$$

   On a alors pour $\xi' + \xi''$:

$$\begin{array}{l} \xi \uparrow \text{ et } \xi' + \xi'' \uparrow \text{ , ou} \\ \xi \downarrow \text{ et } \xi' + \xi'' \downarrow, \end{array}$$

   ce qui conclut la preuve pour le cas 2.

□

Le plongement avec norme $\preceq_\perp$ est une variante plus fine du plongement $\preceq$ entre des arbres, cette fois avec une condition d'intervalle en voyant le prédicat $\downarrow$ comme une étiquette sur les sous-arbres. La condition d'intervalle est ici que toutes les étiquettes sur les sous-arbres de ξ' dans lequel est plongé un même sous-arbre ζ de ξ doivent être identiques à l'étiquette de ζ.

Le théorème de [Kri89] montre que le plongement avec condition d'intervalle où on demanderait que les étiquettes des sous-arbres de ξ' soient supérieures (pour un ordre linéaire bien fondé) à l'étiquette de ζ est un beau préordre.

Le plongement où on demanderait l'égalité ne vérifie pas la propriété de beau préordre en général. Mais dans notre modèle, les étiquettes $\downarrow$ ne peuvent pas prendre des valeurs arbitraires (cf. section 6.5): si un noeud d'un arbre est étiqueté par $\uparrow$, alors tous ces descendants doivent porter la même étiquette.

Avec cette idée, on peut considérer notre plongement $\preceq_\perp$ comme une variante de l'ordre de [Kri89], de sorte que

Théorème 8.3.4. [Kru60]
$(M_G, \preceq_\perp)$ est un bel ordre.

Puisque la propriété d'être normé pour des états hiérarchiques est décidable (cf. la section 6.5) l'ordre $\preceq_\perp$ est décidable.

De plus, il est compatible avec des transitions dans le sens suivant:

Lemme 8.3.5. *(Compatibilité vers le haut pour $\preceq_\perp$ (fig. 8.6))[KS96a] Pour tous $\xi_1 \preceq_\perp \xi_1'$ et toutes transitions $\xi_1 \stackrel{\alpha}{\longmapsto} \xi_2$, il existe un $n \geq 2$ et une séquence*

$$\xi_1' \stackrel{\alpha_1}{\longmapsto} \xi_2' \stackrel{\alpha_2}{\longmapsto} \cdots \stackrel{\alpha_{n-1}}{\longmapsto} \xi_n'$$

tels que $\alpha = \alpha_{n-1}$, $\xi_2 \preceq_\perp \xi_n'$ et $\xi_1 \preceq_\perp \xi_k'$ pour $k = 1, \ldots, n-1$.

Preuve. ξ_1 peut s'écrire sous la forme $\xi_1[q]$ t.q. la transition $\xi_1 \stackrel{\alpha}{\longmapsto} \xi_2$ est l'occurrence de $q \stackrel{\alpha}{\longmapsto} \zeta$ dans le contexte $\xi_1[.]$ (et donc, $\xi_2 = \xi_1[\zeta]$). Par hypothèse du lemme $\xi_1 \preceq_\perp \xi_1'$ c.-à-d. ξ_1' peut être écrit sous la forme $\xi_1'[q]$, d'une façon respectant le plongement $\preceq_\perp$, c.-à-d. que l'occurrence de q que nous avons isolée dans ξ_1' correspond au plongement de l'occurrence de q dans ξ_1.

Dans le cas le plus simple, la transition $q \stackrel{\alpha}{\longmapsto} \zeta$ est possible dans le contexte $\xi_1'[.]$ et on obtient une transition $\xi_1' \stackrel{\alpha}{\longmapsto} \xi_2' \stackrel{déf}{=} \xi_1'[\zeta]$. Ensuite, par induction sur la structure de ξ_1', et en procédant par cas sur la définition de

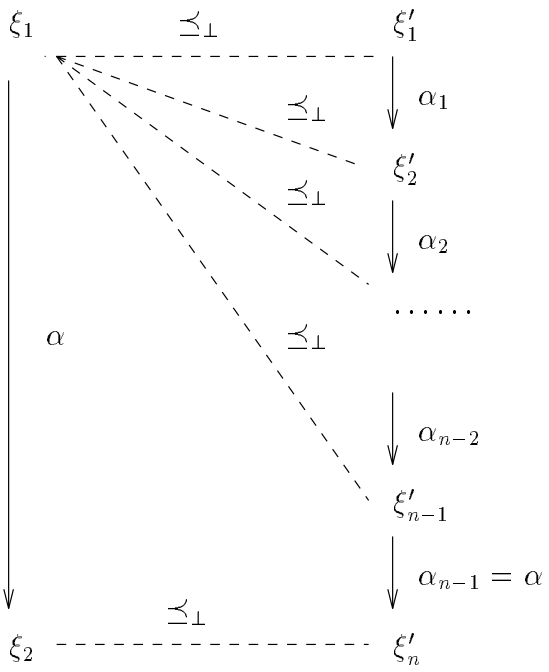


Figure 8.6 : Compatibilité vers le haut pour $\preceq_\perp$

$\xi_1[q] \preceq_\perp \xi'_1[q]$, on peut vérifier que $\xi_2 \preceq_\perp \xi'_2$. Là, l'idée consiste en ce que le seul changement effectué est le remplacement de l'occurrence de q par ζ dans ξ_1 ainsi que dans ξ'_1. En vérifiant que $\xi_2 \preceq_\perp \xi'_2$ on se retrouve facilement à l'étape correspondant à l'occurrence de ζ (de la définition inductive 8.3.1) et on voit que

- soit $\zeta \uparrow$,
- soit $\zeta \downarrow$,

dans ξ_2 et ξ'_2 tous les deux (car sinon on a une contradiction avec $\xi_1 \preceq_\perp \xi'_1$).

Quand la transition $q \stackrel{\alpha}{\longmapsto} \zeta$ n'est pas possible dans le contexte $\xi'_1[.]$ c'est parce que $q \stackrel{\alpha}{\longmapsto} \zeta$ est une `wait`-transition (q est un noeud `wait`) qui ne peut être exécutée que si q n'a pas d'invocations-"filles", condition remplie dans le contexte $\xi_1[.]$, mais pas dans $\xi'_1[.]$. Alors, $\xi'_1[q]$ peut être écrit sous la forme $\xi''[q, \zeta_q]$, où ζ_q ($\neq \emptyset$) sont les invocations-"filles" de q dans $\xi'_1[q]$. Là, une `wait`-transition $\xi''[q, \emptyset] \stackrel{\alpha}{\longmapsto} \xi''[\zeta, \emptyset]$ est possible. Il suffit maintenant de remarquer que ζ_q peut terminer parce que le plongement $\preceq_\perp$ exige $\emptyset \preceq_\perp \zeta_q$. Ainsi, il existe une séquence

$$(\zeta_q =)\ \zeta_1 \longmapsto \zeta_2 \cdots \longmapsto \zeta_m = \emptyset$$

(évidemment, avec $\emptyset \preceq_\perp \zeta_i$ pour $i = 1, \ldots, m$) permettant

$$(\xi'_1 =)\ \xi''[q, \zeta_1] \longmapsto \xi''[q, \zeta_2] \cdots\ \xi''[q, \emptyset] \stackrel{\alpha}{\longmapsto} \xi''[\zeta, \emptyset]$$

Soit $n \stackrel{déf}{=} m + 1$ et $\xi'_i \stackrel{déf}{=} \xi''[q, \zeta_i]$. Il nous reste seulement à vérifier que $\xi_1 \preceq_\perp \xi'_i\ (= \xi''[q, \zeta_i])$ pour $i = 1, \ldots, m$ (c'est une conséquence de $\emptyset \preceq_\perp \zeta_i$) ainsi que $\xi_2 \preceq_\perp \xi'_n$; ce qu'on fait par induction sur la structure de ξ'_1 en procédant par cas selon la définition 8.3.1 du plongement $\preceq_\perp$.

□

Notons que cette propriété de compatibilité vers le haut n'est pas exactement celle requise dans [Fin90, ACJY96] dans la mesure où la nôtre requiert en général une séquence de transitions pour imiter une transition simple. Ainsi, il n'est pas possible d'appliquer tels quels les résultats de [ACJY96]. Néanmoins, il nous est possible d'en énoncer et d'en démontrer des variantes. Par exemple la décidabilité du problème de l'arrêt comme un cas spécial du

Théorème 8.3.6. *Le problème de savoir si, étant donnés un état hiérarchique ξ et (une base finie d') un ensemble $I \subseteq M(G)$ fermé vers le haut (pour $\preceq_\perp$), tous les calculs partant de ξ passent inévitablement par un état qui n'est pas dans I, est décidable.*

Corollaire 8.3.7. *On peut décider si tous les calculs partant d'un état ξ terminent inévitablement.*

Preuve. En effet, $M(G) \setminus \{\emptyset\}$, l'ensemble des états qui ne sont pas terminés, est un ensemble fermé vers le haut pour lequel une base finie (par rapport du plongement $\preceq_\perp$) est construite. Alors il ne reste qu'à appliquer le théorème 8.3.6.

□

Le théorème 8.3.6 généralise (et s'inspire d') un théorème similaire de [ACJY96] pour notre propriété de la compatibilité vers le haut. Nous en donnons néanmoins une preuve complète parce que

1. notre présentation s'occupe moins de perfectionnements algorithmiques et nous la trouvons plus claire, et

2. la preuve explique où on utilise la condition spécifique (le lemme 8.3.5) comme quoi $\xi_1 \preceq_\perp \xi'_k$ *pour* $k = 1, \ldots, n-1$.

Pour cela nous avons besoin d'une construction auxiliaire qui est un ingrédient omniprésent algorithmique pour des STBS et nous définissons $RT(\xi)$, l'*arbre d'atteignabilité* partant de ξ, comme un arbre enraciné avec des noeuds étiquetés par des états. Plus précisément,

- Les noeuds de $RT(\xi)$ sont *morts* ou *vivants*.
- La racine est un noeud vivant n_0, étiqueté par ξ (noté $n_0 : \xi$).
- Un noeud mort n'a pas de noeuds "fils".
- Tout noeud vivant étiqueté par un ζ ($n : \zeta$) a des "fils" étiquetés par les successeurs immédiats (s'ils existent) de ζ, i.e. un "fils" pour chaque $n' : \zeta'$ t.q. $\zeta \longmapsto \zeta'$ dans le système de transitions $\mathcal{M}_G$.
- Un noeud-"fils" $n' : \zeta'$ est vivant sauf s'il existe un noeud $n : \xi'$ avec $n \neq n'$ situé sur le chemin allant de la racine $n_0 : \xi$ au noeud $n' : \zeta'$ t.q. $\xi' \preceq_\perp \zeta'$. Dans ce cas, on dit que n *subsume* n' et n' est un noeud mort.

Exemple 8.2. Un arbre d'atteignabilité.

La figure 8.7 contient un arbre d'atteignabilité fini admis par notre schéma de l'exemple 6.1 avec la racine étiquetée par $(q_5, (q_3, \emptyset))$. Là, le noeud $n_0 : (q_5, (q_3, \emptyset))$ subsume $n' : (q_5, (q_{10}, (q_3, \emptyset)))$.

Ainsi, les noeuds feuilles dans $RT(\xi)$ sont exactement

1. les noeuds étiquetés par des états terminaux, et
2. les noeuds qui sont subsumés.

Puisque $\mathcal{M}_G$ est un système de transitions à branchement fini, $RT(\xi)$ est un arbre fini. C'est un argument classique: on suppose que $RT(\xi)$ est infini, alors il y a un chemin infini (par le lemme de König) et, puisque $\preceq_\perp$ est un bel ordre, nous pouvons trouver au long de ce chemin un noeud n qui subsume un noeud n' (plus bas que n le long du chemin). Ainsi, n' doit être un noeud mort, contredisant le fait que le chemin est infini. Puisque $RT(\xi)$ est fini, la construction itérative sous-jacente à notre définition termine forcément et $RT(\xi)$ peut être construit effectivement.

La construction de $RT(\xi)$ n'exige pas la compatibilité entre $\preceq_\perp$ et $\longmapsto$. Mais lorsque nous avons cette compatibilité, $RT(\xi)$ contient (d'une forme finie) une information suffisante pour répondre à plusieurs questions sur des chemins de calculs partant de ξ.

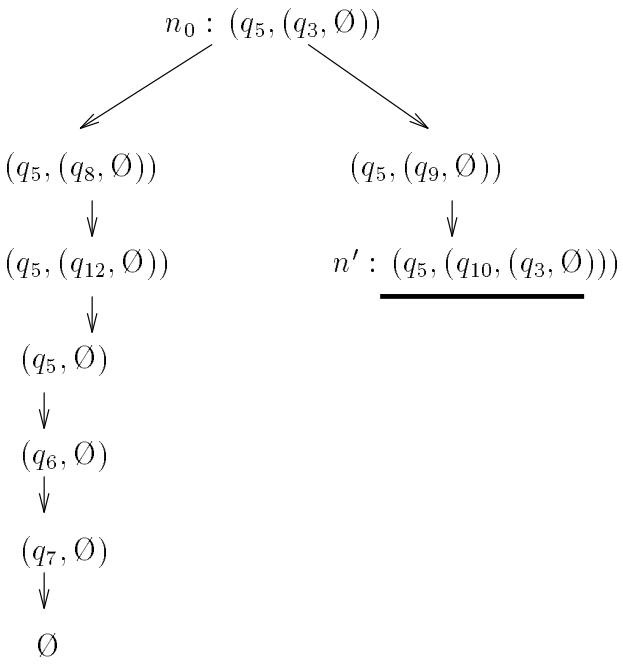


Figure 8.7 : Un arbre d'atteignabilité fini

Maintenant, la preuve du théorème 8.3.6 est basée sur le lemme suivant, traduisant la question initiale en une question équivalente sur $RT(\xi)$, laquelle peut être décidée aisément (parce que $RT(\xi)$ est fini) quand I est donné par une base finie.

Lemme 8.3.8. *Tous les calculs (dans le système de transitions) partant de ξ atteignent inévitablement un état qui n'est pas dans I ssi tous chemins maximaux (dans $RT(\xi)$) atteignent inévitablement un (noeud étiqueté par un) état qui n'est pas dans I.*

Preuve. La direction "$\Leftarrow$" se montre facilement parce que chaque calcul partant de ξ dans $\mathcal{M}_G$ a un préfixe sous la forme d'un chemin maximal dans $RT(\xi)$. La direction "$\Rightarrow$" est plus difficile à démontrer. On suppose (pour arriver à une contradiction) qu'il existe un chemin maximal dans $RT(\xi)$ dont les noeuds sont tous étiquetés par des états de I. Ce chemin visite les noeuds $n_0, \ldots, n_k$ étiquetés par $\zeta_0, \ldots, \zeta_k$.

On va montrer qu'il existe un chemin maximal (une exécution) $(\xi =)\, \xi_0 \longmapsto \xi_1 \longmapsto \ldots$ dans $\mathcal{M}_G$, où tous les états sont plus grands (via $\preceq_\perp$) que l'un des ζ_i, et donc sont dans I.

On construit les ξ_i inductivement à partir de $\xi_0 \stackrel{déf}{=} \xi = \zeta_0$. Supposons qu'on a déjà construit $\xi_0, \ldots, \xi_n$. ξ_n est plus grand qu'un des ζ_i. Deux cas peuvent se présenter:

- Si $i < k$, alors il existe une transition $\zeta_i \longmapsto \zeta_{i+1}$. Grâce à la compatibilité vers le haut, il existe une séquence $\xi_n \longmapsto \cdots \longmapsto \xi_m$ $(m > n)$ avec $\zeta_i \preceq_\perp \xi_n, \ldots, \xi_{m-1}$ et $\zeta_{i+1} \preceq_\perp \xi_m$. On l'utilise pour allonger notre séquence jusqu'à ξ_m.

- Si $i = k$, alors ζ_i est une feuille n_k de $RT(\xi)$. Si c'est un noeud vivant, alors ζ_i n'a pas de successeurs et donc $\zeta_i = \xi_n = \emptyset$. Par conséquent, nous avons construit un chemin maximal (une exécution).
  Si n_k est un noeud mort, alors un des ζ_j rencontré plus tôt subsume ζ_i et ainsi $\zeta_j \preceq_\perp \xi_n$. Dès lors, on revient au cas précédent et nous pouvons allonger notre séquence comme déjà vu.

□

[ACJY96] utilise un arbre fini similaire pour montrer que le problème de simulation d'un système de transitions bien structuré par un système de transitions d'états finis, ce problème est donc décidable.

Remarquons que le théorème 8.3.6 n'aurait pas pu être démontré si on avait simplement dit que $\longmapsto^+$, la fermeture transitive de $\longmapsto$, a la propriété de compatibilité utilisée dans [ACJY96]. En effet

- $\longmapsto^+$ n'est pas à branchement fini en général,

- $\longmapsto^+$ n'est pas décidable en général,

- (le plus important) dire que tous les calculs *au sens de* $\longmapsto^+$ passent inévitablement un ensemble donné n'est pas du tout équivalent à la propriété qui nous intéresse !!

Exemple 8.3. Un petit exemple.

C'est le fait de passer(arriver) *inévitablement* qui nous intéresse et pas celui d'une possibilité (qui s'exprime par la fermeture transitive de $\longmapsto$ avec la propriété de compatibilité utilisée dans [ACJY96]).

Dans la figure 8.8, il existe un calcul (le pointillé) $(q_1, \emptyset) \longmapsto^+ (q_3, \emptyset)$ au sens de $\longmapsto^+$ qui partant du noeud q_1 ne passe pas par le `wait`-noeud q_2; la propriété de compatibilité (page 120) utilisée dans [ACJY96] nous dit que si $(q_1, \emptyset) \leq (q_1, q_4)$, alors il existe un calcul $(q_1, q_4) \longmapsto^+ (q_3, q_4)$ et $(q_3, \emptyset) \leq (q_3, q_4)$. Ce calcul ne passe pas non plus par le `wait`-noeud q_2 et donc, la terminaison de la procédure invoquée (si elle termine) n'est pas prise en considération.

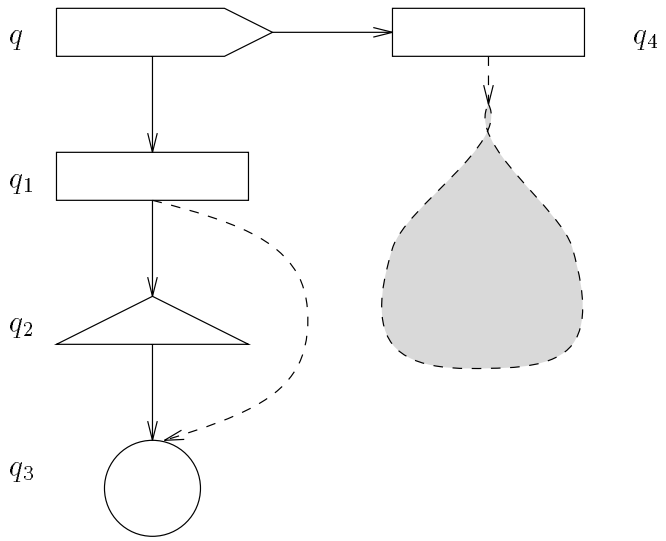


Figure 8.8 : Un petit exemple explicatif

Cette situation change dès qu'on prend la propriété de la compatibilité vers le haut pour $\preceq_\perp$.

8.4 Conclusion

Nous avons montré dans ce chapitre comment notre modèle d'états hiérarchiques peut être muni d'une structure de système de transitions bien structuré, ceci de 2 façons différentes, permettant de montrer la décidabilité de certains problèmes d'analyse.

On trouve de nombreux exemples de cette démarche dans [ACJY96], où il est écrit (à propos de systèmes de transitions bien structurés)
"for different classes of such systems (e.g., hybrid automata, data independed systems, relational automata, Petri nets, and lossy channel systems) this research has resulted in numerous highly nontrivial algorithms."

Pour notre part, nous nous sommes intéressés aux méthodes permettant d'analyser les programmes non interprétés présentés dans notre modèle.

Nous aimerions interpréter les résultats de ce chapitre de façon plus générale, p.ex. en faisant abstraction des particularités propres à RPPS.

Dans cette optique, la compatibilité vers le bas que nous avons proposée pour la relation $\preceq$ du lemme 8.2.3 permet d'obtenir une autre famille de

systèmes de transitions bien structurés qui est complètement différente. Une forme de la compatibilité vers le bas admettant une séquence de transitions (comme pour la relation $\preceq_\perp$ du lemme 8.3.5) peut être utilisée pour considérer les *channel systems with insertion errors* de [CFP95] comme un STBS en tirant des conséquences de cette structure pour leur analyse.

Puisque les systèmes de transition finis sont la base de nombreuses méthodes et outils d'analyse des programmes concurrents (cf. [IP91] pour une comparaison des principaux outils), nous aimerions actuellement établir une propriété de simulation entre notre modèle de schéma RPPS et un système d'états finis comme une des implémentations plus élaborées du fait d'"être bien structuré" que notre modèle possède. Comme cette idée est inspirée de la preuve de [ACJY96], nous avons bon espoir de pouvoir l'obtenir malgré le fait que notre modèle est moins simple que des exemples déjà connus de systèmes de transitions bien structurés. Ainsi, notre étude pourrait élargir le domaine d'application des méthodes de [Fin90, ACJY96].

Troisième partie

Des modèles d'implémentation

Introduction

Dans notre travail, nous ne cherchons pas à proposer une implémentation de RPC; notre but est de trouver un modèle formel pour l'implémentation, modèle qui permet de comprendre l'implémentation effective des programmes RPC telle qu'elle a été réalisée à l'IPTC.

Le compilateur RPC (cf. [VEKM94] pour sa description) qui a été développé pour une machine parallèle à l'IPTC, utilise une stratégie d'implémentation qui, en fait, revient à donner une nouvelle sémantique aux programmes RPC. Cette sémantique n'a pas été décrite formellement par les implémenteurs de RPC. Pour notre part, nous nous sommes intéressés à proposer une telle sémantique qui nous semble éclairante.

Dans cette partie de la thèse[1], nous allons formaliser (en deux étapes successives) la stratégie d'implémentation de [VEKM94] et nous allons énoncer en quel sens précis l'implémentation de RPC est correcte. Notre travail vise plusieurs objectifs:

1. nous voulons pouvoir établir quels liens formels existent entre l'implémentation actuelle de RPC et notre modèle d'états hiérarchiques,

2. nous voulons comprendre comment utiliser des résultats obtenus sur une abstraction non-interprétée des schémas pour produire une information pertinente sur des programmes réels; en particulier, quels types de propriétés sont préservés,

3. nous voulons illustrer comment la théorie sémantique du parallélisme peut éclairer les questions et problèmes d'implémentation.

Nous allons présenter deux modèles d'implémentation de RPC: la première (chapitre 9) est une réalisation distribuée à base de compteurs de synchronisation, avec parallélisme non-borné; la deuxième (avec parallélisme borné, voir chapitre 10) est une réalisation qui utilise elle aussi des compteurs, mais en plus, des files d'attente pour gérer le parallélisme. Nous allons appliquer une démarche de formalisation similaire à celle de la sémantique d'états hiérarchiques, c.-à-d. qu'on va procéder de façon opérationnelle.

[1] qui reprend et étend [KS96b].

Un des avantages de notre approche est la facilité avec laquelle les sémantiques différentes peuvent être comparées.

Nous montrerons la correction de deux sémantiques proposées dans cette partie de la thèse par rapport à celle d'états hiérarchiques. Le critère que nous utilisons est une notion de τ-*simulation* (au sens de Milner) préservant à la fois les propriétés de *sûreté* et de *terminaison*.

Pourquoi veut-on exhiber tel genre de liens et pourquoi se contentera-t-on de simulation? Nous justifions notre choix par un souci de donner un modèle formel pour RPC, modèle sur lequel on pourrait construire une nouvelle génération d'outils d'analyse de programmes récursifs-parallèles.

En ayant en vue l'outil d'analyse de programmes récursifs-parallèles, nous avons dû choisir quelles constructions allaient figurer dans notre modèle. Nous nous en sommes tenus à la synchronisation, au non-déterminisme influencé par l'extérieur, aux événements invisibles de l'extérieur pour allouer, déplacer, stocker, etc. des invocations. Un point important est qu'on cherche à traiter la divergence.

On se contentera de simulation car elle nous permet de choisir un niveau d'abstraction dans le modèle d'états hiérarchiques tel qu'il nous permet d'obtenir des résultats de décidabilité de plusieurs problèmes classiques.

Même si notre modèle, où il existe un événement invisible, est compliqué à manipuler, on est amené à notre notion de correction pour les raisons développées ci-dessus.

Une raison de plus pour choisir notre notion de correction consiste en ce que (comme on le verra dans le chapitre 11) la même relation d'implémentation relie la sémantique des schémas non-interprétée et celle avec des actions de base interprétées.

On va montrer comment raffiner nos modèles en incorporant l'interprétation des actions de base. Le résultat de correction permettra d'affirmer que toute propriété de sûreté ou de terminaison démontrée pour notre modèle $\mathcal{M}_G$ est vraie de la réalisation effective $\mathcal{P}_G^I$. A la fin du chapitre 11 on va énoncer un résultat de *complétude* de notre méthode de modélisation.

Chapitre 9

Une réalisation distribuée

La réalisation avec des compteurs (une réalisation distribuée) est un modèle d'implémentation de RPC avec parallélisme non-borné. C'est en fait une sémantique comportementale alternative des schémas de $RPPS_{\Lambda_\tau}$.

9.1 Une sémantique distribuée

La réalisation avec des compteurs a reçu son nom de l'utilisation de compteurs pour implémenter un langage de programmation. Nous allons donner une intuition de ce qui se passe quand un programme RPC non interprété s'exécute en utilisant des compteurs.

Tout d'abord, on précise que l'implémentation actuelle de RPC est telle qu'une instruction d'invocation `pcall` ne fait que créer une "activation" d'une procédure parallèle (sans transmettre le contrôle) en la mettant dans un buffer spécial. Chacune de ces activations est étroitement liée avec (on dirait qu'elle possède) une variable spéciale (dans une mémoire locale de processeur) qui s'appelle un *compteur de synchronisation*. La valeur initiale de chaque compteur est zéro. Une exécution d'une instruction d'invocation `pcall` d'une procédure parallèle à l'intérieur d'une activation donnée fait ajouter 1 à la valeur du compteur de cette activation; la terminaison d'une procédure parallèle (invoquée d'une activation donnée) soustrait 1 de la valeur du compteur de cette activation parentale. Une instruction de synchronisation `wait` peut être exécutée seulement à condition que la valeur du compteur associé à une activation donnée soit égale à 0.

Pour notre part, nous avons décidé de donner une sémantique comportementale des schémas $RPPS_{\Lambda_\tau}$ (d'un autre niveau d'abstraction) qui prend

en considération des valeurs de compteurs de synchronisation. Là, notre hypothèse du parallélisme non-borné nous permet de ne pas nous occuper (dans un premier temps) du problème des buffers où (d'où) les activations sont mises (prises).

Pour ce faire, pour donner une sémantique comportementale avec des raisonnements en terme de compteurs, nous allons suivre une démarche similaire à celle utilisée pour le modèle d'états hiérarchiques (cf. chapitre 6).

Un des avantages de cette approche est la facilité avec laquelle différentes sémantiques peuvent être comparées.

On considère un schéma G. La notion fondamentale dans cette section est celle d'un *état distribué*, défini formellement par

Définition 9.1.1. *(Etat distribué) Un* état distribué *(de G) est une séquence*

$$\sigma = \langle \omega_1; \ldots; \omega_i; \ldots; \omega_m \rangle$$

dans laquelle chaque composante ω_i est un bloc *d'activation, c.-à-d. une structure de la forme $l_i : q_i, n_i, r_i$ où*

$l_i \in \mathbb{N}$	*est l'*étiquette *du bloc*
$q_i \in Q_G \cup \{\perp\}$	*est l'*instruction *du bloc*
$n_i \in \mathbb{N}$	*est le* nombre de fils
$r_i \in \mathbb{N} \cup \{\perp\}$	*est l'*étiquette du père.

Informellement, un état distribué σ est la mise en parallèle de blocs d'activation où chaque bloc ω de la forme $l : q, n, r$ est repéré par son adresse (un entier l), contient une instruction courante q (ou $\perp$ s'il n'y a plus rien à faire), le nombre n de coroutines "filles" activées par le bloc, et l'adresse r du bloc ayant activé ω (ou $\perp$ si ω n'a pas de père).

Exemple 9.1. Un état distribué.

Notre schéma de l'exemple 6.1 admet, parmi d'autres, un état distribué

$$\sigma = \langle \underbrace{1 : q_2, 2, \perp;}_{\omega_1} \underbrace{2 : q_9, 0, 1;}_{\omega_2} \underbrace{3 : q_3, 0, 1}_{\omega_3} \rangle$$

où les blocs ω_2 et ω_3 ont été activés par le bloc ω_1 qui n'a pas de père (cf. la figure 9.1).

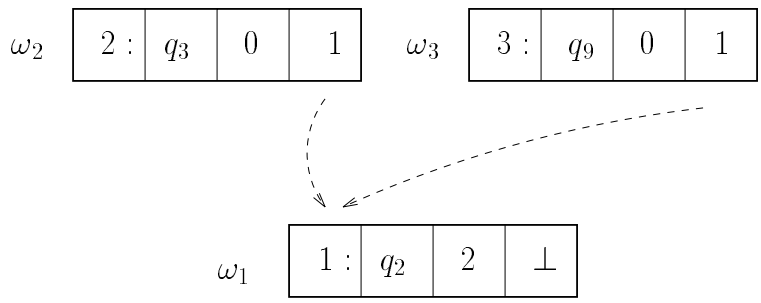


Figure 9.1 : Un état distribué

L'état distribué $\sigma_0 \stackrel{déf}{=} \langle \omega_{init} \rangle$ (où $\omega_{init} \stackrel{déf}{=} (l_1 : q_0, 0, \perp)$) est l'*état distribué initial*. On note Σ_G l'ensemble de tous les états distribués, et $Bloc_G$ l'ensemble de tous les blocs.

Au schéma G, on va associer un système de transitions étiqueté

$$\mathcal{S}_G = (\Sigma_G, \Lambda_\tau, \mapsto, \sigma_0)$$

Comme dans la définition 6.3.1, nous définissons la relation de transition $\mapsto \subseteq \Sigma_G \times \Lambda_\tau \times \Sigma_G$ en fonction du type des noeuds du schéma G.

Définition 9.1.2. *(Sémantique à états distribués)*
Le système de transitions étiqueté $\mathcal{S}_G \stackrel{\text{déf}}{=} (\Sigma_G, \Lambda_\tau, \mapsto, \sigma_0)$ *a un état initial* σ_0 *et une relation de transitions étiquetées* $\mapsto \subseteq \Sigma_G \times \Lambda_\tau \times \Sigma_G$ *définie comme la plus petite relation obéissant aux règles suivantes:*

(ACT_v) *si q est un noeud d'action étiqueté par α, et si q' est un des suivants de q, alors*

$$\langle \ldots; l : q, n, r; \ldots \rangle \stackrel{\alpha}{\mapsto} \langle \ldots; l : q', n, r; \ldots \rangle$$

(CALL_v) *si q est un noeud* `pcall`*, de noeud suivant q' et d'invoqué q'', alors*

$$\langle \ldots; l : q, n, r; \ldots \rangle \stackrel{\tau}{\mapsto} \langle \ldots; l : q', n+1, r; \ldots; l_{new} : q'', 0, l \rangle$$

où l_{new} est un label (un entier) non utilisé dans l'état courant.

(WAIT_v) *si q est un noeud* `wait` *de noeud suivant q', alors*

$$\langle \ldots; l : q, 0, r; \ldots \rangle \stackrel{\tau}{\mapsto} \langle \ldots; l : q', 0, r; \ldots \rangle$$

($\mathrm{END_v}$) *si q est un noeud* `end`*, alors*

$$\langle \ldots; l : q, n, r; \ldots \rangle \overset{\tau}{\mapsto} \langle \ldots; l : \bot, n, r; \ldots \rangle$$

($\mathrm{FREE_v}$) *si $q = \bot$, alors* [1]

$$\langle \ldots; \omega_{i-1}; l : \bot, 0, l'; \omega_{i+1}; \ldots; l' : q', n', r'; \ldots \rangle \overset{\tau}{\mapsto} \langle \ldots; \omega_{i-1}; \omega_{i+1}; \ldots; l' : q', n' - 1, r'; \ldots \rangle$$

- Avec la règle $\mathrm{ACT_v}$, seule une des composantes est modifiée: on avance dans l'instruction courante.
- La règle $\mathrm{CALL_v}$ ajoute un nouveau bloc dans l'état distribué, correspondant à la coroutine invoquée. Remarquons comment on a incrémenté le nombre de fils activés par le bloc l et comment le nouveau bloc contient l comme adresse de père.
- La règle $\mathrm{WAIT_v}$ remplace la condition "l'ensemble des fils est-il vide ?" de la définition 6.3.1 par une condition plus locale "le compteur de fils est il nul ?".
- La règle $\mathrm{END_v}$ ne retire pas de l'état courant un bloc qui termine. Simplement, le compteur d'instruction est avancé à $\bot$.
- C'est la règle $\mathrm{FREE_v}$ qui retire un bloc terminé, ceci à condition que son nombre de fils soit nul. Cette règle est la seule qui ne soit pas totalement locale, dans la mesure où elle va aller décrémenter, à l'intérieur du bloc l', le nombre de fils du père de l.

Exemple 9.2. Un comportement distribué.

Notre schéma de l'exemple 6.1 admet, parmi d'autres, une exécution débutant par

$$\langle 1 : q_0, 0, \bot \rangle \overset{a_1}{\mapsto} \langle 1 : q_1, 0, \bot \rangle \overset{\tau}{\mapsto} \langle 1 : q_2, 1, \bot;\ 2 : q_3, 0, 1 \rangle \overset{b_2}{\mapsto}$$

$$\langle 1 : q_2, 1, \bot;\ 2 : q_9, 0, 1 \rangle \overset{a_2}{\mapsto} \langle 1 : q_4, 1, \bot;\ 2 : q_9, 0, 1 \rangle \overset{b_1}{\mapsto} \langle 1 : q_5, 1, \bot;\ 2 : q_9, 0, 1 \rangle \overset{\tau}{\mapsto}$$

$$\langle 1 : q_5, 1, \bot;\ 2 : q_{10}, 1, 1;\ 3 : q_3, 0, 2 \rangle \cdots$$

où le `wait`-noeud q_5 ne peut pas être passé tant que le compteur associé est non-nul.

[1] Bien sûr la règle s'applique même si l' est à gauche de l dans σ. Par ailleurs, si l'adresse l' est $\bot$, alors la règle reste applicable, cette fois sans décrémentation d'un n'.

On voit ainsi comment, à l'intérieur d'un état distribué, les blocs utilisent les pointeurs qui les relient les uns aux autres. La définition ci-dessous est très naturelle:

Définition 9.1.3. *(Relation de dépendance)*
A l'intérieur d'un état σ donné, le bloc $\omega = l : q, n, r$ dépend *du bloc $\omega' = l' : q', n', r'$ ssi $r = l'$.*

Ainsi, dans notre exemple 9.1, le bloc ω_2 dépend du bloc ω_1 (ainsi que le bloc ω_3) car l'étiquette de père du bloc ω_2 est égale à l'étiquette du bloc ω_1 (les pointillés de la figure 9.1).

Définition 9.1.4. *(Correction de la valeur du compteur)*
Dans un état σ donné, on dit que la valeur du compteur n d'un bloc $\omega = l : q, n, r$ est correcte *s'il existe dans σ exactement n blocs $\omega_{i_1}, \omega_{i_2}, \ldots, \omega_{i_n}$ dépendant de ω.*

Dans notre exemple 9.1, la valeur du compteur du bloc ω_1 est correcte car il existe dans σ exactement 2 blocs ω_2, ω_3 dépendant de ω_1. Les valeurs des compteurs de ω_2 et de ω_3 sont aussi correctes.

Définition 9.1.5. *(Etat distribué cohérent)*
Un état distribué σ est dit cohérent *ssi*

1. *la relation de dépendance entre les blocs est acyclique,*
2. *les valeurs des compteurs sont toutes correctes,*
3. *les blocs ont tous des étiquettes différentes,*
4. *les étiquettes de père correspondent bien à des blocs de σ.*

On espère que le lecteur voit bien que notre définition d'état distribué cohérent est inspirée de l'intention de construire un modèle d'implémentation de RPC avec parallélisme non-borné pour produire une information sur des programmes réels.

Certaines conséquences de ces définitions peuvent être formulées immédiatement.

Lemme 9.1.6. *σ_0 est cohérent.*

Lemme 9.1.7. *Si σ est cohérent et $\sigma \overset{\alpha}{\mapsto} \sigma'$, alors σ' est cohérent.*

Preuve. On démontre la propriété par cas suivant les règles de transition.

1. Pour les règles $\mathrm{ACT_v}$, $\mathrm{WAIT_v}$, $\mathrm{END_v}$ qui ne changent rien aux pointeurs, la propriété est évidente.

2. Pour la règle $\mathrm{CALL_v}$ on ajoute 1 au compteur n de ω et on ajoute une nouvelle composante ω_{new} dépendant de ω_i ($r_{new} := l_i$). Clairement σ' est cohérent.

3. Pour la règle $\mathrm{FREE_v}$, la condition $n = 0$ et la cohérence de σ garantissent la cohérence de l'état σ' obtenu en retirant un bloc l et en décrémentant le compteur du bloc père.

□

9.2 Correspondances entre $\mathcal{M}_G$ et $\mathcal{S}_G$

Comme nous l'avons mentionné dans l'introduction de ce chapitre, un des avantages de notre approche est la facilité avec laquelle différentes sémantiques peuvent être comparées.

Dans cette section on va étudier les correspondances qui peuvent être faites entre les deux systèmes de transitions que nous avons associés à un schéma G.

Tout d'abord, on va montrer comment relier les états distribués cohérents et les états hiérarchiques. L'intuition est qu'un état distribué cohérent peut être vu comme une distribution sur des processeurs d'un état hiérarchique. Afin d'établir ce lien on donne quelques définitions indispensables.

Définition 9.2.1. *On dit qu'un état distribué σ' est une* permutation *de l'état distribué σ si σ' est obtenu par une permutation des blocs de σ.*

Il convient de noter que si les états distribués σ et σ' sont des permutations l'un de l'autre, cela veut dire que ce sont deux distributions différentes d'un même "état", qui ont donc le même comportement (cf. le corollaire 9.2.10).

Lemme 9.2.2. *Si σ est cohérent et σ' est une permutation de σ, alors σ' est cohérent.*

Preuve. Une permutation ne change pas les valeurs des compteurs, ni les pointeurs des blocs. D'où la cohérence de σ'.

□

Définition 9.2.3. *(Permutation de dépendance) A chaque état distribué σ cohérent on associe un état distribué σ_{dep}, appelé* permutation de dépendance *de σ. σ_{dep} est une permutation de σ de la forme*

$$\sigma_{dep} = \langle \omega_1 ; \sigma_1 ; \omega_2 ; \sigma_2 ; \ldots ; \omega_k ; \sigma_k \rangle$$

où pour chaque bloc ω_i, on a

- $r_i = \perp$, *et*
- σ_i *contient les blocs de σ qui dépendent transitivement de l_i.*

σ_{dep} peut être facilement construit en extrayant de σ tous les blocs $\{\omega_1, \ldots, \omega_k\}$ avec $r = \perp$. Ensuite, la cohérence implique que les autres blocs dépendent transitivement d'un des ω_i. On les range dans les σ_i correspondants.

Exemple 9.3. Permutation de dépendance.

Soit par exemple l'état distribué σ:
$\sigma = \langle 1 : q_1, 1, \perp ;\ 2 : q_5, 1, 1 ;\ 3 : q_7, 1, \perp ;\ 4 : q_3, 0, 2 ;\ 5 : q_2, 0, 3 \rangle$.
Sa permutation de dépendance σ_{dep} est:
$\sigma_{dep} = \langle \underbrace{1 : q_1, 1, \perp}_{\omega_1} ;\ \underbrace{2 : q_5, 1, 1 ;\ 4 : q_3, 0, 2}_{\sigma_1} ;\ \underbrace{3 : q_7, 1, \perp}_{\omega_2} ;\ \underbrace{5 : q_2, 0, 3}_{\sigma_2} \rangle$.

Maintenant on va associer un état hiérarchique $\xi_\sigma \in M$ à chaque état distribué cohérent $\sigma \in \Sigma$. Formellement

Définition 9.2.4. *A tout état distribué σ cohérent on associe un état hiérarchique $\xi_\sigma \in M$ défini inductivement par*

$$\xi_\sigma \stackrel{\text{déf}}{=} \sum_{i=1}^{k} \xi_{\langle \omega_i,\, \sigma_i \rangle}$$

si

$$\sigma_{dep} = \langle \omega_1 ; \sigma_1 ; \omega_2 ; \sigma_2 ; \ldots ; \omega_k ; \sigma_k \rangle$$

et où $\forall\, i\ (1 \leq i \leq k)$:

$$\xi_{\langle \omega_i,\, \sigma_i \rangle} \stackrel{\text{déf}}{=} \begin{cases} \xi_{\sigma_i} & si\ \omega_i = l_i : \perp, n_i, r_i \\ q_i, \xi_{\sigma_i} & si\ \omega_i = l_i : q_i, n_i, r_i \end{cases}$$

Exemple 9.4. Correspondance entre σ et ξ_σ.

A partir de la permutation de dépendance de l'état distribué σ de l'exemple 9.3 on construit un état hiérarchique ξ_σ suivant la définition 9.2.4:
$\xi_\sigma = \xi_{\langle\omega_1;\sigma_1\rangle} + \xi_{\langle\omega_2;\sigma_2\rangle}$, où
$\xi_{\langle\omega_1;\sigma_1\rangle} = q_1, \xi_{\sigma_1} = q_1, (q_5, \xi_{\omega_4}) = q_1, (q_5, (q_3, \emptyset))$ et où
$\xi_{\langle\omega_2;\sigma_2\rangle} = q_7, \xi_{\sigma_2} = q_7, (q_2, \emptyset)$. Donc, on obtient finalement
$\xi_\sigma = q_1, (q_5, (q_3, \emptyset)) + q_7, (q_2, \emptyset)$.

La construction de l'état hiérarchique ξ_σ à partir d'un état distribué σ que nous venons de décrire est insensible aux permutations:

Lemme 9.2.5. *Si σ est cohérent et σ' est une permutation de σ, alors $\xi_\sigma = \xi_{\sigma'}$.*

Preuve. A σ cohérent on associe $\xi_{\sigma_{dep}}$. A σ' qui est une permutation cohérente de σ (d'après le lemme 9.2.2) on associe $\xi_{(\sigma')_{dep}}$. Puisque $(\sigma')_{dep}$ est une permutation de σ_{dep} on voit aisément que $\xi_{\sigma_{dep}} = \xi_{(\sigma')_{dep}}$ par définition 9.2.4 et en raison des propriétés des multi-ensembles introduites ci-dessus. □

Nous pouvons maintenant établir des liens entre le système de transitions $\mathcal{M}_G$ basé sur les états hiérarchiques, et le système de transitions $\mathcal{S}_G$, basé sur les états distribués.

Lemme 9.2.6. *Soit σ un état distribué cohérent. Si $\sigma \overset{\alpha}{\mapsto} \sigma'$ alors*

$$\begin{array}{l} \textit{soit } \xi_\sigma \overset{\alpha}{\Longrightarrow} \xi_{\sigma'} \\ \textit{soit } \alpha = \tau,\ |\sigma'| < |\sigma| \textit{ et } \xi_\sigma = \xi_{\sigma'} \end{array}$$

Preuve. Par induction sur la longueur de σ. On considère les différents cas possibles pour la transition $\sigma \overset{\alpha}{\mapsto} \sigma'$.

On va procéder par type des règles de transitions.

1. Si $q \in Q_G$ est un noeud d'action étiqueté par $\alpha \in \Lambda$, alors par règle $\mathrm{ACT_v}$ on a

$$\underbrace{\langle \ldots, l : q, n, r; \ldots \rangle}_{\sigma} \overset{\alpha}{\mapsto} \underbrace{\langle \ldots, l : q', n, r; \ldots \rangle}_{\sigma'}$$

et σ, σ' sont cohérents. A σ cohérent on associe $\xi_\sigma = \xi_\sigma[q]$, à σ' cohérent on associe $\xi_{\sigma'} = \xi_{\sigma'}[q']$. Les états hiérarchiques ξ_σ et $\xi_{\sigma'}$ ont exactement la même structure sauf que une occurrence de q dans ξ_σ a été remplacée par q'. Par définition 6.3.1 on a:

$$\mathrm{ACT} \ :\ q, \xi_q \overset{\alpha}{\longmapsto} q', \xi_q \qquad \forall\, \xi_q,$$

où ξ_q est une famille d'invocations-"filles". Par les règles PAR_1, PAR_2, le lemme 6.4.3 et par structure des états hiérarchiques ξ_σ, $\xi_{\sigma'}$ on est amené à $\xi_\sigma \stackrel{\alpha}{\longmapsto} \xi_{\sigma'}$.

2. Si $q \in Q_G$ est un `pcall`-noeud étiqueté par τ, alors l'exécution de la règle $\mathrm{CALL_v}$ produit une composante nouvelle ω_{new} dépendant de ω_i:

$$\langle \ldots; \underbrace{l_i : q, n, r}_{\omega_i}; \ldots \rangle \stackrel{\tau}{\mapsto} \langle \ldots; \underbrace{l_i : q', n+1, r}_{\omega_i}; \ldots; \underbrace{l_{new} : q'', 0, l_i}_{\omega_{new}} \rangle,$$

et σ, σ' sont cohérents. A l'état distribué σ on associe $\xi_\sigma = \xi_\sigma[q]$, à l'état distribué σ' on associe $\xi_{\sigma'} = \xi_{\sigma'}[\zeta]$. Les états hiérarchiques ξ_σ et $\xi_{\sigma'}$ ont exactement la même structure (par construction) sauf que une occurrence de q dans ξ_σ a été remplacée par ζ avec une transition valide $q \stackrel{\tau}{\mapsto} \zeta$. Puisque q est un `pcall`-noeud avec le suivant q' et l'invoqué q'' par règle CALL on a

$$q, \xi_q \stackrel{\tau}{\longmapsto} q', \xi_q + (q'', \emptyset).$$

Par des règles PAR_1, PAR_2, lemme 6.4.3 et par structure des états hiérarchiques ξ_σ, $\xi_{\sigma'}$ on déduit immédiatement $\xi_\sigma \stackrel{\tau}{\longmapsto} \xi_{\sigma'}$.

3. Pour la règle $\mathrm{WAIT_v}$ on le montre facilement, car on ne modifie pas les structures de σ, σ' par transition.

4. Si $q \in Q_G$ est un `end`-noeud étiqueté par τ, alors l'exécution de la règle $\mathrm{END_v}$ remplace q par $\perp$:

$$\mathrm{END_v} \; : \; \underbrace{\langle \ldots; l : q, n, r; \ldots \rangle}_{\sigma} \stackrel{\tau}{\mapsto} \underbrace{\langle \ldots; l : \perp, n, r; \ldots \rangle}_{\sigma'}$$

et en outre σ, σ' sont cohérents. A σ cohérent on associe $\xi_\sigma = \xi_\sigma[q]$, à l'état distribué σ' cohérent on associe un état hiérarchique $\xi_{\sigma'} = \xi_{\sigma'}[\zeta]$. Les états hiérarchiques $\xi_{\sigma'}$ et ξ_σ ont exactement la même structure de forêt mais les arbres liés à q sont différents. Cette différence consiste en un collage convenable des "fils" de l'état hiérarchique (q, ξ_q) à leur "père". Par la définition 6.3.1 on a :

$$\mathrm{END} : \; q, \xi_q \stackrel{\tau}{\longmapsto} \xi_q.$$

Les règles PAR_1, PAR_2, le lemme 6.4.3 et la structure des états hiérarchiques ξ_σ et $\xi_{\sigma'}$ nous conduisent à

$$\xi_\sigma \stackrel{\tau}{\longmapsto} \xi_{\sigma'}.$$

5. Pour la règle FREE_v on a (d'après la définition 9.1.2):

$$\underbrace{\langle \ldots;\omega_{i-1};l:\perp,0,l_j;\omega_{i+1};\ldots;l_j:q_j,n_j,r_j;\ldots\rangle}_{\sigma} \overset{\tau}{\mapsto}$$

$$\underbrace{\langle \ldots;\omega_{i-1};\omega_{i+1};\ldots;l_j:q_j,n_j-1,r_j;\ldots\rangle}_{\sigma'}.$$

A σ cohérent on associe ξ_σ, à σ' (qui est cohérent selon le lemme 9.1.7) on associe $\xi_{\sigma'}$. On voit sans peine que $\xi_{\sigma'}$ a exactement la même structure (par la définition 9.2.4) que ξ_σ et donc, $\xi_\sigma = \xi_{\sigma'}$, ce qui complète la preuve.

□

Il est utile de remarquer que ce lemme établit en fait que la fonction (partielle) de Σ dans M, qui à σ associe ξ_σ, est une τ-simulation entre $\mathcal{M}_G$ et $\mathcal{S}_G$.

De plus la divergence est préservée:

Lemme 9.2.7. *Si σ cohérent diverge, alors ξ_σ diverge.*

Preuve. Considérons une suite infinie de τ-transitions partant de σ:

$$\sigma = \sigma_0 \overset{\tau}{\mapsto} \sigma_1 \overset{\tau}{\mapsto} \cdots \sigma_i \overset{\tau}{\mapsto} \cdots$$

Forcément on a $|\sigma_{i+1}| \geq |\sigma_i|$ infiniment souvent, de sorte qu'il existe une infinité de suites de transitions $\xi_{\sigma_i} \overset{\tau}{\Rightarrow} \xi_{\sigma_{i+1}}$ entre les états hiérarchiques correspondants. Comme une séquence $\overset{\tau}{\Rightarrow}$ de τ-transitions ne peut pas être vide (contrairement à $\overset{\epsilon}{\Rightarrow}$), on en déduit que ξ_σ diverge.

□

On peut exhiber une simulation dans l'autre direction:

Lemme 9.2.8. *Si $\xi \overset{\alpha}{\longmapsto} \xi'$ et $\xi = \xi_\sigma$, alors il existe σ' tel que $\sigma \overset{\alpha}{\mapsto} \sigma'$ et $\xi' = \xi_{\sigma'}$.*

Preuve. On démontre la propriété par le type des règles de transitions.

1. Si $q \in Q_G$ est un noeud d'action étiqueté par $\alpha \in \Lambda$, alors on a la règle ACT à appliquer: ACT : $\underbrace{\xi[q]}_{\xi_\sigma} \overset{\alpha}{\longmapsto} \xi[\zeta]$ avec une transition valide $q,\xi_q \overset{\alpha}{\longmapsto} \zeta$, où $\zeta = q',\xi_q$. Les états hiérarchiques ξ_σ et $\xi[\zeta]$ ont exactement la même structure sauf que une occurrence de q dans ξ_σ a été remplacée par q'. L'état hiérarchique $\xi_\sigma[q]$ est associé à l'état

distribué σ donc, il est possible d'effectuer la transition qui obéit à la règle $\mathrm{ACT_v}$ (d'après la définition 9.1.2):

$$\underbrace{\langle \ldots; l_i : q, n, r; \ldots \rangle}_{\sigma} \stackrel{a}{\mapsto} \underbrace{\langle \ldots; l_i : q', n, r; \ldots \rangle}_{\sigma'}.$$

A σ' qui est cohérent (d'après le lemme 9.1.7) on associe $\xi_{\sigma'}$ et par comparaison de $\xi[\zeta]$ et $\xi_{\sigma'}$ on obtient le résultat $(\xi' =)\xi[\zeta] = \xi_{\sigma'}$.

2. Pour les règles CALL, WAIT, END on le vérifie de même façon.

□

Maintenant on est prêt à énoncer le résultat principal de cette section qui nous permet de relier formellement la sémantique d'états hiérarchiques et celle d'états distribués:

Théorème 9.2.9. *Si σ cohérent, alors σ et ξ_σ sont d-bisimilaires.*

Preuve. On considère la relation R qui associe à chaque σ cohérent le ξ_σ correspondant. Les lemmes 9.2.6, 9.2.7 et 9.2.8 énoncent que R a la propriété de transfert de la d-bisimulation.

□

Nous pouvons en déduire le corollaire suivant:

Corollaire 9.2.10. *Si σ et σ' sont des permutations l'une de l'autre, alors σ, σ' sont d-bisimilaires.*

Même si en fait σ et σ' sont bisimilaires au sens fort.

C'est un corollaire immédiat qui valide formellement l'idée introduite p. 146 comme quoi les permutations de σ ne changent rien.

C'est le théorème 9.2.9 qui nous permet d'affirmer que $\mathcal{S}_G$, la réalisation distribuée de G à base de compteurs, est une implémentation correcte de la sémantique $\mathcal{M}_G$ basée elle sur la notion d'états hiérarchiques.

Nous allons maintenant passer à un autre modèle d'implémentation de niveau d'abstraction plus bas, plus proche encore de la stratégie utilisée dans le compilateur RPC de [VEKM94].

Dans ce cadre, notre réalisation avec des compteurs (une réalisation distribuée) est une étape intermédiaire entre la sémantique d'états hiérarchiques (cf. chapitre 6) et un modèle d'implémentation qui est basé, là aussi, sur l'idée de compteurs de synchronisation, mais en plus, qui prend en compte une gestion du parallélisme.

Chapitre 10

Implémentation avec parallélisme borné

Dans ce chapitre, on va développer un modèle pour RPC, à base de compteurs (comme dans le chapitre précédent), mais utilisant en plus des files d'attente et des *dequeues* ("double-ended queues", ou files d'attente à double entrée) pour gérer un parallélisme borné (cf. [MVVK88, VEKM94] pour une description de gestion).

10.1 Une sémantique vectorielle

Nous allons proposer encore une sémantique comportementale alternative de schémas $RPPS_{\Lambda_\tau}$. Il convient de noter qu'elle correspond à un niveau d'abstraction plus bas que la sémantique d'états distribués.

On suppose fixé un entier $m \in \mathbb{N}$, nombre de processeurs du système distribué ($m > 0$). On considère un schéma G.

Concrètement, la stratégie utilisée dans [VEKM94] pour implémenter le langage RPC sur une architecture à m processeurs parallèles (et mémoire partagée) utilise, pour chaque processeur, une file d'attente (locale) et une *dequeue* pour les invocations parallèles en attente d'allocation. Le problème quand on a du parallélisme borné consiste à gérer les invocations/activations. Il faut les allouer, les déplacer, les stocker, etc. Cette stratégie peut être formalisée au moyen d'*états vectoriels*.

Définition 10.1.1. *Un* vecteur-état *(de taille m) est une séquence*

$$\pi = \langle \rho_1;\ \ldots;\ \rho_m \rangle$$

dans laquelle chaque composante ρ_j *est de la forme* ω_j, f_j, d_j *où*
- $\omega_j \in Bloc \cup \{\bot\}$ *est un* bloc *(ou bien* $\bot$*),*
- $f_j \in Bloc^*$ *est une* file *de blocs,*
- $d_j \in Bloc^*$ *est une* dequeue *de blocs.*

Les blocs sont les mêmes blocs que dans la définition 9.1.1.

Exemple 10.1.

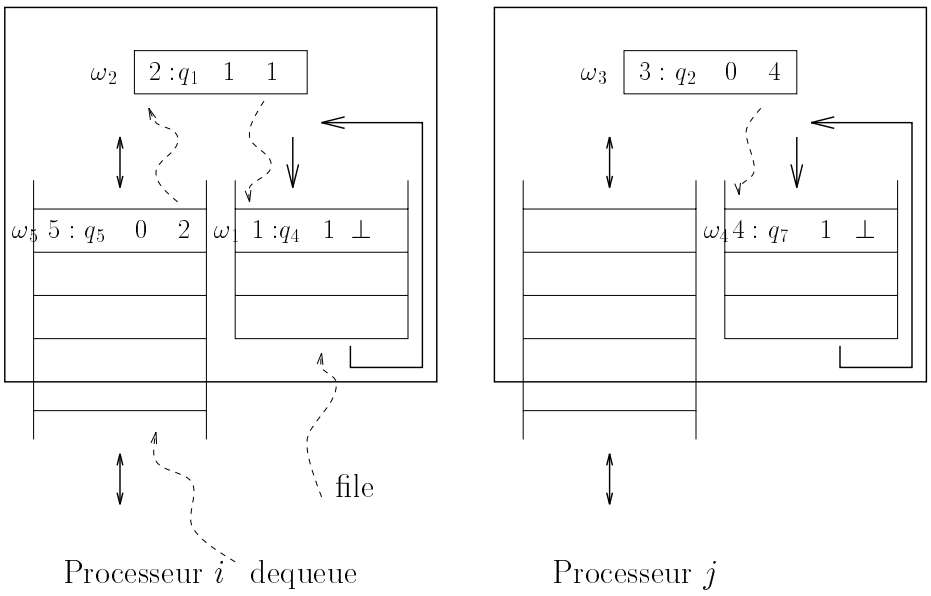


Figure 10.1 : Un état vectoriel

L'état vectoriel

$$\pi = \langle \ldots \underbrace{(2:q_1,1,1),((1:q_4,1,\bot)),((5:q_5,0,2))}_{\rho_i}; \ldots \underbrace{(3:q_2,0,4),((4:q_7,1,\bot)),()}_{\rho_j} \ldots \rangle$$

a, en particulier, deux composantes ρ_i et ρ_j, où le processeur i est en train d'exécuter le bloc $\omega_2 = (2 : q_1, 1, 1)$ dépendant du bloc $\omega_1 = (1 : q_4, 1, \bot)$ qui est en file d'attente; le bloc ω_5 (crée par ω_2) est mis en dequeue et peut être pris par n'importe quel processeur (cf. la figure 10.1). Les processeurs i et j exécutent les blocs ω_2 et ω_3 en parallèle.

Dans l'état π, le processeur j est en train d'exécuter ω_j et les blocs appartenant à f_j ou d_j sont en attente. Les blocs de f_j (gérés avec une discipline FIFO) sont des processus qui ont été interrompus (sur le processeur j) lors

d'une instruction `wait`. Ils seront (peut-être) réactivés plus tard. Les blocs de d_j (file accessible par tous les processeurs) sont des invocations créées par les instructions `pcall`. Elles peuvent être exécutées par n'importe quel processeur.

On va formaliser une stratégie d'implémentation dont on a donné les idées (avec un exemple) ci-dessus.

On note Π_G l'ensemble de tous les états vectoriels. Le vecteur-état $\pi_0 = \langle \omega_{init}, (), (); \underbrace{\bot; \ldots; \bot}_{m-1} \rangle$ s'appelle *vecteur-état initial.*

A un schéma G on associe un système de transitions étiqueté $\mathcal{P}_G = (\Pi_G, \Lambda_\tau, \mapsto, \pi_0)$.

Définition 10.1.2. *(Sémantique vectorielle)*
$\mathcal{P}_G \stackrel{\text{déf}}{=} (\Pi_G, \Lambda_\tau, \mapsto, \pi_0)$ *est le système de transitions étiqueté où la relation* $\mapsto \subseteq \Pi_G \times \Lambda_\tau \times \Pi_G$ *est obtenue comme la plus petite relation qui obéit aux règles suivantes:*

ACT$_a$: *si q est un noeud d'action (ou de test) étiqueté par α et si q' est un des suivants de q, alors*

$$\langle \ldots; l : q, n, r, f, d; \ldots \rangle \stackrel{\alpha}{\mapsto} \langle \ldots; l : q', n, r, f, d; \ldots \rangle$$

END$_a$: *si q est un noeud* `end`*, alors*

$$\langle \ldots; l : q, n, l_j, f, d; \ldots \rangle \stackrel{\tau}{\mapsto} \langle \ldots; l : \bot, n, l_j, f, d; \ldots \rangle$$

CALL$_a$: *si q est un noeud* `pcall`*, de suivant q' et d'invoqué q'', alors*

$$\langle \ldots; l : q, n, r, f, d; \ldots \rangle \stackrel{\tau}{\mapsto} \langle \ldots; l : q', n+1, r, f, (l_{new} : q'', 0, l).d; \ldots \rangle$$

où l_{new} est un label non utilisé dans l'état courant.

WAIT$_a$: *si q est un noeud* `wait` *de suivant q', alors*

$$\langle \ldots; l : q, 0, r, f, d; \ldots \rangle \stackrel{\tau}{\mapsto} \langle \ldots; l : q', 0, r, f, d; \ldots \rangle$$

PUSH$_a$: *si q est un noeud* `wait` *de suivant q' et si $n > 0$, alors*

$$\langle \ldots; l : q, n, r, f, d; \ldots \rangle \stackrel{\tau}{\mapsto} \langle \ldots; \bot, f.(l : q, n, r), d; \ldots \rangle$$

FREE$_a$:

$$\langle \ldots;\ \underbrace{l:\bot, 0, l'}_{\omega}, f, d;\ \ldots \cdots \underbrace{l':q', n', r'}_{\omega'} \cdots\ \ldots \rangle$$
$$\stackrel{\tau}{\mapsto} \langle \ldots;\ \bot, f, d;\ \ldots \cdots l':q', n'-1, r' \cdots\ \ldots \rangle$$

où la notation $\cdots\omega'\cdots$ signifie que ω' apparaît quelque part (sur un processeur, dans un des d_j ou des f_j) dans l'état vectoriel, ce que le système sait localiser directement à partir de l'étiquette l'.

POP – FILE:

$$\langle \ldots;\ \bot, \underbrace{(\omega_1, \ldots, \omega_k)}_{f}, d;\ \ldots \rangle \stackrel{\tau}{\mapsto} \langle \ldots;\ \omega_i, (\omega_{i+1}, \ldots, \omega_k, \omega_1, \ldots, \omega_{i-1}), d;\ \ldots \rangle$$

où ω_i est le premier bloc de f tel que $n_i = 0$.

POP – DEQUEUE:

$$\langle \ldots;\ \bot, f, (\omega.d);\ \ldots \rangle \stackrel{\tau}{\mapsto} \langle \ldots;\ \omega, f, d;\ \ldots \rangle$$

si f n'a pas de bloc tel que $n = 0$ (i.e. la règle POP – FILE *ne s'applique pas).*

POP – AUTRE – DEQUEUE:

$$\langle \ldots;\ \bot, f, ();\ \ldots;\ \omega_j, f_j, (d_j.\omega);\ \ldots \rangle \stackrel{\tau}{\mapsto} \langle \ldots;\ \omega, f, ();\ \ldots;\ \omega_j, f_j, d_j;\ \ldots \rangle$$

si f n'a pas de bloc tel que $n = 0$ (i.e. la règle POP – FILE *ne s'applique pas).*

Les règles que nous venons de donner définissent quelles transitions $\pi \stackrel{\alpha}{\mapsto} \pi'$ sont possibles. Ces règles reprennent la gestion des compteurs de fils de la définition 9.1.2 mais la présence des files d'attente nécessite des ajustements:

- Seuls les blocs *actifs*, i.e. présents sur un processeur plutôt que dans une file d'attente, peuvent évoluer.
- En cas d'invocation d'une coroutine, la coroutine est rangée dans la dequeue locale (et donc pas active).
- Un noeud `wait` avec compteur $n \neq 0$ ne reste pas simplement en attente, il sera rangé dans la file locale, libérant ainsi le processeur.
- La stratégie de dépilage qui alloue des blocs aux processeurs libres impose une priorité sur les trois sortes de dépilage: POP – FILE passe avant POP – DEQUEUE qui elle-même passe avant POP – AUTRE– DEQUEUE.

A un état vectoriel on peut associer un état distribué correspondant. Là, notre idée est très simple: pour chaque composante ρ d'un état vectoriel

1. d'abord, on extrait successivement les contenus (des blocs d'activations) d'une file d'attente et on les met à côté pour former des nouvelles composantes d'un état distribué;

2. ensuite, on applique la même démarche pour des blocs de dequeue.

Formellement,

Définition 10.1.3. *Au vecteur-état*

$$\pi = \underbrace{\langle \ldots;\ \omega_i, (\omega'_1, \ldots, \omega'_{|f_i|}), (\omega''_1, \ldots, \omega''_{|d_i|});\ \ldots \rangle}_{m}$$

on associe un état distribué σ_π (une allocation *de π) défini inductivement par*

1.

$$\underbrace{\langle \ldots; \omega_i, (\omega'_2, \ldots, \omega'_{|f_i|}), (\omega''_1, \ldots, \omega''_{|d_i|}); \ldots;}_{m} \underbrace{\ldots\ldots\ldots\ldots; \omega'_1;}_{\sum_{k=1}^{i-1}(|f_k|+|d_k|)+1} \rangle;$$

2.

$$\underbrace{\langle \ldots; \omega_i, (), (\omega''_2, \ldots, \omega''_{|d_i|}); \ldots;}_{m} \underbrace{\ldots; \omega'_1; \ldots\ldots\ldots; \omega'_{|f_i|}; \omega''_1}_{\sum_{k=1}^{i-1}(|f_k|+|d_k|)+|f_i|+1} \rangle.$$

Exemple 10.2. Un état vectoriel et un état distribué correspondant.

Au vecteur-état π de l'exemple 10.1 on associe un état distribué σ_π de la forme

$$\sigma_\pi = \langle \ldots; (2 : q_1, 1, 1); \ldots; (3 : q_2, 0, 4); \ldots; (1 : q_4, 1, \perp); (5 : q_5, 0, 2); (4 : q_7, 1, \perp) \rangle$$

Dans la suite nous nous intéressons aux états vectoriels *cohérents*, c'est à dire tels que la relation de dépendance entre les blocs est acyclique, les valeurs des compteurs sont correctes, ... comme dans la définition 9.1.5. Là, notre motivation est la même que dans le cas d'état distribué cohérent: on cherche à construire un modèle d'implémentation de RPC possédant une information sur des programmes réels.

Lemme 10.1.4. *Un vecteur-état π est cohérent ssi son allocation σ_π l'est.*

Preuve. Dans la définition 10.1.3, la construction de σ_π ne change ni l'ensemble des blocs, ni les pointeurs, ni les compteurs.

□

Nous pouvons alors montrer

Lemme 10.1.5. *Si π est cohérent et $\pi \stackrel{\alpha}{\mapsto} \pi'$, alors π' est cohérent.*

Preuve. Se montre par type des règles de transitions de la définition 10.1.2.

□

Lemme 10.1.6. *Si π est cohérent et si $\pi \stackrel{\alpha}{\mapsto} \pi'$, alors $\sigma_\pi \stackrel{\alpha}{\Longrightarrow} \sigma_{\pi'}$.*

Preuve. Se montre par type des règles de transitions de la définition 10.1.2 à partir de la définition 10.1.3 et celle 9.1.2 en utilisant le lemme 9.1.7. Chaque type de transitions obéissant aux règles de la définition 10.1.2 est simulé

- soit par zéro τ-transitions partant de σ_π car on n'a pas besoin de gérer des invocations/activations dans le cas d'états distribués,
- soit par une seule transition correspondante de la définition 9.1.2.

□

Lemme 10.1.7. *Pour un état cohérent π, on a $\pi \underline{\leftrightarrow}_d \mathbf{0}$ ssi $\sigma_\pi \underline{\leftrightarrow}_d \mathbf{0}$.*

Preuve.

- La direction $\Rightarrow$ suit du lemme 10.1.6.
- Nous voyons la preuve de l'autre implication. La preuve est par l'absurde. On suppose que l'état distribué σ_π est bisimilaire à $\mathbf{0}$ mais l'état vectoriel π ne l'est pas, c.-à-d.

  1. soit $\exists \alpha \in \Lambda : \pi \stackrel{\alpha}{\Longrightarrow}$;
  2. soit le nombre k de τ-transitions partant de π n'est pas fini.

  La première situation implique ce que $\alpha \in \Lambda : \sigma_\pi \stackrel{\alpha}{\Longrightarrow}$ d'après le lemme 10.1.6 et donc, $\sigma_\pi \not\underline{\leftrightarrow}_\tau \mathbf{0}$ (la contradiction). La deuxième situation implique l'existence (d'après l'application multiple du lemme 10.1.6) de la suite infinie de τ-transitions partant de σ_π et donc, $\sigma_\pi \not\underline{\leftrightarrow}_\tau \mathbf{0}$ (la contradiction).

□

Il est alors possible de relier formellement le comportement des deux modèles d'implémentation de niveaux d'abstraction différents: du modèle d'états distribués et celui d'états vectoriels. Le critère que nous utilisons est une notion de τ-simulation au sens de [Mil89] préservant à la fois les propriétés de sûreté et la terminaison. Nous notons $\sqsubseteq_d$ ce préordre d'implémentation.

Définition 10.1.8. *$p \sqsubseteq_d q$ ssi il existe une relation R telle que*

1. *R a la propriété de (τ)-simulation;*
2. *$p\ R\ q$ implique ($p \underline{\leftrightarrow}_d \mathbf{0}$ ssi $q \underline{\leftrightarrow}_d \mathbf{0}$).*

Il est alors possible de démontrer

Corollaire 10.1.9.

$$\mathcal{P}_G \sqsubseteq_d \mathcal{S}_G$$

C'est en ce sens que la stratégie de [VEKM94] est une implémentation correcte de la sémantique d'états hiérarchiques. Notons que seule une simulation est garantie, et qu'en particulier, le langage engendré (et a fortiori le comportement arborescent) n'est pas préservé. Là, une raison intuitive est la suivante: une instruction courante q d'un bloc ω d'un état distribué σ_π peut être exécutée, tandis qu'il est tout à fait possible que ce bloc ω d'un état vectoriel π ne puisse pas être dépilé d'une des files d'attente ou bien des dequeues d'un des m processeurs qui sont tous en train d'exécuter leurs blocs.

Exemple 10.3. Un état vectoriel et un état hiérarchique correspondant.
A partir d'un état vectoriel

$$\pi = \langle \ldots \underbrace{(2 : q_1, 1, 1), ((1 : q_4, 1, \bot)), ((5 : q_5, 0, 2))}_{\rho_i}; \ldots \underbrace{(3 : q_2, 0, 4), ((4 : q_7, 1, \bot)), ()}_{\rho_j} \ldots \rangle$$

de l'exemple 10.1 par l'intermédiaire de l'état distribué correspondant

$$\sigma_\pi = \langle \ldots (2 : q_1, 1, 1); \ldots; (3 : q_2, 0, 4); \ldots;$$
$$(1 : q_4, 1, \bot); (5 : q_5, 0, 2); \ldots; (4 : q_7, 1, \bot); \ldots \rangle$$

on peut construire un état hiérarchique ξ:

$$\xi_{\sigma_\pi} = \{(q_4, (q_1, (q_5, \emptyset))), (q_7, (q_2, \emptyset))\}$$

L'instruction q_5 peut être exécutée dans ξ_{σ_π} et σ_π tous les deux, mais le bloc $\omega = (5 : q_5, 0, 2)$ ne peut pas être dépilé de dequeue du processeur i si tous les processeurs sont occupés.

Maintenant on peut expliquer en quel sens on aurait pu proposer une "meilleure" notion de correction. En premier lieu, on aurait pu chercher un modèle d'implémentation tel qu'il préserve le langage engendré par le modèle RPPS; en deuxième lieu, un autre qui préserve le comportement arborescent.

En ayant en vue l'outil d'analyse des programmes récursifs-parallèles, nous avons dû choisir quelles constructions allaient figurer dans notre modèle. Nous nous en sommes tenu à la synchronisation, au non-déterminisme influencé par l'extérieur, aux événements invisibles (les τ-transitions). On a cherché à traiter la divergence (suite infinie de transitions invisibles). Même si notre modèle, où il existe un événement invisible est compliqué à manipuler, nous avons préféré notre notion de correction pour les raisons développées ci-dessus.

Dans les chapitres précédents, nous avons décrit une façon par laquelle un programme récursif-parallèle P est transformé en un schéma RPPS $G_P \in RPPS$ pour lequel nous avons proposé trois sémantiques différentes, reliées par:

$$\mathcal{P}_G \quad \sqsubseteq_d \quad \mathcal{S}_G \quad \underline{\leftrightarrow}_d \quad \mathcal{M}_G \tag{10.1}$$

Les trois modèles ont leurs propres vues sur le sens du parallélisme et de la synchronisation du langage récursif-parallèle, mais ils restent compatibles au sens de "$\sqsubseteq_d$". Nous allons voir que cette même notion de compatibilité relie les modèles sans interprétation que nous avons utilisés jusqu'à présent, et les modèles interprétés qui donnent une sémantique plus réaliste.

Chapitre 11

Schémas RPPS interprétés

11.1 Propriétés de programmes

Une propriété d'un programme est un attribut qui est vrai de chaque exécution possible d'un programme considéré. Chaque propriété peut être formulée en termes de deux sortes spéciales de propriétés: *sûreté* et *vivacité*. Une propriété de sûreté exprime que le programme n'admet jamais un mauvais état, i.e. un état où certaines des variables ont des valeurs indésirables. Une propriété de vivacité exprime qu'un programme passe inévitablement par un bon état, i.e. un état où toutes les variables d'un programme ont des valeurs correctes.

La *correction partielle* qui est un exemple de propriété de sûreté exprime que si un programme termine, alors l'état final est correct, i.e. un résultat correct a été obtenu. L'absence de deadlock est un autre exemple de propriété de sûreté. La *terminaison* qui est un exemple d'une propriété de vivacité exprime qu'un programme sera inévitablement terminé, i.e. chaque exécution d'un programme est finie. La *correction totale* est une propriété qui combine la correction partielle et la terminaison. La correction totale dit qu'un programme termine toujours, et avec un résultat correct.

11.2 Modèles interprétés

L'intérêt du résultat (10.1) ne se limite pas aux propriétés de préservation de $\sqsubseteq_d$. En effet, la même relation d'implémentation relie le comportement $\mathcal{P}_G$ des programmes et leur sémantique "interprétée" $\mathcal{P}_G^I$.

Dans notre définition du comportement des schémas RPPS, les actions de base restaient sans interprétation. Mais il est possible d'étoffer notre

modèle de façon à prendre en compte une interprétation des actions de base, donnant une sémantique complète du langage récursif-parallèle RPC. (Bien sûr, on perd alors presque tous les résultats de décidabilité.) Dans l'ordre nous expliquons

- ce que seraient les modèles interprétés;
- comment on les relie formellement aux modèles non-interprétés.

Les modèles interprétés ont seulement besoin que soit modélisée la mémoire. Elle est découpée en

- une mémoire globale à laquelle toutes les invocations peuvent accéder;
- pour chaque invocation, une mémoire locale.

Il faudrait définir un ensemble (généralement infini)

$$GMem = \{u, \ldots\}$$

d'états-mémoire globaux, ainsi qu'un ensemble (généralement infini)

$$LMem = \{v, \ldots\}$$

d'états-mémoire locaux. Dans ce cas, les actions de base auront un effet précis sur l'état-mémoire (le mécanisme de synchronisation reste le même).

On munit alors les actions de base et les actions de tests d'une sémantique sur $GMem \times LMem$. Par exemple une action $\alpha \in \Lambda$ sera interprétée comme une application $\stackrel{\alpha}{\longmapsto} \subseteq GMem \times LMem \times GMem \times LMem$. On écrira $u, v \stackrel{\alpha}{\longmapsto} u', v'$ pour dénoter les *modifications de l'état-mémoire* dues à l'accomplissement des actions α.

Des actions `pcall` auront, elles aussi, un effet sur l'état local et sur l'état global, mais en plus elles donnent lieu à un nouvel état local pour la procédure invoquée, ce qui est modélisé par une application $GMem \times LMem \longmapsto (GMem \times LMem) \times LMem$.

Maintenant, on peut définir un état hiérarchique qui aura la forme

$$\langle u, \xi \rangle$$

avec

$$\xi \stackrel{déf}{=} \{(q_1, v_1, \xi_1), \ldots, (q_n, v_n, \xi_n)\}$$

où chaque composante parallèle est munie de son état-mémoire local. Les transitions prennent en compte l'effet des actions sur l'état-mémoire.

Nous n'allons pas écrire explicitement la façon dont les règles de la définition 6.3.1 sont adaptées à ce nouveau cadre et préférons donner un exemple:

Exemple 11.1. Comportement avec actions interprétées.

1. Si $q \in Q_G$ est un noeud `pcall` de suivant q' et d'invoqué q'' et
2. si $((u,v),(u',v'),v'') \in \stackrel{\tau}{\longmapsto}$,

alors $\langle u,(q,v,\xi)\rangle \stackrel{\tau}{\longmapsto} \langle u',(q',v',(\xi+(q'',v'',\emptyset)))\rangle$.

Il y a toutefois une différence importante: les actions et (surtout) les tests donnent un résultat déterministe.

Finalement, on voit la façon par laquelle cette construction produit un modèle $\mathcal{M}_G^I$ pour lequel il ne reste qu'à choisir un état initial précis $\xi_0 \stackrel{déf}{=} (q_0,v_0,\emptyset)$. Même si nous ne donnons pas la définition exacte des sémantiques interprétées, le lecteur peut se convaincre que le théorème suivant permet de la relier au modèle sans interprétation:

Théorème 11.2.1. *Compatibilité de la sémantique interprétée*

$$\mathcal{M}_G^I \sqsubseteq_d \mathcal{M}_G$$

où le modèle non-interprété ne fait qu'une τ-simulation du modèle interprété (à cause du non-déterminisme des tests dans le modèle non-interprété), mais néanmoins, une simulation qui respecte la divergence.

De façon semblable, nous pourrions définir un modèle d'états distribués et un modèle d'états vectoriels *avec interprétation*. A la fin, les six modèles sont reliés comme le résume la figure 11.1.

Dans ce cadre, le point important est que les modèles non-interprétés implémentent correctement les modèles interprétés en prenant $\sqsubseteq_d$ comme préordre d'implémentation correcte. Or c'est le même préordre qui explique en quoi la sémantique "idéalisée" à base d'états hiérarchiques ne fait qu'approximer le comportement de l'implémentation vectorielle du programme. Ainsi, les approximations et simplifications que nous faisons en nous limitant à des schémas non-interprétés (ceci dans un but de clarté mais aussi afin de pouvoir obtenir des résultats de décidabilité) sont compatibles avec celles qu'introduit le modèle à parallélisme non-borné.

Finalement, n'importe quelle propriété "φ compatible avec $\sqsubseteq_d$" (voir la définition 11.2.2) peut être significativement établie sur le modèle non-interprété $\mathcal{M}_G$, où nous pouvons appliquer les méthodes d'analyse développées dans la deuxième partie de cette thèse: elle sera alors vérifiée par le "vrai" comportement $\mathcal{P}_G^I$.

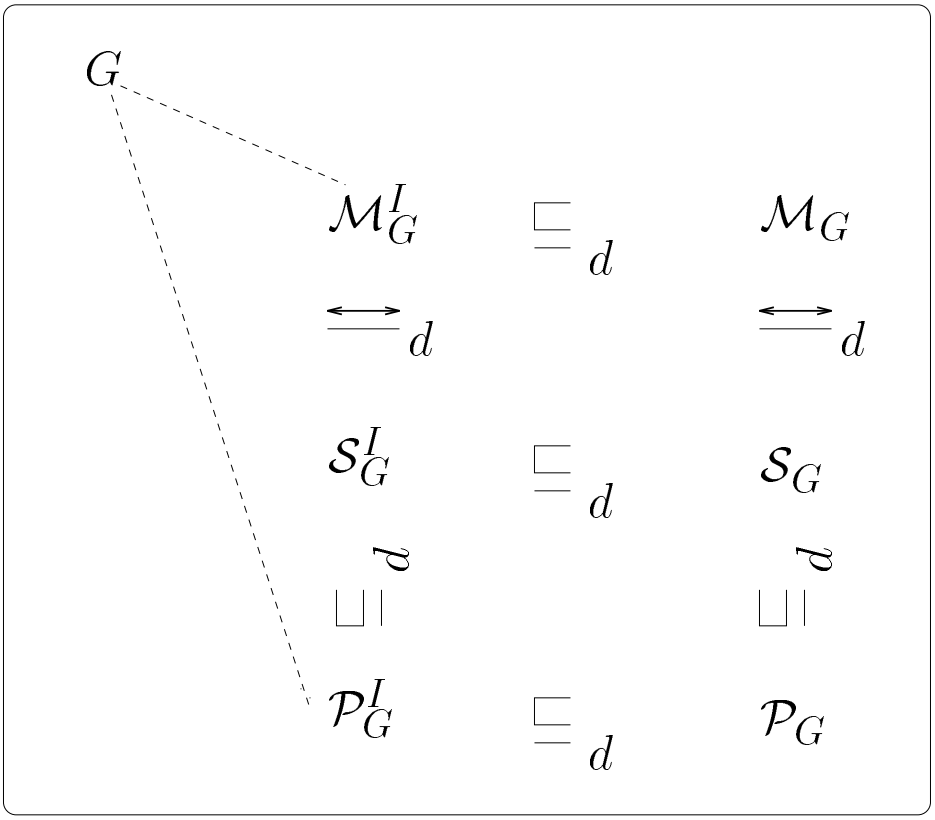


Figure 11.1 : Comparaison des modèles formels

Définition 11.2.2. *φ est compatible avec $\sqsubseteq_d$ ssi $\mathcal{P} \sqsubseteq_d \mathcal{P}'$ et $\mathcal{P}' \models \varphi$ entraînent $\mathcal{P} \models \varphi$.*

Il existe de nombreuses propriétés compatibles avec $\sqsubseteq_d$, et par exemple:

Proposition 11.2.3. *1. Toutes les propriétés de sûreté sont compatibles avec $\sqsubseteq_d$.*

2. La terminaison est compatible avec $\sqsubseteq_d$.

Les limitations de notre notion de correction $\sqsubseteq_d$ de notre cadre d'analyse sont:

- Les propriétés qui ne sont pas compatibles avec $\sqsubseteq_d$ (par exemple, la "terminaison possible") sont seulement significatives si nous voulons faire l'analyse du modèle non-interprété sans obtenir aucune information sur le modèle interprété.

- Les propriétés qui sont compatibles avec $\sqsubseteq_d$ ne peuvent être analysées que dans une direction. Ainsi, si on démontre que le système $\mathcal{M}_G$ ne termine pas, on ne peut pas déduire que $\mathcal{P}_G^I$ ne termine pas.

Toutefois, on pourrait d'énoncer un résultat de *complétude*. La correction que nous avons choisie n'est pas triviale. Le résultat (10.1) nous permet de dire que si le modèle non-interprété vérifie une propriété φ qui est compatible avec notre notion de correction $\sqsubseteq_d$, alors le modèle $\mathcal{P}_G^I$ interprété (c.-à-d. avec des actions de base interprétées) vérifie φ:

$$\forall G \, \forall I \, (\mathcal{M}_G \models \varphi \;\; \Rightarrow \;\; \mathcal{P}_G^I \models \varphi)$$

d'où

$$\forall G \, (\mathcal{M}_G \models \varphi \;\; \Rightarrow \;\; \forall I \, \mathcal{P}_G^I \models \varphi)$$

Maintenant on considère l'ensemble de toutes les interprétations possibles d'un schéma G en admettant le choix non-déterministe pour des actions et des tests. Dans ces circonstances, si le modèle non-interprété $\mathcal{M}_G$ ne vérifie pas une propriété φ compatible avec $\sqsubseteq_d$, alors il existe forcément une interprétation I du schéma G qui ne vérifie pas φ:

$$\forall G \, (\mathcal{M}_G \not\models \varphi \;\; \Rightarrow \;\; \exists I \, \mathcal{P}_G^I \not\models \varphi)$$

et c'est le résultat de complétude de notre méthode de modélisation.

11.3 Conclusion

Pour un schéma G de $RPPS_{\Lambda_\tau}$ nous avons proposé trois sémantiques comportementales différentes:

1. une sémantique d'états *hiérarchiques*, $\mathcal{M}_G$,
2. une sémantique d'états *distribués* (parallélisme non-borné), $\mathcal{S}_G$,
3. une sémantique d'états *vectoriels* (parallélisme borné), $\mathcal{P}_G$.

Ces trois sémantiques correspondent à des visions différentes sur le parallélisme et la synchronisation du langage récursif-parallèle. En fait la première sémantique $\mathcal{M}_G$ correspond au niveau d'abstraction le plus élevé, sans faire références aux détails d'implémentation, et la dernière $\mathcal{P}_G$ au niveau le plus bas (proche de la stratégie d'implémentation).

On a démontré qu'il est possible de relier formellement les trois modèles, et ainsi d'énoncer en quel sens précis l'implémentation de RPC est correcte. Le critère que nous avons utilisé est une notion de τ-simulation préservant à la fois les propriétés de sûreté et la non-terminaison.

D'autre part, on a montré comment raffiner ces trois modèles en incorporant l'interprétation des actions de base. Les modèles obtenus ($\mathcal{M}_G^I$, $\mathcal{S}_G^I$, $\mathcal{P}_G^I$) correspondent là aussi à des niveaux d'abstraction décroissants. Ce fait permet d'affirmer que toute propriété de sûreté ou de terminaison démontrée pour notre modèle $\mathcal{M}_G$ est vraie de la réalisation effective $\mathcal{P}_G^I$. Ainsi, notre étude théorique sur la vérification de propriétés sur $\mathcal{M}_G$ trouve des applications en ce qui concerne l'analyse du comportement des programmes interprétés.

Conclusions

Le parallélisme est un concept majeur en informatique, ceci à la fois d'un point de vue théorique et d'un point de vue pratique.

Les programmes parallèles sont des objets compliqués et aujourd'hui on ne sait pas encore comment profiter pleinement de la puissance des machines parallèles. On ne sait pas encore vraiment raisonner correctement sur les programmes parallèles, et on admet communément qu'il faut développer des modèles, des concepts, des notations, des langages, etc. adaptés à la programmation parallèle.

Dans cette thèse, nous avons proposé un modèle formel pour une classe de programmes parallèles particulière. Une particularité de notre étude réside dans la considération explicite d'une récursivité au niveau de programmes parallèles dans un contexte où la synchronisation n'est possible que de façon limitée. Ces restrictions permettent d'obtenir de bons résultats de décidabilité.

Tout d'abord, nous avons défini une notion formelle d'un schéma de RPPS et de son comportement sous la forme d'un système de transitions étiqueté. La notion fondamentale de notre modèle est celle d'état hiérarchique.

Nous nous sommes ensuite intéressés à l'étude théorique de notre modèle. Dans ce cadre, nous avons étudié ses propriétés de base et nous avons obtenu le résultat d'expressivité permettant de relier le modèle de RPPS avec celui d'algèbre de processus de PA. Ensuite, nous avons montré comment le modèle d'états hiérarchiques, qui n'est pas d'états finis, peut être vu comme un système de transitions bien structuré. Ce modèle n'a pas de lien évident avec d'autres systèmes de transitions bien structurés connus dans la littérature.

Nous avons proposé deux façons différentes de voir notre modèle sous l'angle des systèmes de transitions bien structurés. Cela nous a permis de montrer la décidabilité de certains problèmes d'atteignabilité (par exemple d'atteignabilité d'un état de contrôle) et de terminaison (par exemple le problème de l'arrêt).

Nous avons conclu ce travail par une étude montrant en quoi le mo-

dèle formel développé au début de la thèse et les méthodes d'analyse qui l'accompagnent restent pertinentes dans le cadre réaliste du langage RPC tel qu'il est implémenté actuellement. Ceci nous a amené à en envisager deux modèles d'implémentation: une réalisation distribuée avec parallélisme non-borné et une réalisation à base de files d'attente qui est un modèle avec parallélisme borné. Nous avons montré l'intérêt de notre modèle, en indiquant que toute propriété de sûreté ou de terminaison démontrée pour notre modèle est vraie de la réalisation effective.

De nombreuses voies ont été explorées dans cette thèse, mais plusieurs points nous semblent aujourd'hui mériter une étude complémentaire:

- Il serait sans doute très intéressant de chercher à construire et à étudier une algèbre de processus parallèles basée sur une notion d'invocation de processus parallèle *avec hiérarchisation* et une notion de synchronisation père-fils uni-directionnelle à la RPC. Ceci permettrait de transporter nos résultats dans le cadre plus classique des algèbres de processus, où de nouvelles questions (axiomatisations équationnelles, ...) deviendraient pertinentes.

- La question de la décidabilité de la bisimulation entre schémas RPPS nous semble être un problème ouvert très intéressant, apparemment proche des frontières de ce qui est connu aujourd'hui sur la décidabilité de la bisimulation des systèmes d'états infinis [Hüt91, CH93, CHM94, Mol96, Esp96].

- Les deux points de vue "bien structurés" que nous proposons dans le chapitre 8 nous semblent dignes d'être explorés en tant que tels, indépendamment des applications à l'analyse des schémas RPPS.

Bibliographie

[ACJY96] Abdulla (P. A.), Cerans (K.), Jonsson (B.) & Yih-Kuen (T.). – General decidability theorems for infinite-state systems. *In : Proc. 11th IEEE Symp. Logic in Computer Science, New Brunswick, NJ.*

[AG85] Aleksandrov (V.) & Gorsky (N.). – *Image Representation and Processing.* – Nauka, Leningrad, Russia, 1985.

[AJ93] Abdulla (P. A.) & Jonsson (B.). – Verifying programs with unreliable channels. *In : Proc. 8th IEEE Symp. Logic in Computer Science, Montreal.*

[AK95] Abdulla (P. A.) & Kindahl (M.). – Decidability of simulation and bisimulation between lossy channel systems and finite state systems. *In : Proc. CONCUR'95, LNCS 962.* pp. 333–347. – Springer-Verlag.

[Bak72] Baker (H.). – Petri nets and languages. – CSJ, Memo 68, Project MAC, MIT, 1972.

[BB90] Baeten (J. C. M.) & Bergstra (J. A.). – *Process Algebra with a Zero Object.* – Research Report n° CS-R9028, CWI, juin 1990.

[BBK87a] Baeten (J. C. M.), Bergstra (J. A.) & Klop (J. W.). – Decidability of bisimulation equivalence for processes generating context-free languages. *In : Proc. PARLE'87, vol. II: Parallel Languages, Eindhoven, LNCS 259.* pp. 94–111. – Springer-Verlag.

[BBK87b] Baeten (J. C. M.), Bergstra (J. A.) & Klop (J. W.). – On the consistency of Koomen's fair abstraction rule. *Theoretical Computer Science*, vol. 51, n° 1, 1987, pp. 129–176. – Available as CWI Report CS-R8511.

[BBK87c] Baeten (J. C. M.), Bergstra (J. A.) & Klop (J. W.). – Term rewriting systems with priorities. *In : Proc. Rewriting Techniques and Applications 87, Bordeaux, LNCS 256.* pp. 83–94. – Springer-Verlag.

[BK86] Bergstra (J. A.) & Klop (J. W.). – Algebra of communicating processes. *In : CWI Monographs I, Proc. CWI Symp. Math. and Comp. Sci.*, éd. par de Bakker et al. (J. W.), pp. 89–138. – North-Holland, 1986.

[BK88] Bergstra (J. A.) & Klop (J. W.). – A complete inference system for regular processes with silent moves. *In : Proc. Logic Colloquium '86, Hull, UK*, éd. par Drake (F. R.) & Truss (J. K.). pp. 21–81. – North-Holland.

[BK89] Bergstra (J. A.) & Klop (J. W.). – Process theory based on bisimulation semantics. *In : Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, Noordwijkerhout, LNCS 354.* pp. 50–122. – Springer-Verlag.

[BW90] Baeten (J. C. M.) & Weijland (W. P.). – *Process Algebra.* – Cambridge Univ. Press, 1990, *Cambridge Tracts in Theoretical Computer Science*, volume 18.

[CDG+89] Cardelli (L.), Donahue (J.), Glassman (L.), Jordan (M.), Kalsow (B.) & Nelson (G.). – *Modula-3 Report.* – Rapport technique, Dec Systems Resherch Center, 1989.

[Cer94] Cerans (K.). – Deciding properties of integral relational automata. *In : Proc. 21st ICALP, Jerusalem, LNCS 820.* pp. 35–46. – Springer-Verlag.

[CFP95] Cécé (G.), Finkel (A.) & Purushothaman Iyer (S.). – Unreliable channels are easier to verify than perfect channels. *Information and Computation*, vol. 124, n° 1, janvier 1995, pp. 20–31.

[CH93] Christensen (S.) & Hüttel (H.). – Decidability issues for infinite-state processes - a survey. *EATCS Bull.*, vol. 51, octobre 1993, pp. 156–166.

[CHM93] Christensen (S.), Hirshfeld (Y.) & Moller (F.). – Bisimulation equivalence is decidable for basic parallel processes.

In : Proc. CONCUR'93, Hildesheim, Germany, LNCS 715. pp. 143–157. – Springer-Verlag.

[CHM94] Christensen (S.), Hirshfeld (Y.) & Moller (F.). – Decidable subsets of CCS. *The Computer Journal*, vol. 37, n° 4, 1994, pp. 233–242.

[Chr93] Christensen (S.). – *Decidability and Decomposition in Process Algebras.* – Thèse de PhD, Univ. Edinburgh, septembre 1993.

[CHS92] Christensen (S.), Hüttel (H.) & Stirling (C.). – Bisimulation equivalence is decidable for all context-free processes. *In : Proc. CONCUR'92, Stony Brook, NY, LNCS 630.* pp. 138–147. – Springer-Verlag.

[DH66] Dennis (J. B.) & Horn (E. C. van). – Programming semantics for multiprogrammed computations. *Communications of the ACM*, vol. 9, n° 3, mars 1966, pp. 143–155.

[DH87] De Nicola (R.) & Hennessy (M.). – CCS without τ's. *In : Proc. CAAP'87, Pisa, LNCS 249.* pp. 138–151. – Springer-Verlag.

[DJ90] Dershowitz (N.) & Jouannaud (J.-P.). – Rewrite systems. *In : Handbook of Theoretical Computer Science, vol. B*, éd. par Leeuwen (J. van), chap. 6, pp. 243–320. – Elsevier Science Publishers, 1990.

[Esp96] Esparza (J.). – More infinite results. *In : Proc. Int. Workshop Verification of Infinite State Systems, Pisa*, pp. 4–20.

[FF84] Filman (Robert E.) & Friedman (Daniel P.). – *Coordinated Computing. Tools and Techniques for Distributed Software.* – McGraw-Hill, 1984.

[Fin90] Finkel (A.). – Reduction and covering of infinite reachability trees. *Information and Computation*, vol. 89, n° 2, décembre 1990, pp. 144–179.

[GG89] Glabbeek (R. J. van) & Goltz (U.). – Equivalence notions for concurrent systems and refinement of actions. *In : Proc. Math. Found. Comp. Sci., Porabka-Kozubnik, LNCS 379.* pp. 237–248. – Springer-Verlag.

[Gla90] GLABBEEK (R. J. VAN). – The linear time - branching time spectrum. *In : Proc. CONCUR'90, Amsterdam, LNCS 458.* pp. 278–297. – Springer-Verlag.

[Gla94] GLABBEEK (R. J. VAN). – What is branching time semantics and why to use it ? *EATCS Bull.*, vol. 53, juin 1994, pp. 191–198.

[GRS95] GORRIERI (R.), ROCCETTI (M.) & STANCAMPIANO (E.). – A theory of processes with durational actions. *Theoretical Computer Science*, vol. 140, n° 1, mars 1995, pp. 73–94.

[Hac75] HACK (M.). – Petri net languages. – CSG, Memo 124, Project MAC, MIT, 1975.

[Hen88] HENNESSY (M.). – *Algebraic Theory of Processes.* – MIT Press, 1988.

[Hen95] HENZIGER (T. A.). – Hybrid automata with finite bisimulations. *In : Proc. 22nd ICALP, Szeged, Hungary, LNCS 944.* – Springer-Verlag.

[Hig52] HIGMAN (G.). – Ordering by divisibility in abstract algebras. – Proc. London Math. Soc., 1952.

[HM85] HENNESSY (M.) & MILNER (R.). – Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, vol. 32, n° 1, janvier 1985, pp. 137–161.

[Hoa78] HOARE (C. A. R.). – Communicating sequential processes. *Communications of the ACM*, vol. 21, n° 8, août 1978, pp. 666–677.

[Hoa85] HOARE (C. A. R.). – *Communicating Sequential Processes.* – Prentice Hall Int., 1985.

[Hüt91] HÜTTEL (H.). – *Decidability, Behavioural Equivalences and Infinite Transition Graphs.* – Thèse de PhD, Univ. Edinburgh, 1991.

[IP91] INVERARDI (P.) & PRIAMI (C.). – Evaluation of tools for the analysis of communicating systems. *EATCS Bull.*, vol. 45, octobre 1991, pp. 158–187.

[Jan86] JANTZEN (M.). – Language theory of Petri nets. *In : Petri Nets: Central Models and Their Properties, Bad Honnef, LNCS 254*. pp. 397–412. – Springer-Verlag.

[JP93] JONSSON (B.) & PARROW (J.). – Deciding bisimulation equivalences for a class of non-finite-state programs. *Information and Computation*, vol. 107, n° 2, décembre 1993, pp. 272–302.

[Kel73] KELLER (R. M.). – Parallel program schemata and maximal parallelism. *Journal of the ACM*, vol. 20, n° 3/4, 1973, pp. 514–537, and 696–710.

[Kel76] KELLER (R. M.). – Formal verification of parallel programs. *Communications of the ACM*, vol. 19, n° 7, juillet 1976, pp. 371–384.

[KM69] KARP (R. M.) & MILLER (R. E.). – Parallel program schemata. *Journal of Computer and System Sciences*, vol. 3, n° 2, 1969, pp. 147–195.

[Kot91] KOTOV (V.). – *Program Scheme Theory.* – Nauka, Moscow, 1991.

[Kri89] KRIZ (I.). – Well-quasiordering finite trees with gap-condition. Proof of Harvey Friedman's conjecture. *Annals of Mathematics*, vol. 130, 1989, pp. 215–226.

[Kru60] KRUSKAL (J. B.). – Well-quasi-ordering, the Tree Theorem, and Vazsonyi's conjecture. *Trans. Amer. Math. Soc.*, vol. 95, 1960, pp. 210–225.

[KS96a] KOUCHNARENKO (O.) & SCHNOEBELEN (PH.). – A model for recursive-parallel programs. *In : Proc. Int. Workshop Verification of Infinite State Systems, Pisa*, pp. 127–138.

[KS96b] KOUCHNARENKO (O.) & SCHNOEBELEN (PH.). – Modèles formels pour les programmes récursifs-parallèles. *In : Proc. RENPAR'8, Bordeaux*, pp. 85–88.

[LHG86] LISKOV (B.), HERLIHY (M.) & GILBERT (L.). – Limitations of synchronous communication with static process structure in languages for distributed computing. *In : Proc. 13th ACM Symp. Principles of Programming Languages, St. Petersburg Beach, Florida*, pp. 150–159.

[May83] MAY (D.). – OCCAM. *SIGPLAN Notices*, vol. 13, n° 4, avril 1983.

[Mil80] MILNER (R.). – *A Calculus of Communicating Systems.* – Springer-Verlag, 1980, *Lecture Notes in Computer Science*, volume 92.

[Mil81] MILNER (R.). – A modal characterisation of observable machine-behaviour. *In : Proc. CAAP'81, Genoa, LNCS 112.* pp. 25–34. – Springer-Verlag.

[Mil89] MILNER (R.). – *Communication and Concurrency.* – Prentice Hall Int., 1989.

[Mil90] MILNER (R.). – Operational and algebraic semantics of concurrent processes. *In : Handbook of Theoretical Computer Science, vol. B*, éd. par Leeuwen (J. van), chap. 19, pp. 1201–1242. – Elsevier Science Publishers, 1990.

[Mol96] MOLLER (F.). – Infinite results. *In : Proc. CONCUR'96, Pisa, Italy, LNCS 1119.* pp. 195–216. – Springer-Verlag.

[Mos90] MOSSES (P. D.). – Denotational semantics. *In : Handbook of Theoretical Computer Science, vol. B*, éd. par Leeuwen (J. van), chap. 11, pp. 575–631. – Elsevier Science Publishers, 1990.

[MVVK88] MAMATOV (Y.), VASILCHIKOV (V.), VOLCHENKOV (S.) & KURCHIDIS (V.). – *Multiprocessor Computer System with Dynamic Parallelism.* – Rapport technique n° 7160, Moscow, Russia, VINITI, septembre 1988.

[NPW81] NIELSEN (M.), PLOTKIN (G. D.) & WINSKEL (G.). – Petri nets, event structures and domains, Part I. *Theoretical Computer Science*, vol. 13, n° 1, 1981, pp. 85–108.

[Pan89] PANOV (S.). – *Programming Particularities for Recursive Computer System.* – IPVT RAN, Yaroslavl, Russia, 1989.

[Par81] PARK (D.). – Concurrency and automata on infinite sequences. *In : Proc. 5th GI Conf. on Th. Comp. Sci., LNCS 104.* pp. 167–183. – Springer-Verlag.

[Pet62] PETRI (C. A.). – *Kommunikation mit Automaten.* – Thèse de PhD, Univ. Bonn, 1962. Schriften des Instituts für Instrumentelle Mathematik.

[Pet74] PETERSON (J.). – *Modeling of Parallel Systems.* – Rapport technique n° STAN-CS-74-410, Computer Science Department, Stanford University, 1974.

[Plo81] PLOTKIN (G. D.). – A structural approach to operational semantics. – Lect. Notes, Aarhus University, Aarhus, DK, 1981.

[Pnu85] PNUELI (A.). – Linear and branching structures in the semantics and logics of reactive systems. *In : Proc. 12th ICALP, Nafplion, LNCS 194.* pp. 15–32. – Springer-Verlag.

[Pod91] PODLOVCHENKO (R.). – Recursive programs and hierarchy of their models. *Programming*, no6, 1991, pp. 44–52.

[RT74] RITCHIE (D. M.) & THOMPSON (K.). – The unix timesharing system. *Communications of the ACM*, vol. 17, n° 7, juillet 1974, pp. 365–375.

[Shi85] SHIELDS (M. W.). – Deterministic asynchronous automata. *In : Formal Methods in Programming.* – North-Holland.

[SK95] SOKOLOV (V.) & KOUCHNARENKO (O.). – Analysis of semantic properties of a class of recursive-parallel programs. *In : IV Conf. Application Logic, Irkutsk, Russia*, pp. 72–73.

[Smi96] SMITH (E.). – A survey on high-level Petri-net theory. *EATCS Bull.*, vol. 59, juin 1996, pp. 267–293.

[Sta78] STARKE (P. H.). – Free Petri net languages. *In : Proc. 7th Math. Found. Comp. Sci., Zakopane, Poland, LNCS 64*, pp. 506–515.

[Sto77] STOY (J. E.). – *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* – MIT Press, 1977.

[Tho90] THOMAS (W.). – Infinite trees and automaton definable relations over ω-words. *In : Proc. STACS 90, Rouen, LNCS 415.* – Springer-Verlag.

[VEKM94] VASILCHIKOV (V.), EMIELYN (V.), KURCHIDIS (V.) & MAMATOV (Y.). – *Recursive-parallel programming and work in RPMSHELL.* – IPVT RAN, Iaroslavl, Russia, 1994.

[VKT90] Volchenkov (S.), Krichmara (A.) & Tcivkunov (A.). – Programming particularities for recursive computer system on some problem examples. *In : Computer Systems and their models*, pp. 19–28. – Yaroslavl, Russia, 1990.

[Wir78] Wirth (N.). – *Modula-2.* – Rapport technique n° 36, Institut fur Informatik, ETH, décembre 1978.

[WN92] Winskel (G.) & Nielsen (M.). – Models for concurrency. *In : Handbook of Logic in Computer Science*, éd. par Abramsky (S.), Gabbay (D.) & Maibaum (T.). – Oxford Univ. Press, 1992.

[Wol86] Wolper (P.). – Expressing interesting properties of programs in propositional temporal logic (extended abstract). *In : Proc. 13th ACM Symp. Principles of Programming Languages, St. Petersburg Beach, Florida*, pp. 184–193.

Résumé

Cette thèse s'inscrit dans le cadre des travaux consacrés au développement des modèles sémantiques destinés aux langages de programmation concurrents. Une particularité de notre étude réside dans la considération explicite d'une récursivité au niveau des programmes parallèles. Pour ces programmes nous proposons une sémantique originale, dont nous étudions en détail les propriétés. Bien que le modèle proposé ne soit pas d'états finis, il est possible de le munir d'une structure de systèmes de transitions bien structurés au sens de Finkel ce qui établit la décidabilité de nombreux problèmes de vérification. Cette sémantique à la Plotkin permet de mieux comprendre le comportement des programmes récursifs-parallèles, d'analyser formellement certaines de leurs propriétés, de décrire la stratégie d'implémentation et d'énoncer en quel sens elle est correcte.

Abstract

Semantics of recursive-parallel programs and methods for their analysis.

We define a formal model for a class of recursive-parallel programs with specific invocation and synchronization primitives and study the corresponding program schemes. This model is not finite-state but can still be turned into a well-structured transition systems in the sense of Finkel, so that some interesting reachability problems are clearly decidable. Building on Plotkin's approach, the new semantics of these recursive-parallel program schemes gives a clear understanding of the program behaviour, and a formal model in which to analyze the program properties. In this framework we can study the implementation model and prove its correctness.

Mots clés : sémantique du parallélisme, récursivité, système de transitions, bisimulation, algèbre de processus, implémentation correcte