



**HAL**  
open science

# Sémantique relationnelle des programmes non-déterministes et des processus communicants

Pedro J. V. D Guerreiro

► **To cite this version:**

Pedro J. V. D Guerreiro. Sémantique relationnelle des programmes non-déterministes et des processus communicants. Modélisation et simulation. Université Joseph-Fourier - Grenoble I, 1981. Français. NNT: . tel-00297312

**HAL Id: tel-00297312**

**<https://theses.hal.science/tel-00297312v1>**

Submitted on 15 Jul 2008

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

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

# THESE

TU 2424  
013264

présentée à

## l'Université Scientifique et Médicale de Grenoble

pour obtenir le grade de

DOCTEUR DE 3<sup>ème</sup> CYCLE

Informatique

par

INSTITUT IMAG

Informatique, Mathématiques Appliquées de Grenoble

CNRS - INPG - USMG

Pedro J.V.D. GUERREIRO MÉDIATHÈQUE

B.P. 68

38402 ST-MARTIN-D'HÈRES CED

FRANCE

Tél. (76) 51.46.36



### SEMANTIQUE RELATIONNELLE DES PROGRAMMES NON-DETERMINISTES ET DES PROCESSUS COMMUNICANTS



Thèse soutenue le 3 juillet 1981 devant la Commission d'Examen :

Monsieur	C. DELOBEL	Président
Messieurs	Ph. JORRAND	} Examineurs
	M. NIVAT	
	C. PAIR	
	J. SIFAKIS	



# UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

Monsieur Gabriel CAU : Président

Monsieur Joseph KLEIN : Vice-Président

## MEMBRES DU CORPS ENSEIGNANT DE L'U.S.M.G.

### PROFESSEURS TITULAIRES

MM.	AMBLARD Pierre	Clinique de dermatologie
	ARNAUD Paul	Chimie
	ARVIEU Robert	I.S.N.
	AUBERT Guy	Physique
	AYANT Yves	Physique approfondie
Mme	BARBIER Marie-Jeanne	Electrochimie
MM.	BARBIER Jean-Claude	Physique expérimentale
	BARBIER Reynold	Géologie appliquée
	BARJON Robert	Physique nucléaire
	BARNOUD Fernand	Biosynthèse de la cellulose
	BARRA Jean-René	Statistiques
	BARRIE Joseph	Clinique chirurgicale A
	BEAUDOING André	Clinique de pédiatrie et puériculture
	BELORIZKY Elie	Physique
	BARNARD Alain	Mathématiques pures
Mme	BERTRANDIAS Françoise	Mathématiques pures
MM.	BERTRANDIAS Jean-Paul	Mathématiques pures
	BEZES Henri	Clinique chirurgicale et traumatologie
	BLAMBERT Maurice	Mathématiques pures
	BOLLIET Louis	Informatique (I.U.T. B)
	BONNET Jean-Louis	Clinique ophtalmologie
	BONNET-EYMARD Joseph	Clinique hépato-gastro-entérologie
Mme	BONNIER Marie-Jeanne	Chimie générale
MM.	BOUCHERLE André	Chimie et toxicologie
	BOUCHEZ Robert	Physique nucléaire
	BOUSSARD Jean-Claude	Mathématiques appliquées
	BOUTET DE MONVEL Louis	Mathématiques pures
	BRAVARD Yves	Géographie
	CABANEL Guy	Clinique rhumatologique et hydrologique
	CALAS François	Anatomie
	CARLIER Georges	Biologie végétale
	CARRAZ Gilbert	Biologie animale et pharmacodynamie

.../...



MM.	CAU Gabriel	Médecine légale et toxicologie
	CAUQUIS Georges	Chimie organique
	CHABAUTY Claude	Mathématiques pures
	CHARACHON Robert	Clinique ot-rhino-laryngologique
	CHATEAU Robert	Clinique de neurologie
	CHIBON Pierre	Biologie animale
	COEUR André	Pharmacie chimique et chimie analytique
	COUDERC Pierre	Anatomie pathologique
	DEBELMAS Jacques	Géologie générale
	DEGRANGE Charles	Zoologie
	DELORMAS Pierre	Pneumophtisiologie
	DEPORTES Charles	Chimie minérale
	DESRE Pierre	Métallurgie
	DODU Jacques	Mécanique appliquée (I.U.T. I)
	DOLIQUE Jean-Michel	Physique des plasmas
	DREYFUS Bernard	Thermodynamique
	DUCROS Pierre	Cristallographie
	FONTAINE Jean-Marc	Mathématiques pures
	GAGNAIRE Didier	Chimie physique
	GALVANI Octave	Mathématiques pures
	GASTINEL Noël	Analyse numérique
	GAVEND Michel	Pharmacologie
	GEINDRE Michel	Electroradiologie
	GERBER Robert	Mathématiques pures
	GERMAIN Jean-Pierre	Mécanique
	GIRAUD Pierre	Géologie
	JANIN Bernard	Géographie
	KAHANE André	Physique générale
	KLEIN Joseph	Mathématiques pures
	KOSZUL Jean-Louis	Mathématiques pures
	KRAVTCHENKO Julien	Mécanique
	LACAZE Albert	Thermodynamique
	LACHARME Jean	Biologie végétale
Mme	LAJZEROWICZ Janine	Physique
MM.	LAJZEROWICZ Joseph	Physique
	LATREILLE René	Chirurgie générale
	LATURAZE Jean	Biochimie pharmaceutique
	LAURENT Pierre	Mathématiques appliquées
	LEDRU Jean	Clinique médicale B
	LE ROY Philippe	Mécanique (I.U.T. I)

MM.	LLIBOUTRY Louis	Géophysique
	LOISEAUX Jean-Marie	Sciences nucléaires
	LONGEQUEUE Jean-Pierre	Physique nucléaire
	LOUP Jean	Géographie
Mlle	LUTZ Elisabeth	Mathématiques pures
MM.	MALINAS Yves	Clinique obstétricale
	MARTIN-NOEL Pierre	Clinique cardiologique
	MAYNARD Roger	Physique du solide
	MAZARE Yves	Clinique Médicale A
	MICHEL Robert	Minéralogie et pétrographie
	MICOUD Max	Clinique maladies infectieuses
	MOURIQUAND Claude	Histologie
	MOUSSA André	Chimie nucléaire
	NEGRE Robert	Mécanique
	NOZIERES Philippe	Spectrométrie physique
	OZENDA Paul	Botanique
	PAYAN Jean-Jacques	Mathématiques pures
	PEBAY-PEYROULA Jean-Claude	Physique
	PERRET Jean	Séméiologie médicale (neurologie)
	RASSAT André	Chimie systématique
	RENARD Michel	Thermodynamique
	REVOL Michel	Urologie
	RINALDI Renaud	Physique
	DE ROUGEMONT Jacques	Neuro-Chirurgie
	SARRAZIN Roger	Clinique chirurgicale B
	SEIGNEURIN Raymond	Microbiologie et hygiène
	SENGEL Philippe	Zoologie
	SIBILLE Robert	Construction mécanique (I.U.T. I)
	SOUTIF Michel	Physique générale
	TANCHE Maurice	Physiologie
	VAILLANT François	Zoologie
	VALENTIN Jacques	Physique nucléaire
Mme	VERAIN Alice	Pharmacie galénique
MM.	VERAIN André	Physique biophysique
	VEYRET Paul	Géographie
	VIGNAIS Pierre	Biochimie médicale

**PROFESSEURS ASSOCIES**

MM. CRABBE Pierre  
SUNIER Jules

CERMO  
Physique

**PROFESSEURS SANS CHAIRE**

Mlle	AGNIUS-DELORS Claudine	Physique pharmaceutique
	ALARY Josette	Chimie analytique
MM.	AMBROISE-THOMAS Pierre	Parasitologie
	ARMAND Gilbert	Géographie
	BENZAKEN Claude	Mathématiques appliquées
	BIAREZ Jean-Pierre	Mécanique
	BILLET Jean	Géographie
	BOUCHET Yves	Anatomie
	BRUGEL Lucien	Energétique (I.U.T. I)
	BUISSON René	Physique (I.U.T. I)
	BUTEL Jean	Orthopédie
	COHEN-ADDAD Jean-Pierre	Spectrométrie physique
	COLOMB Maurice	Biochimie médicale
	CONTE René	Physique (I.U.T. I)
	DELOBEL Claude	M.I.A.G.
	DEPASSEL Roger	Mécanique des fluides
	GAUTRON René	Chimie
	GIDON Paul	Géologie et minéralogie
	GLENAT René	Chimie organique
	GROULADE Joseph	Biochimie médicale
	HACQUES Gérard	Calcul numérique
	HOLLARD Daniel	Hématologie
	HUGONOT Robert	Hygiène et médecine préventive
	IDELMAN Simon	Physiologie animale
	JOLY Jean-René	Mathématiques pures
	JULLIEN Pierre	Mathématiques appliquées
Mme	KAHANE Josette	Physique
MM.	KRAKOWIACK Sacha	Mathématiques appliquées
	KUHN Gérard	Physique (I.U.T. I)
	LUU DUC Cuong	Chimie organique - pharmacie
	MICHOULIER Jean	Physique (I.U.T. I)
Mme	MINIER Colette	Physique (I.U.T. I)

MM.	PELMONT Jean	Biochimie
	PERRIAUX Jean-Jacques	Géologie et minéralogie
	PFISTER Jean-Claude	Physique du solide
Mlle	PIERY Yvette	Physiologie animale
MM.	RAYNAUD Hervé	M.I.A.G.
	REBECQ Jacques	Biologie (CUS)
	REYMOND Jean-Charles	Chirurgie générale
	RICHARD Lucien	Biologie végétale
Mme	RINAUDO Marguerite	Chimie macromoléculaire
MM.	SARROT-REYNAULD Jean	Géologie
	SIROT Louis	Chirurgie générale
Mme	SOUTIF Jeanne	Physique générale
MM.	STIEGLITZ Paul	Anesthésiologie
	VIALON Pierre	Géologie
	VAN CUTSEM Bernard	Mathématiques appliquées

#### MAITRES DE CONFERENCES ET MAITRES DE CONFERENCES AGREGES

MM.	ARMAND Yves	Chimie (I.U.T. I)
	BACHELOT Yvan	Endocrinologie
	BARGE Michel	Neuro-chirurgie
	BEGUIN Claude	Chimie organique
Mme	BERIEL Héléne	Pharmacodynamie
MM.	BOST Michel	Pédiatrie
	BOUCHARLAT Jacques	Psychiatrie adultes
Mme	BOUCHE Liane	Mathématiques (CUS)
MM.	BRODEAU François	Mathématiques (I.U.T. B) (Personne étrangère habilitée à être directeur de thèse)
	BERNARD Pierre	Gynécologie
	CHAMBAZ Edmond	Biochimie médicale
	CHAMPETIER Jean	Anatomie et organogénèse
	CHARDON Michel	Géographie
	CHERADAME Hervé	Chimie papetière
	CHIAVERINA Jean	Biologie appliquée (EFP)
	COLIN DE VERDIERE Yves	Mathématiques pures
	CONTAMIN Charles	Chirurgie thoracique et cardio-vasculaire
	CORDONNER Daniel	Néphrologie
	COULOMB Max	Radiologie
	CROUZET Guy	Radiologie

MM.	CYROT Michel	Physique du solide
	DENIS Bernard	Cardiologie
	DOUCE Roland	Physiologie végétale
	DUSSAUD René	Mathématiques (CUS)
Mme	ETERRADOSSI Jacqueline	Physiologie
MM.	FAURE Jacques	Médecine légale
	FAURE Gilbert	Urologie
	GAUTIER Robert	Chirurgie générale
	GIDON Maurice	Géologie
	GROS Yves	Physique (I.U.T. I)
	GUIGNIER Michel	Thérapeutique
	GUITTON Jacques	Chimie
	HICTER Pierre	Chimie
	JALBERT Pierre	Histologie
	JUNIEN-LAVILLAVROY Claude	O.R.L.
	KOLODIE Lucien	Hématologie
	LE NOC Pierre	Bactériologie-virologie
	MACHE Régis	Physiologie végétale
	MAGNIN Robert	Hygiène et médecine préventive
	MALLION Jean-Michel	Médecine du travail
	MARECHAL Jean	Mécanique (I.U.T. I)
	MARTIN-BOUYER Michel	Chimie (CUS)
	MASSOT Christian	Médecine interne
	NEMOZ Alain	Thermodynamique
	NOUGARET Marcel	Automatique (I.U.T. I)
	PARAMELLE Bernard	Pneumologie
	PECCOUD François	Analyse (I.U.T. B) (Personnalité étrangère habilitée à être directeur de thèse)
	PEFFEN René	Métallurgie (I.U.T. I)
	PERRIER Guy	Géophysique-glaciologie
	PHELIP Xavier	Rhumatologie
	RACHALL Michel	Médecine interne
	RACINET Claude	Gynécologie et obstétrique
	RAMBAUD Pierre	Pédiatrie
	RAPHAEL Bernard	Stomatologie
Mme	RENAUDET Jacqueline	Bactériologie (pharmacie)
MM.	ROBERT Jean-Bernard	Chimie-physique
	ROMIER Guy	Mathématiques (I.U.T. B) (Personnalité étrangère habilitée à être directeur de thèse)
	SAKAROVITCH Michel	Mathématiques appliquées

MM.	SCHAERER René	Cancérologie
Mme	SEIGLE-MURANDI Françoise	Cryptogamie
MM.	STOEBNER Pierre	Anatomie pathologie
	STUTZ Pierre	Mécanique
	VROUSOS Constantin	Radiologie

#### MAITRES DE CONFERENCES ASSOCIES

MM.	DEVINE Roderick	Spectro Physique
	KANEKO Akira	Mathématiques pures
	JOHNSON Thomas	Mathématiques appliquées
	RAY Tuhina	Physique

#### MAITRE DE CONFERENCES DELEGUE

M.	ROCHAT Jacques	Hygiène et hydrologie (pharmacie)
----	----------------	-----------------------------------

Fait à Saint Martin d'Hères, novembre 1977



Je tiens à remercier :

Claude DELOBEL, Professeur à l'Université Scientifique et Médicale de Grenoble, qui m'a fait l'honneur de présider le jury de ma thèse ;

Philippe JORRAND, Maître de Recherches au CNRS, qui m'a reçu dans son équipe, tout en m'accordant une grande liberté dans mon travail. Ses conseils ont été précieux et encourageants ;

Maurice NIVAT, Professeur à l'Université de Paris VII, qui avait participé au jury de mon D.E.A. et a bien voulu continuer à s'intéresser à mon travail ;

Claude PAIR, Professeur à l'Institut National Polytechnique de Lorraine, qui m'a fait l'honneur de juger mon travail ;

Joseph SIFAKIS, Chargé de Recherches au CNRS, dont les idées ont été l'inspiration initiale de mes recherches et avec qui j'ai eu de nombreuses discussions très fructueuses.

Je remercie également :

Jacques VOIRON, qui a voulu s'intéresser à ce projet, pour son aide sympathique et ses remarques et propositions toujours pertinentes ;

Mme Madalena QUIRINO, responsable de mon travail à l'Université Nouvelle de Lisbonne, qui m'a encouragé pour entreprendre cette thèse ;

La Fondation Calouste Gulbenkian, qui m'a accordé une bourse d'études pour réaliser ce travail. Je tiens à remercier particulièrement le Service de Bourses d'Etudes de la Fondation et sa Directrice Mme Maria Raquel ALMEIDA DIAS, pour leur sympathie, leur compréhension et leur efficacité.



*Je suis très reconnaissant à Marie-Jo DOREL qui a dactylographié une partie importante du manuscrit, dans des conditions difficiles. Je remercie également Véronique CATRIS et Elisabeth DUBOIS qui ont assuré le reste de la dactylographie avec beaucoup de compétence et de patience. Je remercie enfin le personnel du Service de Reprographie de l'IMAG pour le soin apporté à l'impression de cette thèse.*

*Ce travail a été subventionné par la Fondation Calouste Gulbenkian,  
Lisbonne (bourse de recherche 14/78/B).*



SEMANTIQUE RELATIONNELLE  
DES PROGRAMMES NON-DETERMINISTES  
ET DES PROCESSUS COMMUNICANTS



Chapitre I. Introduction.....	1
Chapitre II. Relations et transformateurs de prédicats.....	7
1. Introduction. Sémantique relationnelle.....	9
2. Relations.....	10
3. Transformateurs de prédicats.....	15
4. Les transformateurs de prédicats $\text{image}[R]$ , $\text{pre}[R]$ et $\tilde{\text{pre}}[R]$ .....	16
5. Opérations sur transformateurs de prédicats.....	21
6. Points fixes de transformateurs de prédicats.....	25
7. Continuité des transformateurs de prédicats $\text{pre}[R]$ et $\tilde{\text{pre}}[R]$ .....	27
8. Conclusion.....	29
Chapitre III. Relations programmables et transformateurs de prédicats de correction totale.....	31
1. Introduction.....	33
2. Transformateur de prédicats de correction totale.....	34
3. La représentation de la non-terminaison.....	36
4. Le transformateur de prédicats $\text{wpr}[R]$ .....	38
5. Les relations programmables.....	46
6. Conclusion.....	52
Chapitre IV. Invariants et trajectoires.....	53
1. Introduction.....	55
2. Le concept d'invariant.....	56
3. Le concept de trajectoire.....	58
4. Invariants sans blocage et terminaison d'exécutions répétitives..	61
5. L'opération de fermeture itérative.....	64
6. Les invariants de la fermeture itérative.....	70
7. Conclusion.....	75

Chapitre V. Sémantique relationnelle du langage des commandes gardées..	77
1. Introduction .....	79
2. Syntaxe .....	80
3. Sémantique .....	81
4. Les transformateurs de prédicats associés aux relations sémantiques .	85
5. Conclusion .....	90
 Chapitre VI. Invariants et correction de programme .....	91
1. Introduction .....	93
2. Utilisation des invariants dans les preuves de correction totale des programmes .....	95
3. Maximum de trois entiers .....	99
4. Algorithme d'Euclide .....	101
5. Dénombrement des nœuds d'un arbre .....	102
6. Conclusion .....	107
 Chapitre VII. Invariants et fonctions définis récursivement .....	109
1. Introduction .....	111
2. Induction structurelle et induction noetherienne .....	112
3. Fonction de Ackermann .....	114
4. Un schéma de transformation .....	117
5. Fonctions définies récursivement sans pile .....	123
6. Conclusion .....	136
 Chapitre VIII. Sémantique relationnelle de processus communicants .....	137
1. Introduction .....	139
2. Syntaxe .....	140
3. Sémantique intuitive .....	144
4. Sémantique a priori .....	147
5. Sémantique de réseaux de processus .....	153

6. Reconstruction d'un processus composé .....	17
7. "Partitionnement d'un ensemble" .....	16
8. Les cinq philosophes .....	16
9. Conclusion .....	17
Chapitre IX. Une expérience de programmation parallèle .....	17
1. Introduction .....	17
2. Structure du Langage .....	17
3. Le problème des télégrammes .....	18
4. Suggestion pour l'implantation .....	18
5. Conclusion .....	20
Chapitre X. Conclusion .....	20
Références .....	20





**CHAPITRE I**

**INTRODUCTION**



L'Informatique couvre aujourd'hui un domaine de connaissances et de recherches assez large dont les frontières sont mal définies. Nous estimons, de toute façon, que les deux problèmes fondamentaux qui caractérisent cette discipline, ou ses deux grands pôles d'intérêt restent les deux questions suivantes :

- i. Comment écrire des programmes.
- ii. Comment construire des machines qui exécutent des programmes.

Conceptuellement, cette deuxième question devrait être subsidiaire de la première. Dans la pratique, on constate que les programmes que l'on écrit généralement sont fortement conditionnés par le type de machines dont on dispose. Une des préoccupations majeures en Programmation (appelons ainsi la partie de l'Informatique qui s'occupe du problème i. ci-dessus), réside précisément dans l'étude de langages de programmation et de techniques d'écriture de programmes, qui soient aussi indépendantes que possible des contraintes matérielles d'implantation.

Cet éloignement entre programme et machine crée le besoin de connaître quel type d'objets mathématiques sont les programmes, et, en particulier, à quoi correspondent les objets formels manipulés par les programmes. C'est-à-dire, on veut savoir exactement ce que l'on décrit quand on écrit un programme. En principe les réponses à ces questions devraient dépendre essentiellement du type de réalité que les programmes sont sensés décrire. Pourtant, ceci n'est pas sans problème car, d'une part la réalité à laquelle on s'intéresse est comprise de manière incomplète, et d'autre part la connaissance des mécanismes de description est imparfaite.

Evidemment, on a tout intérêt à ce que le langage utilisé s'adapte bien au style de description que l'on veut faire. En particulier, il sera d'autant plus utile que les descriptions formelles, i.e., les programmes, permettent d'analyser et d'étudier les propriétés des objets

décrits. Pour que ceci soit possible on doit disposer d'une sémantique du langage, laquelle a pour but de préciser la signification mathématique des constructions utilisées, et par conséquent, des descriptions faites.

Parmi les sémantiques possibles nous préférons celles qui saisissent le mieux notre intuition des objets réels décrits avec le langage. Evidemment, le choix d'une telle sémantique reste en partie subjectif et ne peut être justifié que de manière informelle. Par contre, il faut que la sémantique permette une confrontation formelle des programmes avec d'autres descriptions, plus abstraites, des mêmes objets "réels", en vue de la détermination de la compatibilité du programme vis-à-vis des autres descriptions. Il s'agit ici du problème de la preuve de la correction d'un programme par rapport à une spécification.

Les objets auxquels nous nous intéressons et que nous voulons décrire sont des "processus". Intuitivement, un processus est un mécanisme abstrait susceptible d'entreprendre des actions qui provoquent en lui un changement d'état. Ainsi, dans le contexte de ce rapport un programme est une description formelle d'un système de processus, dans laquelle sont explicitement représentés, d'une manière ou d'une autre, l'état de chaque processus et les actions qu'il peut entreprendre. Dans le cas simple où il y a un seul processus à décrire nous retrouvons la notion classique de programme séquentiel, les actions étant représentées par les commandes et l'état par la configuration des valeurs des variables et du contrôle. Si l'on ne donne pas de restrictions sur le nombre d'actions possibles à chaque état on a affaire à un programme séquentiel non-déterministe.

Un aspect fondamental des programmes séquentiels est leur comportement "entrée-sortie". Dans le cas des programmes déterministes ce comportement est nécessairement fonctionnel, c'est-à-dire, pour chaque donnée le programme produit un résultat, ou ne produit aucun résultat. Cette absence de résultat correspond aux cas où l'exécution du programme ne se termine pas. Pour les programmes non-déterministes le comportement

devient, en général, non fonctionnel, car pour certaines données le programme peut produire plusieurs résultats. Il semble raisonnable d'espérer pouvoir représenter le comportement des programmes non-déterministes à l'aide de relations binaires d'entrée-sortie.

Quand on décrit un système formé par plusieurs processus on obtient ce que l'on appelle un programme parallèle. Ici la situation est plus compliquée. En général, chaque processus ne peut pas être caractérisé uniquement par son comportement entrée-sortie, certaines actions requérant la participation de plusieurs processus. Ces actions sont habituellement appelées "communications".

Ce travail est une réflexion sur les thèmes que nous venons de suggérer. Nous commençons par l'étude du cas séquentiel, en prenant soin de bien modéliser tout d'abord notre intuition sur des systèmes formés d'un seul processus, que nous ne distinguerons pas dans la pratique des programmes séquentiels non-déterministes. Cette modélisation nous permet de donner une nouvelle sémantique pour le langage des commandes gardées de Dijkstra [Dij75]. Ce langage est un langage simple, mais il s'adapte bien à nos besoins de description. Par contre, les langages parallèles existants ne nous satisfont pas complètement. Pour cette raison, nous proposons un formalisme expérimental dont nous sommes capables de fournir une caractérisation sémantique, suggérée d'ailleurs par nos discussions sur les programmes séquentiels.

Ce rapport est organisé de la manière suivante. Dans le chapitre II nous étudions les programmes séquentiels non-déterministes, et la modélisation de leur comportement par des relations binaires "donnée-résultat". Nous développons, à l'aide de ces relations, un certain nombre d'outils mathématiques importants d'un point de vue sémantique et de caractérisation de propriétés de programmes. Dans le chapitre III nous discutons des limitations du modèle présenté et nous proposons une solution dont les incidences sont étudiées en détail. En particulier, les concepts fondamentaux d'invariant et de trajectoire sont introduits

et leur importance est soulignée dans le chapitre IV. Dans le chapitre V nous utilisons les résultats présentés précédemment pour fournir une sémantique relationnelle du langage des commandes gardées. Dans les chapitres VI et VII nous illustrons l'utilisation de cette sémantique dans des preuves de correction de programmes. A ce sujet nous détaillons particulièrement l'étude d'une classe de programmes qui présentent un intérêt particulier : les programmes qui calculent des fonctions définies récursivement. Dans le reste du rapport nous traitons les systèmes de processus. Dans le chapitre VIII, nous proposons un formalisme de description permettant de modéliser agréablement l'intuition suggérée dans cette Introduction. Le problème de la vérification des propriétés des systèmes de processus est également envisagé. Nous traitons, à ce propos, deux petits exemples classiques. Finalement, dans le chapitre IX nous examinons comment transformer le formalisme de description en un véritable langage de programmation. Nous illustrons nos suggestions à l'aide d'un exemple de programmation parallèle.

Le travail décrit ici vient en continuation de notre rapport de D.E.A [Gue79], soutenu en Juin 1979. Une partie des premiers chapitres est une réélaboration d'idées qui y sont présentées et dont l'inspiration initiale était la thèse de Sifakis [Sif79a].

## CHAPITRE II

### RELATIONS ET TRANSFORMATEURS DE PREDICATS





## 1. INTRODUCTION. SEMANTIQUE RELATIONNELLE

L'idée d'étudier les programmes et leurs propriétés à travers des relations caractérisant d'une certaine manière leur comportement n'est pas nouvelle. On peut la trouver notamment chez Mazurkiewicz [Maz74], de Roever [Roe76], Blikle [Bli77] et Schmidt [Sch79]. Les fondements du calcul relationnel sous-jacents à ces travaux sont beaucoup plus anciens ; citons par exemple Tarski [Tar41]. Très récemment un livre écrit par Sanderson [San80] est totalement consacré à l'étude de la Théorie Relationnelle de la Programmation.

Un modèle relationnel un peu plus élaboré est celui des systèmes de transitions. Un système de transitions n'est autre qu'une famille finie de relations binaires sur un ensemble (d'"états") donné. Ce modèle s'avère très pratique pour la vérification de propriétés de programmes parallèles, Keller [Kel76], pour la caractérisation et l'analyse des propriétés des systèmes asynchrones, Sifakis [Sif79a], ainsi que pour la définition de la sémantique de certaines classes de programmes parallèles, Cousot et Cousot [CoC80a].

Un des aspects les plus importants et les plus intéressants de la théorie relationnelle de la programmation est le fait qu'elle permet d'unifier, en tant qu'outil de travail, plusieurs branches de l'Informatique, comme le remarque Sanderson. Le cas le plus flagrant et le plus connu est celui de la théorie relationnelle des bases de données, Codd [Cod70].

Notre but dans ce premier chapitre est d'introduire les deux concepts fondamentaux qui sont à la base des constructions ultérieures : les relations binaires et les transformateurs de prédicats. Le concept de relation binaire est un concept mathématique élémentaire. En l'incluant nous ne prétendons qu'expliquer les notations utilisées, éclairer l'intuition derrière son usage pour caractériser le comportement des programmes et souligner les problèmes que cet usage soulève. Les transformateurs de prédicats sont un concept un peu plus élaboré. Néanmoins, la manière de les utiliser est, d'un certain point de vue, analogue à celle des relations. A la limite, nous pourrions nous passer

des transformateurs de prédicats et ne travailler qu'avec des relations. Cette approche ne serait pas prudente car les transformateurs de prédicats permettent d'exprimer d'une manière naturelle certaines situations courantes en Programmation qu'il serait maladroit de traduire en termes de relations.

## 2. RELATIONS

### 2.1. Le concept de relation

Le concept de relation est un concept élémentaire de la théorie des ensembles. Par définition, étant donné deux ensembles  $Q$  et  $Q'$ , une relation (binaire) de  $Q$  vers  $Q'$  est un sous-ensemble du produit cartésien  $Q \times Q'$ . Nous nous intéressons particulièrement au cas des relations binaires définies dans un ensemble non-vide, que nous nommons  $Q$  tout au long de ce rapport. Une relation binaire dans  $Q$  est un sous-ensemble de  $Q^2$ . L'ensemble des relations binaires dans  $Q$  est noté  $R$ , i.e.,  $R = 2^{Q^2}$  ( $2^{Q^2}$  est l'ensemble des parties de  $Q^2$ ).

Dans l'ensemble  $R$  on distingue la relation d'identité dans  $Q$ , notée  $I$ ,  $I = \{(a,a) \mid a \in Q\}$ , et on définit plusieurs opérations. Soient  $R, S \in R$  :

i. composition :  $R \circ S = \{(a,b) \mid \exists c (a,c) \in R \wedge (c,b) \in S\}$ .

La composition est associative et non commutative.  $\square$

Pour  $i \in \mathbb{N}$ ,  $R^i$  est la relation binaire définie inductivement par :

$$\begin{aligned} R^0 &= I, \\ R^{i+1} &= R \circ R^i \end{aligned}$$

ii. union :  $R \cup S$ . C'est l'union ensembliste.  $\square$

iii. fermeture réflexive transitive :  $R^* = \bigcup_{i \in \mathbb{N}} R^i$ .  $\square$

iv. inversion :  $R^{-1} = \{(a,b) \mid (b,a) \in R\}$ .  $\square$

Exemple II. 2.1. Soit  $Q = \mathbb{Z}$  et  $R = \{(m,n) \mid n=m+1\}$ . Alors

$$R^i = \{(m,n) \mid n=m+i\}, \quad R^* = \{(m,n) \mid n \geq m\},$$

$$R^{-1} = \{(m,n) \mid n=m-1\}.$$

On définit aussi une relation d'ordre, qui correspond à la relation d'inclusion entre sous-ensembles et que nous notons par le même symbole :  $R \subseteq S$ , ( $R$  inclus dans  $S$ ). Il s'agit d'une relation d'ordre partiel.

## 2.2. Relations en tant que transformateurs d'ensembles

Le concept de relation binaire dans un ensemble est un concept que l'on peut utiliser de plusieurs manières différentes. Ceci est dû au fait qu'il existe plusieurs objets mathématiques plus élaborés isomorphes à l'ensemble de relations  $R$ . On peut choisir parmi eux celui qui s'adapte le mieux à la situation que l'on veut décrire.

Considérons l'ensemble  $Q \rightarrow 2^Q$  des applications (fonctions totales) de  $Q$  dans les parties de  $Q$ . A chaque  $R \in \mathcal{R}$  on peut associer un  $r \in Q \rightarrow 2^Q$  tel que  $r(a) = \{b \mid (a,b) \in R\}$ . De même à chaque  $r \in Q \rightarrow 2^Q$  on peut associer un  $R \in \mathcal{R}$  tel que  $R = \{(a,b) \mid b \in r(a)\}$ . La distinction entre les deux objets isomorphes  $R$  et  $Q \rightarrow 2^Q$  n'étant pas essentielle nous nous permettons d'écrire, par exemple,  $(a,b) \in R$  et  $b \in R(a)$ , laissant au contexte le soin de fournir l'interprétation voulue.

Soit  $I$  un ensemble (d'indices), et  $\{A_i\}_{i \in I}$  une famille de sous-ensembles de  $Q$ . L'union  $\bigcup_{i \in I} A_i$ , définie par  $\bigcup_{i \in I} A_i = \{a \mid \exists i \in I, a \in A_i\}$ , est un sous-ensemble de  $Q$ . Si  $I = \emptyset$ , on a, en particulier  $\bigcup_{i \in \emptyset} A_i = \emptyset$ . (Dans la pratique, si l'ensemble d'indices est arbitraire, on se permet de le laisser implicite, et d'écrire par exemple  $\{A_i\}$  et  $\bigcup A_i$ ).

Soit maintenant  $A \subseteq Q$ . L'image de l'ensemble  $A$  par la relation  $R$ , notée habituellement  $R(A)$ , est définie par  $R(A) = \bigcup_{a \in A} R(a)$ , en prenant en compte les remarques que l'on vient de faire. Dans la partie gauche

de cette égalité, la relation  $R$  peut être considérée comme une application de  $2^Q \rightarrow 2^Q$ . (On pourrait l'appeler ici un "transformateur d'ensembles"). Voici une nouvelle interprétation pour les relations. Cette interprétation est assez riche, comme le montrent les propriétés suivantes, toutes facilement démontrables.

#### Propriétés II.2.2.1.

Soient  $R, S \in \mathcal{R}$  ; soient  $A$  et  $B$  deux sous-ensembles de  $Q$ , et  $\{A_i\}_{i \in I}$  une famille de sous-ensembles de  $Q$ . Nous avons :

1.  $R(\emptyset) = \emptyset$
2.  $R(A \cup B) = R(A) \cup R(B)$
3.  $R(\bigcup_{i \in I} A_i) = \bigcup_{i \in I} R(A_i)$
4.  $A \subseteq B \Rightarrow R(A) \subseteq R(B)$
5.  $R(A \cap B) \subseteq R(A) \cap R(B)$ .  $\square$

Toutes ces propriétés ne sont pas indépendantes. En fait 1 et 2 sont des cas particuliers de 3. 4 est une conséquence de 2 :  $A \subseteq B \Rightarrow A = A \cup B \Rightarrow R(A) = R(A \cup B) \Rightarrow R(A) = R(A) \cup R(B) \Rightarrow R(A) \subseteq R(B)$ . Finalement 5 est conséquence de 4 :  $A \cap B \subseteq A \wedge A \cap B \subseteq B \Rightarrow R(A \cap B) \subseteq R(A) \wedge R(A \cap B) \subseteq R(B) \Rightarrow R(A \cap B) \subseteq R(A) \cap R(B)$ .

Dans le groupe de propriétés suivant, nous utilisons la notation  $I_X$  pour désigner la relation d'identité sur  $X$ ,  $X \subseteq Q$ , i.e.,  $I_X = \{(a, a) \mid a \in X\}$ .

#### Propriétés II.2.2.2.

Pour  $R, S \in \mathcal{R}$  et  $A, X, Y \subseteq Q$ , on a

1.  $(R \circ S)(A) = S(R(A))$
2.  $(R \cup S)(A) = R(A) \cup S(A)$
3.  $(X \times Y)(A) = \underline{\text{si}} A \cap X = \emptyset \underline{\text{alors}} \emptyset \underline{\text{sinon}} Y$
4.  $I_X(A) = A \cap X$

$\square$

Proposition II.2.2.3. Pour  $R, S \in \mathcal{R}$  on a  $R \subseteq S$  ssi  $\forall A \subseteq Q R(A) \subseteq S(A)$ .

Démonstration ( $\Rightarrow$ ) triviale.

( $\Leftarrow$ ) La partie droite implique  $\forall a \in Q \ R(a) \subseteq S(a)$  ; donc, pour tout  $a$ , si  $(a,b) \in R$  alors  $(a,b) \in S$ , i.e.,  $R \subseteq S$ .  $\square$

Pour terminer ce paragraphe, remarquons qu'il existe des transformateurs d'ensembles, i.e., des applications de  $2^Q \rightarrow 2^Q$  qui ne "sont" pas des relations. En fait, on a, formellement :

Proposition II.2.2.4. Pour  $r \in 2^Q \rightarrow 2^Q$  il existe  $R \in \mathcal{R}$ , tel que pour tout  $A \subseteq Q$  on a  $r(A) = R(A)$  ssi pour toute famille  $\{A_i\}$  on a  $r(\cup A_i) = \cup r(A_i)$ .

Démonstration ( $\Rightarrow$ ) triviale par II.2.2.1.3.

( $\Leftarrow$ ) Soit  $R$  définie par  $R = \{(a,b) \mid b \in r(\{a\})\}$ . Alors  $R(a) = r(\{a\})$  et  $R(A) = \cup_{a \in A} R(a) = \cup_{a \in A} r(\{a\}) = r(\cup_{a \in A} \{a\}) = r(A)$ .  $\square$

### 2.3. Représentation du comportement d'un programme par une relation

Habituellement le comportement d'un programme "déterministe" est défini comme une fonction (partielle) d'un ensemble de données vers un ensemble de résultats, ou, plus abstraitement d'un ensemble d'états approprié vers lui-même. En étendant cette idée au cas des programmes "non-déterministes" (i.e., les programmes pour lesquels à partir d'une donnée on peut obtenir plusieurs résultats), il est naturel d'essayer de décrire le comportement de ces programmes par des relations binaires dans un tel ensemble d'états. Pour un programme  $r$  donné, notons  $Q$  l'ensemble d'états de  $r$ . Pour qu'une relation binaire  $R$ , définie dans  $Q$ , représente le comportement de  $r$ , on doit exiger que l'hypothèse de base suivante soit respectée.

Hypothèse. La relation  $R$  représente le comportement du programme  $r$  si  $R$  est telle que  $(a,b) \in R$  si et seulement si il est possible que l'exécution de  $r$  initialisé dans l'état  $a$  se termine dans l'état  $b$ .

Dans ces conditions, on peut dire que le programme  $r$  est représenté par la relation  $R$ .

On peut toutefois considérer d'un point de vue inverse que c'est le programme qui représente la relation. Ceci est d'ailleurs cohérent avec le fait qu'un objet peut avoir plusieurs représentations mais chaque représentation ne doit désigner qu'un seul objet. On sait bien que plusieurs programmes syntaxiquement différents peuvent "réaliser" la même fonction (ou relation). D'autre part, comme le remarque Sanderson [San80] on peut schématiser la démarche de programmation en affirmant que le programmeur conçoit d'abord la fonction et ensuite cherche à la représenter dans un langage particulier. Ce schéma quoique agréable n'est pas sans problème. En fait, les relations que les programmes non-déterministes calculent ne sont pas toujours celles auxquelles on pourrait s'attendre à première vue. Nous abordons cette question dans le chapitre suivant.

Ayant fait allusion au problème, nous l'ignorerons dans la pratique en confondant l'objet et la représentation pourvu que cette ambiguïté ne soit pas gênante. Ainsi, quand nous dirons "exécuter la relation R" ceci doit être compris comme une abréviation de "exécuter le programme r que, ou qui (selon le point de vue) représente la relation R".

Soit un programme r représentant une relation R. Si a est un état initial pour r, l'ensemble d'états finals atteignables par r est  $R(a)$ . Si A est un ensemble d'états initiaux, l'ensemble d'états finals atteignables à partir de A est  $R(A)$ . De même, si b est un état final, l'ensemble des états initiaux à partir desquels b est atteignable est  $\{a \mid (a,b) \in R\} = \{a \mid (b,a) \in R^{-1}\} = R^{-1}(b)$ . Si B est un ensemble d'états finals, l'ensemble d'états initiaux à partir desquels B est atteignable est  $\bigcup_{b \in B} R^{-1}(b) = R^{-1}(B)$ .

### 3. TRANSFORMATEURS DE PREDICATS

Nous pourrions continuer le développement de la théorie à l'aide des concepts de relation et d'ensemble. Cependant, cette attitude deviendrait vite encombrante surtout parce que nous aurons besoin de considérer des transformateurs d'ensembles qui ne satisfont pas les conditions de la proposition II.2.2.4 ; ils ne peuvent donc pas être directement représentés par des relations. Autrement dit, il nous faut une classe de transformateurs d'ensembles plus large que celle fournie par les relations.

Profitions de l'élargissement des opérateurs pour changer aussi le "point de vue" : au lieu de prendre les ensembles d'états eux-mêmes, nous les considérerons par l'intermédiaire des prédicats qui les caractérisent. Nos opérateurs deviennent ainsi des transformateurs de prédicats. Ce changement de point de vue, n'étant pas essentiel, rejoint une pratique courante en Programmation qui consiste à caractériser des situations dans les programmes par des assertions sur les variables et l'état du contrôle. Précisons les notions de prédicat et de transformateur de prédicats.

Un prédicat sur  $Q$  (ou simplement un prédicat si  $Q$  est implicite), est une application de  $Q$  dans l'ensemble de valeurs logiques  $\{\text{vrai}, \text{faux}\}$ , i.e., un élément de  $Q \rightarrow \{\text{vrai}, \text{faux}\}$ .

Soit  $P$  l'ensemble des prédicats sur  $Q$ ,  $P = Q \rightarrow \{\text{vrai}, \text{faux}\}$ . A chaque prédicat  $P$  correspond l'ensemble  $\{q | P(q)\}$ , noté  $\underline{P}$ , et appelé l'ensemble caractéristique de  $P$ . Réciproquement à chaque sous-ensemble  $X$  de  $Q$  correspond le prédicat  $\lambda q. q \in X$ , dont  $X$  est l'ensemble caractéristique. La relation d'inclusion  $\subseteq$  dans  $Q$  se transforme en la relation d'ordre partiel dans  $P$ , notée aussi par  $\subseteq$ , définie par  $A \subseteq B$  si  $\underline{A} \subseteq \underline{B}$ , pour  $A, B \in P$ , i.e.,  $A \subseteq B$  si  $\forall q A(q) \Rightarrow B(q)$ .

Dans  $P$  nous distinguons deux prédicats particuliers notés  $\top$  et  $\perp$  dont les ensembles caractéristiques sont respectivement  $Q$  et  $\emptyset$  :  $\top = \lambda q. \text{vrai}$  (le prédicat "toujours vrai"),  $\perp = \lambda q. \text{faux}$  (le prédicat "toujours faux"). Dans  $P$  on définit aussi les opérations négation



(notée " $\neg$ "), disjonction (notée " $\cup$ ") et conjonction (notée " $\cap$ ") par :  
 $\neg A = \lambda q. \neg(A(q))$ ,  $A \cup B = \lambda q. A(q) \vee B(q)$ ,  $A \cap B = \lambda q. A(q) \wedge B(q)$ . La notation  
 $A \Rightarrow B$  est utilisée comme une abréviation de  $\neg A \cup B$ , ainsi que  $A-B$  pour  
 $A \cap \neg B$ .

Les opérations de disjonction et conjonction se généralisent au  
cas d'une famille de prédicats  $\{P_i\}_{i \in I}$  :  $\bigcup_{i \in I} P_i = \lambda q. \bigvee_{i \in I} P_i(q)$ ,

$\bigcap_{i \in I} P_i = \lambda q. \bigwedge_{i \in I} P_i(q)$ . (Si  $I = \emptyset$  on a  $\bigcup_{i \in \emptyset} P_i = \perp$  et  $\bigcap_{i \in \emptyset} P_i = \top$ ).

La structure  $(P, \subseteq, \neg, \cup, \cap, \top, \perp)$  que l'on vient de définir est un  
treillis booléen. Il est isomorphe du treillis des parties de  $Q$   
 $(2^Q, \subseteq, \neg, \cup, \cap, Q, \emptyset)$ . Par conséquent tout raisonnement fait en termes de  
prédicats peut être facilement traduit en termes d'ensembles, et  
réciproquement. Dans ces conditions,  $P$  étant un prédicat, nous nous  
permettons de dire "un état de  $P$ ", avec le sens "un état de l'ensemble  
 $\underline{P}$ ", ou "un état qui vérifie le prédicat  $P$ ".

Un transformateur de prédicats est une application de  $P \rightarrow P$ . Les  
ensembles d'applications  $P \rightarrow P$  et  $2^Q \rightarrow 2^Q$  sont isomorphes. Ceci implique  
que pour chaque relation  $R$ , il existe un transformateur de prédicats  $F$   
tel que  $F(A) = B$  si  $R(\underline{A}) = \underline{B}$ , (cf. II.2.2.4). Nous  
étudions cette question en détail dans le paragraphe suivant.

#### 4. LES TRANSFORMATEURS DE PRÉDICATS $\text{image}[R]$ , $\text{pre}[R]$ et $\tilde{\text{pre}}[R]$

Nous avons remarqué qu'une relation  $R$  peut être considérée de  
plusieurs manières : comme un ensemble de doublets de  $Q^2$ , comme une  
application de  $Q \rightarrow 2^Q$ , ou comme une application de  $2^Q \rightarrow 2^Q$ . Il suit du  
paragraphe précédent que  $R$  peut être considérée encore comme un  
transformateur de prédicats. Cependant, nous préférons, dans ce cas,  
introduire une nouvelle notation, et définir le transformateur de  
prédicats  $\text{image}[R]$  qui se comporte comme le "transformateur de prédicats"  
 $R$ .

Définition II.4.1.  $\text{image}[R]$ . Pour  $R \in \mathcal{R}$  on définit  $\text{image}[R] \in \mathcal{P} \rightarrow \mathcal{P}$  par

$$\text{image}[R](A) = \lambda b. \exists a A(a) \wedge (a,b) \in R. \quad \square$$

La proposition suivante montre que  $\text{image}[R]$  joue le même rôle pour les prédicats que  $R$  joue pour les ensembles.

Proposition II.4.2.  $\text{image}[R](A) = B$  ssi  $R(A) = \underline{B}$ .

Démonstration. Il suffit de remarquer que la déf. II.4.1 est équivalente à  $\text{image}[R](A) = \lambda b. b \in R(A)$ .  $\square$

Dans  $\text{image}[R]$  on peut considérer que le symbole  $\text{image}$  désigne une application de  $\mathcal{R} \rightarrow (\mathcal{P} \rightarrow \mathcal{P})$  définie par  $\text{image} = \lambda R \lambda A \lambda b. \exists a A(a) \wedge (a,b) \in R$ .

Les propriétés du transformateur de prédicats  $\text{image}[R]$  sont, naturellement, analogues aux propriétés II.2.2.1 et II.2.2.2.

Propriétés II.4.3. Pour  $R \in \mathcal{R}$ ,  $A, B \in \mathcal{P}$ , et famille  $\{A_i\}_{i \in I}$  de prédicats de  $\mathcal{P}$ , on a

1.  $\text{image}[R](\perp) = \perp$
2.  $\text{image}[R](A \cup B) = \text{image}[R](A) \cup \text{image}[R](B)$
3.  $\text{image}[R](\bigcup_{i \in I} A_i) = \bigcup_{i \in I} \text{image}[R](A_i)$
4.  $A \subseteq B \Rightarrow \text{image}[R](A) \subseteq \text{image}[R](B)$
5.  $\text{image}[R](A \cap B) \subseteq \text{image}[R](A) \cap \text{image}[R](B)$ .  $\square$

Dans la suite, la relation d'identité dans l'ensemble caractéristique d'un prédicat  $P$  est notée  $I_P$ , i.e., par définition  $I_P = \underline{I_P}$ .

Propriétés II.4.4. Pour  $R, S \in \mathcal{R}$  et  $A, X, Y \in \mathcal{P}$ , on a

1.  $\text{image}[R \circ S](A) = \text{image}[S](\text{image}[R](A))$
2.  $\text{image}[R \cup S](A) = \text{image}[R](A) \cup \text{image}[S](A)$
3.  $\text{image}[X \times Y](A) = (\lambda b. A \cap X \neq \perp) \cap Y$
4.  $\text{image}[I_X](A) = A \cap X$

En Programmation on préfère en général les transformateurs de prédicats "arrière", qui transforment les post-conditions établies pour un programme en pré-conditions garantissant un certain comportement

du programme. Nous sommes ainsi amenés à considérer le transformateur de prédicats  $\text{image}[R^{-1}]$ , que nous nommons  $\text{pre}[R]$ . En d'autres termes, nous avons la définition :

Définition II.4.5.  $\text{pre}[R]$ . Pour  $R \in \mathcal{R}$  on définit  $\text{pre}[R] \in \mathcal{P} \rightarrow \mathcal{P}$  par  

$$\text{pre}[R](B) = \text{image}[R^{-1}](B). \quad \square$$

Les propositions et propriétés suivantes sont des conséquences directes de la définition de  $\text{pre}[R]$ .

Proposition II.4.6.  $\text{pre}[R](B) = A$  ssi  $R^{-1}(B) = A$   $\square$

Proposition II.4.7.  $\text{pre}[R](B) = \lambda a. \exists b (a, b) \in R \wedge B(b)$   $\square$

Propriétés II.4.8. Pour  $R \in \mathcal{R}$ ,  $A, B \in \mathcal{P}$  et famille  $\{A_i\}_{i \in I}$  de prédicats de  $\mathcal{P}$ , on a

1.  $\text{pre}[R](\perp) = \perp$
2.  $\text{pre}[R](A \cup B) = \text{pre}[R](A) \cup \text{pre}[R](B)$
3.  $\text{pre}[R](\bigcup_{i \in I} A_i) = \bigcup_{i \in I} \text{pre}[R](A_i)$
4.  $A \subseteq B \Rightarrow \text{pre}[R](A) \subseteq \text{pre}[R](B)$
5.  $\text{pre}[R](A \cap B) \subseteq \text{pre}[R](A) \cap \text{pre}[R](B)$   $\square$

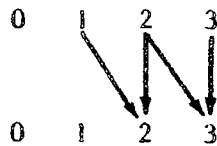
Propriétés II.4.9. Pour  $R, S \in \mathcal{R}$ ,  $B, X, Y \in \mathcal{P}$  on a

1.  $\text{pre}[R \circ S](B) = \text{pre}[R](\text{pre}[S](B))$
2.  $\text{pre}[R \cup S](B) = \text{pre}[R](B) \cup \text{pre}[S](B)$
3.  $\text{pre}[\underline{X} \times \underline{Y}](B) = (\lambda b. \{B \cap Y \neq \perp\}) \cap X$
4.  $\text{pre}[\underline{I}_X](B) = B \cap X$   $\square$

Si la relation  $R$  représente un programme, les significations de  $\text{image}[R](A)$  et de  $\text{pre}[R](B)$  sont respectivement analogues à celles de  $R(A)$  et  $R^{-1}(B)$ . Si  $A$  est le prédicat caractérisant l'ensemble d'états initiaux de  $R$ , alors  $\text{image}[R](A)$  caractérise l'ensemble d'états finals possibles. Ceci implique que si  $R$  se termine alors ce sera nécessairement dans un état de  $\text{image}[R](A)$

Si  $B$  caractérise un ensemble d'états finals alors  $\text{pre}[R](B)$  est le prédicat caractérisant l'ensemble des états à partir desquels il est possible que le programme se termine dans un état de  $B$ .

Exemple II.4.1. Soit  $Q = \{0, 1, 2, 3\}$  et  $R = \{(1, 2), (2, 2), (2, 3), (3, 3)\}$ .  
Graphiquement :



Soit  $q$  la variable qui parcourt  $Q$ . En utilisant une écriture informelle des prédicats<sup>(1)</sup> (comme nous le ferons souvent dans la suite), nous avons par exemple :

$$\text{image}[R](\tau) = q \geq 2, \quad \text{image}[R](q=0) = \perp,$$

$$\text{image}[R](q \neq 0) = \text{image}[R](q \geq 2) = q \geq 2 ;$$

$$\text{pre}[R](\tau) = \{q \neq 0\}^{(1)}, \quad \text{pre}[R](q=0) = \text{pre}[R](q \leq 1) = \perp$$

$$\text{pre}[R](q=3) = q \geq 2. \quad \square$$

On peut observer, au sujet de l'interprétation de  $\text{pre}[R]$ , qu'il est généralement préférable de caractériser l'ensemble des états à partir desquels il est certain (et non seulement possible) que le programme termine dans un état vérifiant une post-condition donnée. Nous reviendrons sur ce point plus tard. Pour le moment, nous nous contentons de remarquer que si le programme est initialisé en dehors de  $\text{pre}[R](B)$ , alors il est sûr qu'il ne se termine pas dans  $B$ .

La double négation implicite dans la phrase précédente suggère un troisième transformateur de prédicats, associé à  $R$ , défini par :

Définition II.4.10.  $\tilde{\text{pre}}[R]$ . Pour  $R \in \mathcal{R}$  on définit  $\tilde{\text{pre}}[R] \in \mathcal{P} \rightarrow \mathcal{P}$  par

$$\tilde{\text{pre}}[R](P) = \neg \text{pre}[R](\neg P) \quad \square$$

Le transformateur de prédicats  $\tilde{\text{pre}}[R]$  est le dual de  $\text{pre}[R]$ . Ses propriétés sont, par conséquent, duales de celles de  $\text{pre}[R]$ .

(1) En cas d'ambiguïté dans la lecture d'expressions contenant des prédicats écrits à l'aide des signes  $=$ ,  $\neq$ , on entourera les expressions définissant les prédicats par  $\{ \}$ .

Proposition II.4.11.  $\tilde{\text{pre}}[R](B)=A$  ssi  $R^{-1}(\neg B) = \neg A$   $\square$

Proposition II.4.12.  $\tilde{\text{pre}}[R](B) = \lambda a. \forall b (a, b) \in R \Rightarrow B(b)$   $\square$

Proposition II.4.13.  $\tilde{\text{pre}}[R](B) = \lambda a. R(a) \subseteq B$

Démonstration. Utiliser II.4.12.  $\square$

Propriétés II.4.14. Pour  $R \in \mathcal{R}$ ,  $A, B \in \mathcal{P}$ , et  $\{A_i\}_{i \in I}$  une famille de prédicats de  $\mathcal{P}$  on a

1.  $\tilde{\text{pre}}[R](\tau) = \tau$
2.  $\tilde{\text{pre}}[R](A \cap B) = \tilde{\text{pre}}[R](A) \cap \tilde{\text{pre}}[R](B)$
3.  $\tilde{\text{pre}}[R](\bigcap_{i \in I} A_i) = \bigcap_{i \in I} \tilde{\text{pre}}[R](A_i)$
4.  $A \subseteq B \Rightarrow \tilde{\text{pre}}[R](A) \subseteq \tilde{\text{pre}}[R](B)$
5.  $\tilde{\text{pre}}[R](A) \cup \tilde{\text{pre}}[R](B) \subseteq \tilde{\text{pre}}[R](A \cup B)$   $\square$

Propriétés II.4.15. Pour  $R, S \in \mathcal{R}$ ,  $B, X, Y \in \mathcal{P}$  on a

1.  $\tilde{\text{pre}}[R \circ S](B) = \tilde{\text{pre}}[R](\tilde{\text{pre}}[S](B))$
2.  $\tilde{\text{pre}}[R \cup S](B) = \tilde{\text{pre}}[R](B) \cap \tilde{\text{pre}}[S](B)$
3.  $\tilde{\text{pre}}[X \times Y](B) = (\lambda b. Y \subseteq B) \cup \neg X$
4.  $\tilde{\text{pre}}[\text{I}_X](B) = X \Rightarrow B$   $\square$

En nous reportant à l'interprétation de  $\text{pre}[R]$ , il est clair que  $\tilde{\text{pre}}[R](B)$  caractérise l'ensemble d'états initiaux à partir desquels il est garanti que  $R$  se termine dans  $B$  ou ne se termine pas.

Exemple II.4.2. Avec la relation  $R$  de l'exemple II.4.1 on a, par exemple,  $\tilde{\text{pre}}[R](\perp) = \{q=0\}$ ,  $\tilde{\text{pre}}[R](q \geq 2) = \tau$ ,  $\tilde{\text{pre}}[R](q=3) = \{q=0 \vee q=3\}$ ,  $\tilde{\text{pre}}[R](q \leq 1) = \{q=0\}$ .  $\square$

La propriété suivante de la conjonction de  $\text{pre}[R](P)$  et  $\tilde{\text{pre}}[R](P)$  aura des conséquences importantes dans la suite.

- Propriété II.4.16. 1.  $\text{pre}[R](P) \cap \tilde{\text{pre}}[R](P) = \text{pre}[R](\tau) \cap \tilde{\text{pre}}[R](P)$   
 2.  $\text{pre}[R](P) \cup \tilde{\text{pre}}[R](P) = \tilde{\text{pre}}[R](\perp) \cup \text{pre}[R](P)$

Démonstration

1.  $\text{PD} = \text{pre}[R](P \cup \neg P) \cap \tilde{\text{pre}}[R](P)$   
 $= (\text{pre}[R](P) \cup \text{pre}[R](\neg P)) \cap \tilde{\text{pre}}[R](P)$  (par II.4.8.2)  
 $= \text{pre}[R](P) \cap \tilde{\text{pre}}[R](P) \cup \text{pre}[R](\neg P) \cap \tilde{\text{pre}}[R](P)$   
 $= \text{PG} \cup \perp = \text{PG}$

B.P. 68

2. par dualité.  $\square$

Les trois résultats suivants, dont la démonstration est analogue à celle de II.2.2.4, donnent des caractérisations intéressantes des transformateurs de prédicats que nous venons de présenter.

Proposition II.4.17. Un transformateur de prédicats  $F, F \in P \rightarrow P$ , est tel que, pour toute famille de prédicats  $\{A_i\}, F(\cup A_i) = \cup F(A_i)$  ssi il existe une relation  $R, R \in R$  telle que  $F = \text{image}[R]$ .  $\square$

Proposition II.4.18. Un transformateur de prédicats  $F, F \in P \rightarrow P$ , est tel que, pour toute famille de prédicats  $\{A_i\}, F(\cup A_i) = \cup F(A_i)$  ssi il existe une relation  $R, R \in R$ , telle que  $F = \text{pre}[R]$ .  $\square$

Proposition II.4.19. Un transformateur de prédicats  $G, G \in P \rightarrow P$ , est tel que, pour toute famille de prédicats  $\{B_i\}, G(\cap B_i) = \cap G(B_i)$  ssi il existe une relation  $R, R \in R$ , telle que  $G = \tilde{\text{pre}}[R]$ .  $\square$

## 5. OPERATIONS SUR LES TRANSFORMATEURS DE PREDICATS

Les transformateurs de prédicats étant des fonctions de  $P \rightarrow P$  on peut bien sûr les composer pour obtenir d'autres transformateurs de prédicats. Pour  $F, G \in P \rightarrow P$  la composition est définie, comme habituellement, par  $F \circ G = \lambda P. F(G(P))$ . De plus,  $F \circ F$  peut s'écrire  $F^2$ . En général, nous avons  $F^0 = I$ , où  $I$  est le transformateur de prédicats identité,  $I = \lambda P. P$ , et  $F^{i+1} = F \circ F^i$ .

Nous définissons encore deux autres opérations binaires entre transformateurs de prédicats, la somme (notée " $\vee$ ") et le produit (noté " $\wedge$ "), par

Définition II.5.1. Somme de transformateurs de prédicats,  $\vee$ .

Pour  $F, G \in P \rightarrow P$ ,  $F \vee G \in P \rightarrow P$  et  $F \vee G = \lambda P. F(P) \cup G(P)$ .  $\square$

Définition II.5.2. Produit de transformateurs de prédicats,  $\wedge$ .

Pour  $F, G \in P \rightarrow P$ ,  $F \wedge G \in P \rightarrow P$  et  $F \wedge G = \lambda P. F(P) \cap G(P)$ .  $\square$

Ces définitions s'étendent facilement au cas d'une famille  $\{F_i\}_{i \in I}$  de transformateurs de prédicats

$$\bigvee_{i \in I} F_i = \lambda P. \bigcup_{i \in I} F_i(P)$$

$$\bigwedge_{i \in I} F_i = \lambda P. \bigcap_{i \in I} F_i(P)$$

Un cas particulier important d'utilisation de ces opérations est celui où la famille considérée est la famille des puissances d'un transformateur de prédicats  $F$ , c'est-à-dire  $\{F^i\}_{i \in \mathbb{N}}$ . Nous les distinguons par deux opérations unaires  $*$  et  $\times$  définies par :

Définition II.5.3. L'opération  $*$ . Pour  $F \in \mathcal{P} \rightarrow \mathcal{P}$ ,  $F^* \in \mathcal{P} \rightarrow \mathcal{P}$  et

$$F^* = \bigvee_{i \in \mathbb{N}} F^i. \quad \square$$

Définition II.5.4. L'opération  $\times$ . Pour  $F \in \mathcal{P} \rightarrow \mathcal{P}$ ,  $F^\times \in \mathcal{P} \rightarrow \mathcal{P}$  et

$$F^\times = \bigwedge_{i \in \mathbb{N}} F^i. \quad \square$$

Une autre opération unaire sur transformateurs de prédicats que nous avons déjà eu l'occasion de rencontrer, à propos de  $\tilde{\text{pre}}[R]$ , consiste à prendre le dual d'un transformateur de prédicats donné.

Définition II.5.5 ("Prendre le dual")  $\sim$ . Pour  $F \in \mathcal{P} \rightarrow \mathcal{P}$ ,  $\tilde{F} \in \mathcal{P} \rightarrow \mathcal{P}$  et  $\tilde{F} = \lambda P. \neg F(\neg P)$ .  $\square$

Les relations entre l'opération  $\sim$  et les autres opérations sont données par les égalités suivantes, toutes faciles à démontrer.

- i.  $\tilde{\tilde{I}} = I$
- ii.  $\tilde{\tilde{F}} = F$
- iii.  $(F \circ G)^\sim = \tilde{F} \circ \tilde{G}$
- iv.  $(F \vee G)^\sim = \tilde{F} \wedge \tilde{G}$
- v.  $(F \wedge G)^\sim = \tilde{F} \vee \tilde{G}$
- vi.  $(F^i)^\sim = (\tilde{F})^i$
- vii.  $(F^\times)^\sim = (\tilde{F})^*$
- viii.  $(F^*)^\sim = (\tilde{F})^\times$

Les opérations que nous venons de présenter nous permettent de réécrire les propriétés II.4.9.1 et 2 II.4.15.1 et 2 de façon plus concise.

Propriétés II.5.6. Pour  $R, S \in \mathcal{R}$ , on a

1.  $\text{pre}[R \circ S] = \text{pre}[R] \circ \text{pre}[S]$
2.  $\text{pre}[R \cup S] = \text{pre}[R] \vee \text{pre}[S]$
3.  $\tilde{\text{pre}}[R \circ S] = \tilde{\text{pre}}[R] \circ \tilde{\text{pre}}[S]$
4.  $\tilde{\text{pre}}[R \cup S] = \tilde{\text{pre}}[R] \wedge \tilde{\text{pre}}[S]$   $\square$

Nous avons aussi les nouvelles propriétés suivantes.

Propriétés II.5.7. Pour  $R \in \mathcal{R}$ , on a

1.  $\text{pre}[R^i] = \text{pre}[R]^i$
2.  $\text{pre}[R^*] = \text{pre}[R]^*$
3.  $\tilde{\text{pre}}[R^i] = \tilde{\text{pre}}[R]^i$
4.  $\tilde{\text{pre}}[R^*] = \tilde{\text{pre}}[R]^*$   $\square$

Dans  $P \rightarrow P$  on peut définir une relation d'ordre partiel, notée aussi " $\underline{\subseteq}$ " par :

Définition II.5.8. Relation d'ordre partiel pour transformateurs de prédicats,  $\underline{\subseteq}$ . Pour  $F, G \in P \rightarrow P$  on a  $F \underline{\subseteq} G$  si  $\forall P \in \mathcal{P} F(P) \subseteq G(P)$ .  $\square$

Propriété II.5.9.  $F \underline{\subseteq} G$  ssi  $\tilde{G} \underline{\subseteq} \tilde{F}$ .  $\square$

Nous pouvons maintenant présenter l'analogie de la proposition II.2.2.3 en termes de  $\text{pre}[R]$  et de  $\tilde{\text{pre}}[R]$  :

Proposition II.5.10. Pour  $R, S \in \mathcal{R}$  on a  $R \underline{\subseteq} S$  ssi  $\text{pre}[R] \subseteq \text{pre}[S]$  ssi  $\tilde{\text{pre}}[S] \subseteq \tilde{\text{pre}}[R]$ .

Démonstration. Il suffit de remarquer que  $R \underline{\subseteq} S$  ssi  $R^{-1} \underline{\subseteq} S^{-1}$  et utiliser les définitions II.2.2.3 et II.5.9.  $\square$

Corollaire II.5.11.  $\text{pre}[R] = \text{pre}[S]$   
 ssi  $\tilde{\text{pre}}[R] = \tilde{\text{pre}}[S]$   
 ssi  $R=S$ .  $\square$



Ce corollaire montre que  $R$ ,  $\text{pre}[R]$  et  $\tilde{\text{pre}}[R]$  sont tous sémantiquement équivalents. Par conséquent, il n'y a pas perte d'information dans le passage de  $R$  à  $\text{pre}[R]$  ou  $\tilde{\text{pre}}[R]$ .

Dans la suite nous utilisons beaucoup les deux définitions suivantes.

Définition II.5.12. Continuité (supérieure).  $F \in \mathcal{P} \rightarrow \mathcal{P}$  est dit (supérieurement) continu si pour toute suite croissante de prédicats  $\{A_i\}_{i \in \mathbb{N}}$ ,  $A_i \subseteq A_{i+1}$ , on a l'égalité  $F(\bigcup_{i \in \mathbb{N}} A_i) = \bigcup_{i \in \mathbb{N}} F(A_i)$ .  $\square$

Définition II.5.13. Continuité inférieure.  $G \in \mathcal{P} \rightarrow \mathcal{P}$  est dit inférieurement continu si pour toute suite décroissante de prédicats  $\{B_i\}_{i \in \mathbb{N}}$ ,  $B_{i+1} \subseteq B_i$  on a l'égalité  $G(\bigcap_{i \in \mathbb{N}} B_i) = \bigcap_{i \in \mathbb{N}} G(B_i)$ .  $\square$

Les deux résultats suivants sont des conséquences immédiates respectivement des propositions II.4.8.3 et II.4.14.3.

Proposition II.5.14. Pour tout  $R \in \mathcal{R}$ ,  $\text{pre}[R]$  est supérieurement continu.  $\square$

Proposition II.5.15. Pour tout  $R \in \mathcal{R}$ ,  $\tilde{\text{pre}}[R]$  est inférieurement continu.  $\square$

En effet, on a la propriété générale suivante

Propriété II.5.16.  $F \in \mathcal{P} \rightarrow \mathcal{P}$  est supérieurement continu ssi  $\tilde{F}$  est inférieurement continu.

Démonstration ( $\Rightarrow$ ). Soit  $\{B_i\}_{i \in \mathbb{N}}$  une suite décroissante de prédicats. Alors  $\{\neg B_i\}_{i \in \mathbb{N}}$  est une suite croissante. Donc  $F(\bigcup_{i \in \mathbb{N}} \neg B_i) = \bigcup_{i \in \mathbb{N}} F(\neg B_i)$  ce qui implique  $\tilde{F}(\bigcap_{i \in \mathbb{N}} B_i) = \neg F(\bigcup_{i \in \mathbb{N}} \neg B_i) = \neg \bigcup_{i \in \mathbb{N}} F(\neg B_i) = \bigcap_{i \in \mathbb{N}} \neg F(\neg B_i) = \bigcap_{i \in \mathbb{N}} \tilde{F}(B_i)$ . ( $\Leftarrow$ ) similaire.  $\square$

Le transformateur de prédicats identité et les transformateurs de prédicats constant sont, trivialement, supérieurement et inférieurement continus. De plus, nous avons les propriétés générales suivantes, facilement démontrables à partir des définitions.

Propriété II.5.17. Si  $F, G \in \mathcal{P} \rightarrow \mathcal{P}$  sont supérieurement continus alors  $F \cup G$ ,  $F \cap G$ ,  $F \circ G$ , et  $F^*$  le sont aussi.  $\square$

Propriété II.5.18. Si  $F, G \in P \rightarrow P$  sont inférieurement continus alors  $F \cup G$ ,  $F \cap G$ ,  $F \circ G$ , et  $F^{\times}$  le sont aussi.  $\square$

## 6. POINTS FIXES DE TRANSFORMATEUR DE PREDICATS

Un résultat bien connu de la théorie des treillis, Tarski [Tar55], a pour conséquence que si un transformateur de prédicats  $F$  est monotone (i.e.,  $F$  est tel que  $A \subseteq B \Rightarrow F(A) \subseteq F(B)$ ) alors  $F$  admet un plus petit point fixe, c'est-à-dire il existe un prédicat  $A_0$  tel que  $F(A_0) = A_0$  et pour tout  $A$  tel que  $F(A) = A$  on a  $A_0 \subseteq A$ . De plus, pour chaque  $P$  tel que  $P \subseteq F(P)$  il existe le plus petit point fixe de  $F$  qui est supérieur à  $P$ . Si  $F$  est supérieurement continu alors ce plus petit point fixe est précisément  $F^*(P)$ , (def.II.5.3). Et, en particulier, le plus petit point fixe de  $F$  est  $F^*(\perp)$ .

Par dualité, nous pouvons conclure que  $F$  admet aussi un plus grand point fixe et que pour chaque  $P$  tel  $F(P) \subseteq P$ , il existe le plus grand point fixe de  $F$  qui est inférieur à  $P$ . Finalement, si  $F$  est inférieurement continu ce point fixe est  $F^{\times}(P)$ , (def.II.5.4). Dans ces conditions, le plus grand point fixe de  $F$  est  $F^{\times}(\top)$ .

Nous allons être amenés à résoudre des inéquations de la forme  $P \subseteq F(P)$ , où  $F$  est un transformateur de prédicats inférieurement continu. La résolution de cette inéquation est en fait équivalente à la résolution d'une équation car

$$\begin{aligned} P \subseteq F(P) &\Leftrightarrow P = P \cap F(P) \\ &\Leftrightarrow P = (I \wedge F)(P). \end{aligned}$$

Le transformateur de prédicats  $I \wedge F$  étant tel que  $(I \wedge F)(A) \subseteq A$ , pour tout  $A$ , et clairement inférieurement continu, on peut conclure que pour tout  $A$  la plus grande solution de  $P \subseteq F(P)$  inférieure à  $A$  est  $(I \wedge F)^{\times}(A)$ .

Le raisonnement dual nous permet de résoudre des inéquations de la forme  $G(P) \subseteq P$ , quand  $G$  est supérieurement continu. Dans ce cas, nous obtenons que pour tout  $A$  la plus petite solution de  $G(P) \subseteq P$  qui est supérieure à  $A$  est  $(I \vee G)^*(A)$ .

Remarquons ici que le théorème du point fixe de Tarski [Tar55] nous indique que la plus petite solution de  $G(P) \subseteq P$  est précisément le plus petit point fixe de  $G$  ; ceci entraîne directement  $(I \vee G)^*(\perp) = G^*(\perp)$ . De manière duale nous avons  $(I \wedge F)^x(\top) = F^x(\top)$ .

La proposition suivante et sa duale permettent parfois de simplifier l'expression de la solution des inéquations.

Proposition II.6.1. Si  $G$  est distributive par rapport à la disjonction alors  $(I \vee G)^* = G^*$ .

Démonstration. Il suffit de montrer que  $\forall i (I \vee G)^i = \bigvee_{j \in [0, i]} G^j$   
 $i=0$  : trivial  
 $i+1$  :  $(I \vee G)^{i+1} = (I \vee G) \circ (I \vee G)^i$   
 $= (I \vee G) \circ \left( \bigvee_{j \in [0, i]} G^j \right)$  (hypothèse d'induction)  
 $= \bigvee_{j \in [0, i]} G^j \vee G \circ \left( \bigvee_{j \in [0, i]} G^j \right)$   
 $= \bigvee_{j \in [0, i]} G^j \vee \bigvee_{j \in [1, i+1]} G^j$  (distributivité)  
 $= \bigvee_{j \in [0, i+1]} G^j \quad \square$

Proposition II.6.2. Si  $F$  est distributive par rapport à la conjonction alors  $(I \wedge F)^x = F^x$ .  $\square$

Nous résumons les résultats concernant la résolution d'inéquations avec transformateurs de prédicats dans les deux propositions suivantes :

Proposition II.6.3. Si  $G \in P \rightarrow P$  est supérieurement continu et  $A \in P$  alors la plus petite solution de l'inéquation  $G(P) \subseteq P$  qui est supérieure à  $A$  est le prédicat  $(I \vee G)^*(A)$ . De plus, si  $G$  est distributif par rapport à la disjonction cette solution est aussi donnée par  $G^*(A)$ .  $\square$

Proposition II.6.4. Si  $F \in P \rightarrow P$  est inférieurement continu et  $A \in P$  alors la plus grande solution de l'inéquation  $P \subseteq F(P)$  qui est inférieure à  $A$  est le prédicat  $(I \wedge F)^x(A)$ . De plus, si  $F$  est distributif par rapport à la conjonction cette solution est aussi donnée par  $F^x(A)$ .  $\square$

## 7. CONTINUITÉ DE $\text{pre}[R]$ ET $\tilde{\text{pre}}[R]$

Nous revenons à l'étude des transformateurs de prédicats  $\text{pre}[R]$  et  $\tilde{\text{pre}}[R]$ , maintenant en ce qui concerne leurs propriétés de continuité. Les propositions II.5.14 et II.5.15 nous indiquent que  $\text{pre}[R]$  est toujours supérieurement continu et que  $\tilde{\text{pre}}[R]$  est toujours inférieurement continu. Pour caractériser la continuité inférieure de  $\text{pre}[R]$  et supérieure de  $\tilde{\text{pre}}[R]$  il nous faut quelques concepts supplémentaires.

**Définition II.7.1. Déterminisme.** La relation  $R, R \in \mathcal{R}$ , est déterministe si  $\forall_{q \in Q} |R(q)| = 1$ .  $\square$

Si une relation n'est pas déterministe on peut dire qu'elle est non-déterministe. Il est cependant plus courant de dire que le non-déterminisme est l'absence de la condition de déterminisme et non la négation. Dans ce sens toutes les relations sont non-déterministes.

Une classe intéressante de relations est celle des relations pour lesquelles chaque élément a un nombre fini de successeurs. Nous disons que ces relations sont à non-déterminisme fini. (On les appelle aussi relations à image finie).

**Définition II.7.2. Non-déterminisme fini.** La relation  $R$  est à non-déterminisme fini si  $\forall_{q \in Q} \exists_{k \in \mathbb{N}} |R(q)| \leq k$ .  $\square$

**Exemple II.7.1.** Les relations  $\geq$  dans  $\mathbb{N}$ , ainsi que

$\{(x, y) \in \mathbb{N}^2 \mid x \text{ est un multiple de } y\}$   
et  $\{(x, y) \in \mathbb{Z}^2 \mid |x - y| \leq 1\}$  sont à non-déterminisme fini.

Les relations  $\leq$  dans  $\mathbb{N}$ ,  $\geq$  et  $\leq$  dans  $\mathbb{Z}$  ainsi que  $\{(x, y) \in \mathbb{N}^2 \mid x \text{ est un diviseur de } y\}$  ne sont pas à non-déterminisme fini.  $\square$

La continuité inférieure de  $\text{pre}[R]$  étant équivalente à la continuité supérieure de  $\tilde{\text{pre}}[R]$  (par II.5.16) nous choisissons de traiter d'abord  $\tilde{\text{pre}}[R]$ .

Théorème II.7.3. Le transformateur de prédicats  $\tilde{\text{pre}}[R]$  est supérieurement continu ssi la relation  $R$  est à non-déterminisme fini.

Démonstration. ( $\Leftarrow$ ) Soit  $\{P_i\}$  une suite croissante de prédicats.

Utilisant II.4.13 il nous suffit de montrer que, pour tout  $a$ ,

$$R(a) \subseteq \bigcup_i P_i \Leftrightarrow \bigcap_i R(a) \subseteq P_i.$$

L'implication  $\Leftarrow$  étant évidente, il reste à prouver que

$$(*) \quad R(a) \subseteq \bigcup_i P_i \Rightarrow \bigcap_i R(a) \subseteq P_i$$

Si  $R(a) = \emptyset$  alors  $(*)$  est trivialement vrai. Sinon définissons pour chaque  $b \in R(a)$  l'entier  $s(b) = \min\{i \mid b \in P_i\}$ .

L'ensemble  $\{s(b) \mid b \in R(a)\}$  est fini et non-vide. Il a donc un maximum

$m$ . Les  $P_i$  formant une suite croissante d'ensembles on a certainement

$R(a) \subseteq P_m$ , ce qui établit la partie droite de  $(*)$ .

( $\Rightarrow$ ). Supposons que  $R$  n'est pas à non-déterminisme fini et prenons  $a$  tel que  $R(a)$  est infini. Prenons  $D$  un sous-ensemble dénombrable de  $R(a)$ , et numérotons ses éléments :  $D = \{b_0, b_1, \dots, b_i, \dots\}$ .

Définissons une suite croissante de prédicats  $\{P_i\}$  par

$$P_i(x) = \begin{cases} i > j & \text{si } x \in D \text{ et } x = b_j \\ \text{vrai} & \text{si } x \notin D \end{cases}$$

Puisque  $\bigcup_i P_i = \top$ , nous avons, d'après II.4.14.1, que  $\tilde{\text{pre}}[R](\bigcup_i P_i)(a) = \text{vrai}$ .

D'autre part, il est évident que  $\forall i \ R(a) \not\subseteq P_i$  d'où, d'après II.4.13,

$\forall i \ \tilde{\text{pre}}[R](P_i)(a) = \text{faux}$  et donc  $(\bigcap_i \tilde{\text{pre}}[R](P_i))(a) = (\bigcup_i \tilde{\text{pre}}[R](P_i))(a) = \text{faux}$  ce qui implique que  $\tilde{\text{pre}}[R]$  n'est pas continu.  $\square$

Théorème II.7.4. Le transformateur de prédicats  $\text{pre}[R]$  est inférieurement continu ssi la relation  $R$  est à non-déterminisme fini.

Démonstration. Par II.7.3 et II.5.16.  $\square$

## 8. CONCLUSION

Dans ce chapitre, nous avons introduit les idées de base et les concepts fondamentaux de notre théorie relationnelle, concepts sur lesquels s'appuient les développements ultérieurs. Jusqu'à maintenant, nous n'avons discuté que très brièvement de l'utilité des relations et des transformateurs de prédicats dans le domaine de la Programmation. En particulier, nous n'avons rien dit sur les méthodes pour calculer les relations représentées par des programmes donnés, ni de la façon de calculer en pratique les transformateurs de prédicats associés à ces relations. En fait, ce sujet n'est traité que dans le chapitre V. Il nous faut auparavant raffiner encore notre modèle avec de nouveaux concepts mieux adaptés aux types de problèmes qui se posent dans la réalité. Ainsi le chapitre suivant est consacré à l'étude d'un nouveau transformateur de prédicats, qu'on pourrait appeler "de correction totale", et le chapitre IV à l'introduction de quelques outils, dont les invariants, qui vont permettre l'étude des programmes itératifs.



### CHAPITRE III

## RELATIONS PROGRAMMABLES ET TRANSFORMATEURS DE PREDICATS DE CORRECTION TOTALE





## 1. INTRODUCTION

Les transformateurs de prédicats  $\text{pre}[R]$  et  $\tilde{\text{pre}}[R]$  ne correspondent pas à la notion de transformateurs de prédicats de correction totale. Celle-ci s'applique à des transformateurs de prédicats qui produisent des prédicats (pre-conditions) caractérisant des ensembles d'états initiaux pour un programme, et qui garantissent la terminaison de ce programme dans un état satisfaisant un prédicat (post-condition) donné. Dans ce chapitre, nous partons à la recherche d'un transformateur de prédicats de correction totale bâti à partir des concepts déjà présentés. Nous commençons par mettre en cause l'hypothèse faite dans II.2.3 sur la représentation d'un programme par une relation, puis nous remplaçons cette hypothèse par une autre, plus fine. Le transformateur de prédicats que nous proposons finalement a toutes les propriétés de "salubrité" indiquées par Dijkstra [Dij76]. De plus, comme nous le montrerons dans le chapitre V, il correspond en fait à la fonction  $\text{wp}$  de Dijkstra [Dij75], [Dij76]. L'étude du transformateur de prédicats proposé est faite en termes de relations binaires comme dans le chapitre précédent, ce qui est très avantageux, car outre sa simplicité conceptuelle, il permet de faire abstraction des particularités du langage avec lequel seraient écrits les programmes.

Un résultat important de ce chapitre est que toutes les relations ne sont pas représentables par des programmes. Il serait intéressant de caractériser précisément la classe des relations qui sont représentables par des programmes. Ce problème est évidemment lié au langage de programmation utilisé et n'est pas traité dans ce rapport. Néanmoins nous étudions dans ce chapitre une classe de relations parmi lesquelles on trouvera nécessairement les relations représentables par des programmes.

## 2. TRANSFORMATEURS DE PREDICATS DE CORRECTION TOTALE

Rappelons brièvement les interprétations de  $\text{pre}[R]$  et de  $\tilde{\text{pre}}[R]$  :  $\text{pre}[R](P)$  est le prédicat caractérisant l'ensemble d'états initiaux à partir desquels il est possible que  $R$  se termine dans un état de  $P$  ;  $\tilde{\text{pre}}[R](P)$  caractérise les états initiaux à partir desquels il est certain que  $R$  se termine dans un état de  $P$  ou ne se termine pas. Si l'on dit qu'un transformateur de prédicats  $F$  associé à la relation  $R$  est de correction totale quand, pour tout  $P$ ,  $F(P)$  caractérise un ensemble d'états à partir desquels il est certain que  $R$  se termine dans un état de  $P$ , les interprétations données pour  $\text{pre}[R]$  et  $\tilde{\text{pre}}[R]$  montrent qu'ils ne sont pas des transformateurs de prédicats de correction totale. Remarquons néanmoins que si l'on sait que  $R$  se termine toujours (i.e., pour chaque état initial), alors  $\tilde{\text{pre}}[R]$  est effectivement de correction totale. Malheureusement ce n'est pas toujours le cas car il y a beaucoup de programmes qui peuvent ne pas de terminer.

Considérons un programme dont l'exécution à partir d'un état  $q$  peut ne pas se terminer et supposons que ce comportement est représenté dans la relation  $R$  associée par le fait que  $R(q)=\emptyset$ . Dans cette hypothèse, si l'on considère le transformateur de prédicats  $\text{pre}[R] \wedge \tilde{\text{pre}}[R]$  on peut constater qu'il est de correction totale. Définissons, pour alléger l'écriture,  $\hat{\text{pre}}[R] = \text{pre}[R] \wedge \tilde{\text{pre}}[R]$ . Si, pour un prédicat  $P$  donné, un état  $q$  vérifie  $\hat{\text{pre}}[R](P)$  alors, d'une part  $q$  vérifie  $\text{pre}[R](P)$ , ce qui implique que  $q$  a au moins un successeur et, d'après l'hypothèse faite, que le programme initialisé à  $q$  se termine toujours ; d'autre part  $q$  vérifie aussi  $\tilde{\text{pre}}[R]$  ce qui implique que, le programme devant se terminer, il doit se terminer dans un état de  $P$ . Sous une autre forme le transformateur de prédicats  $\hat{\text{pre}}[R](P)$  a servi à Wand [Wan77] pour donner une caractérisation relationnelle de la fonction  $\text{wp}$  de Dijkstra [Dij76].

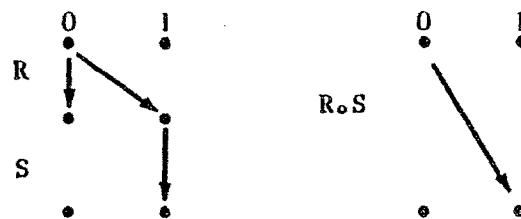
Il faut cependant remarquer que l'hypothèse faite ci-dessus sur la représentation de la non-terminaison contredit celle de II.2.3, à savoir qu'un doublet  $(a,b)$  est dans  $R$  si et seulement si il est possible qu'un calcul commencé à  $a$  se termine dans  $b$ . Il existe des programmes

où, pour un état initial donné, outre la possibilité de non-terminaison, il est aussi possible que le programme se termine. D'après l'hypothèse ci-dessus les calculs qui se terminent ne sont pas représentés. D'autre part, dans l'hypothèse de II.2.3, c'est le calcul sans fin qui n'est pas représenté.

L'issue correcte de ce dilemme est, à notre avis, l'abandon de  $\text{pre}[R] \wedge \tilde{\text{pre}}[R]$  en tant que transformateur de prédicats de correction totale, et le raffinement de l'hypothèse de II.2.3 de façon à permettre la représentation des calculs qui ne se terminent pas.

Une raison supplémentaire pour abandonner  $\hat{\text{pre}}[R]$  est le fait que, en général,  $\hat{\text{pre}}[R \circ S] \neq \hat{\text{pre}}[R] \circ \hat{\text{pre}}[S]$ . (Ceci est un peu surprenant, au vu de II.5.6.1. et II.5.6.3.).

Exemple III.2.1. Soit  $Q = \{0,1\}$ ,  $R = \{(0,0), (0,1)\}$ ,  $S = \{(1,1)\}$  ; Dans ce cas  $R \circ S = \{(0,1)\}$ . Graphiquement :



On obtient :  $\hat{\text{pre}}[R \circ S](\tau) = \{q=0\}$  et  $\hat{\text{pre}}[R](\hat{\text{pre}}[S](\tau)) = \hat{\text{pre}}[R](q=1) = \perp$ .  $\square$

En effet on peut seulement garantir que  $\hat{\text{pre}}[R] \circ \hat{\text{pre}}[S] \subseteq \hat{\text{pre}}[R \circ S]$   
 (Démonstration :  $(\text{pre}[R] \wedge \tilde{\text{pre}}[R])((\text{pre}[S] \wedge \tilde{\text{pre}}[S])(P))$   
 $= \text{pre}[R](\text{pre}[S](P) \cap \tilde{\text{pre}}[S](P)) \cap \tilde{\text{pre}}[R](\text{pre}[S](P) \cap \tilde{\text{pre}}[S](P))$   
 $\subseteq \text{pre}[R](\text{pre}[S](P)) \cap \tilde{\text{pre}}[R](\tilde{\text{pre}}[S](P))$ , (par II.4.8.4. et II.4.14.4.)  
 $= \text{pre}[R \circ S](P) \cap \tilde{\text{pre}}[R \circ S](P)$ , (par II.4.9.1. et II.4.15.1.)).

### 3. LA REPRESENTATION DE LA NON-TERMINAISON

La conclusion à tirer de la discussion précédente est qu'il faut un moyen explicite de représentation des calculs qui ne se terminent pas. La manière la plus naturelle de le faire est d'élargir l'ensemble d'états  $Q$  avec un élément distingué, servant précisément à "marquer" les calculs sans fin. (Cette solution est aussi adoptée par de Roever [Roe76]).

Notons par  $\omega$  cet élément distingué. Dans la suite, nous supposons toujours que l'élément  $\omega$  appartient à l'ensemble d'états  $Q$ . Prenons une relation  $R$  dans l'ensemble de relations  $\mathcal{R}$ . La signification d'un doublet  $(a,b)$  de  $R$ , en termes d'exécution d'un programme est maintenant la suivante : si  $b \neq \omega$  alors il existe un calcul qui commence à l'état  $a$  et se termine à l'état  $b$  ; si  $b = \omega$  alors il existe un calcul commençant à l'état  $a$  et ne se terminant pas.

L'introduction de l'état distingué  $\omega$  nous permet aussi de restreindre la classe des relations qui servent à représenter des programmes. En fait, il est évident que  $\omega$  ne doit admettre d'autre successeur que lui-même. D'autre part, tout programme peut, en principe, être initialisé dans n'importe quel état, c'est-à-dire, tout état de  $Q$  peut être l'état initial d'une exécution du programme. Ceci implique que tout état de  $Q$  doit avoir au moins un successeur par la relation, ce successeur pouvant être  $\omega$ , dans le cas où le calcul "n'est pas défini".

En conclusion, les relations qui servent à représenter des programmes ont (au moins) les propriétés de  $\omega$ -absorption et de totalité définies maintenant :

Définition III.3.1.  $\omega$ -absorption. Une relation  $R$ ,  $R \in \mathcal{R}$ , est  $\omega$ -absorbante si  $R(\omega) \subseteq \{\omega\}$ . (Dans la pratique, on dit simplement que  $R$  est absorbante).

Définition III.3.2. Totalité. Une relation  $R$ ,  $R \in \mathcal{R}$ , est totale si  $\forall q \in Q \ R(q) \neq \emptyset$ .

Il est évident que si  $R$  est  $\omega$ -absorbante et totale alors  $R(\omega) = \{\omega\}$ .

Ces définitions peuvent être données en termes de transformateurs de prédicats. Pour le faire, notons  $\Omega$  le prédicat qui est vrai à l'état  $\omega$  et faux ailleurs :

$$\Omega = \{\lambda q. q = \omega\}$$

Proposition III.3.3.  $R$  est absorbante ssi  $\text{image}[R](\Omega) \subseteq \Omega$ ,  
 ssi  $\Omega \subseteq \tilde{\text{pre}}[R](\Omega)$ ,  
 ssi  $\text{pre}[R](\neg\Omega) \subseteq \neg\Omega$ .  $\square$

Proposition III.3.4. 1.  $\text{pre}[R](\top) = \lambda a. R(a) \neq \emptyset$   
 2.  $\tilde{\text{pre}}[R](\perp) = \lambda a. R(a) = \emptyset$

Démonstration. 2. par II.4.13.  
 1. par dualité.  $\square$

Proposition III.3.5.  $R$  est totale ssi  $\text{pre}[R](\top) = \top$   
 ssi  $\tilde{\text{pre}}[R](\perp) = \perp$ .  $\square$

Soit  $R$  une relation absorbante et totale et interprétons  $\tilde{\text{pre}}[R](P)$  en termes de l'exécution de  $R$ . Puisque tous les éléments de  $Q$  ont au moins un successeur par  $R$  (totalité) on peut reprendre l'argument exposé dans la première partie de III.2 pour conclure qu'un état  $a$  vérifie  $\tilde{\text{pre}}[R](P)$  si et seulement si tous les calculs commençant à  $a$  vont certainement "se terminer" dans  $P$ . Nous écrivons "se terminer" (entre guillemets), car rien n'exclut que le prédicat  $P$  porte aussi sur  $\omega$ , i.e.,  $P(\omega) = \text{vrai}$  ; dans ce cas un des successeurs de  $a$  pourrait être  $\omega$ , ce qui représenterait précisément un calcul qui ne se termine pas.

Pour éviter cette possibilité il faut exclure de  $\tilde{\text{pre}}[R](P)$  tous les états pouvant mener à  $\Omega$ . Ces états sont caractérisés par le prédicat  $\text{pre}[R](\Omega)$ . En conclusion, si  $R$  est totale et absorbante, l'ensemble des états à partir desquels le calcul de  $R$  se termine certainement est caractérisé par le prédicat  $\tilde{\text{pre}}[R](P) \cap \neg \text{pre}[R](\Omega)$ , qui peut se réécrire  $\tilde{\text{pre}}[R](P) \cap \tilde{\text{pre}}[R](\neg\Omega)$ , et  $\tilde{\text{pre}}[R](P - \Omega)$  (en utilisant II.4.14.2.).

(Remarquons que seule la propriété de totalité joue un rôle important dans cet argument ; la  $\omega$ -absorption n'intervient qu'implicitement, pour éviter l'occurrence de doublets de la forme  $(\omega, b)$  avec  $b \neq \omega$ , dans la relation).

Cette discussion montre que le transformateur de prédicats  $\lambda P. \tilde{\text{pre}}[R](P-\Omega)$  est un transformateur de prédicats de correction totale, pourvu que  $R$  soit absorbante et totale (ce qui est vrai chaque fois que  $R$  représente un programme). Nous l'étudions en détail dans le reste de ce chapitre.

Nous terminons ce paragraphe avec une autre caractérisation intéressante des relations totales.

Proposition III.3.6.  $R$  est totale ssi  $\tilde{\text{pre}}[R] \subseteq \text{pre}[R]$ .

Démonstration. ( $\Rightarrow$ ) D'après II.4.16.1 et III.3.5  $\text{pre}[R](P) \cap \tilde{\text{pre}}[R](P) = \tilde{\text{pre}}[R](P)$

ce qui implique  $\tilde{\text{pre}}[R](P) \subseteq \text{pre}[R](P)$ .

( $\Leftarrow$ ) Si  $R$  n'est pas totale alors par III.3.5  $\tilde{\text{pre}}[R](\perp) \neq \perp$

tandis que d'après II.4.8.1  $\text{pre}[R](\perp) = \perp$ .  $\square$

#### 4. LE TRANSFORMATEUR DE PREDICATS $wpr[R]$

Nous avons vu que tout programme peut être décrit par une relation absorbante et totale. Nous pourrions donc ne considérer que ce type de relations dans la suite. Toutefois, pour des raisons techniques qui seront évidentes à partir du chapitre V, il est avantageux de ne pas exiger la totalité dans tous les cas. La levée de la restriction de totalité provient essentiellement du fait que l'on peut construire une relation totale par la réunion de deux relations non totales. Pour étudier facilement les relations totales ainsi obtenues, il convient de pouvoir aussi traiter les relations non totales.

En ce qui concerne la  $\omega$ -absorption, la situation est différente. En fait, il n'y a aucun intérêt à étudier des relations pour lesquelles l'état  $\omega$  pourrait avoir d'autres successeurs que lui-même. Ceci nous permet de ne considérer à partir de maintenant que des relations absorbantes. Pour abréger l'écriture, nous les appelons des  $\omega$ -relations.

Définition III.4.1.  $\omega$ -relation. Une relation  $R$ ,  $R \in \mathcal{R}$ , est une  $\omega$ -relation si  $R$  est  $\omega$ -absorbante.  $\square$

L'ensemble des  $\omega$ -relations dans  $Q$  est noté  $\mathcal{R}_\omega$ , i.e.,  
 $\mathcal{R}_\omega = \{R \in \mathcal{R} \mid R(\omega) \subseteq \{\omega\}\}$ . Nous avons :

Proposition III.4.2.  $\mathcal{R}_\omega$  est fermé pour l'union, la composition et la fermeture réflexive et transitive de relations.

Démonstration. Soient  $R, S \in \mathcal{R}_\omega$

- i.  $(R \cup S)(\omega) = R(\omega) \cup S(\omega) \subseteq \{\omega\} \cup \{\omega\} = \{\omega\}$
- ii.  $(R \circ S)(\omega) = (R \circ S)(\{\omega\}) = S(R(\{\omega\}))$ , (II.2.2.2.1.)  
 $\subseteq S(\{\omega\})$ , (II.2.2.1.4.),  
 $\subseteq \{\omega\}$
- iii. Par induction  $R^i(\omega) \subseteq \{\omega\}$  pour tout  $i \in \mathbb{N}$  et donc  $R^*(\omega) \subseteq \{\omega\}$ .  $\square$

Pour les  $\omega$ -relations, nous définissons un nouveau transformateur de prédicats.

Définition III.4.3.  $wpr[R]$ . Pour  $R \in \mathcal{R}_\omega$ , on définit  $wpr[R] \in P \rightarrow P$  par  
 $wpr[R] = \lambda P. \tilde{pre}[R](P - \Omega)$ .  $\square$

D'après la discussion du paragraphe précédent,  $wpr[R]$  est un transformateur de prédicats de correction totale si  $R$  est une  $\omega$ -relation totale.

Les propriétés intéressantes de  $wpr[R]$  se déduisent de celles de  $\tilde{pre}[R]$ , à savoir II.4.12. à II.4.14.

Proposition III.4.4.  $wpr[R](P) = \lambda a. \forall b.(a, b) \in R \Rightarrow P(b) \wedge b \neq \omega$ .  $\square$

Proposition III.4.5.  $wpr[R](P) = \lambda a. R(a) \subseteq P - \Omega$ .  $\square$

Propriétés III.4.6. Pour  $R \in \mathcal{R}_\omega$ ,  $A, B \in P$  et  $\{A_i\}_{i \in I}$  une famille non vide de prédicats de  $P$ , on a

1.  $wpr[R](A \cap B) = wpr[R](A) \cap wpr[R](B)$
2.  $wpr[R](\bigcap_{i \in I} A_i) = \bigcap_{i \in I} wpr[R](A_i)$  ( $I \neq \emptyset$ )



3.  $A \subseteq B \Rightarrow \text{wpr}[R](A) \subseteq \text{wpr}[R](B)$   
 4.  $\text{wpr}[R](A) \cup \text{wpr}[R](B) \subseteq \text{wpr}[R](A \cup B)$ .  $\square$

La définition de  $\text{wpr}[R]$  implique clairement la proposition suivante

Proposition III.4.7.  $\text{wpr}[R] \subseteq \tilde{\text{pre}}[R]$ .  $\square$

Examinons maintenant la continuité de  $\text{wpr}[R]$ . La propriété III.4.6.2 montre qu'il est toujours inférieurement continu. D'autre part, le théorème II.7.3. entraîne que le non-déterminisme fini de  $R$  est une condition suffisante pour la continuité supérieure de  $\text{wpr}[R]$ . Cette constatation ne peut nous satisfaire complètement car on trouve dans la réalité des programmes qui ne peuvent pas être décrits par des relations à non-déterminisme fini. L'exemple classique est le "goon-program" qui dans le langage des commandes gardées de Dijkstra [Dij75], s'écrit

```

do goon  $\rightarrow$   $x := x + 1$ 
     $\square$  goon  $\rightarrow$  goon := false
od

```

Si  $x$  est une variable entière la relation associée à ce programme est, à première vue, la relation  $\leq$  (plus petit ou égal) dans  $Z$ , (on ne s'intéresse qu'aux valeurs de  $x$ ). Cette relation n'est pas à non-déterminisme fini (cf. exemple II.7.1.). Néanmoins, en regardant de plus près, nous nous apercevons que pour toute valeur initiale de  $x$  le "goon-program" peut aussi ne pas se terminer. Ceci veut dire que la relation associée est, en fait,  $\leq \cup \{(m, \omega) \mid m \in Z\} \cup \{(\omega, \omega)\}$ , (qui est définie dans  $Z \cup \{\omega\}$ ).

Une propriété générale des programmes qu'on sait implémenter est que si à partir d'un état donné il est possible d'obtenir un nombre infini de résultats, alors il est aussi possible que le programme ne se termine pas. Informellement, comme le fait Dijkstra [Dij76], ceci s'explique en remarquant qu'une machine ne peut pas être obligée de faire un choix arbitraire parmi un nombre infini de possibilités en un temps fini. Un argument de style plus mathématique utilise le lemme de König, [Knu75] :

"tout arbre infini à branchement fini a au moins une branche infinie". Les programmes auxquels on s'intéresse sont à "branchement fini", car au niveau le plus élémentaire de la décomposition on ne trouve que des choix parmi un nombre fini de possibilités. En appliquant le lemme de König, nous concluons que si pour une certaine donnée le nombre de résultats est infini, alors il doit y avoir aussi un calcul infini pour cette donnée.

Ces observations nous conduisent à la définition d'un nouveau type de non-déterminisme qui prend en compte la possibilité de non-terminaison dans les cas où il y a un nombre infini de résultats possibles.

Définition III.4.8. Non-déterminisme borné. La  $\omega$ -relation  $R$  est à non-déterminisme borné si  $\forall q \in Q (\exists k \in \mathbb{N} |R(q)| \leq k \vee (q, \omega) \in R)$ .  $\square$

Le théorème suivant est fondamental.

Théorème III.4.9. Le transformateur de prédicats  $wpr[R]$  est supérieurement continu ssi la  $\omega$ -relation  $R$  est à non-déterminisme borné.

Démonstration. ( $\Leftarrow$ ) Soit  $\{P_i\}$  une suite croissante de prédicats. Nous voulons montrer que, pour tout  $a$

$$(*) \quad wpr[R](\cup P_i)(a) = \exists i \quad wpr[R](P_i)(a)$$

Soit  $a$  tel que  $\exists k |R(a)| \leq k$ . Ecrivons  $R_a = \{(x, y) \in R \mid x=a\}$ .

$R_a$  est clairement à non-déterminisme fini. De plus, nous avons, trivialement,  $R(a) = R_a(a)$ , ce qui implique, en utilisant II.4.13 que  $\tilde{pre}[R](A)(a) = \tilde{pre}[R_a](A)(a)$ , pour tout  $A$ .

Calculant à partir de la partie gauche de (\*) nous avons

$$\begin{aligned} wpr[R](\cup P_i)(a) &= \tilde{pre}[R](\cup P_i - \Omega)(a) \\ &= \tilde{pre}[R](\cup (P_i - \Omega))(a) \\ &= \tilde{pre}[R_a](\cup (P_i - \Omega))(a) \\ &= (\cup \tilde{pre}[R_a](P_i - \Omega))(a) \quad (\text{théorème II.7.3}) \\ &= \exists i \quad \tilde{pre}[R_a](P_i - \Omega)(a) \\ &= \exists i \quad \tilde{pre}[R](P_i - \Omega)(a) \\ &= \exists i \quad wpr[R](P_i)(a), \text{ ce qui établit } (*). \end{aligned}$$

Soit maintenant  $a$  tel que  $(a, \omega) \in R$ . Alors pour tout  $A$  nous avons  $wpr[R](A)(a) = \text{faux}$ , ce qui établit (\*) trivialement.

( $\Rightarrow$ ) Supposons que  $R$  n'est pas à non-déterminisme borné. Alors  $\exists a \forall k |R(a)| > k \wedge (a, \omega) \notin R$ . Pour un tel  $a$ , nous avons  $R(a) \subseteq A$  ssi  $R(a) \subseteq A - \Omega$ , pour tout  $A$ , ce qui veut dire, utilisant II.4.13 et II.4.5 que  $\tilde{\text{pre}}[R](A)(a) = \text{wpr}[R](A)(a)$ . Nous pouvons, par conséquent, utiliser le contre-exemple de la démonstration du théorème II.7.3 pour montrer que, dans ce cas, (\*) n'est pas toujours vérifié.  $\square$

Le transformateur de prédicats  $\tilde{\text{wpr}}[R]$ , dual de  $\text{wpr}[R]$ , sera très utilisé dans la suite. Nous présentons maintenant ses propriétés intéressantes qui sont, naturellement, les duales de celles de  $\text{wpr}[R]$ .

Proposition III.4.10.  $\tilde{\text{wpr}}[R] = \lambda P. \text{pre}[R](P \cup \Omega)$ .  $\square$

Rappelons que, comme  $\text{wpr}[R]$ ,  $\tilde{\text{wpr}}[R]$  n'est défini que si  $R$  est une  $\omega$ -relation.

Proposition III.4.11.  $\tilde{\text{wpr}}[R](P) = \lambda a. \exists b (a, b) \in R \wedge (P(b) \vee b = \omega)$ .  $\square$

Proposition III.4.12.  $\tilde{\text{wpr}}[R](B) = A$  ssi  $R^{-1}(B \cup \Omega) = A$ .  $\square$

Propriétés III.4.13. Pour  $R \in \mathcal{R}_\omega$ ,  $A, B \in \mathcal{P}$  et  $\{A_i\}_{i \in I}$  une famille non-vide de prédicats de  $\mathcal{P}$  on a

1.  $\tilde{\text{wpr}}[R](A \cup B) = \tilde{\text{wpr}}[R](A) \cup \tilde{\text{wpr}}[R](B)$
2.  $\tilde{\text{wpr}}[R](\bigcup_{i \in I} A_i) = \bigcup_{i \in I} \tilde{\text{wpr}}[R](A_i)$  ( $I \neq \emptyset$ )
3.  $A \subseteq B \Rightarrow \tilde{\text{wpr}}[R](A) \subseteq \tilde{\text{wpr}}[R](B)$
4.  $\tilde{\text{wpr}}[R](A \cap B) \subseteq \tilde{\text{wpr}}[R](A) \cap \tilde{\text{wpr}}[R](B)$ .  $\square$

Proposition III.4.14.  $\text{pre}[R] \subseteq \tilde{\text{wpr}}[R]$ .  $\square$

La propriété III.4.13.2 montre que  $\tilde{\text{wpr}}[R]$  est toujours supérieurement continu. Pour la continuité inférieure, nous avons le dual du théorème III.4.9. :

Théorème III.4.15. Le transformateur de prédicats  $\tilde{\text{wpr}}[R]$  est inférieurement continu ssi la  $\omega$ -relation  $R$  est à non-déterminisme borné.  $\square$

Rappelons la proposition II.5.10 qui nous dit, informellement, que la relation d'inclusion ensembliste entre relations est préservée par la fonction  $\text{pre}$  et que son inverse est préservée par la fonction  $\tilde{\text{pre}}$ . Le problème que nous nous posons maintenant est de trouver une relation, notée  $\hat{\subseteq}$ , entre relations, telle que :  
 $R \hat{\subseteq} S$  ssi  $\text{wpr}[R] \subseteq \text{wpr}[S]$ . On souhaiterait que  $\hat{\subseteq}$  soit une relation d'ordre, mais ce n'est pas possible, car s'il en était ainsi, l'antisymétrie de  $\hat{\subseteq}$  impliquerait que si  $\text{wpr}[R] = \text{wpr}[S]$  alors  $R=S$  ; or, cette dernière assertion est fautive, comme le montre l'exemple suivant :

Exemple III.4.1. Soient  $R$  et  $S$  les relations représentées graphiquement par



On a  $R \neq S$  mais  $\text{wpr}[R] = \text{wpr}[S] = \lambda P.1$   $\square$

Par conséquent, le mieux que l'on puisse espérer est que  $\hat{\subseteq}$  soit un pré-ordre (i.e., une relation réflexive et transitive).

Il est plus agréable de traiter cette question en utilisant le transformateur de prédicats  $\tilde{\text{wpr}}[R]$ , à cause de sa distributivité par rapport à la disjonction de prédicats (prop. III.4.13.2), et de prendre le dual des résultats à la fin, en utilisant II.5.9. Nous définissons :

Définition III.4.16. Pré-ordre  $\overset{\vee}{\subseteq}$ . Pour  $R, S \in \mathcal{R}_\omega$  on définit  $R \overset{\vee}{\subseteq} S$  si  $R \subseteq S \cup S^{-1}(\omega) \times Q$ .  $\square$

La signification de cette définition est traduite par la proposition suivante.

Proposition III.4.17.

$R \overset{\vee}{\subseteq} S$  ssi  $\forall a, b \in Q (a, b) \in R \Rightarrow (a, b) \in S \vee (a, \omega) \in S$ .

Démonstration : omise.  $\square$

Vérifions que  $\underline{\subseteq}^{\vee}$  est, en fait, un pré-ordre. La réflexivité étant évidente par la définition, il suffit de prouver la transitivité. Nous utilisons la proposition précédente : supposons  $R \underline{\subseteq}^{\vee} S$  et  $S \underline{\subseteq}^{\vee} T$ , et soit  $(a,b) \in R$ . Dans ces conditions, nous avons  $(a,b) \in S$  ou  $(a,\omega) \in S$ . Si  $(a,b) \in S$  alors  $(a,b) \in T$  ou  $(a,\omega) \in T$  ; si  $(a,\omega) \in S$  alors  $(a,\omega) \in T$ . Par conséquent, dans tous les cas, nous avons  $(a,b) \in T$  ou  $(a,\omega) \in T$ , ce qui nous permet de conclure que  $R \underline{\subseteq}^{\vee} T$ . Pour montrer que la relation  $\underline{\subseteq}^{\vee}$  n'est pas nécessairement antisymétrique nous pouvons utiliser les relations de l'exemple III.4.1.

Le résultat que nous cherchons est le suivant :

Proposition III.4.18.  $R \underline{\subseteq}^{\vee} S$  ssi  $\tilde{wpr}[R] \subseteq \tilde{wpr}[S]$ .

Démonstration : ( $\Rightarrow$ ) Nous avons  $R \subseteq S \cup S^{-1}(\omega) \times Q$ , ce qui implique, pour tout P

$$\begin{aligned} \text{pre}[R](P) &\subseteq \text{pre}[S \cup S^{-1}(\omega) \times Q](P) \quad (\text{prop. II.5.10}) \\ &= \text{pre}[S](P) \cup \text{pre}[S^{-1}(\omega) \times Q](P) \quad (\text{prop. II.4.9.2}). \end{aligned}$$

Si  $P \neq 1$  alors  $\text{pre}[S^{-1}(\omega) \times Q](P) = \text{pre}[S](\Omega)$ , (II.4.9.3), et, par conséquent

$$\text{pre}[R](P) \subseteq \text{pre}[S](P) \cup \text{pre}[S](\Omega) \quad (P \neq 1)$$

Cette inégalité implique

$$\begin{aligned} (*) \quad \text{pre}[R](\Omega) &\subseteq \text{pre}[S](\Omega), \text{ et} \\ \text{pre}[R](P) &\subseteq \tilde{wpr}[S](P) \quad (P \neq 1) \end{aligned}$$

ce qui montre que, pour  $P \neq 1$ , nous avons

$$\tilde{wpr}[R](P) \subseteq \tilde{wpr}[S](P)$$

Si  $P=1$  cette dernière expression est équivalente à (\*) et, donc, dans tous les cas  $\tilde{wpr}[R](P) \subseteq \tilde{wpr}[S](P)$ , i.e.,  $\tilde{wpr}[R] \subseteq \tilde{wpr}[S]$ .

( $\Leftarrow$ ). Nous avons, par hypothèse

$\text{pre}[R](P) \cup \text{pre}[R](\Omega) \subseteq \text{pre}[S](P) \cup \text{pre}[S](\Omega)$ , pour tout P, ce qui implique

$$\text{pre}[R](P) \subseteq \text{pre}[S](P) \cup \text{pre}[S](\Omega), \text{ pour tout P.}$$

Si  $P \neq 1$  alors  $\text{pre}[S](\Omega) = \text{pre}[S^{-1}(\omega) \times Q](P)$  (II.4.9.3) et l'inégalité ci-dessus implique

$$\text{pre}[R](P) \subseteq \text{pre}[S \cup S^{-1}(\omega) \times Q](P) \quad (\text{II.4.9.2})$$

Cette expression est aussi vraie quand  $P=1$  (II.4.8.1). Par conséquent, elle est vraie pour tout P. D'après II.5.10, ceci entraîne

$$R \subseteq S \cup S^{-1}(\omega) \times Q. \quad \square$$

Pour obtenir le résultat concernant  $wpr$  il suffit d'appliquer II.5.9.

Définition III.4.19. Pré-ordre  $\hat{\underline{\subseteq}}$ . Pour  $R, S \in \mathcal{R}_\omega$  on définit

$$R \hat{\underline{\subseteq}} S \text{ si } S \underline{\subseteq} R \text{ (i.e. } \hat{\underline{\subseteq}} = \underline{\subseteq}^{-1} \text{)} \quad \square$$

Proposition III.4.20.  $R \hat{\underline{\subseteq}} S$  ssi  $\forall a, b \in Q (a, b) \in S \Rightarrow (a, b) \in R \vee (a, \omega) \in R$ .  $\square$

Proposition III.4.21.  $R \hat{\underline{\subseteq}} S$  ssi  $wpr[R] \underline{\subseteq} wpr[S]$ .  $\square$

A partir d'un pré-ordre on peut définir, de manière standard une relation d'équivalence. Nous utilisons ceci pour définir une relation d'équivalence entre  $\omega$ -relations, notée  $\hat{\vee}$ , et appelée "équivalence du point de vue de la correction totale".

Définition III.4.22. Equivalence du point de vue de la correction

$$\text{totale, } \hat{\vee}. \text{ Pour } R, S \in \mathcal{R} \text{ on définit } R \hat{\vee} S \text{ si } R \hat{\underline{\subseteq}} S \text{ et } R \underline{\subseteq} S. \quad \square$$

Le nom donné à cette relation provient naturellement de la propriété suivante :

Proposition III.4.23.  $R \hat{\vee} S$  ssi  $wpr[R] = wpr[S]$   
ssi  $\tilde{wpr}[R] = \tilde{wpr}[S]$ .

La relation d'équivalence du point de vue de la correction totale est évidemment moins forte que la relation d'égalité, qui est elle aussi, une relation d'équivalence. Quand on passe de  $=$  à  $\hat{\vee}$  on perd l'information concernant les situations à partir desquelles un programme peut se terminer et peut ne pas se terminer. Si on ne s'intéresse qu'à la correction totale cette perte d'information n'est pas importante car, du point de vue de la correction totale, la possibilité de non-terminaison est aussi indésirable que la certitude de non-terminaison. (Pour une étude plus détaillée des relations d'équivalence entre programmes, on peut consulter Broy, Pepper et Wirsing [BPW80], quoique leur approche soit différente).

## 5. LES RELATIONS PROGRAMMABLES

D'après Dijkstra [Dij76] un "bon" transformateur de prédicats de correction totale doit satisfaire cinq propriétés, appelées "critères de salubrité".

III.5.1. Critères de salubrité. Soit  $F$  un transformateur de prédicats,  $F \in \mathcal{P} \rightarrow \mathcal{P}$ . On dit que  $F$  est salubre, ou que  $F$  remplit les cinq critères de salubrité si  $F$  satisfait les cinq propriétés suivantes :

CS1.  $F(\perp) = \perp$

CS2.  $F(\bigcap_{i \in I} A_i) = \bigcap_{i \in I} F(A_i)$ , pour toute famille non-vide de prédicats  $\{A_i\}_{i \in I}$

CS3.  $A \subseteq B \Rightarrow F(A) \subseteq F(B)$ , pour tous prédicats  $A, B$

CS4.  $F(A) \cup F(B) \subseteq F(A \cup B)$ , pour tous prédicats  $A, B$

CS5.  $F$  est supérieurement continu.  $\square$

Remarques : Les critères CS1, CS2, CS3, CS4 ont été introduits par Dijkstra dans [Dij75] ; le critère CS5 n'apparaît que dans [Dij76]. Le critère CS2 tel que nous le présentons ici est dû à Hoare [Hoa78a] et est plus fort que celui de Dijkstra, lequel s'applique seulement à des familles finies de prédicats. Le critère CS1 est la version, dans notre modèle avec l'état  $\omega$ , de la "loi du miracle exclu", présentée par Dijkstra [Dij75] dans la forme  $F(\perp) = \perp$ .  $\square$

Par un raisonnement analogue à celui présenté à propos des propriétés II.2.2.1 on constaterait que les critères CS3 et CS4 sont en fait impliqués par CS2. Par conséquent, pour qu'un transformateur de prédicats soit salubre, il suffit qu'il remplisse les critères CS1, CS2 et CS5. Le critère CS1 signifie que si on peut garantir qu'un programme se termine, alors ce programme admet au moins un état final ; CS2 dans sa forme plus faible,  $F(A \cap B) = F(A) \cap F(B)$ , signifie que l'on peut garantir la terminaison d'un programme satisfaisant simultanément deux post-conditions si et seulement si on peut garantir la terminaison satisfaisant chacune des post-conditions ; CS5 intervient pour garantir le non-déterminisme borné des programmes considérés (cf. théorème III.4.9).

Au vu des propriétés III.4.6, nous constatons que  $wpr[R]$  remplit les critères CS2, CS3 et CS4 pour toute  $\omega$ -relation  $R$ . De plus, si  $R$  est à non-déterminisme borné  $wpr[R]$  satisfait aussi CS5 (théorème III.4.9). Quant à la "loi du miracle exclu" nous avons :

- Lemme III.5.2. 1.  $wpr[R](\Omega) = wpr[R](\perp) = \{\lambda a. R(a) = \emptyset\}$   
 2.  $\tilde{wpr}[R](\neg\Omega) = \tilde{wpr}[R](\top) = \{\lambda a. R(a) \neq \emptyset\}$

Démonstration. Par III.3.4.

Proposition III.5.3. La  $\omega$ -relation  $R$ ,  $R \in \mathcal{R}_\omega$ , est totale

$$\text{ssi } wpr[R](\Omega) = \perp$$

$$\text{ssi } \tilde{wpr}[R](\neg\Omega) = \top. \quad \square$$

Nous concluons que lorsque  $R$  est une  $\omega$ -relation totale et à non-déterminisme borné le transformateur de prédicats  $wpr[R]$  est salubre. Rappelant les discussions de III.3 et III.4 nous constatons que les  $\omega$ -relations totales à non-déterminisme borné sont précisément celles qui peuvent servir à représenter des programmes. Elles sont donc particulièrement importantes. Nous proposons la définition :

Définition III.5.4. Relation programmable. Une  $\omega$ -relation  $R$ ,  $R \in \mathcal{R}_\omega$ , est une relation programmable si  $R$  est totale et à non-déterminisme borné.  $\square$

L'ensemble des relations programmables dans  $Q$  est noté  $R_p$ .

Les relations programmables sont celles qui peuvent servir à représenter des programmes. Pour des raisons techniques, il est utile de distinguer les  $\omega$ -relations à non-déterminisme borné mais non nécessairement totales, telles que l'image de l'état  $\omega$  soit l'ensemble  $\{\omega\}$ . Ces relations sont dites semi-programmables. Cette distinction est importante car nous voudrions, par des considérations analogues à celles du paragraphe III.4, construire des relations programmables, par une opération de "fermeture" sur des relations programmables. L'intuition sous-jacente à ces propos est la description des programmes itératifs, qui bouclent sur une construction, qui n'est pas nécessairement un programme. Nous détaillerons ces questions dans la suite et nous en verrons des applications dans les chapitres suivants. Nous proposons donc la définition :



Définition III.5.5. Relation semi-programmable.

Une  $\omega$ -relation  $R$ ,  $R \in \mathcal{R}_\omega$ , est une relation semi-programmable si  $R$  est à non-déterminisme borné et  $R(\omega) = \{\omega\}$ .  $\square$

L'ensemble des relations semi-programmables dans  $Q$  est notée  $\mathcal{R}_{sp}$ .  
Nous avons évidemment  $\mathcal{R}_p \subseteq \mathcal{R}_{sp} \subseteq \mathcal{R}_\omega \subseteq \mathcal{R}$ .

La proposition III.4.2. nous dit que  $\mathcal{R}_\omega$  est fermé pour l'union et la composition de relations. Avant de montrer qu'il en est de même pour  $\mathcal{R}_{sp}$  et  $\mathcal{R}_p$  nous présentons quelques autres propriétés du transformateur de prédicats  $wpr[R]$ .

Propriété III.5.6. Pour  $R, S \in \mathcal{R}_\omega$  on a

$$wpr[R \cup S] = wpr[R] \wedge wpr[S]$$

Démonstration.  $wpr[R \cup S](P) = \tilde{pre}[R \cup S](P-\Omega) = \tilde{pre}[R](P-\Omega) \cap \tilde{pre}[S](P-\Omega)$ , par (II.4.15.2)  
 $= wpr[R](P) \cap \tilde{wpr}[S](P)$ .  $\square$

Propriété III.5.7. Pour  $R \in \mathcal{R}_\omega$  on a  $R(\omega) = \{\omega\}$

$$\text{ssi } wpr[R](\tau) \subseteq \neg\Omega$$

$$\text{ssi } \Omega \subseteq \tilde{wpr}[R](\perp).$$

Démonstration (premier  $\Rightarrow$ ) Le seul état qui ne vérifie pas  $\neg\Omega$  est  $\omega$ .

Donc il suffit de montrer que

$$(*) wpr[R](\tau)(\omega) = \text{faux},$$

Par III.4.5 (\*) est équivalent à  $R(\omega) \not\subseteq \neg\Omega$ , ce qui est évidemment vrai.

(premier  $\Leftarrow$ ) Si  $R(\omega) \neq \{\omega\}$  alors  $R(\omega) = \emptyset$  et  $R(\omega) \subseteq \neg\Omega$ , trivialement, ce qui entraîne, par III.4.5 que  $wpr[R](\tau)(\omega) = \text{vrai}$  et donc  $wpr[R](\tau) \not\subseteq \neg\Omega$ .

Le résultat pour  $\tilde{wpr}[R]$  suit par dualité.  $\square$

Corollaire III.5.8. Pour  $R \in \mathcal{R}_\omega$ , si  $R(\omega) = \{\omega\}$  alors pour tout  $P \in \mathcal{P}$

$$wpr[R](P) \subseteq \neg\Omega \text{ et } \Omega \subseteq \tilde{wpr}[R](P).$$

Démonstration. Utiliser III.4.63 et III.4.13.3.  $\square$

Propriétés III.5.9. Pour  $R \in \mathcal{R}_\omega$  tel que  $R(\omega) = \{\omega\}$  et  $S \in \mathcal{R}_\omega$  on a

1.  $wpr[S \circ R] = wpr[S] \circ wpr[R]$
2.  $wpr[R^i] = wpr[R]^i$ , pour  $i \geq 1$
3.  $wpr[R^*] = wpr[R]^*$
4.  $\tilde{wpr}[S \circ R] = \tilde{wpr}[S] \circ \tilde{wpr}[R]$
5.  $\tilde{wpr}[R^i] = \tilde{wpr}[R]^i$ , pour  $i \geq 1$
6.  $\tilde{wpr}[R^*] = \tilde{wpr}[R]^*$

Démonstration

1.  $wpr[S \circ R](P) = \tilde{pre}[S \circ R](P - \Omega) = \tilde{pre}[S](\tilde{pre}[R](P - \Omega))$ , (par II.4.15.1)  
 $= \tilde{pre}[S](wpr[R](P)) = \tilde{pre}[S](wpr[R](P) - \Omega)$ , (parce que par III.5.7  
 $wpr[R](P) - \Omega = wpr[R](P)$ ),  $= wpr[S](wpr[R](P))$ .
2. Par induction, utilisant 1.
3.  $wpr[R^*](P) = \tilde{pre}[R^*](P - \Omega) = \tilde{pre}[R]^X(P - \Omega)$ , (par II.5.7.4)  
 $= \bigcap_{i \in \mathbb{N}} \tilde{pre}[R]^i(P - \Omega) = \bigcap_{i \in \mathbb{N}} \tilde{pre}[R^i](P - \Omega)$ , par (II.5.7.3),  
 $= \bigcap_{i \in \mathbb{N}} wpr[R^i](P) = wpr[I](P) \bigcap_{\substack{i \geq 0 \\ wpr[I](P) = P - \Omega}} wpr[R]^i(P)$ , (par 2.)  $= wpr[R]^X(P)$ , car
- 4., 5. et 6. Par dualité.  $\square$

Proposition III.5.10.  $R_{sp}$  est fermé pour l'union et la composition de relations.

Démonstration. Soient  $R, S \in R_{sp}$ .

- i.  $(R \cup S)(\omega) = R(\omega) \cup S(\omega) = \{\omega\} \cup \{\omega\} = \{\omega\}$ . De plus  
 $wpr[R \cup S] = wpr[R] \wedge wpr[S]$ , par III.5.6;  $wpr[R]$  et  $wpr[S]$  étant  
supérieurement continus, par III.4.9,  $wpr[R] \wedge wpr[S]$  l'est aussi,  
par II.5.17, ce qui implique par III.4.9 que  $R \cup S$  est à non-  
déterminisme borné.  $\square$
- ii. Alors  $(R \circ S)(\omega) = (R \circ S)(\{\omega\}) = S(R(\{\omega\}))$ , (par II.2.2.2.1)  
 $= S(\{\omega\}) = \{\omega\}$ . De plus,  $wpr[R \circ S] = wpr[R] \circ wpr[S]$  et, comme pour l'union  
on conclut que  $R \circ S$  est à non-déterminisme borné.  $\square$

Proposition III.5.11.  $R_p$  est fermé pour l'union et la composition de relations.

Démonstration. Soient  $R, S \in R_p$ . Au vu des démonstrations de III.4.2 et III.5.10, il suffit de prouver la totalité de  $R \cup S$  et  $R \circ S$ . Nous utilisons III.3.4. :  $pre[R \cup S](1) = pre[R](1) \cup pre[S](1)$ , (par II.4.9.2)  $= 1 \cup 1 = 1$  ;  $pre[R \circ S](1) = pre[R](pre[S](1))$ , (par II.4.9.1)  $= pre[R](1) = 1$ .  $\square$

Contrairement à  $R_\omega$ , les ensembles  $R_{sp}$  et  $R_p$  ne sont pas fermés pour la fermeture réflexive transitive, comme l'illustre l'exemple suivant.

Exemple III.5.1. Soit  $R$  définie dans  $(Z \cup \{\omega\})^2$  par  
 $R = \{(m, n) \mid m, n \in Z \wedge n = m + 1\} \cup \{(\omega, \omega)\}$ .  $R$  est évidemment une relation programmable. Toutefois,

$R^* = \{(m,n) \mid m,n \in \mathbb{Z} \wedge n \geq m\} \cup \{(\omega,\omega)\}$ , n'est pas à non-déterminisme borné (cf. exemple II.7.1) et donc  $R^*$  n'est pas programmable, ni semi-programmable.  $\square$

L'exemple ci-dessus montre aussi que la fermeture réflexive transitive d'une relation semi-programmable n'est pas nécessairement programmable. Cette situation est assez embarrassante car elle remet un peu en question nos justifications pour l'introduction des relations semi-programmables, à savoir qu'elles doivent permettre obtenir des relations programmables par une opération de fermeture. Nous reviendrons sur cette question dans le chapitre suivant avec une nouvelle opération de "fermeture" qui a toutes les bonnes propriétés.

Nous terminons ce chapitre avec une nouvelle caractérisation des relations programmables, inspirée par des problèmes similaires résolus par Wand [Wan77] et Best [Bes79].

**Théorème III.5.12.** Un transformateur de prédicats  $F, F \in \mathcal{P} \rightarrow \mathcal{P}$  est salubre et tel que  $F(\tau) \subseteq \neg\Omega$  ssi il existe une relation programmable  $R, R \in \mathcal{R}_P$ , telle que  $F = \text{wpr}[R]$ .

Démonstration. ( $\Leftarrow$ ) triviale par III.4.6., III.4.9. et III.5.7.

( $\Rightarrow$ )  $\tilde{F}$ , le dual de  $F$ , vérifie  $\Omega \subseteq \tilde{F}(\perp)$  et, pour toute famille non vide de prédicats  $\{A_i\}_{i \in I}, I \neq \emptyset, \tilde{F}(\cup A_i) = \cup \tilde{F}(A_i)$ . Soit  $\langle a \rangle$  le prédicat  $\lambda q. \{q=a\}$ , et pour un prédicat  $P$ , soit  $\underline{\tilde{F}}(P)$  une notation abrégée de l'ensemble  $\underline{\tilde{F}}(P)$ .

Définissons une relation  $S$  par

$$S(a) = \begin{cases} \underline{\tilde{F}}(\perp) & \text{si } a = \omega \\ \underline{\tilde{F}}(\langle a \rangle) - \{\omega\} & \text{si } a \neq \omega \end{cases}$$

Vu que  $\Omega \subseteq \underline{\tilde{F}}(\perp)$ , nous avons  $S^{-1}(\omega) = \{\omega\}$ . De plus, pour  $P \neq \perp$  on a

$$S(\underline{P \cup \Omega}) = S(\underline{P}) \cup S(\underline{\Omega}), \text{ (par II.2.2.1.2)}$$

$$= S(\cup_{a \in \underline{P}} \{a\}) \cup S(\omega) = \cup_{a \in \underline{P}} S(\{a\}) \cup \underline{\tilde{F}}(\perp)$$

$$= (\cup_{a \in \underline{P}} \underline{\tilde{F}}(\langle a \rangle) - \{\omega\}) \cup \underline{\tilde{F}}(\perp) = \underline{\tilde{F}}(\cup_{a \in \underline{P}} \langle a \rangle) - \{\omega\} \cup \underline{\tilde{F}}(\perp)$$

$$= \underline{\tilde{F}}(\underline{P}) - \{\omega\} \cup \underline{\tilde{F}}(\perp) = \underline{\tilde{F}}(\underline{P}) \cup \underline{\tilde{F}}(\perp), \text{ (car } \Omega \subseteq \underline{\tilde{F}}(\perp))$$

$$= \underline{\tilde{F}}(\underline{P}). \text{ Si } P = \perp \text{ alors, par construction } S(\underline{\perp \cup \Omega}) = \underline{\tilde{F}}(\perp).$$

En résumé, pour tout  $P$ ,  $S(P \cup \Omega) = \tilde{F}(P)$ , c'est-à-dire,  $\text{image}[S](P \cup \Omega) = \tilde{F}(P)$ , par II.4.2.

Soit  $R=S^{-1}$ . Alors  $\text{pre}[R](P \cup \Omega)=\tilde{F}(P)$ , par II.4.5. De plus  $R(\omega)=\{\omega\}$ , ce qui veut dire que  $R$  est une  $\omega$ -relation. Nous avons donc  $\tilde{\text{wpr}}[R](P)=\tilde{F}(P)$ , par III.4.10, et par conséquent  $\text{wpr}[R](P)=F(P)$ .

$F$  satisfaisant les critères III.5.1 CS1 et CS5,  $\text{wpr}[R]$  les satisfait aussi, ce qui entraîne respectivement par III.5.3 et III.4.9 que  $R$  est totale et à non-déterminisme borné.  $\square$

Le théorème que nous venons de démontrer suggère que dans notre modèle la propriété  $F(\tau) \subseteq \neg\Omega$  doit avoir le statut de critère de salubrité ;

III.5.13. Critère de salubrité. A la liste de III.5.1. on doit ajouter un sixième critère de salubrité

CS6.  $F(\tau) \subseteq \neg\Omega$ .  $\square$

Ce critère est en quelque sorte une deuxième loi du miracle exclu. En fait, sa signification est la suivante : pour pouvoir garantir qu'un programme termine ( $F(\tau)$ ) il faut pouvoir garantir qu'il commence ( $\neg\Omega$ ). Remarquons à ce propos que la première loi du miracle exclu qui, par III.5.12 et III.5.3, est équivalente à la totalité de la  $\omega$ -relation en cause, signifie que tout programme qui commence doit soit se terminer soit ne pas se terminer. Le miracle exclu dans ce cas est l'existence d'une troisième alternative. D'ailleurs, de ce point de vue le critère de salubrité III.5.1. CS5, est aussi une loi de miracle exclu : par III.4.9. il traduit l'impossibilité d'avoir un programme qui, à partir d'un état donné, se termine toujours et peut quand même fournir un nombre infini de résultats. En fait, les seuls critères qui sont inhérents à la définition de  $\text{wpr}[R]$  sont CS2 et, par conséquent, CS3 et CS4.

## 6. CONCLUSION

Nous avons proposé et étudié en détail un transformateur de prédicats de correction totale,  $wpr[R]$ . Les résultats principaux de ce chapitre trouvent leur synthèse dans le théorème final (théorème III.5.12), qui présente une caractérisation assez forte de la salubrité de  $wpr[R]$  en termes des propriétés de la relation  $R$ . Cette caractérisation est néanmoins suffisamment générale pour contenir tous les cas intéressants qu'on trouve en pratique.

La question qui se pose maintenant est de savoir si  $wpr$  correspond réellement à la fonction  $wp$  de Dijkstra [Dij75]. Celle-ci est définie en termes des programmes d'un langage particulier, le langage des commandes gardées ; pour pouvoir répondre à la question, il faut avoir un moyen de calculer les relations associées aux programmes de ce langage. Autrement dit, il faut avoir une sémantique relationnelle du langage. Ceci sera l'objet du chapitre V. Auparavant, il convient d'introduire deux autres concepts fondamentaux : invariant et trajectoire. Nous le faisons dans le chapitre suivant, en utilisant largement les résultats présentés jusqu'ici.

## CHAPITRE IV

### INVARIANTS ET TRAJECTOIRES



## 1. INTRODUCTION

Une partie importante du travail de vérification ou analyse des propriétés de programmes concerne la découverte (et la preuve) de certains "invariants". Habituellement, les invariants sont utilisés en tant qu'assertions (ou prédicats) sur les valeurs de certaines variables, associées à des points du programme, et on prétend que ces assertions sont validées chaque fois que le contrôle passe par les points auxquels elles sont associées, Floyd [Flo67], Hoare [Hoa69].

Dans ce chapitre, nous introduisons, formellement, un concept d'invariant qui traduit d'une certaine manière l'intuition sous-jacente à l'approche décrite. La démarche suivie est assez abstraite, étant donné que nous ne nous intéressons pas maintenant à la preuve de programmes mais seulement à la caractérisation du concept d'invariant, en termes de la théorie relationnelle développée dans les chapitres précédents. L'application des résultats de ce chapitre fera l'objet du chapitre VI.

Le concept d'invariant présenté ici est inspiré de celui que propose Sifakis [Sif79a], dans le cadre de l'étude des propriétés des systèmes de transitions (voir aussi Keller [Kel76]), et il généralise, d'une certaine manière, celui utilisé par Dijkstra [Dij76], comme nous le montrerons dans le chapitre VI.

Un concept complémentaire d'invariant correspondrait, en termes classiques, à des assertions associées à des points du programme tels que l'on ne puisse pas exclure que lorsque le contrôle passe par ces points les assertions soient validées. Cette situation est formalisée à l'aide du concept de trajectoire, proposé par Sifakis [Sif79a], [Sif79b], [Sif80], que nous adaptons à notre modèle. Quoique les trajectoires ne soient pas essentielles pour les preuves de correction habituelles elles sont très importantes du point de vue sémantique comme nous aurons l'occasion de le montrer.



## 2. LE CONCEPT D'INVARIANT

Soit  $R$  une relation programmable, représentant un certain programme. Nous voulons définir un invariant de  $R$  (ou du programme) comme un prédicat  $J$  qui, étant vérifié avant l'exécution de  $R$ , garantit la terminaison dans un état vérifiant encore  $J$ . Ceci se traduit formellement par  $\forall a \in Q \ J(a) \Rightarrow (\forall b \ (a,b) \in R \Rightarrow J(b) \wedge b \neq \omega)$ .

Pour les raisons exposées à propos de la définition des relations semi-programmables (définition III.5.5), il est utile d'élargir la notion d'invariant à ce type de relations. Pour une relation semi-programmable  $R$ , un état  $a$  tel que  $R(a) = \emptyset$  doit être interprété comme un état à partir duquel  $R$  ne peut pas être exécutée. Ainsi, nous disons que  $J$  est un invariant de  $R$  si, quand  $J$  est vérifié et l'exécution de  $R$  peut avoir lieu, alors elle se termine certainement dans un état de  $J$ . Cette notion se traduit par la même formule que ci-dessus. Nous proposons la définition générale suivante :

Définition IV.2.1. Invariant. Soient  $R$  une relation semi-programmable,  $R \in \mathcal{R}_{sp}$ , et  $J$  un prédicat,  $J \in \mathcal{P}$ . On dit que  $J$  est un invariant de  $R$  si  $\forall a \in Q \ J(a) \Rightarrow (\forall b \in Q \ (a,b) \in R \Rightarrow J(b) \wedge b \neq \omega)$ .  $\square$

Nous avons immédiatement la proposition :

Proposition IV.2.2.  $J$  est un invariant de  $R$  ssi  $J \subseteq \text{wpr}[R](J)$ .

Démonstration. Directe, par III.4.4.  $\square$

Les propositions suivantes sont maintenant faciles à démontrer.

Proposition IV.2.3. Si  $J_1$  et  $J_2$  sont des invariants de  $R$ , alors

$J_1 \cap J_2$  et  $J_1 \cup J_2$  le sont aussi.

Démonstration. i.  $J_1 \cap J_2 \subseteq \text{wpr}[R](J_1) \cap \text{wpr}[R](J_2)$ , (par hypothèse),  
 $= \text{wpr}[R](J_1 \cap J_2)$ , (par III.4.6.1).

ii.  $J_1 \cup J_2 \subseteq \text{wpr}[R](J_1) \cup \text{wpr}[R](J_2)$ , (par hypothèse),  
 $\subseteq \text{wpr}[R](J_1 \cup J_2)$ , (par III.4.6.4).  $\square$

Proposition IV.2.4. Si  $J$  est un invariant de  $R$  alors  $J \subseteq \bar{\Omega}$ .

Démonstration. Par IV.2.2. et III.5.8.  $\square$

Proposition IV.2.5. Si  $J$  est un invariant de  $R$  et de  $S$  alors  $J$  est un invariant de  $R \circ S$  et de  $R \cup S$ .

Démonstration.  $J \subseteq \text{wpr}[R](J)$  et  $J \subseteq \text{wpr}[S](J)$  implique :

- i.  $J \subseteq \text{wpr}[R](\text{wpr}[S](J))$ , par III.4.6.3, =  $\text{wpr}[R \circ S](J)$ , par III.5.9.1.
- ii.  $J \subseteq \text{wpr}[R](J) \cap \text{wpr}[S](J) = \text{wpr}[R \cup S](J)$ , par III.5.6.  $\square$

Proposition IV.2.6. Pour  $R \in \mathcal{R}_{sp}$  et  $P \in \mathcal{P}$ , si  $R(\underline{P}) = \delta$  alors  $P$  est un invariant de  $R$ .

Démonstration. Utiliser, par exemple, IV.2.2 et III.4.5.  $\square$

Le théorème suivant est fondamental car, à notre avis, il isole l'essence du concept d'invariant.

Théorème IV.2.7. (Théorème fondamental des invariants)

$J$  est un invariant de  $R$  ssi, pour tout  $n \in \mathbb{N}$ ,  $J$  est un invariant de  $R^n$ .

Démonstration. ( $\Leftarrow$ ) triviale.

( $\Rightarrow$ ) Par induction.  $n=0$  : Par IV.2.4  $J \subseteq \bar{\Omega}$  et donc  $J = J - \Omega = \text{wpr}[I](J)$  ;  $n+1$  :  $J$  invariant de  $R$  (par hypothèse) et  $J$  invariant de  $R^n$  (hypothèse d'induction) implique  $J$  invariant de  $R^{n+1}$ , par IV.2.5.  $\square$

Corollaire IV.2.8. Si  $J$  est un invariant de  $R$  alors  $J = \text{wpr}[R]^x(J)$ .

Démonstration.  $J \subseteq \text{wpr}[R^i](J) = \text{wpr}[R]^i(J)$ , pour  $i \geq 1$ , et  $J = \text{wpr}[R]^0(J)$  impliquent  $J \subseteq \text{wpr}[R]^x(J)$ , et d'autre part  $\text{wpr}[R]^x(J) \subseteq J$ , trivialement.

Corollaire IV.2.9. Si  $J$  est un invariant de  $R$  alors  $J$  est un invariant de  $R^*$ .

Démonstration. Par IV.2.8 et III.5.9.3.  $\square$

L'interprétation du théorème est la suivante. Si à partir d'un état d'un invariant  $J$ , on décide d'exécuter  $R$  de façon répétitive alors à chaque pas on se retrouve dans un état de  $J$ . Si un état  $a$  tel que  $R(a)=\emptyset$  est atteint, ceci signifie que la répétition est terminée. De plus, cet état final est un état de  $J$ .

Proposition IV.2.10. Les invariants de  $R$  sont les points fixes du transformateur de prédicats  $I \wedge \text{wpr}[R]$ .

Démonstration.  $J \subseteq \text{wpr}[R](J)$ , (proposition IV.2.2.), est équivalent à  $J = J \cap \text{wpr}[R](J)$ , i.e.,  $J = (I \wedge \text{wpr}[R])(J)$ .  $\square$

Proposition IV.2.11. Pour tout  $P \in \mathcal{P}$ , le plus grand invariant de  $R$  inférieur à  $P$  est le prédicat  $\text{wpr}[R]^\times(P)$ .

Démonstration. Par II.6.4.  $\square$

Ces deux dernières propositions signifient aussi (d'après le théorème de Tarski, [Tar55]) que les invariants d'une relation semi-programmable  $R$  forment un treillis complet dont l'infimum est  $\perp$  et le supremum est  $\text{wpr}[R]^\times(\top)$ . (Voir aussi Amy et Caplain [AmC78], Sifakis [Sif79a]). D'ailleurs la proposition IV.2.4. nous permet de conclure que le plus grand invariant de  $R$  est aussi donné par  $\text{wpr}[R]^\times(\neg\Omega)$ . D'autre part, la proposition IV.2.3. montre que les opérations de borne supérieure et borne inférieure dans les treillis des invariants coïncident avec les opérations correspondantes dans le treillis des prédicats. Par contre, il est évident que le treillis des invariants n'est pas booléen.

### 3. LE CONCEPT DE TRAJECTOIRE

Nous avons vu que, pour une relation semi-programmable  $R$ , un invariant de  $R$  est un prédicat  $J$  tel que pour tout état de  $J$  à partir duquel l'exécution de  $R$  est possible, cette exécution se termine certainement dans un état de  $J$ . De manière complémentaire, une trajectoire de  $R$  est un prédicat  $W$  tel que pour tout état de  $W$  l'exécution de  $R$  est possible à partir de cet état, mais on ne peut pas garantir qu'elle se termine dans un état de  $\neg W$ , (c'est-à-dire, il se peut qu'elle se termine dans  $W$  ou qu'elle ne se termine pas. Formellement :

Définition IV.3.1. Trajectoire. Soient  $R$  une relation semi-programmable,  $R \in \mathcal{R}_P$ , et  $W$  un prédicat,  $W \in \mathcal{P}$ . On dit que  $W$  est une trajectoire de  $R$  si  $\forall a \in Q \ W(a) \Rightarrow \exists b \in Q \ (a,b) \in R \wedge (W(b) \vee b=\omega)$ .  $\square$

Comme pour les invariants, nous avons :

Proposition IV.3.2.  $W$  est une trajectoire de  $R$  ssi  $W \subseteq \tilde{wpr}[R](W)$ .

Démonstration. Directe, par III.4.11.  $\square$

Proposition IV.3.3. Si  $W_1$  et  $W_2$  sont des trajectoires de  $R$  alors  $W_1 \cup W_2$  l'est aussi.

Démonstration. Utiliser III.4.13.1.  $\square$

Remarquons que l'intersection de deux trajectoires n'est pas nécessairement une trajectoire.

Proposition IV.3.4. Si  $W$  est une trajectoire de  $R$  alors  $W \subseteq \lambda a.R(a) \neq \emptyset$ .

Démonstration.  $W \subseteq \tilde{wpr}[R](W) \subseteq \tilde{wpr}[R](\tau)$ , par III.4.13.3,  
 $= \lambda a.R(a) \neq \emptyset$ , par III.5.2.2.  $\square$

Proposition IV.3.5. Si  $W$  est une trajectoire de  $R$  et de  $S$  alors  $W$  est une trajectoire de  $R \circ S$ .

Démonstration. Utiliser III.5.9.4.  $\square$

Comme pour les invariants nous avons un théorème fondamental des trajectoires.

Théorème IV.3.6.  $W$  est une trajectoire de  $R$  ssi pour tout  $n \in \mathbb{N}$

$W$  est une trajectoire de  $R^n$ .

Démonstration. Analogue à celle de IV.2.7.  $\square$

Corollaire IV.3.7. Si  $W$  est une trajectoire de  $R$  alors  $W = \tilde{wpr}[R]^x(W)$ .

Démonstration. Analogue à celle de IV.2.8.  $\square$

L'interprétation du théorème est la suivante. Si à partir d'un état d'une trajectoire on décide d'itérer sur  $R$  alors on ne peut pas exclure la possibilité qu'après un nombre quelconque d'itérations on se retrouve encore dans la trajectoire, donc dans un état à partir duquel une autre itération est possible, tout ceci à condition qu'entre-temps une des itérations n'ait pas dégénéré en un calcul sans fin. En résumé, si on initialise l'exécution répétitive de  $R$  dans une trajectoire, on ne peut pas garantir qu'elle se termine. Toutefois, si elle se termine, c'est nécessairement en dehors de la trajectoire.

Remarque. L'interprétation donnée ci-dessus montre que le concept de trajectoire proposé correspond au concept de trajectoire sans fin que l'on trouve chez Sifakis [Sif79a][Sif79b][Sif80]. Cet auteur définit un concept de trajectoire, plus général que le nôtre, qui se traduirait informellement dans notre modèle par : une trajectoire de R est un prédicat W tel qu'à partir de tout état de W, si l'exécution de R est possible à partir de cet état alors il se peut qu'elle ne se termine pas dans un état de  $\neg W$ . Formellement ceci s'écrit

$$\forall a W(a) \Rightarrow R(a) \neq \emptyset \vee \exists b (a,b) \in R \wedge (W(b) \vee b = \omega).$$

Rappelant III.5.2.1, ceci est équivalent à

$W \subseteq \text{wpr}[R](\Omega) \cup \widetilde{\text{wpr}}[R](P)$ . La partie droite de cette inéquation est encore équivalente à  $\text{wpr}[R](P) \cup \widetilde{\text{wpr}}[R](P)$  (utiliser un argument analogue à celui de II.4.16), ce qui donnerait pour les trajectoires de Sifakis la caractérisation  $W \subseteq (\text{wpr}[R] \vee \widetilde{\text{wpr}}[R])(W)$ .  $\square$

Proposition IV.3.8. Les trajectoires de R sont les points fixes du transformateur de prédicats  $I \wedge \widetilde{\text{wpr}}[R]$ .

Démonstration. Utiliser IV.3.2.  $\square$

Proposition IV.3.9. Pour tout  $P \in \mathcal{P}$  la plus grande trajectoire de R inférieure à P est le prédicat  $(I \wedge \widetilde{\text{wpr}}[R])^{\times}(P)$ .  $\square$

Par conséquent, la plus grande trajectoire de R est  $(I \wedge \widetilde{\text{wpr}}[R])^{\times}(\tau)$ . Ce prédicat peut aussi être donné par d'autres expressions. En effet :

Lemme IV.3.10. Si P est un invariant de R alors

$$1. (I \wedge \widetilde{\text{wpr}}[R])^{\times}(\neg P) = \widetilde{\text{wpr}}[R]^{\times}(\neg P)$$

$$2. (I \vee \text{wpr}[R])^{\star}(P) = \text{wpr}[R]^{\star}(P).$$

Démonstration. 2. Il suffit de montrer que  $\forall i (I \vee \text{wpr}[R])^i = \text{wpr}[R]^i$ .  $i=0$ , trivial.

$i+1$ .  $(I \vee \text{wpr}[R])^{i+1}(P) = (I \vee \text{wpr}[R])(\text{wpr}[R]^i(P))$ , hypothèse d'induction,  $= \text{wpr}[R]^i(P) \cup \text{wpr}[R]^{i+1}(P) = \text{wpr}[R]^{i+1}(P)$ , car  $P \subseteq \text{wpr}[R](P)$  implique  $\text{wpr}[R]^i(P) \subseteq \text{wpr}[R]^{i+1}(P)$ .

1. Par dualité.  $\square$

Proposition IV.3.11. La plus grande trajectoire de R est le prédicat  $\widetilde{\text{wpr}}[R]^{\times}(\lambda a. R(a) \neq \emptyset)$ .

Démonstration. Par IV.3.4, la plus grande trajectoire est la plus grande trajectoire contenue dans  $\lambda a. R(a) \neq \emptyset$ . Le complément de ce prédicat est un invariant, par IV.2.6., d'où le résultat, utilisant IV.3.9 et IV.3.10.1.  $\square$

Comme pour les invariants, nous concluons que les trajectoires d'une relation semi-programmable  $R$  forment un treillis complet, avec infimum  $\perp$  et supremum  $\tilde{wpr}[R]^X (\lambda a. R(a) \neq \emptyset)$ . Cependant, tandis que l'opération de borne supérieure dans ce treillis coïncide avec l'opération correspondante dans le treillis de prédicats (prop. IV.3.3.), il n'en est pas de même pour l'opération de borne inférieure.

#### 4. INVARIANTS SANS BLOCAGE ET TERMINAISON DES EXECUTIONS REPETITIVES

On s'intéresse parfois à des programmes qui bouclent indéfiniment. Il s'agit de programmes itératifs tels que chaque itération se termine et chaque itération est suivie par une autre. Il est important de déterminer dans quelles conditions on peut garantir qu'un programme boucle "sans blocage".

Symétriquement il existe des situations où on souhaite garantir qu'un programme répétitif se termine, mais sans borner le nombre de répétitions autorisées. Cette question peut être étudiée par l'intermédiaire de la question complémentaire : dans quelles conditions ne peut-on pas assurer la terminaison de l'exécution répétitive ?

Pour traiter le premier de ces problèmes, nous introduisons le concept d'invariant sans blocage, adapté à partir de Sifakis [Sif79]. Précisons tout d'abord un détail de notation :

Définition IV.4.1. Le prédicat  $\text{exit}_R$ . Pour une relation semi-programmable  $R$  le prédicat  $\text{exit}_R$  est défini par  $\text{exit}_R = \{\lambda a. R(a) = \emptyset\}$ .  $\square$

Proposition IV.4.2. 1.  $\text{exit}_R = \tilde{pre}[R](\perp) = wpr[R](\Omega)$   
 2.  $\neg \text{exit}_R = pre[R](\top) = \tilde{wpr}[R](\neg \Omega)$ .

Démonstration. Par III.3.4 et III.5.2.  $\square$

Si un invariant  $J$  d'une relation semi-programmable  $R$  est contenu dans le complémentaire de  $\text{exit}_R$  alors l'exécution de  $R$  peut effectivement avoir lieu à partir de n'importe quel état de  $J$ . Dans ces conditions, il est certain que l'exécution répétitive de  $R$  initialisée dans  $J$  ne se terminera pas. Nous avons ainsi l'intuition pour la définition suivante :

Définition IV.4.3. Invariant sans blocage. Un invariant  $J$  de  $R$  est un invariant sans blocage si  $J \subseteq \neg \text{exit}_R$ .  $\square$

Proposition IV.4.4.  $J$  est un invariant sans blocage ssi  $J \subseteq (\text{wpr}[R] \wedge \tilde{\text{wpr}}[R])(J)$ .

Démonstration. Utilisant IV.4.2.,  $J$  est invariant sans blocage ssi  $J \subseteq \text{wpr}[R](J)$  et  $J \subseteq \tilde{\text{wpr}}[R](\neg \Omega)$ , i.e., ssi  $J \subseteq \text{wpr}[R](J) \cap \tilde{\text{wpr}}[R](\neg \Omega)$ . Maintenant, utilisant un argument analogue à la démonstration de II.4.16, il est facile de montrer que  $\text{wpr}[R](J) \cap \tilde{\text{wpr}}[R](\neg \Omega) = \text{wpr}[R](J) \cap \tilde{\text{wpr}}[R](J)$ .  $\square$

Il est intéressant de comparer cette proposition avec la remarque du paragraphe IV.3. La caractérisation des invariants sans blocage qu'elle nous offre montre qu'un prédicat est un invariant sans blocage si et seulement si il est simultanément un invariant et une trajectoire, ce qui nous permet d'énoncer le théorème suivant :

Théorème IV.4.5.  $J$  est un invariant sans blocage de  $R$  ssi pour tout  $n \in \mathbb{N}$ ,  $J$  est un invariant sans blocage de  $R^n$ .

Démonstration. Utiliser IV.2.7 et IV.3.6.  $\square$

Ce théorème montre bien que si l'on initialise l'exécution répétitive de  $R$  dans un invariant sans blocage alors, après n'importe quel nombre d'itérations, (chaque itération s'est bien terminée), on se trouve nécessairement dans un état de l'invariant sans blocage ; ceci implique qu'une nouvelle itération est possible. Dans ces conditions, on peut assurer la non-terminaison de l'exécution répétitive de  $R$ .

Les deux propositions suivantes peuvent être démontrées immédiatement à partir de IV.4.4 et IV.2.10, respectivement.

Proposition IV.4.6. Les invariants sans blocage de  $R$  sont les points fixes du transformateur de prédicats  $I \wedge \text{wpr}[R] \wedge \tilde{\text{wpr}}[R]$ .  $\square$

Proposition IV.4.7. Le plus grand invariant sans blocage de  $R$  est le prédicat  $\text{wpr}[R]^X(\neg \text{exit}_R)$ .  $\square$

Considérons maintenant le problème qui se pose quand on veut assurer la terminaison d'une exécution répétitive. Nous le traitons à l'aide du concept de trajectoire. Le résultat important est une conséquence immédiate du théorème suivant.

**Théorème IV.4.8.** (Théorème de la plus grande trajectoire).

La plus grande trajectoire de R est le prédicat  $\lambda a. \forall i \in \mathbb{N} R^i(a) \neq \emptyset$ .

**Démonstration.** Par IV.3.11 et IV.4.1 la plus grande trajectoire de R est  $\tilde{wpr}[R]^X(\neg \text{exit})$ . Par IV.4.2 ceci est équivalent à

$$\begin{aligned} \tilde{wpr}[R]^X(\tilde{wpr}[R](\neg \Omega)) &= \bigcap_{i \in \mathbb{N}} \tilde{wpr}[R]^i(\tilde{wpr}[R](\neg \Omega)) \\ &= \bigcap_{i \in \mathbb{N}} \tilde{wpr}[R]^{i+1}(\neg \Omega) = \bigcap_{i \in \mathbb{N}} \tilde{wpr}[R]^{i+1}(\neg \Omega), \text{ par III.5.9.5} \\ &= \bigcap_{i \in \mathbb{N}} \tilde{wpr}[R]^i(\neg \Omega), \text{ (car } \tilde{wpr}[R]^0(\neg \Omega) = \text{pre}[I](\tau) = \tau), \\ &= \bigcap_{i \in \mathbb{N}} \{\lambda a. R^i(a) \neq \emptyset\}, \text{ par III.5.2.2,} = \lambda a. \forall i \in \mathbb{N} R^i(a) \neq \emptyset. \quad \square \end{aligned}$$

**Proposition IV.4.9.** Le complément de la plus grande trajectoire est le prédicat  $\tilde{wpr}[R]^*(\text{exit})$ .  $\square$

**Démonstration.** Rappel IV.3.11 et IV.4.1.  $\square$

**Théorème IV.4.10.** (Théorème du complément de la plus grande trajectoire).

Le complément de la plus grande trajectoire de R est le prédicat  $\lambda a. \exists i \in \mathbb{N} R^i(a) = \emptyset$ .

**Démonstration.** Directe par IV.4.8.  $\square$

Ce dernier théorème montre clairement que les états du complément de la plus grande trajectoire sont précisément les états à partir desquels on ne peut pas répéter indéfiniment l'exécution de R.

**Corollaire IV.4.11.** Le complément de la plus grande trajectoire de R est un invariant de R.

**Démonstration.** Représentons par P le complément de la plus grande trajectoire de R. Soit a tel que P(a), et, soit i tel que  $R^i(a) = \emptyset$ , (ce i existe, d'après le théorème). Si b est tel que  $(a, b) \in R$  alors  $R^{i-1}(b) = \emptyset$  et donc on a P(b). De plus,  $b \neq a$ , car  $R^i(a) = \{a\}$  pour tout i. D'où par la définition IV.2.1, P est un invariant.  $\square$



## 5. L'OPERATION DE FERMETURE ITERATIVE

Nous avons signalé, dans le paragraphe III.5, que la fermeture réflexive et transitive d'une relation semi-programmable  $R$  n'est pas nécessairement programmable. Nous voudrions utiliser  $R^*$  pour représenter le programme qui itère sur  $R$ . Puisque la terminaison du programme est obtenue lorsqu'un état  $b$  pour lequel  $R(b)=\emptyset$  est atteint, il est bien évident que les calculs qui se terminent sont représentés par la relation  $\{(a,b) \mid (a,b) \in R^* \wedge R(b)=\emptyset\} = \{(a,b) \mid (a,b) \in R^* \wedge \text{exit}(b)\}^{(1)}$ . Pourtant, en général, dans les programmes itératifs il n'y a pas que des calculs qui se terminent. Il y a aussi éventuellement des calculs infinis qui doivent être représentés. La discussion du paragraphe précédent montre clairement que les états à partir desquels il y a des calculs infinis sont précisément ceux de la plus grande trajectoire. Ces considérations nous conduisent à la définition suivante.

Définition IV.5.1. Fermeture itérative. Pour une relation semi-programmable  $R$ ,  $R \in \mathcal{R}_{sp}$ , la fermeture itérative de  $R$ , notée  $R^\Theta$ , est définie par  $R^\Theta = \{(a,b) \mid (a,b) \in R^* \wedge \text{exit}(b)\} \cup \{(a,\omega) \mid \text{wpr}[R]^\times(\neg\text{exit})(a)\}$ .  $\square$

La proposition suivante nous donne une autre expression pour  $R^\Theta$ .

Proposition IV.5.2.  $R^\Theta = R^* \circ I_{\text{exit}} \cup \{a \mid \text{wpr}[R]^\times(\neg\text{exit})(a)\} \times \{\omega\}$ .  $\square$

Le résultat qui suit exprime que l'exécution répétitive de  $R$  à partir d'un état de exit est sans effet.

Proposition IV.5.3.  $I_{\text{exit}} \circ R^\Theta = I_{\text{exit}}$ .

Démonstration

$(a,b) \in I_{\text{exit}} \circ R^\Theta$  ssi  $\text{exit}(a) \wedge ((a,b) \in R^* \vee \text{wpr}[R]^\times(\neg\text{exit})(a))$   
 ssi  $\text{exit}(a) \wedge (a,b) \in R^*$  (car  $\text{exit}(a) \wedge \text{wpr}[R]^\times(\neg\text{exit})(a) = \text{faux}$ , par IV.3.4.),  
 ssi  $\text{exit}(a) \wedge (a,b) \in I_{\text{exit}}$ , (car  $R(a)=\emptyset$ ),  
 ssi  $(a,b) \in I_{\text{exit}}$ .  $\square$

En plus, si l'on répète deux fois de suite l'exécution répétitive de  $R$ , la deuxième répétition ne sert à rien :

(1) Habituellement on omet l'indice dans  $\text{exit}_R$ ,  $R$  restant implicite.

Proposition IV.5.4.  $R^{\circledast} \circ R^{\circledast} = R^{\circledast}$ .

Démonstration. Abrégeons  $I_{\text{exit}}$  par E et  $\tilde{\text{wpr}}^*[R](\neg \text{exit})$  par W. Alors

$$\begin{aligned} R^{\circledast} \circ R^{\circledast} &= (R^* \circ E \cup \underline{W} \times \{\omega\}) \circ R^{\circledast} \\ &= R^* \circ E \circ R^{\circledast} \cup \underline{W} \times \{\omega\} \circ R^{\circledast} \\ &= R^* \circ E \cup \underline{W} \times \{\omega\} \circ R^{\circledast} \\ &= R^* \circ E \cup \underline{W} \times \{\omega\}, \text{ (d'après IV.5.3., et car } R^{\circledast}(\omega) = \{\omega\}), \\ &= R^{\circledast}. \quad \square \end{aligned}$$

Remarquons à propos de cette proposition que l'opération  $\circledast$  n'est pas une fermeture au sens strict du terme car, en général, l'inclusion  $R \subseteq R^{\circledast}$  n'est pas vérifiée.

Nous montrons maintenant que  $R^{\circledast}$  est une relation programmable pourvu que R soit semi-programmable. Nous pourrions le faire en utilisant directement les définitions. Il est toutefois plus instructif de le faire à l'aide du transformateur de prédicats  $\text{wpr}[R^{\circledast}]$ . (Remarquons que  $R^{\circledast}$  est évidemment une  $\omega$ -relation). Nous commençons par le calculer.

Proposition IV.5.5. Pour tout prédicat P,  $P \in \mathcal{P}$ , on a

$$\text{wpr}[R^{\circledast}](P) = \text{wpr}[R]^{\times}(\text{exit} \Rightarrow P) \cap \text{wpr}[R]^*(\text{exit}).$$

Démonstration.

$$\begin{aligned} \text{wpr}[R^{\circledast}](P) &= \text{wpr}[R^* \circ I_{\text{exit}}](P) \cap \text{wpr}[\{a \mid \tilde{\text{wpr}}[R]^{\times}(\neg \text{exit})(a)\} \times \{\omega\}](P), \\ &\text{ par IV.5.2 et III.5.6.} \end{aligned}$$

D'une part nous avons :

$$\begin{aligned} \text{wpr}[R^* \circ I_{\text{exit}}](P) &= \text{wpr}[R^*](\text{wpr}[I_{\text{exit}}](P)) \\ \text{ où } \text{wpr}[I_{\text{exit}}](P) &= \tilde{\text{pre}}[I_{\text{exit}}](P - \Omega) = \text{exit} \Rightarrow P - \Omega, \text{ par II.4.15.4,} \\ &= \text{exit} \Rightarrow P \cap \text{exit} \Rightarrow \neg \Omega = \text{exit} \Rightarrow P, \text{ (car } \text{exit} \subseteq \neg \Omega) \\ \text{ et } \text{wpr}[R^*](\text{exit} \Rightarrow P) &= \text{wpr}[R]^{\times}(\text{exit} \Rightarrow P); \end{aligned}$$

D'autre part :

$$\begin{aligned} \text{wpr}[\{a \mid \tilde{\text{wpr}}[R]^{\times}(\neg \text{exit})(a)\} \times \{\omega\}](P) &= \tilde{\text{pre}}[\{a \mid \tilde{\text{wpr}}[R]^{\times}(\neg \text{exit})(a)\} \times \{\omega\}](P - \Omega) \\ &= \lambda b. \Omega \subseteq P - \Omega \cup \neg \tilde{\text{wpr}}[R]^{\times}(\neg \text{exit}), \text{ par II.4.15.3} \\ &= \neg \tilde{\text{wpr}}[R]^{\times}(\neg \text{exit}) = \text{wpr}[R]^*(\text{exit}). \quad \square \end{aligned}$$

La formule que nous venons de présenter montre que pour assurer la terminaison de l'exécution répétitive de R il faut et il suffit que l'état initial appartienne au plus grand invariant de R contenu dans  $\text{exit} \Rightarrow P$ , ( $\text{wpr}[R]^{\times}(\text{exit} \Rightarrow P)$ , voir IV.2.11), et au complément de la

plus grande trajectoire de R. Nous découvrons avec cette interprétation la signification profonde de  $wpr[R^\ominus]$ . Ce transformateur de prédicats peut d'ailleurs être caractérisé par une autre formule qui s'avère très utile. Pour la présenter, nous avons besoin d'une nouvelle opération unaire pour relations semi-programmables.

Définition IV.5.6. Opération d'extension par l'identité,  $\dagger$ .

Pour une relation semi-programmable R, l'extension de R par l'identité, notée  $R\dagger$ , est la relation programmable définie par  $R\dagger = R \cup I_{\text{exit}}$ .  $\square$

Les propriétés de cette opération qui nous intéressent sont les suivantes.

Proposition IV.5.7.  $R\dagger \circ R^\ominus = R^\ominus$ .

Démonstration. Abrégerons  $I_{\text{exit}}$  par E et  $\tilde{wpr}[R](\neg\text{exit})$  par W.

Alors  $R\dagger \circ R^\ominus = (R \cup E) \circ R^\ominus = R \circ R^\ominus \cup E \circ R^\ominus$

$= R \circ R^\ominus \cup E$ , (IV.5.3.) ,

$= R \circ R^* \circ E \cup R \circ \underline{W} \times \{\omega\} \cup E$

$= (R \circ R^* \cup I) \circ E \cup R \circ \underline{W} \times \{\omega\}$

$= R \circ E \cup R \circ \underline{W} \times \{\omega\}$ .

Il suffit donc de montrer que

$$R \circ \underline{W} \times \{\omega\} = \underline{W} \times \{\omega\}.$$

Faisons-le : ( $\subseteq$ ) si  $(a, \omega) \in R \circ \underline{W} \times \{\omega\}$  alors  $\exists b (a, b) \in R \wedge W(b)$ , ce qui implique  $W(a)$ , (W est la plus grande trajectoire de R) et par conséquent  $(a, \omega) \in \underline{W} \times \{\omega\}$ .

( $\supseteq$ ) Si  $(a, \omega) \in \underline{W} \times \{\omega\}$  alors  $W(a)$ , ce qui implique  $\exists b (a, b) \in R \wedge (W(b) \vee b = \omega)$ , parce que W est une trajectoire. Comme  $W(\omega) = \text{vrai}$ , on a dans tous les cas  $(a, \omega) \in R \circ \underline{W} \times \{\omega\}$ .  $\square$

Proposition IV.5.8.  $wpr[R\dagger](P) = wpr[R](P) \cap \text{exit} \Rightarrow P$ .

Démonstration.  $wpr[R\dagger](P) = wpr[R](P) \cap wpr[I_{\text{exit}}](P)$ , par IV.5.6 et III.5.6 et  $wpr[I_{\text{exit}}](P) = \text{exit} \Rightarrow P$ , (voir démonstration de IV.5.5).  $\square$

Proposition IV.5.9. J est un invariant de  $R\dagger$  ssi J est un invariant de R.

Démonstration ( $\Rightarrow$ ) trivial.

( $\Leftarrow$ )  $J \subseteq wpr[R](J) \cap \text{exit} \Rightarrow J$  est équivalent à

$J \subseteq wpr[R](J) \cap J \subseteq \text{exit} \Rightarrow J$  et  $\{J \subseteq \text{exit} \Rightarrow J\} = \tau$ .  $\square$

Proposition IV.5.10.  $R^* = R\uparrow^*$ .

Démonstration. Il est évident que  $R^* \subseteq R\uparrow^*$ . Nous montrons par induction que  $\forall i \ R\uparrow^i \subseteq \bigcup_{j=0}^i R^j$ .

$i=0$  : trivial.

$$\begin{aligned} i+1: R\uparrow^{i+1} &= (R \cup I_{\text{exit}}) \circ R\uparrow^i = R \circ R\uparrow^i \cup I_{\text{exit}} \circ R\uparrow^i \subseteq R \circ R\uparrow^i \cup R\uparrow^i \\ &\subseteq (R \circ \bigcup_{j=0}^i R^j) \cup (\bigcup_{j=0}^i R^j) = \bigcup_{j=0}^{i+1} R^j. \quad \square \end{aligned}$$

Proposition IV.5.11.  $\text{wpr}[R]^*(\text{exit}) = \text{wpr}[R\uparrow]^*(\text{exit})$ .

Démonstration. Montrons par induction que

$$\forall i \ \text{wpr}[R\uparrow]^i(\text{exit}) = \text{wpr}[R]^i(\text{exit})$$

$i=0$  : trivial.

$$\begin{aligned} i+1: \text{wpr}[R\uparrow]^{i+1}(\text{exit}) &= \text{wpr}[R\uparrow](\text{wpr}[R]^i(\text{exit})), \text{ par hypothèse d'induction,} \\ &= \text{wpr}[R]^{i+1}(\text{exit}) \cap \text{exit} \Rightarrow \text{wpr}[R]^i(\text{exit}), \text{ par IV.5.8,} \\ &= \text{wpr}[R]^{i+1}(\text{exit}), \text{ car exit étant un invariant de R} \\ &\text{(voir IV.2.6) on a par monotonie de wpr[R] que} \\ &\text{exit} \subseteq \text{wpr}[R]^i(\text{exit}). \quad \square \end{aligned}$$

Nous pouvons maintenant présenter la nouvelle formule pour  $\text{wpr}[R^\ominus]$ .

Proposition IV.5.12. Pour tout prédicat  $P$ ,  $P \in \mathcal{P}$ , on a

$$\text{wpr}[R^\ominus](P) = \text{wpr}[R\uparrow]^*(\text{exit} \cap P).$$

Démonstration. Au vu de IV.5.5, IV.5.10, III.5.9.3 et IV.5.11, il suffit de montrer que  $\text{wpr}[R\uparrow]^x(\text{exit} \Rightarrow P) \cap \text{wpr}[R\uparrow]^*(\text{exit}) =$

$\text{wpr}[R\uparrow]^*(\text{exit} \cap P)$ . On peut le faire en montrant que

$$(*) \ \forall i \ \text{wpr}[R\uparrow]^x(\text{exit} \Rightarrow P) \cap \text{wpr}[R\uparrow]^i(\text{exit}) = \text{wpr}[R\uparrow]^i(\text{exit} \cap P).$$

Procédons par induction

$$i=0: \text{wpr}[R\uparrow]^x(\text{exit} \Rightarrow P) \cap \text{exit}$$

$$= \text{wpr}[R\uparrow]^x(\text{exit} \Rightarrow P) \cap \text{wpr}[R\uparrow]^x(\text{exit}), \text{ car exit est un invariant de R}\uparrow, \\ \text{IV.5.9, IV.2.8,}$$

$$= \text{wpr}[R\uparrow]^x((\text{exit} \Rightarrow P) \cap \text{exit}) = \text{wpr}[R\uparrow]^x(\text{exit} \cap P) = \text{exit} \cap P,$$

car  $\text{exit} \cap P$  est un invariant de  $R\uparrow$ , IV.5.9, IV.2.8.

$$i+1: \text{wpr}[R\uparrow]^x(\text{exit} \Rightarrow P) \cap \text{wpr}[R\uparrow]^{i+1}(\text{exit}) =$$

$$\text{wpr}[R\uparrow]^x(\text{exit} \Rightarrow P) \cap \text{wpr}[R\uparrow](\text{wpr}[R\uparrow]^x(\text{exit} \Rightarrow P)) \cap \text{wpr}[R\uparrow]^{i+1}(\text{exit})$$

car  $\text{wpr}[R\uparrow]^x(\text{exit} \Rightarrow P)$  étant un invariant de  $R\uparrow$  on a

$$\text{wpr}[R\uparrow]^x(\text{exit} \Rightarrow P) \subseteq \text{wpr}[R\uparrow](\text{wpr}[R\uparrow]^x(\text{exit} \Rightarrow P)),$$

$$= \text{wpr}[R\uparrow]^x(\text{exit} \Rightarrow P) \cap \text{wpr}[R\uparrow](\text{wpr}[R\uparrow]^x(\text{exit} \Rightarrow P) \cap \text{wpr}[R\uparrow]^i(\text{exit}))$$

par III.4.6.1.,

$$\begin{aligned}
&= \text{wpr}[R\uparrow]^{\times}(\text{exit} \Rightarrow P) \cap \text{wpr}[R\uparrow](\text{wpr}[R\uparrow]^i(\text{exit} \cap P)) \\
&\quad \text{par hypothèse d'induction,} \\
&= \text{wpr}[R\uparrow]^{\times}(\text{exit} \Rightarrow P) \cap \text{wpr}[R\uparrow]^{i+1}(\text{exit} \cap P) \\
&= \text{wpr}[R\uparrow]^{i+1}(\text{exit} \cap P)
\end{aligned}$$

parce que

$$(**) \text{wpr}[R\uparrow]^{i+1}(\text{exit} \cap P) \subseteq \text{wpr}[R\uparrow]^{\times}(\text{exit} \Rightarrow P)$$

Pour montrer la validité de (\*\*), remarquons que l'hypothèse d'induction nous indique que

$\text{wpr}[R\uparrow]^i(\text{exit} \cap P) \subseteq \text{wpr}[R\uparrow]^{\times}(\text{exit} \Rightarrow P)$ . Les deux membres de cette inégalité sont des invariants de  $R\uparrow$ . Celui de droite étant le plus grand invariant de  $R\uparrow$  contenu dans  $\text{exit} \Rightarrow P$ , par IV.2.11, nous concluons que

$\text{wpr}[R\uparrow]^i(\text{exit} \cap P) \subseteq \text{exit} \Rightarrow P$ . Ceci implique

$$\begin{aligned}
\text{wpr}[R\uparrow]^{i+1}(\text{exit} \cap P) &\subseteq \text{wpr}[R\uparrow](\text{exit} \Rightarrow P), \text{ par III.4.6.3} \\
&\subseteq \text{exit} \Rightarrow (\text{exit} \Rightarrow P), \text{ par IV.5.8} \\
&= \text{exit} \Rightarrow P.
\end{aligned}$$

Par conséquent  $\text{wpr}[R\uparrow]^{i+1}(\text{exit} \cap P) \subseteq \text{wpr}[R\uparrow]^{\times}(\text{exit} \Rightarrow P)$

( $\text{wpr}[R\uparrow]^{i+1}(\text{exit} \cap P)$  étant un invariant, il doit être plus petit que le plus grand invariant inférieur à  $\text{exit} \Rightarrow P$ ).

Nous établissons ainsi (\*\*) et simultanément (\*).  $\square$

Il est maintenant assez facile de montrer que :

Proposition IV.5.13. Pour toute relation semi-programmable  $R$ , la relation  $R^{\ominus}$  est une relation programmable.

Démonstration.  $R^{\ominus}$  est évidemment une  $\omega$ -relation.

i.  $R^{\ominus}$  est totale.  $\text{wpr}[R^{\ominus}](\Omega) = \text{wpr}[R^{\ominus}](\perp) = \text{wpr}[R\uparrow]^*(\perp)$ , (IV.5.12),  $= \perp$ , car  $R\uparrow$  est totale (III.5.3 et III.5.2.1), ce qui implique (III.5.3) que  $R^{\ominus}$  est totale.

ii.  $R^{\ominus}$  est à non déterminisme borné.

$\text{wpr}[R^{\ominus}] = \text{wpr}[R\uparrow]^* \circ (\lambda P. P \cap \text{exit})$ . Le transformateur de prédicats  $\lambda P. P \cap \text{exit}$  est supérieurement continu.  $R\uparrow$  étant une relation programmable (cf. déf. IV.5.6), le théorème III.4.9 nous dit que  $\text{wpr}[R\uparrow]$  est supérieurement continu et d'après II.5.17  $\text{wpr}[R\uparrow]^*$  l'est aussi.

Nous concluons, avec II.5.17, que  $\text{wpr}[R^{\ominus}]$  est supérieurement continu, ce qui implique d'après le théorème III.4.9 que  $R^{\ominus}$  est à non-déterminisme borné.  $\square$

Nous terminons ce paragraphe avec une dernière caractérisation du transformateur de prédicats  $wpr[R^\ominus]$ . Elle se base sur une nouvelle opération d'extension pour relations semi-programmables :

Définition IV.5.14. L'opération de  $\omega$ -extension. Pour une relation semi-programmable  $R$ , la  $\omega$ -extension de  $R$  notée  $R\dagger$  est la relation programmable définie par  $R\dagger = R \cup \underline{\text{exit}} \times \{\omega\}$ .  $\square$

Proposition IV.5.15.  $wpr[R\dagger](P) = wpr[R](P) \cap \neg \text{exit}$ .

Démonstration : par IV.5.14, III.5.6, III.4.3 et II.4.15.3.  $\square$

Proposition IV.5.16. Pour tout prédicat  $P$ ,  $P \in \mathcal{P}$ , on a

$$wpr[R^\ominus](P) = (I \vee wpr[R\dagger])^*(P \cap \text{exit}).$$

Démonstration. Par IV.5.12 et IV.3.10.2, on a que  $wpr[R^\ominus](P) = (I \vee wpr[R\dagger])^*(P \cap \text{exit})$  car  $P \cap \text{exit}$  est un invariant de  $R$  (IV.5.9, IV.2.6). Le résultat vient directement car  $I \vee wpr[R\dagger] = I \vee wpr[R\dagger]$ , (IV.5.8, IV.5.15).  $\square$

Rappelons, en utilisant IV.3.9, que le prédicat  $(I \vee wpr[R\dagger])^*(P \cap \text{exit})$  est le complément de la plus grande trajectoire de  $R\dagger$  contenue dans  $\neg(P \cap \text{exit})$ . Cette interprétation est à notre avis moins agréable que celle de la formule de la proposition IV.5.5. Néanmoins, cette caractérisation est importante car elle correspond directement à la définition axiomatique proposée par Dijkstra [Dij76] pour le transformateur de prédicats associé à la commande répétitive. Nous reviendrons sur ce sujet dans le chapitre suivant.

## 6. INVARIANTS DE LA FERMETURE ITERATIVE

Nous savons, d'après le corollaire IV.2.9, que si un prédicat  $J$  est un invariant d'une relation semi-programmable  $R$  alors  $J$  est aussi un invariant de  $R^*$ . En réfléchissant un peu, nous constatons que la même propriété n'est pas vraie pour  $R^\ominus$ , comme le confirme l'exemple suivant.

Exemple IV.6.1. Prenons  $Q = Z \cup \{\omega\}$  et  $R = \{(a,b) \mid (a,b) \in Z \wedge b=a+1\} \cup \{(\omega,\omega)\}$ . Alors  $R^* = \{(a,b) \mid (a,b) \in Z \wedge a \leq b\} \cup \{(\omega,\omega)\}$  et  $R^\ominus = (Z \cup \{\omega\}) \times \{\omega\}$ . Par conséquent le seul invariant de  $R^\ominus$  est  $\perp$ , tandis que  $\lambda a. a \geq K$  est un invariant de  $R$  et de  $R^*$ , pour tout  $K$ .  $\square$

Dans la pratique, comme nous le verrons au chapitre V, il est utile de pouvoir décider si un prédicat  $P$  donné est invariant de  $R^\ominus$ . En principe, ceci ne pose pas plus de problèmes que n'importe quelle autre relation : il suffit de vérifier si  $P \subseteq \text{wpr}[R^\ominus](P)$ . Cependant, les expressions pour  $\text{wpr}[R^\ominus]$  dont nous disposons ne sont pas simples et rendent cette vérification souvent impraticable. Il faut d'autres techniques qui puissent remplacer la vérification directe. C'est la question que nous abordons maintenant.

Lemme IV.6.1.  $\text{wpr}[R^\ominus](\top) = \text{wpr}[R]^*(\text{exit})$ .

Démonstration. Il suffit d'utiliser IV.5.5 en remarquant que

$\text{wpr}[R]^*(\top)$  est le plus grand invariant de  $R$  et en rappelant que  $\text{wpr}[R]^*(\text{exit})$  est un invariant (IV.4.11, IV.3.11).  $\square$

Lemme IV.6.2.  $\text{wpr}[R^\ominus](P) = \text{wpr}[R]^*(\text{exit} \Rightarrow P) \cap \text{wpr}[R^\ominus](\top)$ .

Démonstration. Par IV.5.5 et IV.6.1.  $\square$

Théorème IV.6.3. Si le prédicat  $J$  est un invariant de  $R$  alors  $J \cap \text{wpr}[R^\ominus](\top) \subseteq \text{wpr}[R^\ominus](J)$ .

Démonstration. Nous avons, trivialement,  $J \subseteq \text{exit} \Rightarrow J$ .

$\text{wpr}[R]^*(\text{exit} \Rightarrow J)$  étant le plus grand invariant inférieur à  $\text{exit} \Rightarrow J$ , (IV.2.11), il est vrai que  $J \subseteq \text{wpr}[R]^*(\text{exit} \Rightarrow J)$ . Ceci implique  $J \cap \text{wpr}[R^\ominus](\top) \subseteq \text{wpr}[R]^*(\text{exit} \Rightarrow J) \cap \text{wpr}[R^\ominus](\top)$   
 $= \text{wpr}[R^\ominus](J)$ , par IV.6.2.  $\square$

Corollaire IV.6.4. Si le prédicat  $J$  est un invariant de  $R$  alors le prédicat  $J \cap \text{wpr}[R^\Theta](\tau)$  est un invariant de  $R^\Theta$ .

Démonstration. Le théorème IV.6.3. implique :

$$J \cap \text{wpr}[R^\Theta](\tau) \subseteq \text{wpr}[R^\Theta](J \cap \text{wpr}[R^\Theta](\tau)).$$

D'autre part  $R^\Theta = R^\Theta \circ R^\Theta$ , (IV.5.4), d'où, par III.5.9.1

l'inégalité ci-dessus peut se réécrire

$$\begin{aligned} J \cap \text{wpr}[R^\Theta](\tau) &\subseteq \text{wpr}[R^\Theta](J \cap \text{wpr}[R^\Theta](\text{wpr}[R^\Theta](\tau))) \\ &= \text{wpr}[R^\Theta](J \cap \text{wpr}[R^\Theta](\tau)), \text{ par III.4.6.1. } \quad \square \end{aligned}$$

Corollaire IV.6.5. Si le prédicat  $J$  est un invariant de  $R$  et  $J \subseteq \text{wpr}[R^\Theta](\tau)$  alors  $J$  est un invariant de  $R^\Theta$ .  $\square$

Remarque. Quand on parle d'invariants dans le contexte d'un programme répétitif  $R^\Theta$ , on veut en général signifier les invariants de  $R$  et non les invariants de  $R^\Theta$ . C'est ce que nous faisons dans la suite.  $\square$

Pour pouvoir utiliser le théorème IV.6.3. et ses corollaires, il faut calculer  $\text{wpr}[R^\Theta](\tau)$ , ou, au moins réussir à montrer qu'un invariant  $J$  donné est tel que  $J \subseteq \text{wpr}[R^\Theta](\tau)$ . Pour ce faire, la technique courante consiste à chercher une fonction partielle de l'espace d'états vers l'ensemble  $\mathbb{N}$  des entiers naturels, telle que l'exécution d'une itération dans le programme répétitif provoque certainement une diminution effective de la valeur de la fonction, pourvu que l'exécution soit initialisée dans un état de  $J$ . Dans ces conditions, la valeur de la fonction ne pouvant diminuer indéfiniment, il est clair intuitivement que l'exécution répétitive doit se terminer, i.e.  $J \subseteq \text{wpr}[R^\Theta](\tau)$ . Cet argument s'applique encore si au lieu de  $\mathbb{N}$  on prend un autre ensemble bien fondé (définition ci-dessous).

Pour formaliser cette technique, il nous faut un nouvel outil. Rappelons que  $\text{wpr}[R](P)$  est la plus faible pré-condition garantissant la terminaison de  $R$  dans  $P$ . Rien n'est dit sur la "manière" dont le prédicat  $P$  est atteint. Si on veut placer des restrictions sur ceci, on peut les décrire par des prédicats binaires (ou par des relations binaires), sur l'espace d'états. Un tel prédicat binaire peut être appelé une transcondition et notre problème est maintenant de calculer la plus faible précondition assurant que l'exécution de  $R$  respecte une transcondition donnée.



Exemple IV.6.2. Soit  $Q = \mathbb{Z} \cup \{\omega\}$  et  $R = \{(a,b) \mid a,b \in \mathbb{Z} \wedge b=a+5\} \cup \{(\omega,\omega)\}$ . Supposons que l'on veuille que, par exécution de  $R$ , la valeur de l'état courante augmente au moins deux fois. La transcondition qui exprime ceci est  $\lambda \langle a,b \rangle . b \geq 2a$ . La plus faible précondition garantissant le respect de la transcondition est  $\lambda a . a \leq 5$ .  $\square$

Présentons les définitions formelles dont nous avons besoin.

L'ensemble des prédicats binaires sur  $Q$  est noté  $PP$ ,  
 $PP = Q \times Q \rightarrow \{\text{vrai}, \text{faux}\}$ .

Définition IV.6.6. Le transformateur de prédicats  $wpr[R]$ .

Pour  $R \in \mathcal{R}_{sp}$  on définit  $wpr[R] \in PP \rightarrow P$  par  
 $wpr[R](PP) = \lambda a . \forall b (a,b) \in R \Rightarrow PP(a,b) \wedge b \neq \omega$ .  $\square$

Définition IV.6.7. Relation d'ordre bien fondé  $\prec$  sur un ensemble  $X$ .

On dit qu'une relation d'ordre partiel  $\prec$  sur un ensemble  $X$  est bien fondée si il n'y a pas de suites strictement croissantes d'éléments de  $X$ ,  $x_0 \prec x_1 \prec \dots \prec x_i \prec \dots$  (où  $x \prec y$  ssi  $x \prec y \wedge x \neq y$ ), i.e. si  $\forall x \in X \exists i \in \mathbb{N} \prec^i(x) = \emptyset$ .  $\square$

Parfois, à la place des relations d'ordre bien fondé on préfère les relations noëthériennes :

Définition IV.6.8. Relation noëthérienne  $\epsilon$  sur un ensemble  $X$ .

On dit qu'une relation binaire  $\epsilon$  sur un ensemble  $X$  est une relation noëthérienne si il n'y a pas de suites infinies  $x_0 \epsilon x_1 \epsilon \dots \epsilon x_i \epsilon \dots$  i.e., si  $\forall x \in X \exists i \in \mathbb{N} \epsilon^i(x) = \emptyset$ .

On a, évidemment, la propriété suivante, Huet [Hue77] :

Proposition IV.6.9. Si  $\epsilon$  est une relation noëthérienne alors  $\epsilon^*$  est une relation d'ordre bien fondé.  $\square$

Le théorème suivant est fondamental.

Théorème IV.6.10. Soient  $R$  une relation semi-programmable,  $J$  un invariant de  $R$  et  $\prec$  une relation d'ordre bien fondé sur  $Q$ . Soit  $PP$  le prédicat binaire défini par  $PP(a,b) = a \prec b$ . Dans ces conditions

si

$$(*) \quad J \subseteq \text{wpr}[R](PP)$$

alors

$$(**) \quad J \subseteq \text{wpr}[R^\emptyset](T).$$

Démonstration. Remarquons tout d'abord que  $\text{wpr}[R^\emptyset](T)$  est le complément de la plus grande trajectoire de  $R$ ; par IV.6.1, IV.4.9 et IV.4.10, nous avons

$$(+)$$

$$\text{wpr}[R^\emptyset](T) = \lambda a. \neg \exists i \in \mathbb{N} R^i(a) = \emptyset.$$

Supposons  $(*)$  est vraie et soit  $a$  tel que  $J(a)$ . Alors

$$\forall b (a, b) \in R \Rightarrow a \prec b \wedge b \neq \omega, \text{ par (IV.6.6), d'où } R(a) \subseteq \prec(a).$$

Montrons maintenant, par induction, que  $R^i(a) \subseteq \prec^i(a)$ , pour tout  $i$  :

$i=0$  : trivial.

$i+1$  :  $R^i(a) \subseteq \prec^i(a)$ , par hypothèse d'induction ;

si  $a' \in R^i(a)$  alors  $J(a')$ , par IV.2.7, et donc  $R(a') \subseteq \prec(a')$ ,

ce qui implique

$$R^{i+1}(a) \underset{a' \in R^i(a)}{\cup} R(a') \subseteq \underset{a' \in R^i(a)}{\cup} \prec(a') \subseteq \underset{a' \in \prec^i(a)}{\cup} \prec(a') = \prec^{i+1}(a)$$

$\prec$  étant un ordre bien fondé, nous avons  $\neg \exists i \in \mathbb{N} \prec^i(a) = \emptyset$ , ce qui implique  $\neg \exists i \in \mathbb{N} R^i(a) = \emptyset$ . Nous concluons à l'aide de (+) que  $J(a) \Rightarrow \text{wpr}[R^\emptyset](T)(a)$ , ce qui établit (\*\*).  $\square$

Corollaire IV.6.11 (Le théorème de la fonction décroissante).

Soient  $R$  une relation semi-programmable et  $J$  un invariant de  $R$ .

Soit  $f$  une fonction de  $Q \rightarrow \mathbb{N}$ . Soit  $PP$  le prédicat binaire défini par

$$PP(a, b) = \begin{cases} f(a) > f(b) & \text{si } f(a) \text{ et } f(b) \text{ sont définis} \\ \text{faux} & \text{sinon.} \end{cases}$$

Dans ces conditions, si  $J \subseteq \text{wpr}[R](PP)$  alors  $J \subseteq \text{wpr}[R^\emptyset](T)$ .

Démonstration : utiliser le théorème avec l'ordre bien fondé  $\prec$  dans  $Q$ , défini par  $a \prec b$  si  $f(a)$  et  $f(b)$  sont définies et  $f(a) > f(b)$  ou  $a=b$ .  $\square$

La fonction  $f$  du corollaire précédent est dite décroissante sous l'invariant  $J$ .

Corollaire IV.6.12. Pour une relation semi-programmable  $R$ , on a

$$\text{wpr}[R^{\Theta}](\tau) = \neg\Omega \text{ ssi il existe une relation noéthérienne } \varepsilon \text{ sur } Q - \{\omega\} \text{ tel que } R = \varepsilon \cup \{\omega, \omega\}.$$

Démonstration. ( $\Rightarrow$ ) D'après l'égalité (+) dans la démonstration du théorème IV.6.10.  $\text{wpr}[R^{\Theta}](\tau) = \neg\Omega$  implique  $\forall a \in Q - \{\omega\} \exists i \in \mathbb{N} R^i(a) = \emptyset$  et aussi  $\forall a \in Q - \{\omega\} \omega \notin R(a)$  car  $R$  est semi-programmable, et  $\forall i \in \mathbb{N} R^i(\omega) = \{\omega\}$ . Il suffit donc de prendre  $\varepsilon = R - \{\omega, \omega\}$ .

( $\Leftarrow$ )  $\neg\Omega$  est un invariant de  $R$ , par construction de  $R$ . Soit l'ordre bien fondé  $\prec = \varepsilon^*$ . Nous avons

$$\begin{aligned} a \neq \omega &\Rightarrow (\forall b (a, b) \in R \Rightarrow b \neq \omega) \\ &\Rightarrow (\forall b (a, b) \in R \Rightarrow (a, b) \in R \wedge b \neq \omega) \\ &\Rightarrow (\forall b (a, b) \in R \Rightarrow a \prec b \wedge b \neq \omega) \end{aligned}$$

i.e.  $\neg\Omega \subseteq \text{wwpr}[R](\lambda \langle a, b \rangle . a \prec b)$ .

D'où, par le théorème  $\neg\Omega \subseteq \text{wpr}[R^{\Theta}](\tau)$ . Mais par III.5.7.  $\text{wpr}[R^{\Theta}](\tau) \subseteq \neg\Omega$ , ce qui entraîne le résultat cherché.  $\square$

Dans la pratique, ce corollaire signifie que si chaque répétition  $R$  d'une exécution répétitive  $R^{\Theta}$  se termine, alors l'exécution répétitive elle-même se termine certainement, pour tout état initial différent de  $\omega$ , si et seulement si la relation  $R - \{(\omega, \omega)\}$  est noéthérienne.

Nous terminons ce paragraphe en présentant le résultat sur lequel se basent les techniques de preuve de correction totale de programmes répétitifs.

Proposition IV.6.13. Si  $J$  est un invariant de  $R^{\Theta}$  alors

$$J \subseteq \text{wpr}[R^{\Theta}](J \cap \text{exit}).$$

Démonstration. Triviale car, par IV.5.5

$$\text{wpr}[R^{\Theta}](J) = \text{wpr}[R^{\Theta}](J \cap \text{exit}). \quad \square$$

Cette proposition signifie que si on arrive à prouver qu'un invariant  $J$  donné est aussi un invariant de  $R^{\Theta}$ , alors on peut garantir que si l'exécution répétitive commence dans un état de  $J$  alors elle se termine dans un état de  $J \cap \text{exit}$ . D'habitude on procède dans un ordre légèrement différent : on cherche d'abord un invariant  $J$  tel que  $J \cap \text{exit}$  implique la post-condition souhaitée et on se préoccupe ensuite de montrer que  $J$  est aussi un invariant de  $R^{\Theta}$ .

## 7. CONCLUSION

Nous terminons ainsi l'étude abstraite de la théorie relationnelle que nous proposons. Nous sommes partis du concept élémentaire de relation binaire dans un ensemble (chapitre II), nous avons discuté de la manière de traiter ce concept pour qu'il serve à représenter le comportement d'un programme, nous l'avons habillé de transformateurs de prédicats qui expriment des besoins courants en Programmation. Les propriétés intéressantes de ces transformateurs de prédicats ont été étudiées en détail et de manière progressive (chapitre III). Finalement, nous avons présenté le concept central d'invariant, et celui de trajectoire, et exposé le principe de leur utilisation dans les preuves de programmes (chapitre IV).

Maintenant, pour que ces résultats soient effectivement utilisables sur des programmes, il faut avoir un moyen de calculer les relations associées à ces programmes. Autrement dit, il faut avoir une sémantique relationnelle du langage de programmation utilisé. Ceci est précisément l'objet du chapitre suivant. Le langage étudié est le langage de commandes gardées de Dijkstra [Dij75][Dij76]. Il s'agit d'un langage qui, tout en restant simple, nous permet d'exprimer de manière relativement concise les programmes auxquels nous nous intéressons et d'exploiter à fond la théorie présentée.



CHAPITRE V

SEMANTIQUE RELATIONNELLE DU LANGAGE  
DES COMMANDES GARDEES



## 1. INTRODUCTION

Dans ce chapitre nous utilisons la théorie développée précédemment pour présenter une sémantique relationnelle du langage des commandes gardées de Dijkstra [Dij75], [Dij76]. Cette sémantique décrit d'une manière assez agréable l'intuition expliquée par Dijkstra pour les différentes commandes du langage. Naturellement, elle est compatible avec la sémantique formelle originale, définie axiomatiquement pour chaque commande, en termes d'un transformateur de prédicats de correction totale (la fonction  $wp$ ). Notre approche consiste plutôt à définir les relations programmables associées à chaque commande et d'en calculer le transformateur de prédicats  $wpr$ . En procédant ainsi nous retrouvons la définition originale, ce qui montre sa cohérence, étant donné que nous en exhibons un modèle. En même temps, nous constatons que le transformateur de prédicats  $wpr$  correspond, dans ce modèle, à la fonction  $wp$  de Dijkstra. Nous concluons que notre sémantique est bonne, d'un point de vue formel, ainsi que d'un point de vue intuitif. En réalité, la sémantique relationnelle est plus forte que la sémantique par transformateur de prédicats de correction totale. En effet, comme nous l'avons signalé dans le paragraphe III.4, celle-ci n'est pas capable d'exprimer à la fois la possibilité de terminaison et de non-terminaison pour un programme, à partir d'un état donné.

Notre approche peut être comparée à celle de Hoare [Hoa78a], pour traiter ce même problème de construction d'une sémantique du langage des commandes gardées, sémantique avec laquelle on puisse établir la cohérence de la définition de Dijkstra. Hoare propose une définition dite "mécanistique"<sup>(1)</sup> du langage, en associant à chaque commande un ensemble de traces. Une trace est une suite finie d'affectations et conditions, dont le rôle est de décrire une exécution possible de la commande. Bien que les traces soient un concept usuel en Programmation, leur manipulation formelle n'est pas toujours simple, ce qui donne à l'approche "mécanistique" une certaine lourdeur, facilement évitable dans l'approche relationnelle.

---

(1) Traduction libre du mot anglais "mechanistic", utilisé par Hoare.



Néanmoins, le fait qu'un langage admette plusieurs sémantiques différentes, compatibles les unes avec les autres, peut être considéré un signe de sa bonne formation (ou "régularité"), comme l'observent Hoare et Lauer [HoL74].

Dans la suite, nous verrons comment les résultats de ce chapitre peuvent être utilisés pour étudier les propriétés des programmes du langage des commandes gardées, en particulier la correction. Nous verrons aussi comment généraliser notre approche de façon à donner une sémantique relationnelle d'un langage de programmation parallèle inspiré du langage des commandes gardées.

## 2. SYNTAXE

La syntaxe abstraite du langage des commandes gardées est la suivante :

$$\begin{aligned}
 p &::= r \\
 r &::= \text{skip} | \text{abort} | e | \text{if } s \text{ fi} | \text{do } s \text{ od} | r_1 ; r_2 \\
 s &::= c_1 \square c_2 \square \dots \square c_n \quad (n \geq 0) \\
 c &::= g \rightarrow r
 \end{aligned}$$

$p$  est un programme. Son espace d'états est noté  $Q$  (avec  $\omega \in Q$ ).  $r$  désigne une commande.  $\text{skip}$  et  $\text{abort}$  sont des commandes particulières.  $e$  est une affectation.  $r_1 ; r_2$  est la composition séquentielle de  $r_1$  et  $r_2$ .  $g$  est appelée garde,  $c$  commande gardée et  $s$  ensemble de commandes gardées.

Un programme est un mécanisme abstrait qui évolue dans son espace d'états. Son évolution est le résultat des changements d'état provoqués par les actions entreprises. Une action consiste en l'exécution d'une commande. Si une action se termine alors son effet pour le programme est précisément le passage de l'état courant à un nouvel état courant, ce passage pouvant se produire de manière non-déterministe. Si une action ne se termine pas, nous disons que le programme passe de son état courant à l'état  $\omega$ . (L'état  $\omega$  est donc le "résultat" d'une action infinie).

En pratique, on utilise des variables pour représenter l'état courant du programme. Nous pouvons faire abstraction des variables effectivement employées et ne considérer qu'une seule variable généralisée, correspondant à la "concaténation" de toutes les autres. Dans la suite, nous notons  $x$  la variable généralisée du programme  $p$ .

Les gardes sont des prédicats sur  $Q$ . Sans perte de généralité, nous les supposons toujours définies. Pour des raisons techniques, que nous essayerons de justifier intuitivement un peu plus tard, nous définissons toujours  $g(\omega) = \text{vrai}$ . En réalité, dans les programmes, une garde  $g$  est écrite sous la forme d'une expression  $g(x)$ , portant sur la variable  $x$  du programme.

Une affectation est une application de  $Q \rightarrow Q$ . Nous devons évidemment définir toujours  $e(\omega) = \omega$ . Les affectations s'écrivent habituellement sous la forme  $x := e(x)$ , où  $e(x)$  est une expression portant sur  $x$ .

L'utilisation des gardes en tant que prédicats et des affectations en tant qu'applications signifie que nous travaillons directement sur les interprétations des expressions apparaissant effectivement dans les programmes, comme d'ailleurs le fait Dijkstra. Nous faisons abstraction de la syntaxe particulière des expressions du langage pour ne nous intéresser qu'aux objets mathématiques qu'elles représentent. Sinon il nous faudrait décrire également la sémantique des expressions. Dans la bonne tradition de l'Informatique, nous laissons ce travail à un "oracle".

### 3. SEMANTIQUE

D'après la syntaxe, un programme du langage des commandes gardées est une commande. Pour donner la sémantique du langage, il suffit de préciser la signification de chacune des commandes. Nous le faisons en définissant une fonction sémantique  $\rho$  qui associe à chaque commande la relation programmable qu'elle représente.

Outre les commandes, il existe dans le langage des constructions qui, sans être des commandes servent à les construire : les gardes, les commandes gardées et les ensembles de commandes gardées. Ces constructions ont une signification relationnelle en elles-mêmes, que nous décrivons à l'aide de fonctions sémantiques particulières. Ainsi pour les gardes nous aurons la fonction mu, pour les commandes gardées la fonction gamma, et pour les ensembles de commandes gardées la fonction sigma. Leurs arguments n'étant pas des commandes, ces trois fonctions sémantiques fournissent en général des relations qui ne sont pas totales. Néanmoins, nous verrons qu'elles sont toujours des relations semi-programmables.

i. skip

skip est la commande qui ne fait rien, qui ne provoque aucun changement d'état. Elle est décrite par

$$\begin{aligned} \text{rho} \llbracket \text{skip} \rrbracket &= \{(a, a) \mid a \in Q\} \\ &= I \end{aligned}$$

$\text{rho} \llbracket \text{skip} \rrbracket$  est évidemment une relation programmable.

ii. abort

L'exécution de la commande abort ne se termine jamais, quel que soit l'état initial ;

$$\begin{aligned} \text{rho} \llbracket \text{abort} \rrbracket &= \{(a, \omega) \mid a \in Q\} \\ &= Q \times \{\omega\} \end{aligned}$$

Il s'agit clairement d'une relation programmable.

iii. affectation

L'exécution d'une affectation e consiste en un changement d'état effectué de manière déterministe :

$$\text{rho} \llbracket e \rrbracket = e$$

e étant une application de  $Q \rightarrow Q$ , tel que  $e(\omega) = \omega$ ,  $\text{rho} \llbracket e \rrbracket$  est une relation programmable.

## iv. garde

Une garde  $g$  sert à filtrer certains états, les états pour lesquels la garde est vraie :

$$\begin{aligned} \text{mu}[[g]] &= \{(a, a) \mid g(a)\} \\ &= I_g \end{aligned}$$

Etant donné que  $g(\omega) = \text{vrai}$ ,  $\text{mu}[[g]]$  est une relation semi-programmable.

## v. commande gardée

Une commande gardée  $g \rightarrow r$  n'est exécutable qu'à partir d'un état validant  $g$ , auquel cas son exécution consiste en l'exécution de  $r$  :

$$\text{gamma}[[g \rightarrow r]] = \text{mu}[[g]] \circ \text{rho}[[r]].$$

En prenant comme hypothèse d'induction sur la structure du langage que  $\text{rho}[[r]]$  est une relation programmable, nous concluons que  $\text{gamma}[[g \rightarrow r]]$  est une relation semi-programmable.

## vi. ensemble de commandes gardées

L'"effet" d'un ensemble de commandes gardées est simplement l'union ensembliste des "effets" des commandes gardées qui le composent :

$$\text{sigma} [[c_1 \square c_2 \square \dots \square c_n]] = \bigcup_{i=1}^n \text{gamma} [[c_i]], \text{ si } n > 0.$$

Si  $n=0$ , nous définissons

$$\text{sigma} [[]] = \{(\omega, \omega)\}.$$

Dans ces conditions, si l'on prend comme hypothèse d'induction que tous les  $\text{gamma}[[c_i]]$  sont des relations semi-programmables, on peut conclure que  $\text{sigma}[[s]]$  est aussi une relation semi-programmable (même si  $n=0$ ).

vii. if s fi

La sémantique intuitive de cette commande peut être résumée dans ces deux phrases de Dijkstra : "One of the guarded commands whose guard is true is selected and its statement list is executed. (...)

Activation in an initial state such that none of the guards is true will lead to abortion", <sup>(1)</sup> [Dij76, pp. 33-34]. En remarquant que,

(1) "Une des commandes gardées dont la garde est vraie est choisie et sa liste d'instructions est exécutée. (...) Si l'état initial est tel

par hypothèse d'induction  $\sigma[[s]]$  est une relation semi-programmable, nous résumons l'idée de Dijkstra par

$$\rho[[\text{if } s \text{ fi}]] = \sigma[[s]] \downarrow$$

(cf. la définition IV.5.14).

Par conséquent  $\rho[[\text{if } s \text{ fi}]]$  est bien une relation programmable.

#### viii. do s od

Dijkstra explique le fonctionnement de cette commande comme suit : "Upon activation the guards are inspected. The activity terminates if there are no true guards ; if there are true guards one of the corresponding statement lists is activated and upon termination the implementation starts over again inspecting the guards"<sup>(1)</sup>, [Dij76, p.35]. On peut penser que les calculs infinis ne sont considérés que de manière implicite (comparons avec if s fi où l'on précisait bien dans quelles conditions l'exécution ne se termine pas). Nous proposons :

$$\rho[[\text{do s od}]] = \sigma[[s]]^{\circ}$$

(cf. la définition IV.5.1).

$\sigma[[s]]$  étant, par hypothèse d'induction, une relation semi-programmable,  $\rho[[\text{do s od}]]$  est une relation programmable (cf. proposition IV.5.13),

#### ix. $r_1 ; r_2$

Ce cas est trivial :

$$\rho[[r_1 ; r_2]] = \rho[[r_1]] \circ \rho[[r_2]]$$

Si, par hypothèse d'induction  $\rho[[r_1]]$  et  $\rho[[r_2]]$  sont des relations programmables  $\rho[[r_1 ; r_2]]$  l'est aussi (cf. III.5.11).

(fin de la définition de la sémantique).  $\square$

---

qu'aucune des gardes n'est vraie, l'élaboration de la commande échoue".

(1) "Lors du début de l'élaboration les gardes sont examinées. L'élaboration termine s'il n'y a pas des gardes vraies ; s'il y a des gardes vraies une des listes d'instructions associées est élaborée, et dès que cette élaboration termine l'implémentation recommence, en examinant les gardes de nouveau".

Pour illustrer une utilisation possible de ces définitions, montrons que if fi et do od sont sémantiquement équivalents à (i.e., ont la même relation sémantique que), respectivement, abort et skip.

Nous écrivons if fi = abort et do od = skip. En effet nous avons :

- i.  $\text{rho}[\text{if fi}] = \{(\omega, \omega)\}^\dagger = \{(a, \omega)\} = \text{rho}[\text{abort}]$  ;
- ii.  $\text{rho}[\text{do od}] = \{(\omega, \omega)\}^\ominus$ , et  $\text{exit} = \neg\Omega$  et  $\text{wpr}[\{(\omega, \omega)\}^\times](\Omega) = \Omega$ ,

d'où

$$\{(\omega, \omega)\}^\ominus = \{(a, a) \mid a \neq \omega\} \cup \{(\omega, \omega)\} = I = \text{rho}[\text{skip}].$$

#### 4. LES TRANSFORMATEURS DE PREDICATS ASSOCIES AUX RELATIONS SEMANTIQUES

Nous calculons maintenant les transformateurs de prédicats associés aux relations sémantiques définies dans la section précédente, ce qui nous permet de montrer la cohérence des axiomes de Dijkstra.

- i.  $\text{wpr}[\text{rho}[\text{skip}]](P) = \text{wpr}[I](P) = P - \Omega$ , (III.4.3 et II.4.15.4)
- ii.  $\text{wpr}[\text{rho}[\text{abort}]](P) = \text{wpr}[Q \times (\omega)](P) = \perp$ , (III.4.3 et II.4.15.3)
- iii.  $\text{wpr}[\text{rho}[e]](P) = \text{wpr}[e](P) = \lambda a. \{e(a)\} \subseteq P - \Omega$ , (III.4.5)  
 $= \lambda a. P(e(a)) \wedge \neg\Omega(e(a)) = P \circ e - \Omega \circ e$ .
- iv.  $\text{wpr}[\text{mu}[g]](P) = \text{wpr}[I_g](P) = g \Rightarrow P - \Omega$  (III.4.3 et II.4.15.4)
- v.  $\text{wpr}[\text{gamma}[g \rightarrow r]](P) = \text{wpr}[\text{mu}[g], \text{rho}[r]](P) =$   
 $= \text{wpr}[\text{mu}[g]](\text{wpr}[\text{rho}[r]](P))$ , (III.5.9.1)  
 $= g \Rightarrow \text{wpr}[\text{rho}[r]](P) - \Omega$ , (iv. ci-dessus)  
 $= g \Rightarrow \text{wpr}[\text{rho}[r]](P)$  (III.5.8)
- vi.  $\text{wpr}[\text{sigma}[c_1 \square c_2 \square \dots \square c_n]](P) = \text{wpr}[\bigcup_{i=1}^n \text{gamma}[c_i]](P) =$   
 $= \bigcap_{i=1}^n \text{wpr}[\text{gamma}[c_i]](P)$  (III.5.6)  
 $= \bigcap_{i=1}^n g_i \Rightarrow \text{wpr}[\text{rho}[r_i]](P)$  (v. ci-dessus)

Ces calculs sont valables si  $n \geq 1$ . Si  $n=0$  alors  
 $wpr[\sigma[\ ]](P) = wpr[\{(\omega, \omega)\}](P) = \Omega \Rightarrow p-\Omega = \Omega \Rightarrow P$  (III.4.3 et II.4.15.4)

$$\begin{aligned} \text{vii. } wpr[\rho[\text{if } s \text{ fi}]](P) &= wpr[\sigma[s] \uparrow](P) \\ &= wpr[\sigma[s]](P) \cap \neg \text{exit}, \quad (\text{IV.5.15}) \\ &= \neg \text{exit} \cap \bigcap_{i=1}^n g_i \Rightarrow wpr[\rho[r_i]](P) \end{aligned}$$

(Ceci est correct pour  $n \geq 1$ . Si  $n=0$  alors  $wpr[\rho[\text{if } s \text{ fi}]](P)=1$ ).

Par rapport à  $\sigma[s]$  nous avons :

$$\begin{aligned} \text{exit} &= \lambda a. \{ \sigma[s](a) = \phi \} = \lambda a. \{ \bigcup_{i=1}^n (\mu[g_i] \circ \rho[r_i])(a) = \phi \} \\ &= \lambda a. \{ \bigcup_{i=1}^n \mu[g_i](a) = \phi \}, \text{ car } \rho[r_i] \text{ est totale} \\ &= \lambda a. \{ \bigcup_{i=1}^n I_{g_i}(a) = \phi \} = \lambda a. \{ I_{\bigcup_{i=1}^n g_i}(a) = \phi \} = \neg \bigcup_{i=1}^n g_i \end{aligned}$$

(Comme dans le cas précédent, ceci n'est correct que pour  $n \geq 1$ .  
 Pour  $n=0$ ,  $\text{exit} = \neg \Omega$ ).

$$\begin{aligned} \text{viii. } wpr[\rho[\text{do } s \text{ od}]](P) &= wpr[\sigma[s]^\oplus](P) \\ &= wpr[\sigma[s]]^{\times}(\text{exit} \Rightarrow P) \cap wpr[\sigma[s]]^*(P), \quad (\text{IV.5.5}) \\ &= wpr[\sigma[s] \uparrow]^*(\text{exit} \cap P), \quad (\text{IV.5.12}) \\ &= (I \vee wpr[\sigma[s] \uparrow])^*(\text{exit} \cap P), \quad (\text{IV.5.16}) \\ &= (I \vee wpr[\rho[\text{if } s \text{ fi}]])^*(\text{exit} \cap P), \quad (\text{vii, ci-dessus}). \end{aligned}$$

(Pour  $\sigma[s] \uparrow$  nous avons, par IV.5.8,  
 $wpr[\sigma[s] \uparrow](P) = \text{exit} \Rightarrow P \cap wpr[\sigma[s]](P)$ ).

$$\begin{aligned} \text{ix. } wpr[\rho[r_1; r_2]](P) &= wpr[\rho[r_1] \circ \rho[r_2]](P), \quad (\text{III.5.9.1}) \\ &= wpr[\rho[r_1]](wpr[\rho[r_2]](P)) \end{aligned}$$

(fin du calcul des transformateurs de prédicats  $wpr$  des relations sémantiques).  $\square$

Pour montrer la cohérence de la définition axiomatique de Dijkstra, il suffit de comparer les formules que nous avons obtenues pour  $wpr[\rho[r]](P)$  avec les axiomes pour  $wp(r,P)$  proposés par Dijkstra. Dans tous les cas, sauf skip et do s od, la correspondance est immédiate. Pour skip nous trouvons  $wp(\text{skip},P) = P$  tandis que d'après i.  $wpr[\rho[\text{skip}]](P) = P \wedge \Omega$ . Cette différence n'est pas significative car elle est due au fait que Dijkstra ne considère pas dans ses définitions l'état  $\omega$ . Quant à do s od Dijkstra donne l'axiome

$$wp(\underline{\text{do s od}}, P) = \bigcup_{i \in \mathbb{N}} H_i(P), \text{ où}$$

$$H_0(P) = \text{exit} \cap P \text{ et}$$

$$H_i(P) = H_0(P) \cup wp(\underline{\text{if s fi}}, H_{i-1}(P)), \text{ pour } i \geq 1$$

qui ne ressemble à aucune de nos formules pour  $wpr[\rho[\underline{\text{do s od}}]]$  <sup>(1)</sup>. Néanmoins nous pouvons obtenir pour  $wpr[R^\Theta]$  une formule dans laquelle la ressemblance devient évidente.

Proposition V.4.1. Soit  $R$  une relation semi-programmable,  $R \in \mathcal{R}_{sp}$ , et  $\{H_i(P)\}_{i \in \mathbb{N}}$  une suite de prédicats définie par récurrence :

$$H_0(P) = \text{exit} \cap P$$

$$H_{i+1}(P) = H_0(P) \cup wpr[R+](H_i(P))$$

Alors

$$wpr[R^\Theta](P) = \bigcup_{i \in \mathbb{N}} H_i(P).$$

Démonstration. Montrons d'abord que  $\{H_i(P)\}$  est une suite croissante, i.e., pour tout  $i$   $H_i(P) \subseteq H_{i+1}(P)$

$i=0$ : trivial

$i+1$ :  $H_{i+1}(P) = H_0(P) \cup wpr[R+](H_i(P)) \subseteq H_0(P) \cup wpr[R+](H_{i+1}(P))$   
(par hypothèse d'induction  $H_i(P) \subseteq H_{i+1}(P)$ , et III.4.6.3),  
 $= H_{i+2}(P)$ .

Rappelant IV.5.16, pour établir la proposition il suffit de montrer que pour tout  $i$   $H_i(P) = (I \vee wpr[R+])^i(\text{exit} \cap P)$ .

(1) Selon Dijkstra,  $H_i(P)$  est "the weakest precondition such that the do-od construct will terminate after at most  $i$  selections of a guarded command leaving the system in a final state satisfying the post-condition  $P$ ", [Dij76, p.35], ["la plus faible pre-condition telle que la commande do-od terminera après au plus  $i$  choix d'une commande gardée, laissant le système dans un état final satisfaisant la post-condition  $P$ "].



$i=0$  : trivial

$$\begin{aligned}
 i+1 &: (I \vee \text{wpr}[R\downarrow])^{i+1}(\text{exit} \cap P) = (I \vee \text{wpr}[R\downarrow])((I \vee \text{wpr}[R\downarrow])^i(\text{exit} \cap P)) \\
 &= (I \vee \text{wpr}[R\downarrow])(H_i(P)), \text{ (hypothèse d'induction)} \\
 &= H_i(P) \cup \text{wpr}[R\downarrow](H_i(P)) \\
 &= H_i(P) \cup H_0(P) \cup \text{wpr}[R\downarrow](H_i(P)), \text{ (car } H_0(P) \subseteq H_i(P), \forall i) \\
 &= H_i(P) \cup H_{i+1}(P), \text{ (par def de } H_{i+1}(P)) \\
 &= H_{i+1}(P), \text{ car } H_i(P) \subseteq H_{i+1}(P). \quad \square
 \end{aligned}$$

Nous pouvons donc ajouter à la liste de formules viii. la formule suivante :

$$\text{viii.bis } \text{wpr}[\text{rho}[\underline{\text{do s od}}]](P) = \bigcup_{i \in \mathbb{N}} H_i(P)$$

$$\text{où } H_0(P) = \text{exit} \cap P$$

$$\text{et } H_{i+1}(P) = H_0(P) \cup \text{wpr}[\text{rho}[\underline{\text{if s fi}}]](H_i(P)). \quad \square$$

Si l'on élargissait la définition de wp de façon à admettre comme argument les prédicats sur Q (à la place des prédicats sur  $Q - \{\omega\}$ ), on aurait en fait l'égalité  $\text{wp}(r, P) = \text{wpr}[\text{rho}[\underline{\text{r}}]](P)$ . Cette équation peut être interprétée comme un théorème dont la démonstration que nous venons de décrire, ou comme une définition formelle de  $\text{wp}(r, P)$ .

Dorénavant, nous représentons les commandes if s fi et do s od respectivement par IF et DO, et le prédicat exit est considéré par rapport à la relation  $\text{sigma}[\underline{\text{s}}]$ .

Conformément aux arguments présentés dans le paragraphe II.2.3. nous confondons souvent les commandes du langage et leurs relations sémantiques, ce qui nous permet, en particulier, d'omettre la référence explicite aux fonctions sémantiques dans le premier argument de wpr : par exemple, nous écrivons  $\text{wpr}[\text{DO}]$  au lieu de  $\text{wpr}[\text{rho}[\underline{\text{DO}}]]$ . Nous trouvons également pratique, pour simplifier l'écriture, d'abrégier  $\text{sigma}[\underline{\text{s}}]$  par GG.

Au vu des propositions IV.5.4 et IV.5.7, nous pouvons écrire  $DO \circ DO = DO$  et  $GC \dagger DO = DO$ . En termes de programmes la première de ces deux formules devient  $DO;DO = DO$ , traduisant une propriété bien connue de la commande  $DO$ . Quant à la seconde, nous pouvons écrire if  $s \square \text{exit} \rightarrow \text{skip}$  fi ;  $DO=DO$ , même si, à la rigueur  $\text{exit}$  ne peut pas être une garde car  $\text{exit}(\omega)=\text{faux}$ , tandis que pour toute garde  $g$  on doit avoir  $g(\omega)=\text{vrai}$  (cf. paragraphe V.2). Nous admettons cette écriture car elle est pratique et ne pose pas de problèmes : il suffit de redéfinir le prédicat  $\text{exit}$  à l'état  $\omega$ , avec  $\text{exit}(\omega)=\text{vrai}$ , dès qu'il apparaît dans une garde.

Nous pouvons, à ce propos, justifier notre choix pour la définition  $g(\omega)=\text{vrai}$ , pour toute garde  $g$ . Il y a deux possibilités : ou bien on prend toujours  $g(\omega)=\text{faux}$  ou bien on prend toujours  $g(\omega)=\text{vrai}$ . L'argument  $\omega$  à l'"entrée" d'une commande peut être interprété comme le résultat fourni par les calculs précédant l'exécution de cette commande quand ces calculs prennent un temps infini. Si on choisissait  $g(\omega)=\text{faux}$  alors d'après la sémantique intuitive de  $DO$  on pourrait conclure que l'exécution d'une commande  $DO$  succédant un calcul infini termine toujours, ce qui est une interprétation curieuse, même si, dans ces conditions, l'état de "terminaison" ne peut être autre que  $\omega$ . Le choix de  $g(\omega)=\text{vrai}$  entraîne  $\text{exit}(\omega)=\text{faux}$ , ce qui permet d'interpréter  $\text{exit}$  comme étant vraiment le prédicat de terminaison de  $DO$ .

Nous terminons ce chapitre en présentant les formules du transformateur de prédicats binaire  $wwpr[R]$  pour quelques-unes des relations. Les calculs étant tout à fait triviaux à partir de la définition de  $wwpr[R]$ , (définition IV.6.6), nous les omettons.

$$\text{x. } wwpr[\text{skip}](PP) = \lambda a. PP(a, a) \wedge a \neq \omega$$

$$\text{xi. } wwpr[\text{abort}](PP) = \perp$$

$$\text{xii. } wwpr[e](PP) = \lambda a. PP(a, e(a)) \cap \neg \Omega \circ e$$

$$\text{xiii. } wwpr[GG](P) = \bigcap_{i=1}^n g_i \Rightarrow wwpr[r_i](PP)$$

$$\text{xiv. } wwpr[IF](PP) = \neg \text{exit} \cap wwpr[GG](PP).$$

## 5. CONCLUSION

Ayant défini la sémantique relationnelle d'un langage, nous pouvons maintenant appliquer les résultats de la théorie développée dans les premiers chapitres à l'étude des programmes de ce langage. C'est ce que nous commencerons à faire dès le chapitre suivant, où nous nous intéressons surtout aux preuves de correction de programmes. Plus tard, quand nous aurons défini une sémantique relationnelle pour un langage d'écriture de programmes parallèles, nous pourrons utiliser une approche similaire pour traiter les programmes de ce type.

**CHAPITRE VI**

**INVARIANTS ET CORRECTION DE PROGRAMMES**



## 1. INTRODUCTION

Comme nous l'avons signalé au début du chapitre IV, une partie importante de la vérification des propriétés des programmes concerne la découverte, et la preuve, d'invariants. Quoique dans des cas particuliers, tels que les programmes de sémaphores, au sens de Clarke [Cla80], et certaines classes de réseaux de Petri, Sifakis [Sif78], les invariants puissent être calculés par la résolution d'un système d'équations linéaires, en général le problème du choix des bons invariants pour un programme donné subsiste.

La caractérisation en termes de points fixes de transformateurs de prédicats continus que nous avons proposée pour les invariants (proposition IV.2.10), et qu'on trouve d'ailleurs chez plusieurs auteurs (Clarke [Cla77], Flon et Suzuki [FlS78], Sifakis [Sif79], Clarke [Cla80], Cousot et Cousot [CoC80b]), nous offre, en principe, un moyen général de construction d'invariants, par calcul itératif de points fixes. Toutefois, cette technique n'est utile que lorsque le calcul itératif converge dans un temps fini, ce qui, malheureusement, ne se produit que dans des cas particuliers (espaces finis d'états, par exemple). Même quand ce calcul converge, son résultat est souvent une longue expression de prédicats, dont l'interprétation n'est pas toujours facile. Les méthodes de calcul approché de points fixes de Cousot [Cou78] et Clarke [Cla80] peuvent être utiles, mais leur applicabilité n'est pas encore suffisamment générale.

N'existant pas de moyen systématique générale pour obtenir des invariants, la seule solution est de les "inventer". Cette activité inventive est souvent méprisée dans notre domaine, comme si elle impliquait de la sorcellerie ... Néanmoins, il est évident que l'on ne peut pas concevoir un programme sans avoir saisi par l'intuition, un certain nombre de propriétés invariantes caractérisant le problème à résoudre, où le système à modéliser. Souvent, la "découverte" d'invariants d'un programme n'est que la mise en évidence des invariants considérés implicitement dès le début. D'autre part, en ce qui concerne l'étude a posteriori des programmes, la recherche d'invariants ne se

fait pas à l'aveuglette. Dans chaque cas, il y aura des heuristiques qui nous guideront. D'ailleurs, en pratique, un argument simple peut souvent remplacer un calcul fastidieux de point fixe. Mais il ne faut pas oublier que si la réussite d'une approche non-systématique requiert de l'ingéniosité, ses chances sont augmentées si l'on dispose d'outils mathématiques adéquats.

Le but de ce chapitre est précisément d'illustrer l'utilisation des outils mathématiques développées dans les chapitres précédents, notamment le concept d'invariant et la sémantique relationnelle du langage des commandes gardées, dans des preuves de correction de quelques programmes classiques. Dans le chapitre suivant, nous incluons d'autres exemples qui présentent un intérêt particulier dans la mesure où ils traitent la preuve de programmes calculant des fonctions définies récursivement.

Les programmes étudiés sont écrits dans le langage des commandes gardées et leur forme générale est une commande do-od précédée par une instruction d'initialisation. Cette forme n'est pas restrictive (il s'agit d'un morceau du folklore de l'Informatique, voir Harel [Har80]), car avec l'addition de variables pour représenter des compteurs de programme, tout programme peut être mis sous cette forme. Par ailleurs, n'ayant pas fait d'hypothèses sur le non-déterminisme des programmes considérés, nous pouvons utiliser la même approche pour étudier des programmes parallèles, pourvu que l'exécution concurrente soit modélisée de manière non-déterministe, comme c'est le cas habituellement. Cette question sera traitée au chapitre VIII.

## 2. UTILISATION DES INVARIANTS DANS LES PREUVES DE CORRECTION TOTALE DE PROGRAMMES

Quoique l'intuition derrière le concept d'invariant soit simple et à peu près la même partout, on peut trouver dans la littérature plusieurs définitions différentes pour ce concept. Comparons, par exemple, Floyd [Flo67], Hoare [Hoa69], Dijkstra [Dij72], Mazurkiewicz [Maz74], Dijkstra [Dij76], Keller [Kel76], Sifakis [Sif79], Clarke [Cla80], Cousot et Cousot [CoC80b]. Au vu des définitions du paragraphe IV.2 et des arguments du paragraphe II.2.3, nous disons qu'un prédicat est un invariant d'une commande s'il est un invariant de la relation sémantique correspondante. Formellement :

Définition VI.2.1. Invariant d'une commande. Un prédicat  $J$  est un invariant d'une commande  $r$  si  $J$  est un invariant de  $\rho[[r]]$ .  $\square$

D'après cette définition, si une commande commence son exécution dans un état d'un invariant  $J$ , elle la termine, et dans un état de  $J$ . Remarquons que les définitions d'invariant que l'on trouve habituellement n'exigent pas la terminaison. D'autre part, la définition proposée nous permet d'utiliser les résultats de la théorie relationnelle que nous avons présentée.

Remarque sur la notation. Dans la suite nous employons souvent une écriture informelle pour les prédicats, en omettant le préfixe  $\lambda$ , i.e., nous écrivons par exemple  $x \geq 0$  et  $a < b$  au lieu de  $\lambda x. x \geq 0$  et  $\lambda \langle a, b \rangle. a < b$ . Nous supposons aussi que le domaine des variables numériques dans les exemples est l'ensemble  $Z$  des entiers, à moins qu'il ne soit indiqué autrement.  $\square$

Exemple VI.2.1. Soit  $r \equiv x := x + 1$ . Le prédicat  $x \geq 0$  est clairement un invariant de  $r$ . En effet, tout prédicat de la forme  $x \geq K$ , pour  $K \in Z$  est un invariant de  $r$  :  $x \geq K \subseteq \text{wpr}[x := x + 1](x \geq K) = x + 1 \geq K$ .  $\square$

Exemple VI.2.2. La commande abort ne se termine jamais. Par conséquent, elle n'a pas d'invariant, autre que le prédicat  $\perp$ , invariant trivial de toute commande.  $\square$



Exemple VI.2.3. Considérons maintenant un exemple traité par Dijkstra dans son article classique sur la programmation structurée [Dij72]. Il s'agit de déterminer si le prédicat  $0 \leq x < dd$  est un invariant de la commande

```
r ≡ dd:=dd/2 ;
  if dd ≤ x → x:=x-dd
  □ dd > x → skip
  fi.
```

Calculons  $wpr[r](0 \leq x < dd)$  :

$$\begin{aligned} wpr[\underline{\text{if}} \dots \underline{\text{fi}}](0 \leq x < dd) &= (dd > x \vee 0 \leq x - dd < dd) \\ &\quad \wedge (dd \leq x \vee 0 \leq x < dd) \\ &= 0 \leq x < 2 * dd \\ wpr[dd:=dd/2](0 \leq x < 2 * dd) &= 0 \leq x < dd \wedge dd/2 \neq \omega \\ &= 0 \leq x < dd \wedge \text{pair}(dd). \end{aligned}$$

L'inclusion  $0 \leq x < dd \subseteq 0 \leq x < dd \wedge \text{pair}(dd)$  étant fausse, nous concluons que  $0 \leq x < dd$  n'est pas un invariant de r.

Dans l'oeuvre citée, Dijkstra présente, à propos des mécanismes intellectuels qui nous aident à comprendre un programme ou une preuve de sa correction, une preuve du fait que "the execution of the composed statement  $dd:=dd/2$  ; if  $dd \leq r$  do  $r:=r-dd$  leaves the relations  $0 \leq r < dd$  invariant"<sup>(1)</sup>. La preuve est semi-formelle car elle est basée sur la notion intuitive de "relation invariante". Le fait que  $0 \leq r < dd$  est une "relation invariante" de l'instruction indiquée est utilisé plus tard dans une preuve de correction d'un programme complet. Cette preuve réussit car l'instruction est placée dans un contexte tel que l'affectation  $dd:=dd/2$  n'est exécutée que si  $dd$  est pair, ce qui n'était pas du tout évident avant. □

On peut dire que les invariants décrivent d'une certaine manière "ce qui ne change pas" lors de l'exécution d'une commande. Au contraire, les relations sémantiques mettent l'accent précisément sur "ce qui change". L'importance de ces aspects statiques de l'exécution d'une commande devient particulièrement décisive dans le cas des commandes

---

(1) "L'exécution de l'instruction composée  $dd:=dd/2$  ; if  $dd \leq r$  do  $r:=r-dd$  laisse les relations  $0 \leq r < dd$  invariantes".

répétitives, la commande DO par exemple. En général, face à une commande DO tout ce que l'on peut affirmer a priori est que, si elle se termine, alors l'état final vérifie le prédicat exit. En général, cette seule information n'est pas très utile. Il faut la compléter par des informations sur ce qui n'a pas changé avec l'exécution. C'est ici que les invariants jouent un rôle fondamental, expliqué de manière formelle dans le paragraphe IV.6.

En pratique nous n'utilisons les invariants que dans le contexte de programmes do s od. Dans ces conditions, conformément à la remarque qui suit le corollaire IV.6.5, le mot "invariant" désigne normalement un invariant de  $\sigma[s]$  et non un invariant de  $\rho[\text{do s od}]$ . Schématiquement, notre approche consiste à trouver des invariants de  $\sigma[s]$  et, en utilisant les résultats du paragraphe IV.6., montrer qu'ils sont aussi des invariants de  $\rho[\text{do s od}]$ . Par conséquent, les formules que nous employons le plus fréquemment sont  $J \subseteq \text{wpr}[GG](J)$  et  $J \subseteq \text{wwpr}[GG](PP)$  où PP est un prédicat binaire de la forme  $\lambda \langle a, b \rangle. a \prec b$ , pour un ordre bien fondé  $\prec$ . Examinons alors l'aspect que ces formules prennent en pratique. Mais avant remarquons que la formule  $J \subseteq \text{wpr}[GG](J)$  montre que les invariants de GG sont les invariants dont Dijkstra parle dans [Dij76, p.39]. Ceux-ci seraient définis par  $J \cap \neg \text{exit} \subseteq \text{wpr}[IF](J)$  ce qui est équivalent à  $J \cap \neg \text{exit} \subseteq \text{wpr}[GG](J) \cap \neg \text{exit}$  (par IV.5.15), et donc à  $J \cap \neg \text{exit} \subseteq \text{wpr}[GG](J)$  ou  $J \subseteq \text{exit} \cup \text{wpr}[GG](J)$  ou  $J \subseteq \text{wpr}[GG](J)$ , car  $\text{exit} = \text{wpr}[GG](\perp) \subseteq \text{wpr}[GG](J)$ , (par III.5.2 et III.4.13.3).

Formulaire VI.2.2. Formules pour  $J \subseteq \text{wpr}[GG](J)$ .

1.  $J \subseteq \bigcap_{i=1}^n g_i \Rightarrow \text{wpr}[r_i](J)$
2.  $\bigwedge_{i=1}^n J \subseteq \neg g_i \cup \text{wpr}[r_i](J)$
3.  $\bigwedge_{i=1}^n J \cap g_i \subseteq \text{wpr}[r_i](J)$

Si tous les  $r_i$  sont des affectations  $x := e_i(x)$ , la formule 1 devient

$$4. J \subseteq \bigcap_{i=1}^n g_i \Rightarrow J \circ e_i \cap \neg \Omega \circ e_i$$

Si  $J \cap g_i \subseteq \neg \Omega \circ e_i$ , pour  $1 \leq i \leq n$ , c'est-à-dire si chaque  $e_i$  est bien défini dans  $J \cap g_i$ , nous avons

$$5. \bigwedge_{i=1}^n J \subseteq \neg g_i \cup J \circ e_i$$

$$6. \bigwedge_{i=1}^n J \cap g_i \subseteq J \circ e_i \quad \square$$

Formulaire VI.2.3. Formules pour  $J \subseteq \text{wwpr}[GG](PP)$ , où  $PP$  est un prédicat binaire défini par  $PP(a,b) = a \prec b$ , pour un ordre bien fondé  $\prec$ . Nous considérons seulement le cas où tous les  $r_i$  sont des affectations  $x := e_i(x)$ , telles que  $J \cap g_i \subseteq \neg \Omega \circ e_i$ .

$$1. J \subseteq \bigcap_{i=1}^n g_i \Rightarrow \lambda a. a \prec e_i(a)$$

$$2. \bigwedge_{i=1}^n J \subseteq g_i \Rightarrow \lambda a. a \prec e_i(a)$$

$$3. \bigwedge_{i=1}^n J \cap g_i \subseteq \lambda a. a \prec e_i(a).$$

Formulaire VI.2.4. Formules pour  $J \subseteq \text{wwpr}[GG](PP)$  où  $PP$  est un prédicat binaire défini comme dans le corollaire IV.6.11, i.e., soit  $f \in Q \rightarrow \mathbb{N}$  et notons par  $\text{DEF}(f)$ <sup>(1)</sup> le prédicat caractérisant le domaine de  $f$ . Alors

$$PP(a,b) = \begin{cases} f(a) > f(b) & \text{si } \text{DEF}(f)(a) \text{ et } \text{DEF}(f)(b) \\ \text{faux} & \text{sinon} \end{cases}$$

Nous considérons seulement le cas où tous les  $r_i$  sont des affectations  $x := e_i(x)$ , telles que  $J \cap g_i \subseteq \neg \Omega \circ e_i$ , pour  $1 \leq i \leq n$ .

$$1. \bigwedge_{i=1}^n J \subseteq g_i \Rightarrow \lambda a. f(a) > f(e_i(a)) \wedge \text{DEF}(f)(a) \wedge \text{DEF}(f)(e_i(a))$$

$$2. \bigwedge_{i=1}^n J \cap g_i \subseteq \lambda a. f(a) > f(e_i(a)) \wedge \text{DEF}(f)(a) \wedge \text{DEF}(f)(e_i(a))$$

(1) En général nous notons par  $\text{DEF}(F)$  le prédicat caractérisant le domaine d'une fonction arbitraire  $F$ .

Si  $J \cap g_i \subseteq \text{DEF}(f)$  ces formules se simplifient de manière immédiate.  $\square$

### 3. MAXIMUM DE TROIS ENTIERS

Considérons le programme suivant :

```

x,y,z:=X,Y,Z ;
do x < y → x:=x+1
  □ y < z → y:=y+1
  □ z < x → z:=z+1
od
{x=y=z=max{X,Y,Z}}
```

Nous voulons prouver qu'il est totalement correct, c'est-à-dire qu'il se termine toujours dans un état satisfaisant la post-condition indiquée (en fait, pour X, Y et Z donnés il n'y a qu'un seul état qui la satisfait).

Montrons tout d'abord que le prédicat J, défini par  $J = \{\max\{x,y,z\} = K\}$  est un invariant, pour tout entier K. Utilisons, par exemple, la formule VI.2.2.6 :

$$\begin{aligned} \max\{x,y,z\} = K \cap x < y &\subseteq \max\{x+1,y,z\} = K \\ \wedge \max\{x,y,z\} = K \cap y < z &\subseteq \max\{x,y+1,z\} = K \\ \wedge \max\{x,y,z\} = K \cap z < x &\subseteq \max\{x,y,z+1\} = K \end{aligned}$$

Ces trois inclusions sont vraies et J est donc un invariant. Comme la preuve est indépendante de la valeur de K, nous avons en fait trouvé une famille  $\{J_K\}_{K \in \mathbb{Z}}$  d'invariants avec  $J_K = \{\max\{x,y,z\} = K\}$ . Ces invariants sont disjoints deux à deux, i.e.,  $J_{K1} \cap J_{K2} = \perp$ , si  $K1 \neq K2$ .

Parmi tous les invariants possibles, seuls nous intéressent ceux dans lesquels le programme peut "entrer". Ainsi, en mettant  $\max\{X,Y,Z\} = M$ , nous devons choisir, pour continuer la preuve, l'invariant  $J_M$ ,  $J_M = \{\max\{x,y,z\} = M\}$ , qui est le seul de la famille  $\{J_K\}$  convenant au programme. En fait, le programme est initialisé dans cet invariant.

Nous avons  $\text{exit} = \{x=y=z\}$ . Il est clair que  $J_M \cap \text{exit}$  implique la post-condition. Il reste seulement à prouver que le programme se termine.

Définissons  $f \in \mathbb{Q} \rightarrow \mathbb{N}$  par  $f(x,y,z) = 3*M-x-y-z$  et montrons que  $f$  est décroissante sous  $J_M$ . Vu que  $J_M \subseteq \text{DEF}(f)$  nous pouvons utiliser la formule VI.2.4.2 :

$$\begin{aligned} \max\{x,y,z\} &= M \cap x < y \subseteq 3*M-x-y-z > 3*M-(x+1)-y-z \\ \wedge \max\{x,y,z\} &= M \cap y < z \subseteq 3*M-x-y-z > 3*M-x-(y+1)-z \\ \wedge \max\{x,y,z\} &= M \cap z < x \subseteq 3*M-x-y-z > 3*M-x-y-(z+1) \end{aligned}$$

Cette expression est manifestement vraie. De plus, rappelons-le, le programme est initialisé dans l'invariant  $J_M$ . Par conséquent, il se termine nécessairement, en validant la post-condition. Ceci termine la preuve.

Notre démarche a été complètement formelle. Dans la pratique, on omet souvent des détails "évidents". Néanmoins la structure des preuves est toujours la même : trouver d'abord une famille d'invariants ; choisir ensuite celui (ou le plus petit, s'ils sont plusieurs) où le programme est initialisé et prouver que son intersection avec le prédicat  $\text{exit}$  implique la post-condition ; définir enfin une fonction décroissante, ou un ordre bien fondé pour prouver que le programme se termine.

On aurait pu obtenir une preuve légèrement différente en considérant d'autres invariants. En effet les prédicats  $x \leq K \wedge y \leq K \wedge z \leq K$ ,  $x \geq x_0$ ,  $y \geq y_0$  et  $z \geq z_0$  sont tous des invariants, pour tout  $K$ ,  $x_0$ ,  $y_0$ ,  $z_0$ . Par conséquent, leur conjonction, i.e., tout prédicat de la forme  $x_0 \leq x \leq K \wedge y_0 \leq y \leq K \wedge z_0 \leq z \leq K$  est un invariant aussi. Il est intéressant de remarquer que ces invariants ne sont pas toujours disjoints deux à deux. Parmi eux, le plus petit avec lequel le programme est initialisé est  $X \leq x \leq M \wedge Y \leq y \leq M \wedge Z \leq z \leq M$ . Ce prédicat implique l'invariant  $J_M$  considéré précédemment et il peut donc le remplacer dans la preuve.

#### 4. L'ALGORITHME D'EUCLIDE

Considérons l'algorithme d'Euclide pour calculer le plus grand commun diviseur de deux entiers positifs  $X$  et  $Y$ , dans la version donnée par Dijkstra [Dij76, p.45] :

```

{X > 0 et Y > 0}
x,y := X,Y ;
do x > y → x:=x-y
  [] y > x → y:=y-x
od
{x=y = pgcd(X,Y)}

```

Pour prouver la correction de ce programme, nous commençons par chercher un invariant dont l'intersection avec  $x=y$  implique la post-condition. Le choix naturel pour la famille d'invariants est

$J_K = \{\text{pgcd}(x,y)=K\}$ . Utilisons la formule VI.2.2.6 :

$$\begin{aligned} \text{pgcd}(x,y) = K \wedge x > y &\subseteq \text{pgcd}(x-y,y) = K \\ \wedge \text{pgcd}(x,y) = K \wedge y > x &\subseteq \text{pgcd}(x,y-x) = K \end{aligned}$$

Les deux implications étant vraies  $J_K$  est en fait un invariant, pour tout  $K$ . (On peut d'ailleurs remarquer que les implications restent vraies même si on enlève les gardes). Les invariants  $J_K$  sont disjoints : celui que l'on doit choisir ne peut être que  $J_{\text{pgcd}(X,Y)}$ . Notons-le  $J$  simplement.

Pour la terminaison, il paraît naturel de choisir comme fonction décroissante la fonction  $f \in \mathbb{Q} \rightarrow \mathbb{N}$  définie par  $f(x,y) = x+y$ . Remarquons que la fonction  $f$  n'est pas totale (on n'exclut pas que  $x$  et  $y$  puissent prendre des valeurs négatives). Par la formule VI.2.4.1, avec  $\text{DEF}(f) = \{x+y \geq 0\}$  on obtient :

$$\begin{aligned} J &\subseteq x \leq y \vee x+y > (x-y)+y \wedge x+y > 0 \wedge (x-y)+y > 0 \\ \wedge J &\subseteq y \leq x \vee x+y > x+(y-x) \wedge x+y > 0 \wedge x+(y-x) > 0 \end{aligned}$$

i.e.  $J \subseteq x=y \vee x>0 \wedge y>0$ , ce qui est faux (on suppose le pgcd défini dans  $\mathbb{Z}^2 - \{0,0\}$ ). Néanmoins il est clair que  $J \cap x>0 \wedge y>0 \subseteq x=y \vee x>0 \wedge y>0$  est vrai. Si  $x>0 \wedge y>0$  était un invariant alors  $J \cap x>0 \wedge y>0$  le serait aussi (d'après la proposition IV.2.3) et  $f$  serait en vérité décroissante sous l'invariant  $J \cap x>0 \wedge y>0$ . Comme cet invariant est vérifié par l'état initial la terminaison serait assurée. Heureusement,  $x>0 \wedge y>0$  est en effet un invariant (ceci est facile à montrer). La preuve est ainsi terminée.

Dans cet exemple, contrairement au précédent, l'invariant original n'était pas suffisamment fort pour assurer la terminaison. Il est intéressant de remarquer que nous avons été conduits vers le bon invariant en essayant de prouver la terminaison sous le premier invariant, à l'aide de la formule VI.2.4.1.

## 5. DENOMBREMENT DES FEUILLES D'UN ARBRE

Nous montrons maintenant un exemple de vérification d'un programme qui manipule des structures de données plus élaborées. Il s'agit du problème du dénombrement des feuilles d'un arbre binaire, pour lequel il existe déjà des preuves, notamment par Burstall [Bur74] et London [Lon77]. Evidemment nous ne nous intéressons pas à la version récursive mais à une version itérative avec gestion explicite de la pile. Cet exemple établit d'ailleurs le lien avec le chapitre suivant où nous traitons en détail le problème de la preuve de programmes itératifs qui calculent des fonctions définies récursivement.

Le nombre de feuilles d'un arbre binaire  $a$ ,  $feui(a)$  est défini récursivement par

$$feui(a) = \begin{cases} a=nil \rightarrow 1 \\ a \neq nil \rightarrow feui(gau(a)) + feui(dro(a)) \end{cases}$$

$gau(a)$  et  $dro(a)$  représentant respectivement le sous-arbre gauche et le sous-arbre droit de l'arbre  $a$ .

Le programme présenté par London [Lon77] est le suivant, où la variable  $p$  représente une pile d'arbres binaires :

```

a:=A ; p:=pilevide ; n:=0 ;
BOUCLE : if a≠nil
          then begin empiler(p,a) ;
              a:=gau(a) ;
              go to BOUCLE
          end
          else begin n:=n+1 ;
              if vide(p) then go to FIN ;
              a:=somet(p) ; dépiler(p) ;
              a:=dro(a) ;
              go to BOUCLE
          end
FIN :
      {n=feui(A)}

```

Nous proposons une version légèrement différente, écrite dans le langage des commandes gardées (avec une petite extension). Etant donné que la pile ne sert qu'à garder des sous-arbres dont les feuilles seront dénombrées plus tard, nous concluons que l'ordre par lequel les sous-arbres sont dépilés n'est pas important. Ainsi, dans notre programme, nous utilisons un ensemble à la place de la pile. Dans le "type" ensemble nous définissons la constante  $\emptyset$ , l'ensemble vide) et les opérations  $+$  (union d'ensembles),  $-$  (soustraction d'ensembles),  $\{ \}$  (création d'un ensemble :  $\{a_1, a_2, \dots, a_n\}$  est l'ensemble dont les éléments sont  $a_1, a_2, \dots, a_n$ ), et une opération non-déterministe notée undans tel que la valeur de undans(s) est un élément (arbitraire) de l'ensemble s.

```

a,s,n:=A,∅,0 ;
do a≠nil → s,a:=s+{a},gau(a)
    □ a=nil and s≠∅ → n,a:=n+1, undans(s) ;
                    s,a:=s-{a}, dro(a)
od ;
n:=n+1
{n=feui(A)}

```



La sémantique de l'opération  $\text{undans}(a)$  doit être précisée. Remarquons que l'écriture  $a := \text{undans}(s)$  est un peu abusive, l'opération  $\text{undans}$  n'étant pas une fonction (cf. paragraphe V.3). Nous la gardons quand même car elle est pratique.

Si l'on représente par  $A$  l'ensemble des arbres binaires finis et par  $S$  l'ensemble des ensembles d'arbres binaires finis (i.e.  $S = 2^A$ ), nous pouvons considérer que l'espace d'états de notre programme est l'ensemble  $Q$  défini par  $Q = A \times S \times \mathbb{N} \cup \{\omega\}$ . Dans ces conditions, la sémantique de  $\text{undans}$  est définie par la relation suivante :

$$\text{rho}[\text{undans}] = \{(\langle a, s, n \rangle, \langle a', s, n \rangle) \mid s \neq \emptyset \wedge a' \in s\} \dagger^{(1)}$$

Ceci implique :

$$\text{rho}[\text{undans}(s)] = \{(\langle a, s, n \rangle, \langle a', s, n+1 \rangle) \mid s \neq \emptyset \wedge a' \in s\} \dagger$$

et

$$\begin{aligned} \text{wpr}[\text{undans}(s)](P(a, s, n)) &= \\ &= s \neq \emptyset \wedge \forall_{u \in s} P(u, s, n+1) \end{aligned}$$

Remarquons que  $\text{gau}$  et  $\text{dro}$  ne sont pas définies quand leur argument est nil, ce qui signifie

$$\text{wpr}[a := \text{gau}(a)](P(a, s, n)) = a \neq \text{nil} \wedge P(\text{gau}(a), s, n)$$

$$\text{wpr}[a := \text{dro}(a)](P(a, s, n)) = a \neq \text{nil} \wedge P(\text{dro}(a), s, n)$$

Pour prouver notre programme nous commençons par essayer l'invariant utilisé par London dans sa preuve

$$J_0 = n + \text{feui}(a) + \sum_{u \in s} \text{feui}(\text{dro}(u)) = K$$

Nous observons tout de suite que l'on doit choisir  $K = \text{feui}(A)$  pour que le programme soit initialisé dans l'invariant. De même, si la boucle se termine, le prédicat  $a = \text{nil} \wedge s = \emptyset$  est vrai, ce qui implique, si  $J_0$  est bien un invariant, que  $n+1 = \text{feui}(A)$  est vrai également. Par conséquent,

---

(1) Il s'agit ici d'une légère extension de la fonction  $\dagger$  qui par IV.5.14. n'est définie que pour des relations semi-programmables.

après la dernière affectation nous avons  $n = \text{feui}(A)$  comme nous le voulons.  
Formellement :

$$\text{wpr}[n := n+1](n = \text{feui}(A)) = \{n+1 = \text{feui}(A)\}$$

Utilisons la formule VI.2.2.2 pour vérifier si  $J_0$  est un invariant.  
Pour la première commande gardée, nous avons

$$\begin{aligned} J_0 &\subseteq a = \text{nil} \cup n + \text{feui}(\text{gau}(a)) + \sum_{u \in S + \{a\}} \text{feui}(\text{dro}(u)) = K \cap a \neq \text{nil} \\ &= a = \text{nil} \cup n + \text{feui}(\text{gau}(a)) + \text{feui}(\text{dro}(a)) + \sum_{u \in S} \text{feui}(\text{dro}(u)) = K \end{aligned}$$

si  $a = \text{nil}$  l'inclusion est vraie ; sinon  $\text{feui}(\text{gau}(a)) + \text{feui}(\text{dro}(a)) = \text{feui}(a)$   
et l'inclusion est vraie aussi.

La deuxième commande gardée donne

$$\begin{aligned} J_0 &\subseteq a \neq \text{nil} \cup s = \emptyset \cup \text{wpr}[n, a := n+1 \text{ undans}(s)] \\ &\quad (n + \text{feui}(\text{dro}(a)) + \sum_{u \in S - \{a\}} \text{feui}(\text{dro}(u))) = K \cap a \neq \text{nil} \\ &= a \neq \text{nil} \cup s = \emptyset \cup (s \neq \emptyset \wedge \forall_{v \in S} (n+1 + \text{feui}(\text{dro}(v)) + \sum_{u \in S - v} \text{feui}(\text{dro}(u))) = K \\ &\quad \wedge \sum_{u \in S - \{v\}} \text{feui}(\text{dro}(u)) = K \wedge v \neq \text{nil})) \\ &= a \neq \text{nil} \cup s = \emptyset \cup \forall_{v \in S} (n+1 + \sum_{u \in S} \text{feui}(\text{dro}(u))) = K \wedge v \neq \text{nil} \\ &= a \neq \text{nil} \cup s = \emptyset \cup n+1 + \sum_{u \in S} \text{feui}(\text{dro}(u)) = K \wedge \forall_{v \in S} v \neq \text{nil} \end{aligned}$$

Si  $a \neq \text{nil}$  ou  $s = \emptyset$  l'inclusion est vraie ; sinon alors  $l = \text{feui}(a)$  et l'inclusion  
serait facilement vérifiée si on avait aussi  $J_0 \subseteq \forall_{v \in S} v \neq \text{nil}$ . Toutefois,

ceci n'est pas vrai, i.e., on ne peut pas déduire  $\forall_{v \in S} v \neq \text{nil}$  de  $J_0$ . On peut,

heureusement, remarquer que  $\forall_{v \in S} v \neq \text{nil}$  est un invariant. Prouvons-le

formellement à l'aide de la formule VI.2.2.2 :

$$\forall_{v \in S} v \neq \text{nil} \subseteq a = \text{nil} \cup (\forall_{v \in S + \{a\}} v \neq \text{nil}) \wedge a \neq \text{nil}$$

^

$$\forall_{v \in S} v \neq \text{nil} \subseteq a \neq \text{nil} \cup s = \emptyset \cup \text{wpr}[n, a := n+1, \text{undans}(a)] ((\forall_{v \in S - \{a\}} v \neq \text{nil}) \wedge a \neq \text{nil})$$

On constate que la première de ces deux implications est vraie. Quant à la seconde, elle se simplifie en

$$\begin{aligned}
 \forall_{ves} v \neq nil \subseteq a \neq nil \cup s = \emptyset \cup \text{wpr}[n, a := n+1, \text{undans}(s)](\forall_{ves} v \neq nil \wedge a \neq nil) \\
 = a \neq nil \cup s = \emptyset \cup (s \neq \emptyset \wedge \forall_{yes} \forall_{ves} v \neq nil \wedge \forall_{ves} v \neq nil) \\
 = a \neq nil \cup s = \emptyset \cup \forall_{ves} v \neq nil
 \end{aligned}$$

ce qui montre qu'elle est vraie aussi.

En résumé : nous avons

$$\begin{aligned}
 J_0 \not\subseteq \text{wpr}[GG](J_0), \\
 \forall_{ves} v \neq nil \subseteq \text{wpr}[GG](\forall_{ves} v \neq nil)
 \end{aligned}$$

et aussi

$$J_0 \cap \forall_{ves} v \neq nil \subseteq \text{wpr}[GG](J_0)$$

Les deux inclusions obtenues impliquent

$$\begin{aligned}
 J_0 \cap \forall_{ves} v \neq nil \subseteq \text{wpr}[GG](J_0) \cap \text{wpr}[GG](\forall_{ves} v \neq nil) \\
 = \text{wpr}[GG](J_0 \cap \forall_{ves} v \neq nil), \text{ par III.4.6.1}
 \end{aligned}$$

Nous concluons que  $J_0 \cap \forall_{ves} v \neq nil$  est un invariant. Il est aussi vrai à l'initialisation et son intersection avec le prédicat  $\text{exit}$  implique  $n+1=K$ , comme nous le souhaitions. Par conséquent, la preuve de correction partielle est complète. Il ne nous reste qu'à prouver la terminaison du programme.

Nous préférons le faire de manière informelle. Remarquons que la première garde garantit que l'opération  $\text{gau}$  est toujours bien définie ; l'invariant  $\forall_{ves} v \neq nil$  et le prédicat  $s \neq \emptyset$  dans la deuxième garde garantissent le même pour les opérations  $\text{dro}$  et  $\text{undans}$ . Les autres opérations ne posent pas de problèmes non plus. Les arbres considérés sont finis et par conséquent l'invariant  $J_0 \cap \forall_{ves} v \neq nil$  implique que la deuxième commande gardée ne peut être exécutée qu'un nombre fini de fois. Si le

programme ne se terminait pas, il devrait boucler indéfiniment sur la première commande gardée. Mais ceci ne peut se produire que si l'arbre a une branche infinie, ce qui est exclu.

Nous considérons que la preuve de notre programme est terminée.

## 6. CONCLUSION

Les trois exemples traités dans ce chapitre illustrent l'utilisation dans les preuves de programmes de la théorie relationnelle développée dans les chapitres précédents. D'une part cette théorie est à la base de la sémantique choisie pour le langage de programmation employé ; d'autre part elle a permis une caractérisation précise du concept d'invariant, qui joue un rôle central dans les preuves.

Dans le chapitre suivant, nous montrons comment les techniques de preuves que nous venons d'exposer s'appliquent également bien à une classe de programmes pour lesquels des preuves avec des invariants ne sont pas usuelles.



## CHAPITRE VII

### INVARIANTS ET FONCTIONS DEFINIES RECURSIVEMENT



## 1. INTRODUCTION

Dans ce chapitre, nous étudions à travers des exemples le problème suivant : comment prouver qu'un programme itératif donné calcule une fonction qui est définie récursivement ? Gries [Gri79] est de l'avis que cette question reste un "bastion" de la méthode des assertions intermittentes. Dans un article sur cette méthode, Manna et Waldinger [MaW78] écrivent que les preuves conventionnelles, (i.e., avec des invariants), de ce type de programmes sont compliquées et peu naturelles et ils lancent un défi pour une preuve simple avec des invariants pour un programme itératif qui calcule la fonction de Ackermann. Gries a déjà répondu à ce défi, d'une manière satisfaisante, dans [Gri79]. Un de nos buts maintenant est de fournir d'avantage d'arguments montrant que l'approche conventionnelle n'est pas nécessairement compliquée ou peu naturelle, bien au contraire, même dans le cas, prétendument plus difficile, des programmes itératifs qui calculent des fonctions définies récursivement.

Nous présentons notre approche à l'aide de la fonction de Ackermann. Le programme que nous prouvons est celui de Gries mais notre invariant est très différent du sien. Le deuxième exemple que nous traitons a aussi été étudié par Gries et par Manna et Waldinger. Il s'agit là d'un programme "générique". Ces deux exemples permettent ainsi de comparer les trois approches.

Dans le reste du chapitre, nous considérons d'autres programmes qui calculent des fonctions définies récursivement, mais qui n'utilisent pas une pile. Dans tous les cas, notre approche est conceptuellement la même, à savoir celle qui a été décrite à propos de l'exemple du paragraphe VI.3.



## 2. INDUCTION STRUCTURELLE ET INDUCTION NOETHERIENNE

Les preuves que nous présentons utilisent, directement ou indirectement, le principe d'induction structurelle. Ce principe nous permet de prouver des assertions sur les éléments d'un ensemble bien fondé, i.e., un ensemble dans lequel est définie une relation d'ordre bien fondé. En termes informels ce principe affirme que si le fait qu'un prédicat  $P$  est vérifié en un point  $x$  est impliqué par le fait que  $P$  est vérifié dans tous les points strictement supérieurs à  $x$  (d'après l'ordre), alors  $P$  est vrai partout. Formellement :

Principe d'induction structurelle. Soient  $X$  un ensemble et  $\prec$  un ordre bien fondé dans  $X$ . Soit  $P$  un prédicat sur  $X$ . Alors <sup>(1)</sup>

$$\text{si } \forall x \in X (\forall y \in \prec(x) P(y)) \Rightarrow P(x)$$

alors  $\forall z \in X P(z)$ .  $\square$

(Pour une preuve de ce principe et des exemples d'utilisation, on peut consulter Manna, Ness et Vuillemin [MNV73]).

Parfois il est préférable de remplacer le principe d'induction structurelle par le principe d'induction noetherienne (cf. def IV.6.8) qui a l'avantage de permettre d'éviter la construction explicite de l'ordre bien fondé engendré par fermeture réflexive transitive d'une relation noetherienne (Huet [Hue77]). Conventionnellement, la notation  $R^+$  désigne la fermeture transitive de la relation  $R$ ,  $R^+ = \bigcup_{i \in \mathbb{N}} R^{i+1}$ .

Principe d'induction noetherienne. Soient  $X$  un ensemble et  $\varepsilon$  une relation noetherienne dans  $X$ . Soit  $P$  un prédicat sur  $X$ . Alors

$$\text{si } \forall x \in X (\forall y \in \varepsilon^+(x) P(y)) \Rightarrow P(x)$$

alors  $\forall z \in X P(z)$ .  $\square$

Ce principe est très pratique dans le contexte de ce qu'on appelle l'ordre bien fondé induit par le calcul d'une fonction définie récursivement. Pour une fonction  $F$ , cet ordre, noté  $\prec$ , est défini par

(1) La notation  $\prec(x)$  représente, comme d'habitude, l'ensemble des successeurs de  $x$  par  $\prec$ ,  $\prec(x) = \{y \mid x \prec y\}$ .

$a \prec b$  si  $F(a)$  est définie et le calcul de  $F(b)$  intervient dans le calcul de  $F(a)$ .

Nous disons que  $F(a)$  est définie quand tous les calculs de  $F(a)$  possibles se terminent et délivrent le même résultat. On ne privilégie donc aucun ordre particulier pour l'évaluation de  $F(a)$ , dans le cas où il y en aurait plusieurs.

Si la fonction  $F$  est définie au moyen d'une équation on constate facilement que  $\prec$  est la fermeture réflexive transitive de la relation noethérienne  $\epsilon$  définie par

$a \epsilon b$  si  $F(a)$  est définie, et la définition de  $F(a)$  est donnée en termes de  $F(b)$ .

Nous appelons cette relation, relation induite par la définition de  $F$ . La condition " $F(a)$  est définie" ci-dessus assure que  $\epsilon$  est en vérité noethérienne. Une séquence infinie  $a \epsilon a_1 \epsilon \dots \epsilon a_i \epsilon \dots$  correspondrait à un calcul infini de  $F(a)$ , en contradiction avec le fait que  $F(a)$  est définie.

D'autres exemples d'ordres bien fondés sont les suivants :

- i.  $\mathbb{N}$  avec  $\geq$  ;
- ii.  $\mathbb{N} \times \mathbb{N}$  avec l'ordre lexicographique  $\geq_2$  :  
 $(a_1, a_2) \geq_2 (b_1, b_2)$  si  $a_1 > b_1 \vee a_1 = b_1 \wedge a_2 \geq b_2$
- iii. L'ensemble des séquences finies d'entiers, que nous notons  $S$ ,  
 $S = \bigcup_{i \in \mathbb{N}} \mathbb{N}^i$ , avec l'ordre d'inclusion des sous-séquences  $\preceq_s$ ,  
 $s_1 \preceq_s s_2$  si  $s_2$  est une sous-séquence de  $s_1$ .

### 3. FONCTION DE ACKERMANN

La fonction de Ackermann A est une application de  $\mathbb{N}^2 \rightarrow \mathbb{N}$  définie par

$$A(m,n) = \begin{cases} m=0 & \rightarrow n+1 \\ m \neq 0 \wedge n=0 & \rightarrow A(m-1,1) \\ m \neq 0 \wedge n \neq 0 & \rightarrow A(m-1, A(m,n-1)) \end{cases}$$

Le programme itératif suivant, qui est dû à Gries [Gri79] calcule la fonction de Ackermann de la manière indiquée par les pre et post-conditions. Le programme utilise une variable de "type" séquence qui représente la pile du calcul récursif et dont les valeurs sont des séquences finies de nombres naturels. L'écriture  $\langle s_n, \dots, s_2, s_1 \rangle$  désigne la séquence dont les éléments sont  $s_n, \dots, s_2$  et  $s_1$ . Si  $s = \langle s_n, \dots, s_2, s_1 \rangle$  les notations  $s(i)$  et  $s(..i)$  désignent respectivement l'élément  $s_i$ , s'il existe, et la séquence  $\langle s_n, \dots, s_{i+1}, s_1 \rangle$ , qui est vide si  $i > n$ ; l'opération  $s|a$  dénote la concaténation de l'élément  $a$  à la séquence  $s$ . Par exemple, si  $s(..2) \neq \langle \rangle$  alors  $s = s(..3)|s(2)|s(1)$ .

```
{m,n ∈ ℕ}
s := <m,n> ;
do s(..2) ≠ <> cand s(2)=0           → s := s(..3)|s(1)+1
  □ s(..2) ≠ <> cand s(2)≠0 and s(1)=0 → s := s(..3)|s(2)-1|1
  □ s(..2) ≠ <> cand s(2)≠0 and s(1)≠0 → s := s(..3)|s(2)-1|s(2)|s(1)-1
od
{s(1)=A(m,n)}
```

Remarque. L'utilisation de l'opérateur cand dont le deuxième argument n'est évalué que si le premier est vrai, assure que les gardes sont toujours définies. □

La principale difficulté de la preuve consiste à trouver un invariant adéquat, comme Gries le reconnaît, du moment que la terminaison peut être établie d'une manière relativement standard. L'invariant que nous proposons est défini en termes d'une fonction entière  $G$  sur des séquences d'entiers,  $G \in S \rightarrow \mathbb{N}$ , définie récursivement par les équations suivantes :

$$G(\langle b \rangle) = b$$

$$G(s|a|b) = \begin{cases} a=0 & \rightarrow G(s|b+1) \\ a \neq 0, b=0 & \rightarrow G(s|a-1|1) \\ a \neq 0, b \neq 0 & \rightarrow G(s|a-1|a|b-1) \end{cases}$$

(Remarquons que  $G(\langle \rangle)$  reste indéfini).

Le théorème suivant établit la liaison entre la fonction de Ackermann et la fonction  $G$  (en montrant d'ailleurs que celle-ci est toujours définie sauf pour la séquence vide).

**Théorème VII.3.1.**  $G(s|a|b) = G(s|A(a,b))$ .

**Démonstration.** Par induction structurelle avec l'ordre lexicographique des paires d'entiers.

i.  $a=0$  (base de l'induction)

$$\begin{aligned} G(s|0|b) &= G(s|b+1) && \text{par def. de } G \\ &= G(s|A(0,b)) && \text{par def. de } A \end{aligned}$$

ii.  $a \neq 0, b=0$

$$\begin{aligned} G(s|a|0) &= G(s|a-1|1) && \text{par def. de } G \\ &= G(s|A(a-1,1)) && \text{par l'hypothèse d'induction} \\ &= G(s|A(a,0)) && \text{par def. de } A \end{aligned}$$

iii.  $a \neq 0, b \neq 0$

$$\begin{aligned} G(s|a|b) &= G(s|a-1|a|b-1) && \text{par def. de } G \\ &= G(s|a-1|A(a,b-1)) && \text{par l'hypothèse d'induction} \\ &= G(s|A(a-1,A(a,b-1))) && \text{idem} \\ &= G(s|A(a,b)) && \text{par def. de } A \end{aligned}$$

□

**Corollaire VII.3.2.**  $G(\langle a,b \rangle) = A(a,b)$ . □

Il est maintenant très facile de montrer que les prédicats  $J_K$ ,  $K \in \mathbb{N}$ ,  $J_K = \{G(s)=K\}$  sont des invariants (appliquer la formule VI.2.2.3). Le corollaire indique que nous devons choisir celui pour lequel  $K=A(m,n)$ , i.e.  $G(s)=A(m,n)$ . Cet invariant implique  $s \neq \langle \rangle$ , qui d'ailleurs est aussi un invariant trivial. Le prédicat exit est  $s(..2)=\langle \rangle$ . Par conséquent, le programme étant initialisé avec  $G(s)=A(m,n)$ , s'il se

termine le prédicat  $\{s(.2)=\langle\rangle\} \cap \{G(s)=A(m,n)\}$  est vérifié à la terminaison, impliquant  $s(1)=A(m,n)$ , comme nous souhaitons.

Il reste à montrer que le programme se termine vraiment. Considérons la relation  $\varepsilon$  entre séquences d'entiers  $\varepsilon \subseteq S \times S$  défini par

$$\begin{aligned} s|o|b &\varepsilon s|b+1 \\ s|a|o &\varepsilon s|a-1|1, \quad \text{si } a \neq 0 \\ s|a|b &\varepsilon s|a-1|a|b-1, \quad \text{si } a \neq 0 \text{ et } b \neq 0 \end{aligned}$$

Il est évident que  $\varepsilon = GG-\{(\omega,\omega)\}$ . (Rappelons que GG est la relation associée à une itération de la commande do-od). Par le corollaire IV.6.12, il suffit de prouver que  $\varepsilon$  est noethérienne pour garantir que le programme se termine :

**Théorème VII.3.3.** La relation  $\varepsilon$  est noethérienne.

*Démonstration.* On montre par induction structurelle sur l'ordre lexicographique des paires d'entiers que

(\*)  $\forall s \in S, a, b \in \mathbb{N} \exists i \in \mathbb{N}, c \in \mathbb{N} \varepsilon^i(s|a|b) = \{s|c\}$  (la démonstration de (\*) étant analogue à celle du théorème VII.3.1, nous l'omettons). Avec (\*), par induction sur la taille de  $s$ , il est facile de montrer que

$$\forall s \in S -\{\langle\rangle\} \exists i \in \mathbb{N}, c \in \mathbb{N} \varepsilon^i(s) = \{\langle c \rangle\}.$$

Mais si  $\varepsilon^i(s) = \{\langle c \rangle\}$  alors  $\varepsilon^{i+1}(s) = \emptyset$ . Vu que  $\varepsilon(\langle\rangle) = \emptyset$ , nous concluons

$$\forall s \in S \exists i \in \mathbb{N} \varepsilon^i(s) = \emptyset$$

ce qui signifie que  $\varepsilon$  est noethérienne.  $\square$

Nous considérons que la preuve du programme est terminée. Néanmoins nous voudrions faire quelques remarques à propos de ce qui a été accompli. D'abord, l'invariant utilisé a été réellement facile à trouver (contrairement à celui utilisé par Gries, comme l'auteur lui-même l'avoue, [Gri79, p.260]). Ce qui a requis plus de réflexion a été la définition de la fonction  $G$  et la formulation correcte du théorème VII.3.1. Après avoir complété cette preuve et bien compris la technique, les autres exemples ont été traités beaucoup plus rapidement.

Nous sommes en accord avec Gries quand il écrit : "Any algorithm is based on certain properties of the objects it manipulates and it seems desirable to keep a clear distinction between these properties and the algorithm that works on the objects"<sup>(1)</sup>, [Cri79, p.264].

Nous croyons que notre preuve respecte bien cette distinction : d'une part nous avons les objets (les nombres naturels et les séquences), et les propriétés qui nous intéressent sont formulées par les théorèmes ; d'autre part nous avons le programme avec sa sémantique relationnelle et ses invariants.

#### 4. UN SCHEMA DE TRANSFORMATION

Soit la fonction  $F$  définie récursivement par

$$F(a) = \begin{cases} p(a) \rightarrow f(a) \\ \neg p(a) \rightarrow h(F(g1(a)), F(g2(a))), \end{cases}$$

où l'on suppose que  $p$ ,  $f$ ,  $g1$ ,  $g2$  et  $h$  sont des fonctions toujours définies, et que  $h$  est associative ( $h(a, h(b, c)) = h(h(a, b), c)$ ), pour tous  $a, b, c$  et possède un élément neutre à gauche noté  $e$  ( $h(e, a) = a$ , pour tout  $a$ ). Manna et Waldinger, avec la méthode des assertions intermittentes [MaW78], et Gries avec des invariants, [Gri79], ont prouvé que la fonction  $F$  est calculée par le programme itératif suivant :

```
{F(X) est défini}
s, z := <X>, e ;
do s ≠ <> cand p(s(1)) → s, z := s(..2), h(z, f(s(1)))
  □ s ≠ <> cand not p(s(1)) → s := s(..2) | g2(s(1)) | g1(s(1))
od
{z = F(X)}.
```

Plus précisément, si  $F(X)$  est défini, le programme se termine avec  $z = F(X)$  ; sinon, le programme ne se termine pas.

---

(1) "Tout algorithme est basé sur certaines propriétés des objets qu'il manipule et il paraît souhaitable de garder une distinction nette entre ces propriétés et l'algorithme qui travaille sur les objets".

Nous présentons une preuve de ce fait, en utilisant la technique introduite par l'exemple du programme de la fonction de Ackermann. Nous verrons que la technique marche très bien. Pour motiver cette preuve générique, nous l'illustrons d'abord sur un cas particulier, la fonction de Fibonacci. On peut d'ailleurs remarquer que la fonction traitée dans la section VI.5 est aussi de ce type.

#### 4.1. La fonction de Fibonacci.

La fonction de Fibonacci, Fib, est une fonction totale,  $\text{Fib} \in \mathbb{N} \rightarrow \mathbb{N}$ , définie par

$$\text{Fib}(a) = \begin{cases} a \leq 1 \rightarrow a \\ a > 1 \rightarrow \text{Fib}(a-1) + \text{Fib}(a-2). \end{cases}$$

Nous constatons que Fib correspond bien au cas général avec  $p(a)=a \leq 1$ ,  $f(a)=a$ ,  $g1(a)=a-1$ ,  $g2(a)=a-2$ ,  $h(a,b)=a+b$  et  $e=0$ . Par conséquent, le programme itératif qui calcule Fib est

```
{X ∈ ℕ}
s, z := <X>, 0 ;
do s ≠ <> cand s(1) ≤ 1 → s, z := s(..2), z+s(1)
□ s ≠ <> cand s(1) > 1 → s := s(..2) | s(1)-2 | s(1)-1
od
{z = Fib(X)}.
```

L'invariant qui nous sert à prouver ce programme est exprimé en termes de la fonction G,  $G \in \mathbb{S} \rightarrow \mathbb{N}$ , définie récursivement par les équations suivantes :

$$G(\langle \rangle) = 0$$

$$G(s|a) = \begin{cases} a \leq 1 \rightarrow a + G(s) \\ a > 1 \rightarrow G(s|a-2|a-1). \end{cases}$$

La relation entre la fonction G et la fonction Fib est établie par :

**Théorème VII.4.1.1.**  $G(s|a) = \text{Fib}(a) + G(s)$ .

**Démonstration.** Par induction structurelle dans  $\mathbb{N}$  avec la relation d'ordre usuelle  $\geq$  (supérieur ou égal)

i.  $a \leq 1$

$$\begin{aligned} G(s|a) &= a + g(s) && \text{par def de } G \\ &= \text{Fib}(a) + G(s) && \text{par def de Fib} \end{aligned}$$

ii.  $a > 1$

$$\begin{aligned} G(s|a) &= G(s|a-2|a-1) && \text{par def. de } G \\ &= \text{Fib}(a-1) + G(s|a-2) && \text{par l'hypothèse d'induction} \\ &= \text{Fib}(a-1) + \text{Fib}(a-2) + G(s) && \text{idem} \\ &= \text{Fib}(a) + G(s) && \text{par def. de Fib} \end{aligned}$$

□

**Corollaire VII.4.1.2.**  $G(\langle a \rangle) = \text{Fib}(a)$ . □

On peut deviner facilement que la famille  $\{J_K\}$ , d'invariants que nous cherchons est donnée par  $J_K = \{z + G(s) = K\}$ , pour  $K \in \mathbb{N}$ . Le corollaire nous dit de choisir  $K = \text{Fib}(X)$ . Notre invariant est donc  $z + G(s) = \text{Fib}(X)$ . Lors de la terminaison on a  $s = \langle \rangle$ , d'où  $G(s) = 0$ , ce qui implique  $z = \text{Fib}(X)$ , la post-condition souhaitée.

Pour prouver que le programme se termine, considérons la relation  $\epsilon$ ,  $\epsilon \subseteq S \times S$  définie par

$$\begin{aligned} s|a \epsilon s & \quad \text{si } a \leq 1 \\ s|a \epsilon s|a-2|a-1 & \quad \text{si } a > 1 \end{aligned}$$

Nous avons, comme dans le paragraphe précédent :

**Théorème VII.4.1.3.** La relation  $\epsilon$  est noethérienne.

**Démonstration.** Utiliser la même technique que pour le théorème VII.3.3. □

Dans ce cas, contrairement à ce qui se passait dans la section précédente, la relation  $\epsilon$  n'est pas égale à la relation  $GG - \{(\omega, \omega)\}$ . ( $GG$  est définie dans  $(S \times \mathbb{N} \cup \omega)^2$ ). Mais il est facile de voir que

$$\text{si } (\langle s, z \rangle, \langle s', z' \rangle) \in GG - \{(\omega, \omega)\} \text{ alors } (s, s') \in \epsilon$$

ce qui implique que  $GG - \{(\omega, \omega)\}$  est noethérienne aussi. Par conséquent, le programme se termine toujours (cf. corollaire IV.6.12).



#### 4.2. Le cas général.

Revenons à l'étude de la fonction "générique"  $F$  présentée au début de cette section. Supposons  $F \in A \rightarrow B$ , pour deux ensembles arbitraires  $A$  et  $B$ . Alors on a  $p \in A \rightarrow \{\text{vrai, faux}\}$ ,  $f \in A \rightarrow B$ ,  $g_1, g_2 \in A \rightarrow A$ ,  $h \in B \times B \rightarrow B$ ,  $e \in B$ . Nous faisons l'hypothèse supplémentaire que  $h$  admet aussi un élément neutre à droite noté  $e'$  ( $h(a, e') = a$ , pour tout  $a$ ). Cette hypothèse n'est pas restrictive et elle facilite la preuve.

Soit  $S_A$  l'ensemble des séquences finies d'éléments de  $A$ ,  
 $S_A = \bigcup_{i \in \mathbb{N}} A^i$ . Comme d'habitude nous commençons par définir récursivement une fonction  $G$ ,  $G \in S_A \rightarrow B$ , par les équations

$$G(\langle \rangle) = e'$$

$$G(s|a) = \begin{cases} p(a) \rightarrow h(f(a), G(s)) \\ \neg p(a) \rightarrow G(s|g_2(a)|g_1(a)) \end{cases}$$

L'analogie du théorème VII.4.1.1 est :

Théorème VII.4.2.1. Si  $F(a)$  est défini alors

$$G(s|a) = h(F(a), G(s)).$$

Démonstration. Par induction noethérienne sur la relation induite par la définition de  $F$

i.  $p(a) = \text{vrai}$

$$\begin{aligned} G(s|a) &= h(f(a), G(s)) \quad \text{par def. de } G \\ &= h(F(a), G(s)) \quad \text{par def. de } F \end{aligned}$$

ii.  $p(a) = \text{faux}$

$$\begin{aligned} G(s|a) &= G(s|g_2(a)|g_1(a)) && \text{par def. de } G \\ &= h(F(g_1(a)), G(s|g_2(a))) && \text{par l'hypothèse d'induction} \\ &= h(F(g_1(a)), h(F(g_2(a)), G(s))) && \text{idem} \\ &= h(h(F(g_1(a)), F(g_2(a))), G(s)) && \text{par associativité de } h \\ &= h(F(a), G(s)) && \text{par def. de } F \end{aligned}$$

□

Corollaire VII.4.2.2. Si  $F(a)$  est défini alors  $G(\langle a \rangle) = F(a)$ . □

Il doit être clair que la famille d'invariants  $\{J_K\}_{K \in B}$  est, dans ce cas, donnée par  $J_K = \{h(z, G(s)) = K\}$  et que pour la preuve nous devons choisir  $K = F(X)$ . Les égalités  $h(z, G(s)) = F(X)$  et  $s = \langle \rangle$  ( $s = \langle \rangle$  est le prédicat exit) impliquent que si le programme se termine, on aura  $z = F(X)$ , comme souhaité.

Pour prouver la terminaison, définissons une relation  $\epsilon$  entre des séquences de  $S_A$ ,  $\epsilon \subseteq S_A \times S_A$ , par

$$\begin{aligned} s | a \in s & \quad \text{si } p(a) = \text{vrai} \\ s | a \in s | g_2(a) | g_1(a) & \quad \text{si } p(a) = \text{faux et } F(a) \text{ est défini} \end{aligned}$$

Comme précédemment, nous avons

**Théorème VII.4.2.3.** La relation  $\epsilon$  est noethérienne.

**Démonstration.** Analogue à celle de VII.3.3, utilisant ici l'induction noethérienne sur la relation induite par la définition de  $F$ .  $\square$

Le rapport entre les relations  $\epsilon$  et  $GG$  est moins étroit que dans les cas précédents :

le fait que  $p(a)$  soit faux ne garantit pas que  $F(a)$  soit définie et par conséquent  $GG - \{(\omega, \omega)\}$  peut ne pas être noethérienne. Ainsi, nous ne pouvons pas utiliser le corollaire IV.6.12. Nous utilisons directement le théorème IV.6.10.

Soit  $\llcorner$  la fermeture réflexive et transitive de  $\epsilon$ ,  $\llcorner = \epsilon^*$ .  $\llcorner$  est une relation d'ordre bien fondé. Soit le prédicat binaire  $PP$ , défini par  $PP(s, s') = s \llcorner s'$ . Il nous faut trouver un invariant  $J'$  tel que  $J' \subseteq \text{wwpr}[GG](PP)$ .

Soit  $DEF(F)$  le prédicat qui caractérise les points où  $F$  est défini. Définissons  $J'$  par

$$J'(s) = \forall i > 0 \ s(..i) = \langle \rangle \vee DEF(F)(s(i)),$$

i.e.,  $J'(s)$  est vrai si  $F$  est définie pour tous les éléments de la séquence  $s$ . La formule VI.2.2.3 confirme que  $J'$  est un invariant :

$$J'(s) \cap s \neq \langle \rangle \wedge P(s(1)) \subseteq J'(s(..2)) \wedge s \neq \langle \rangle \\ \wedge J(s) \cap s \neq \langle \rangle \wedge \neg P(s(1)) \subseteq J(s(..2) | g_2(s(1)) | g_1(s(1))).$$

La première inclusion est trivialement vraie. La deuxième l'est aussi car si  $F(s(1))$  est défini,  $F(g_2(s(1)))$  et  $F(g_1(s(1)))$  le sont aussi. Nous constatons encore que  $J'$  est vrai à l'initialisation.

Nous pourrions montrer directement que  $J' \subseteq \text{wwpr}[GG](PP)$ , mais il suffit de montrer que  $s = \langle \rangle \vee \text{DEF}(F)(s(1)) \subseteq \text{wwpr}[GG](PP)$  et le résultat cherché en découle par transitivité de la relation  $\subseteq$  :

$$(s = \langle \rangle \vee \text{DEF}(F)(s(1))) \cap s \neq \langle \rangle \wedge p(s(1)) \subseteq s \prec s(..2) \wedge s \neq \langle \rangle \\ \wedge (s = \langle \rangle \vee \text{DEF}(F)(s(1))) \cap s \neq \langle \rangle \wedge \neg p(s(1)) \subseteq s \prec s(..2) | g_2(s(1)) | g_1(s(1))$$

La première inclusion est vraie, trivialement. Le premier membre de la deuxième est équivalent à  $\text{DEF}(F)(s(1)) \cap s \neq \langle \rangle \wedge \neg p(s(1))$ , ce qui montre qu'elle est vraie aussi, d'après la définition de  $\prec$ .

Remarquons à propos que le prédicat  $s = \langle \rangle \vee \text{DEF}(F)(s(1))$  n'est pas un invariant. Par conséquent nous ne pourrions pas l'utiliser directement avec le théorème IV.6.10.

Il nous reste à montrer que si  $F(X)$  n'est pas défini, le programme ne se termine pas. Pour ceci il suffit de trouver un invariant sans blocage  $J''$  tel que le programme soit initialisé dans  $J''$  si  $F(X)$  n'est pas défini. Un invariant dans ces conditions est précisément  $\neg J'$ . Montrons-le.

Soit  $J'' = \neg J'$ . Alors  $J''(s)$  est défini par

$$J''(s) = \exists i > 0 \ s(..i) \neq \langle \rangle \wedge \neg \text{DEF}(F)(s(i)).$$

Utilisons la formule VI.2.2.3 pour montrer qu'il s'agit d'un invariant.

$$J''(s) \cap s \neq \langle \rangle \wedge p(s(1)) \subseteq J''(s(..2)) \wedge s \neq \langle \rangle \\ \wedge J''(s) \cap s \neq \langle \rangle \wedge \neg p(s(1)) \subseteq J''(s(..2) | g_2(s(1)) | g_1(s(1))).$$

La première inclusion est vraie car  $p(s(1))$  implique  $\text{DEF}(F)(s(1))$  et par conséquent le premier membre implique  $\exists i > 1 \ s(..i) \wedge \neg \text{DEF}(F)(s(i))$  i.e.  $J''(s(..2))$ . Pour la deuxième, il y a deux possibilités : ou bien

$\neg \text{DEF}(F)(s(1))$  et dans ce cas on doit avoir soit  $\neg \text{DEF}(F)(g_2(s(1)))$  soit  $\neg \text{DEF}(F)(g_1(s(1)))$ , ce qui implique  $J''(s(..2) | g_2(s(1)) | g_1(s(1)))$  ; ou bien  $\exists i > 1 \ s(..i) \neq \langle \rangle \wedge \neg \text{DEF}(F)(s(i))$  et dans ce cas aussi on a, trivialement,  $J''(s(..2) | g_2(s(1)) | g_1(s(1)))$ .

Il est évident que si  $F(X)$  n'est pas défini le programme est initialisé dans  $J''$ . Pour que  $J''$  soit un invariant sans blocage, il faut que  $J'' \subseteq \neg \text{exit}$ , i.e., que  $J'' \cap \text{exit} = \emptyset$ , mais ceci est trivial car  $\text{exit} = \{s = \langle \rangle\}$ .

Nous terminons ainsi l'étude de cet exemple.

## 5. FONCTIONS DÉFINIES RECURSIVEMENT "SANS PILE"

Dans ce paragraphe, nous présentons quelques exemples supplémentaires d'utilisation d'invariants pour prouver la correction de programmes itératifs qui calculent des fonctions définies récursivement. Dans ces exemples, en général, il n'est pas nécessaire de prévoir une variable séquence pour représenter la "pile" du calcul récursif. Néanmoins l'approche reste essentiellement la même, quoique dans certains cas la preuve soit constructive, c'est-à-dire nous construisons le programme et la preuve simultanément, comme il est recommandé habituellement.

### 5.1, Exemple 1.

Soit  $A$  un ensemble arbitraire et considérons la fonction  $F$ ,  $F \in A \rightarrow A$  définie par

$$F(a) = \begin{cases} p(a) \rightarrow a \\ \neg p(a) \rightarrow F(g(a)) \end{cases}$$

où, par hypothèse  $p$  et  $g$  sont des fonctions totales,  $p \in A \rightarrow \{\text{vrai}, \text{faux}\}$ ,  $g \in A \rightarrow A$ .

Il est bien connu que le programme suivant calcule  $F$  :

```
{F(X) est défini}
x:=X ;
do ¬p(x) → x:=g(x) od
{x=F(X)}.
```

La famille  $\{J_K\}_{K \in A}$  d'invariants de ce programme est définie par  $J_K = \{F(x) = K\}$ . Le programme est initialisé dans  $J_{F(X)} = \{F(x) = F(X)\}$ . Nous avons  $\text{exit} = p$  et donc  $J_{F(X)} \cap \text{exit} \subseteq x = F(X)$ , comme souhaité.

Pour prouver que le programme se termine, il suffit de remarquer que le prédicat  $\text{DEF}(F)$  est un invariant, de considérer l'ordre bien fondé induit par le calcul de  $F$  et d'utiliser le théorème IV.6.10.

Le prédicat  $\neg \text{DEF}(F)$  est aussi un invariant. De plus, il est sans blocage. Par conséquent, si le programme était initialisé avec  $F(X)$  non défini, alors il ne se terminerait pas.

## 5.2. Exemple 2.

Cet exemple est une légère généralisation du précédent. Pour deux ensembles arbitraires  $A$  et  $B$ , considérons la fonction  $G$ ,  $G \in A \rightarrow B$ , définie par

$$G(a) = \begin{cases} p(a) \rightarrow h(a) \\ \neg p(a) \rightarrow G(g(a)) \end{cases}$$

où  $p$ ,  $h$  et  $g$  sont des fonctions totales,  $p \in A \rightarrow \{\text{vrai}, \text{faux}\}$ ,  $h \in A \rightarrow B$ ,  $g \in A \rightarrow A$ .

La fonction  $G$  est en rapport avec la fonction  $F$  de la sous-section précédente comme le décrit le théorème suivant.

**Théorème VII.5.2.1.** Si  $F(a)$  est défini alors  $G(a) = h \circ F(a)$ .

**Démonstration.** Par induction noethérienne avec la relation induite par la définition de  $F$ .



## 5.3. Exemple 3.

Soient  $A$  et  $B$  deux ensembles arbitraires, et définissons une fonction  $F$ ,  $F \in A \rightarrow B$ , par

$$F(a) = \begin{cases} p(a) \rightarrow f(a) \\ \neg p(a) \rightarrow h \circ F \circ g(a) \end{cases}$$

où, comme habituellement  $P$ ,  $f$ ,  $h$  et  $g$  sont toujours définies,  $P \in A \rightarrow \{\text{vrai, faux}\}$ ,  $f \in A \rightarrow B$ ,  $g \in A \rightarrow A$ ,  $h \in B \rightarrow B$ . Nous voulons construire un programme qui calcule  $F$ .

Commençons par définir une fonction  $G$ ,  $G \in \mathbb{N} \times A \rightarrow B$  par

$$G(i, a) = \begin{cases} p(a) \rightarrow h^i \circ f(a) \\ \neg p(a) \rightarrow G(i+1, g(a)) \end{cases}$$

Nous avons le théorème :

**Théorème VII.5.3.1.** Pour tout  $a \in A$  tel que  $F(a)$  est défini il existe un  $n \in \mathbb{N}$  tel que pour tout  $i \in [0, n]$  on a  $h^i \circ F \circ g^i(a) = G(i, g^i(a))$ .

**Démonstration.** Soit  $a$  tel que  $F(a)$  est défini. Alors par la relation noethérienne induite par la définition de  $F$  il doit exister une suite finie unique  $a, g(a), \dots, g^{n-1}(a), g^n(a)$ ,  $n \geq 0$ , tel que  $\neg p(a)$ ,  $\neg p(g(a))$ ,  $\dots$ ,  $\neg p(g^{n-1}(a))$  et  $p(g^n(a))$ .

D'autre part la relation  $\varepsilon$  définie dans  $[0, n]$  par  $(k, l) \in \varepsilon$  si  $l = k+1$  (i.e.  $\varepsilon = \{(0, 1), (1, 2), \dots, (n-1, n)\}$ ) est, clairement, noethérienne. Le théorème est prouvé par induction noethérienne avec cette relation :

i.  $i = n$  (base de l'induction)

$$\begin{aligned} h^n \circ F \circ g^n(a) &= h^n \circ f \circ g^n(a) \quad \text{car } p(g^n(a)) \text{ est vrai} \\ &= G(n, g^n(a)) \quad \text{idem} \end{aligned}$$

ii.  $0 \leq i < n$

$$\begin{aligned} h^i \circ F \circ g^i(a) &= h^i \circ h \circ F \circ g \circ g^i(a) \quad \text{car } p(g^i(a)) = \text{faux} \\ &= h^{i+1} \circ F \circ g^{i+1}(a) \\ &= G(i+1, g^{i+1}(a)) \quad \text{par l'hypothèse d'induction} \\ &= G(i, g^i(a)) \quad \text{car } p(g^i(a)) = \text{faux} \end{aligned}$$

□

Corollaire VII.5.3.2. Si  $F(a)$  est défini alors  $F(a)=G(0,a)$ .

Le corollaire indique que pour calculer  $F(X)$  il suffit de calculer  $G(0,X)$ . Puisque la fonction  $G$  est du type considéré dans la sous-section VII.5.2, un programme pour  $F$  est le suivant :

```
{F(X) est défini}
i,x:=0,X ;
do ¬p(x) → i,x:=i+1,g(x) od ;
y:=hi.f(x)
{y=F(X)}.
```

Ce programme, cependant, n'est pas très réaliste, à cause de l'affectation  $y:=h^i.f(x)$ . Nous devons donc le transformer pour que le calcul de  $h^i.f(x)$  soit effectué par itération. Nous obtenons ainsi :

```
{F(X) est défini}
i,x:=0,X ;
do ¬p(x) → i,x:=i+1,g(x) od ;
y:=f(x) ;
do i≠0 → i,y:=i-1, h(y) od
{y=F(X)}.
```

Les invariants de la première boucle sont  $G(i,x)=K1$  et ceux de la seconde sont  $h^i(y)=K2$ . L'opération d'initialisation indique que l'on doit choisir  $K1=G(0,X)=F(X)$ . Quand la première boucle se termine on a  $p(x) \wedge \{G(i,x)=F(X)\}$  ce qui implique  $h^i.f(x)=F(X)$ . Après l'affectation on a certainement  $h^i(y)=F(X)$  car  $wpr\{y:=f(x)\}(h^i(y)=F(X))=\{h^i(f(x))=F(X)\}$ . Par conséquent on doit aussi choisir  $K2=F(X)$ . A la fin de la seconde boucle on a  $i=0$  et donc  $y=F(X)$ , comme souhaité.

La terminaison de la première boucle est assurée par l'ordre bien fondé induit par le calcul de  $F$ . Celle de la deuxième est triviale, pourvu qu'elle soit initialisée avec  $i \geq 0$ , ce qui arrive en fait ( $i \geq 0$  est un invariant de la première boucle et de l'affectation intermédiaire).



## 5.4. Exemple 4.

Considérons, pour deux ensembles arbitraires A et B la fonction F,  $F \in A \rightarrow B$  définie récursivement par

$$F(a) = \begin{cases} p(a) \rightarrow f(a) \\ \neg p(a) \rightarrow g(F(h(a)), a) \end{cases}$$

où p, f, g et h sont toujours définies,  $p \in A \rightarrow \{\text{vrai}, \text{faux}\}$ ,  $f \in A \rightarrow B$ ,  $g \in B \times A \rightarrow B$ ,  $h \in A \rightarrow A$ .

Comme dans l'exemple précédent, nous voulons écrire un programme itératif qui calcule F. Etant donné que dans ce cas la fonction F n'est pas "tail recursive" nous commençons par chercher une solution avec une variable séquence pour représenter la "pile".

Définissons une fonction  $H \in S_A \rightarrow B$  par

$$H(s|a) = \begin{cases} p(a) \rightarrow G(f(a), s) \\ \neg p(a) \rightarrow H(s|a|h(a)) \end{cases}$$

où la fonction G,  $G \in B \times S_A \rightarrow B$ , est définie par les équations

$$\begin{aligned} G(b, \langle \rangle) &= b \\ G(b, s|a) &= G(g(b, a), s). \end{aligned}$$

Le rapport existant entre les trois fonctions F, H et G est décrit par le théorème suivant

**Théorème VII.5.4.1.** Si F(a) est défini alors  $H(s|a) = G(F(a), s)$ .

**Démonstration.** Par induction noethérienne avec la relation induite par la définition de F :

i.  $p(a) = \text{vrai}$

$$\begin{aligned} H(s|a) &= G(f(a), s) && \text{par def. de H} \\ &= G(F(a), s) && \text{par def. de F} \end{aligned}$$

ii.  $p(a)=\text{faux}$

$$\begin{aligned} H(s|a) &= H(s|a|h(a)) && \text{par def. de H} \\ &= G(F(h(a)), s|a) && \text{par l'hypothèse d'induction} \\ &= G(g(F(h(a)), a), s) && \text{par def. de G} \\ &= G(F(a), s) && \text{par def. de G} \end{aligned}$$

□

Corollaire VII.5.4.2. Si  $F(a)$  est défini alors  $F(a)=H(\langle a \rangle)$ .

Au vu de ce corollaire, pour calculer  $F(X)$ , il suffit de calculer  $H(\langle X \rangle)$ . Mais  $H$  est une fonction du type étudié dans le sous-paragraphe VII.5.2. Par conséquent un programme pour calculer  $F(X)$  est le suivant.

```
{F(X) est défini}
x,s:=X,<> ;
do ¬p(x) → x,s:=h(x),s|x od ;
y := G(f(x),s)
{y = F(X)}.
```

Par ailleurs, la fonction  $G$  est aussi du type considéré dans VII.5.2. Le programme précédent peut ainsi être transformé en

```
{F(X) est défini}
x,s:=X,<> ;
do ¬p(x) → x,s:=h(x),s|x od
y := f(x) ;
do s≠<> → y,s:=g(y,s(1)),s(..2) od
{y = F(X)}.
```

Les invariants de la première boucle sont  $H(s|x)=K1$  et ceux de la deuxième  $G(y,s)=K2$ . Pour la première boucle on doit choisir  $H(s|x)=F(x)$  ; ainsi l'invariant est vérifié à l'initialisation. A la terminaison de la première boucle, (elle se termine certainement à cause de l'ordre bien fondé induit par le calcul de  $F$ ), on a  $p(x)=\text{vrai}$ , ce qui implique  $G(f(x),s)=F(X)$ . A cause de l'égalité  $\text{wpr}[y:=f(x)](G(y,s)=F(X)) = \{G(f(x),s)=F(X)\}$ , après l'affectation on a  $\{G(y,s)=F(X)\}$ . C'est ce prédicat que nous

choisissons comme invariant de la deuxième boucle. Quand celle-ci se termine, (ce qui arrive toujours, trivialement),  $s = \langle \rangle$  est vrai, impliquant  $y = F(X)$ , tel que nous le souhaitons.

On peut observer que le rôle de la variable  $s$  pendant la première boucle est de mémoriser les puissances successives de  $h(X)$  (i.e.,  $s = \langle X, h(X), \dots, h^i(X) \rangle$ ) pour qu'elles puissent être utilisées dans l'ordre inverse par la deuxième boucle. Ceci suggère qu'il est possible de se passer de la variable séquence si la fonction  $h$  a une inverse  $h^{-1}$ , car dans ces conditions, on peut calculer les puissances décroissantes de  $h$  directement, ( $h^{i-1}(X) = h^{-1}(h^i(X))$ ). Nous pouvons donc, dans le cas où  $h$  a une inverse, remplacer la variable séquence  $s$  par une simple variable entière qui compte le nombre de fois que la première boucle est exécutée. Nous obtenons ainsi le programme

```
{F(X) est défini}
i, x := 0, X ;
do ¬p(x) → i, x := i+1, h(x) od ;
y, x := f(x), h-1(x) ;
do i ≠ 0 → i, y, x := i-1, g(y, x), h-1(x) od
{y = F(X)}.
```

Nous devons maintenant confirmer notre raisonnement par une preuve formelle de la correction de ce programme.

Commençons par remarquer que le programme est du même type que celui obtenu dans le sous-paragraphe précédent. Nous concluons, par analogie, qu'il calcule une fonction  $\phi$ ,  $\phi \in A \rightarrow B \times A$  définie par

$$\phi(a) = \begin{cases} p(a) \rightarrow \phi(a) \\ \neg p(a) \rightarrow \gamma \circ \phi \circ h(a) \end{cases}$$

avec  $\phi \in A \rightarrow A \times B$  et  $\gamma \in B \times A \rightarrow B \times A$  données par

$$\begin{aligned} \phi(a) &= \langle f(a), h^{-1}(a) \rangle \\ \gamma(b, a) &= \langle g(b, a), h^{-1}(a) \rangle. \end{aligned}$$

Par conséquent pour prouver le programme, il suffit de montrer que si  $F(a)$  est défini alors  $\phi(a)$  l'est aussi, et que la valeur du premier élément du doublet  $\phi(a)$  est égal à  $F(a)$ . Cette question est réglée par le théorème suivant.

Théorème VII.5.4.3. Si  $F(a)$  est défini alors  $\Phi(a) = \langle F(a), h^{-1}(a) \rangle$ .

Démonstration. Par induction noethérienne avec la relation induite par la définition de  $F$

i.  $p(a) = \text{vrai}$

$$\begin{aligned} \Phi(a) &= \phi(a) && \text{par def. de } \Phi \\ &= \langle f(a), h^{-1}(a) \rangle && \text{par def. de } \phi \\ &= \langle F(a), h^{-1}(a) \rangle && \text{par def. de } F \end{aligned}$$

ii.  $p(a) = \text{faux}$

$$\begin{aligned} \Phi(a) &= \gamma(\Phi(h(a))) && \text{par def. de } \Phi \\ &= \gamma(F(h(a)), h^{-1}(h(a))) && \text{par l'hypothèse d'induction} \\ &= \gamma(\langle F(h(a)), a \rangle) \\ &= \langle g(F(h(a)), a), h^{-1}(a) \rangle && \text{par def. de } \gamma \\ &= \langle F(a), h^{-1}(a) \rangle && \text{par def. de } F \end{aligned}$$

□

Nous venons de prouver que le programme obtenu pour le calcul de  $F(X)$  quand  $h$  a une inverse est correct. Il serait toutefois dommage de ne pas essayer de le simplifier. Intuitivement, la variable  $i$  ne devrait pas être indispensable car elle ne sert qu'à arrêter la deuxième boucle, et ceci peut aussi être obtenu en testant  $x = h^{-1}(X)$  comme le suggère le théorème VII.5.4.3, car toutes les puissances  $h^i(X)$  sont différentes, (par la relation noethérienne induite par la définition de  $F$ ).

Nous ne pouvons justifier cette simplification que d'une manière semi-formelle car nous n'avons pas présenté dans ce rapport de résultats précis sur la manipulation de programmes. Néanmoins, le principe de simplification suivant est très intuitif et facilement compréhensible :

Principe de simplification de commandes do-od. (Se reporter à la syntaxe et à la sémantique relationnelle du langage des commandes gardées, paragraphes V.2. et V.3.). Si le prédicat  $J$  est un invariant de  $\text{sigma}[[s]]$  et la commande do s od est placée dans un contexte où l'on peut garantir que le prédicat  $J$  est vérifié quand l'exécution de do s od est initialisé alors on peut remplacer do s od par do s' od

avec  $s' = c'_1 \square \dots \square c'_n$  et  $c'_i = J \text{ and } c_i$ , et réciproquement. (Ce principe est une application de la propriété suivante :

si  $J$  est un invariant de  $R$  alors  $I_{J \cup \Omega} \circ R^{\otimes} = (I_{J \cup \Omega} \circ R)^{\otimes}$ ).

C'est-à-dire, on peut multiplier les gardes d'une commande do od par un invariant, pourvu que l'invariant soit vérifié à l'entrée.

Avant d'appliquer ce principe, observons que les affectations simultanées à  $x$  et  $y$  peuvent être décomposées à condition que  $y$  soit affecté avant  $x$ , ce qui donne le programme

```
{F(X) est défini}
i,x:=0,X ;
do  $\neg p(x) \rightarrow i,x:=i+1,h(x)$  od ;
y := f(x) ;
x :=  $h^{-1}(x)$  ;
do  $i \neq 0 \rightarrow y:=g(y,x)$  ;
       $i,x:=i-1,h^{-1}(x)$ 
od
{y=F(X)}.
```

Dans cette forme, le programme est manifestement équivalent à

```
{F(X) est défini}
i,x:=0,X ;
do  $\neg p(x) \rightarrow i,x:=i+1,h(x)$  od ;
y:=f(x) ;
do  $i \neq 0 \rightarrow i,x:=i-1,h^{-1}(x)$  ;
      y:=g(y,x)
od ;
x:= $h^{-1}(x)$ 
{y=F(X)}.
```

La dernière affectation n'apporte rien à la correction du programme et peut être éliminée. D'autre part, le prédicat  $x=h^{-1}(X)$  est un invariant de la première boucle ainsi que de l'affectation intermédiaire et il est vrai à l'initialisation. Par conséquent, il est vrai à l'entrée de la

deuxième boucle. Par ailleurs, il est aussi un invariant de la deuxième boucle. Nous pouvons donc appliquer le principe de simplification et remplacer la garde de la deuxième boucle par  $i \neq 0$  and  $x = h^i(X)$ . Mais nous avons  $i \neq 0 \wedge x = h^i(X) \subseteq x \neq X$ , ce qui implique  $\{i \neq 0 \wedge x = h^i(X)\} = \{i \neq 0 \wedge x = h^i(X) \wedge x \neq X\}$ . Nous avons aussi  $x = h^i(X) \wedge x \neq X \subseteq i \neq 0$  et par conséquent  $\{i \neq 0 \wedge x = h^i(X)\} = \{x \neq X \wedge x = h^i(X)\}$ . Nous pouvons, donc remplacer la garde  $i \neq 0$  and  $x = h^i(X)$  par  $x \neq X$  and  $x = h^i(X)$ . Il est évident que  $x = h^i(X)$  reste un invariant de la commande ainsi obtenue. Par conséquent, nous pouvons appliquer la transformation en sens inverse pour l'enlever de la garde. En résumé, la deuxième commande do-od peut être remplacée par

$$\underline{\text{do}} \ x \neq X \rightarrow i, x := i-1, h^{-1}(x) ; y := g(y, x) \ \underline{\text{od}}.$$

Ayant été enlevée de la garde, la variable  $i$  devient une variable auxiliaire, au sens de Owicki and Gries [OwG76], qui nous a servi pour la preuve mais qui n'intervient pas en ce qui concerne le résultat du programme. Nous pouvons donc la supprimer partout et obtenir ainsi, comme résultat final de notre simplification, le programme

```

{F(X) est défini}
x := X ;
do ¬p(x) → x := h(x) od ;
y := f(x) ;
do x ≠ X → x := h-1(x) ;
           y := g(y, x)
od
{y = F(X)}.

```

Il est intéressant de constater que ce programme correspond précisément à celui obtenu pour le même problème par Broy et Krieg-Bruckner [BrK80], qui ont utilisé une technique complètement différente.

Nous terminons ici l'étude de cet exemple.

### 5.5. Exemple 5.

Soient  $B$  un ensemble arbitraire et  $F$ , une fonction,  $F \in \mathbb{N} \rightarrow B$  définie par les équations

$$\begin{aligned} F(0) &= b_0 \\ F(n+1) &= h(n, F(n)) \end{aligned}$$

où  $h \in \mathbb{N} \times B \rightarrow B$  est toujours définie. Il est évident que  $F$  est totale et qu'un programme qui la calcule est

```
{X ∈ ℕ}
i, y := 0, b0 ;
do i ≠ X → i, y := i+1, h(i, y) od
{Y = f(X)}.
```

Il est facile de vérifier que le prédicat  $y=F(i)$  est un invariant qui est vérifié à l'initialisation et qui avec le prédicat exit implique la post-condition. Pour obtenir la famille d'invariants correspondante il faut "libérer" la constante  $b_0$ . Définissons alors une fonction  $G$ ,  $G \in B \times \mathbb{N} \rightarrow B$  par

$$\begin{aligned} G(b, 0) &= b \\ G(b, n+1) &= h(n, G(b, n)). \end{aligned}$$

Dans ces conditions,  $F(n) = G(b_0, n)$ , trivialement. Naturellement, la famille d'invariants  $\{J_K\}_{K \in B}$  est définie par  $J_K = \{y=G(K, i)\}$ .

### 5.6. Exemple 6.

Notre dernier exemple est une fonction  $F$ ,  $F \in A \rightarrow A$ , pour un ensemble arbitraire  $A$ , définie par

$$F(a) = \begin{cases} p(a) \rightarrow f(a) \\ \neg p(a) \rightarrow F(F(g(a))) \end{cases}$$

où  $p$ ,  $f$  et  $g$  sont toujours définies,  $p \in A \rightarrow \{\text{vrai}, \text{faux}\}$ ,  $f \in A \rightarrow A$ ,  $g(a) \in A \rightarrow A$ . Un exemple bien connu de fonctions de ce type est la fonction 91 pour laquelle  $A = \mathbb{Z}$ ,  $p(a) = a > 100$ ,  $f(a) = a-10$  et  $g(a) = a+11$ .

Remarquons que

$$F^2(a) = \begin{cases} p(a) \rightarrow F(f(a)) \\ \neg p(a) \rightarrow F(F(g(a))) \end{cases}$$

et, en général, pour  $i \geq 1$

$$F^i(a) = \begin{cases} P(a) \rightarrow F^{i-1}(f(a)) \\ \neg p(a) \rightarrow F^{i+1}(g(a)) \end{cases}$$

Définissons une fonction  $G$ ,  $G \in \mathbb{N} \times A \rightarrow A$ , par  
 $G(i, a) = F^i(a)$  si  $F^i(a)$  est défini.

Cette définition implique que si  $F^i(a)$  est défini alors

$$G(i, a) = \begin{cases} i = 0 \rightarrow a \\ i > 0 \rightarrow \begin{cases} p(a) \rightarrow G(i-1, f(a)) \\ \neg p(a) \rightarrow G(i+1, g(a)) \end{cases} \end{cases}$$

et si  $i > 0$  alors si  $p(a)$  est vrai  $G(i-1, f(a))$  est défini sinon  $G(i+1, g(a))$  est défini. En particulier, si  $F(a)$  est défini alors  $F(a) = G(1, a)$ . Par conséquent, nous pouvons utiliser un programme qui calcule  $G$  pour calculer  $F$ . En remarquant que  $G$  est du type étudié dans le paragraphe VII.5.2, on conclut qu'un programme pour  $F$  est le suivant :

```
{F(X) est défini}
i, x := 1, X ;
do i > 0 → i, x := if p(a) then (i-1, f(x)) else (i+1, g(x)) od
{x = F(X)}.
```

Ce programme est facilement manipulable pour prendre une forme plus agréable :

```
{F(X) est défini}
i, x := 1, X ;
do i > 0 and p(x) → i, x := i-1, f(x)
□ i > 0 and ¬p(x) → i, x := i+1, g(x)
od
{x = F(X)}.
```



La famille d'invariants est  $G(i,x) = K$ , i.e.,  $F^i(x) = K$ . Le programme est initialisé dans  $F(x) = F(X)$  et termine quand  $i=0$ , ce qui implique  $F^0(x) = F(X)$ , c'est-à-dire,  $x=F(X)$ , comme souhaité.

La terminaison peut être prouvée de manière standard avec la relation induite par la définition de  $G$  (remarquons qu'il s'agit d'une relation binaire définie dans  $\mathbb{N} \times A$ ).

Nous terminons le traitement de cet exemple en présentant un joli petit programme qui calcule la fonction 91.

```

i,x := 1,X ;
do i > 0 and x > 100 → i,x:=i-1,x-10
  □ i > 0 and x ≤ 100 → i,x:=i+1,x+11
od
{x=f91(X)}.

```

## 6. CONCLUSION

Nous espérons avoir montré avec les exemples traités que l'approche dite conventionnelle de la preuve de programmes garde tout son pouvoir et toute sa clarté même dans le cas des programmes itératifs qui calculent des fonctions définies récursivement. Ainsi, les critiques de Manna et Waldinger rappelées dans l'introduction de ce chapitre nous apparaissent non justifiées. Une fois leur principe compris, les preuves présentées sont en fait assez naturelles, élégantes et bien structurées, et, finalement, elles ne sont pas fondamentalement différentes d'autres preuves typiques telles que celle du programme pour la fonction pgcd.

D'autre part, ces exemples nous confortent dans la conviction qu'il y aura toujours dans la recherche de preuves une place importante pour l'intuition humaine. En fait, nous ne voyons pas comment les méthodes automatiques de calcul d'invariants par calcul de points fixes pourraient fournir des invariants, pourtant bien simples, comme  $F^i(x)=F(X)$  dans le dernier exemple.

## CHAPITRE VIII

SEMANTIQUE RELATIONNELLE DE PROCESSUS COMMUNICANTS



## 1. INTRODUCTION

Dès la parution du remarquable article de Hoare "Communicating Sequential Processes", [Hoa78b], l'étude des processus qui communiquent directement, pour lesquels le mécanisme primitif de communication n'est pas réalisé explicitement par des variables partagées, a été le sujet de nombreux travaux qu'il serait fastidieux d'énumérer. Parmi tous ces travaux les plus marquants sont ceux du groupe d'Edimbourg notamment l'article "Concurrent Processes and Their Syntax" de Milne et Milner [MiM79] et plus récemment le livre de Milner "A Calculus for Communicating Systems" [Mil80].

Dans son article Hoare propose un langage de programmation de processus communicants, le langage CSP, adapté à la description de la communication directe entre processus. Son approche est pragmatique et il recherche surtout la commodité d'écriture et la lisibilité des programmes sans se soucier de présenter une sémantique formelle (ou de suggérer une implémentation efficace). De son côté Milner se préoccupe principalement des fondements abstraits du mécanisme de communication et leur incidence sur le comportement "observable" des processus et systèmes de processus, qu'il étudie à l'aide d'un formalisme algébrique. Ce formalisme est utilisé par Hennessy, Li et Plotkin [HLP81] pour donner une description sémantique d'un sous-ensemble significatif du langage CSP.

Outre cette approche algébrique, d'autres techniques ont été essayées pour fournir une description formelle de CSP. Nous avons recensé les suivantes :

"mécanistique", par Hoare [Hoa79] (cf. paragraphe V.1.) ; dénotationnelle, par Francez et al. [FHLR79] ; axiomatique, par Apt, Francez et de Roever [AFR80] ; par plus faibles pré-conditions, des mêmes auteurs [AFR79] ; "linear history semantics" par Francez, Lehmann et Pnueli [FLP80] ; opérationnelle, utilisant des relations de transition par Cousot et Cousot [CoC80a] ; réseaux de Petri, par Queille [Que80] et par De Cindio et al. [CMPS80]. Toutes ces propositions sont importantes car elles contribuent d'une manière ou d'une autre à éclairer le concept fondamental de processus communicant. En plus de la description rigoureuse de

la version de CSP considérée dans chaque cas, elles ont pour but commun la découverte de méthodes de preuve pour les programmes parallèles écrits en CSP. Nous constatons néanmoins que les méthodes de preuve proposées, ou suggérées, sont assez particulières et, en générale, assez éloignées des techniques conventionnelles de preuve de programmes.

L'objectif de ce chapitre est de présenter une sémantique pour une classe de processus séquentiels communicants. Il s'agit d'une sémantique relationnelle, basée sur des idées qui découlent directement des chapitres précédents. Elle induit une technique de preuve des propriétés des processus consistant à obtenir un programme non-déterministe dont le comportement est, sous certaines hypothèses, équivalent au comportement du système de processus considéré. Ainsi les propriétés intéressantes du système de processus se reflètent dans le programme non-déterministe, et elles peuvent être étudiées indirectement à l'aide de celui-ci. Cette démarche présente plusieurs avantages, d'une part parce que nous pouvons utiliser une grande quantité de résultats fort bien établis, en particulier ceux issus de la théorie relationnelle développée dans ce rapport, et d'autre part parce que la méthode conventionnelle étant bien connue, elle nous permet parfois de rendre informels certains aspects des preuves. Dans ces conditions on peut éviter certains détails formels encombrants, ce qui rend les preuves plus compréhensibles et naturelles.

## 2. SYNTAXE

La raison définir une sémantique formelle pour un langage n'est pas seulement de préciser une signification abstraite des constructions syntaxiques utilisées, mais aussi de fournir un support mathématique à l'aide duquel on puisse saisir l'intuition que l'on a des objets "réels" que l'on veut décrire.

Une sémantique donnée, étant un objet mathématique, peut être étudiée formellement. Par contre, il est évident que la question de savoir si elle capte bien l'intuition initiale ne peut être décidée qu'informellement.

D'autre part une sémantique sera d'autant plus naturelle et compréhensible que le formalisme choisi s'adapte bien à la description qu'on veut faire. Ainsi, nous devrions commencer par présenter intuitivement les objets que nous voulons traiter, c'est-à-dire, les processus communicants.

Toutefois nous introduisons d'abord le formalisme et seulement ensuite les objets auxquels il s'applique, ce qui est plus commode et représente d'ailleurs une démarche courante en Informatique.

Le formalisme que nous choisissons est inspiré du langage des commandes gardées de Dijkstra (voir chapitre V), du langage CSP et utilise des notions provenant des processus concurrents de Milner [MiM78], notamment le concept de porte <sup>(1)</sup> et la communication bidirectionnelle.

La syntaxe abstraite d'un processus est la suivante :

$$\begin{array}{l}
 X ; \\
 \underline{\text{do}} \quad g_1 \rightarrow r_1 \\
 \quad \quad \square g_2 \rightarrow r_2 \\
 \quad \quad \cdot \\
 \quad \quad \cdot \\
 \quad \quad \cdot \\
 \quad \quad \square g_n \rightarrow r_n \\
 \quad \quad \square k_{\alpha_1} : \alpha_1 \rightarrow o_{\alpha_1}, j_{\alpha_1} \\
 \quad \quad \cdot \\
 \quad \quad \cdot \\
 \quad \quad \cdot \\
 \quad \quad \square k_{\alpha_m} : \alpha_m \rightarrow o_{\alpha_m}, j_{\alpha_m} \\
 \underline{\text{od}}
 \end{array}$$

X est l'état initial du processus. Les  $g \rightarrow r$  sont des commandes gardées comme dans le langage des commandes gardées : les g sont des gardes, appelés ici gardes internes, et les r sont des commandes arbitraires, (qui peuvent être non-déterministes). Les  $\alpha$  sont des portes. Toutes les portes d'un processus sont différentes, i.e.,  $\alpha_i \neq \alpha_j$  si  $i \neq j$ .  $k_\alpha$  est une garde, appelée garde de la porte  $\alpha$ .  $o_\alpha$  est

---

(1)

Nous utilisons une traduction incorrecte du mot anglais "port"

la fonction de sortie de la porte  $\alpha$  et  $j_\alpha$  est la fonction d'entrée de la porte  $\alpha$ . Chaque construction  $k_\alpha : \alpha \rightarrow o_\alpha$ ,  $j_\alpha$  est appelée une communication gardée.

On peut commencer à deviner le fonctionnement d'un processus si on concrétise un peu la syntaxe abstraite :

$$\begin{array}{l}
 P:: x:=X; \\
 \quad \underline{\text{do}} \quad g_1(x) \rightarrow r_1(x) \\
 \quad \quad \cdot \\
 \quad \quad \cdot \\
 \quad \quad \cdot \\
 \quad \square \quad gn(x) \rightarrow rn(x) \\
 \quad \square \quad k_{\alpha_1}(x) : \alpha_1 \rightarrow !o_{\alpha_1}(x), \ ? \ s : x := j_{\alpha_1}(s, x) \\
 \quad \quad \cdot \\
 \quad \quad \cdot \\
 \quad \quad \cdot \\
 \quad \square \quad k_{\alpha_m}(x) : \alpha_m \rightarrow !o_{\alpha_m}(x), \ ? \ s : x := j_{\alpha_m}(s, x) \\
 \quad \underline{\text{od}}
 \end{array}$$

P est le nom du processus.  $x$  est la variable du processus dont la valeur représente l'état courant du processus. Nous notons par  $Q$  l'espace d'états de  $P$  et nous le supposons implicitement défini, et tel que  $\omega \in Q$ . On peut alors dire que  $x$  parcourt  $Q$ , ou est de "type"  $Q$ .  $x:=X$  est l'opération d'initialisation de  $P$ , définissant  $X$ ,  $X \in Q$ , comme état initial de  $P$ .

Les gardes sont des prédicats sur  $Q$ . Il convient de définir  $g(\omega)=\text{vrai}$ , pour les gardes internes, mais  $k_\alpha(\omega)=\text{faux}$  pour les gardes de portes. L'idée d'une commande gardée  $g \rightarrow r$  est la suivante :  $g \rightarrow r$  n'est exécutable qu'à partir d'un état  $x$  tel que  $g(x)=\text{vrai}$  ; l'exécution de  $g \rightarrow r$  consiste dans l'exécution de  $r$ , ce qui amène le processus de l'état  $x$  à un état de  $r(x)$ . (Rappelons que d'après la sémantique étudiée au chapitre V,  $r(x)$  représente un ensemble d'états).

Nous avons déjà dit que toutes les portes du processus  $P$  sont différentes. Nous faisons maintenant l'hypothèse supplémentaire que ce

processus existe dans un système de processus formant un "réseau simple" de processus, où toutes les portes de tous les processus sont différentes et où dans l'ensemble de toutes les portes est définie une bijection, égale à son inverse, qui à une porte d'un processus fait correspondre une autre porte dans un autre processus. Représentant par  $\text{PORTES}(P)$  l'ensemble des portes du processus  $P$ , et par  $(\bar{\quad})$  la bijection mentionnée nous avons formellement, pour un réseau simple :

- i.  $\text{PORTES}(P) \cap \text{PORTES}(P') = \emptyset$  si  $P \neq P'$
- ii.  $\alpha \in \text{PORTES}(P) \Rightarrow \exists P' \neq P \ \bar{\alpha} \in \text{PORTES}(P')$
- iii.  $\bar{\bar{\alpha}} = \alpha$

Ainsi une paire  $\{\alpha, \bar{\alpha}\}$  peut être considérée comme une ligne de transmission entre les processus auxquels appartiennent les portes  $\alpha$  et  $\bar{\alpha}$ .

L'écriture  $o_\alpha(x)$  indique que les arguments de la fonction  $o_\alpha$  sont des états de  $Q$ . On suppose que  $o_\alpha$  est toujours bien définie dans  $k_\alpha$ , c'est-à-dire, si  $k_\alpha(x) = \text{vrai}$  alors  $o_\alpha(x)$  est défini. L'idée est que si le processus se trouve dans un état  $x$  validant la garde  $k_\alpha$  il se peut qu'il envoie par la porte  $\alpha$  la valeur  $o_\alpha(x)$ . Les valeurs susceptibles d'être envoyées par la porte  $\alpha$  appartiennent à un ensemble noté  $U_\alpha$  et appelé le "type" de la porte  $\alpha$ . On peut donc dire que  $o_\alpha$  est une application de  $k_\alpha \rightarrow U_\alpha$ . Nous faisons l'hypothèse que  $\omega \notin U_\alpha$ , ce qui est justifié par le fait que  $k_\alpha(\omega) = \text{faux}$ .

Symétriquement, l'écriture  $?s:x:=j_\alpha(s,x)$  suggère que la tâche de la fonction d'entrée  $j_\alpha$  est de calculer un nouvel état à partir de la valeur  $s$  reçue par la porte  $\alpha$  et de l'état courant  $x$ . Les valeurs qui peuvent "entrer" par la porte  $\alpha$  sont, par hypothèse de construction, celles qui peuvent sortir par la porte complémentaire  $\bar{\alpha}$ . Par conséquent  $j_\alpha$  est une application de  $k_\alpha \times U_\alpha^- \rightarrow Q$ .

Cette discussion montre déjà que pour qu'il y ait transmission de valeurs par la ligne  $\{\alpha, \bar{\alpha}\}$  il faut que les deux processus qui possèdent les portes  $\alpha$  et  $\bar{\alpha}$  soient "simultanément" dans des états validant les gardes de ces portes.



### 3. SEMANTIQUE INTUITIVE

Nous regardons les processus qui nous intéressent comme des mécanismes abstraits, agissant sur des ensembles d'états et capables de réaliser des actions qui provoquent un changement d'état dans ceux qui les entreprennent. Nous considérons, grossièrement, qu'il y a deux types d'actions : celles qui sont réalisées par un seul processus et celles qui exigent la participation de plusieurs processus. Les premières sont appelées actions internes et les secondes communications.

Une action interne est propre à un processus donné. C'est lui seul qui est en mesure de décider de l'entreprendre ou non. Dans le formalisme une action interne est représentée par une commande gardée  $g \rightarrow r$ . Elle ne peut avoir lieu que si le processus se trouve dans un état validant la garde  $g$  auquel cas l'action consiste en l'exécution de la commande  $r$ .

Les communications impliquent la participation de plusieurs processus. Pour qu'une communication puisse avoir lieu il faut que tous les partenaires soient en mesure d'y participer. Le formalisme présenté n'est adapté qu'à l'expression des communications entre deux processus, et pour cela nous pouvons restreindre notre discussion à ce type de communication. Mais il est clair qu'il n'existe rien de particulier dans une communication à deux qui empêche une généralisation à un nombre quelconque des partenaires.

Une communication entre deux processus  $P_1$  et  $P_2$  ne peut avoir lieu que s'il existe une porte  $\alpha$  telle que  $\alpha \in \text{PORTES}(P_1)$  et  $\bar{\alpha} \in \text{PORTES}(P_2)$ . Dans ces conditions elle est représentée par la paire de communications gardées  $k_\alpha : \alpha \rightarrow o_\alpha, j_\alpha$  dans  $P_1$  et  $k_{\bar{\alpha}} : \bar{\alpha} \rightarrow o_{\bar{\alpha}}, j_{\bar{\alpha}}$  dans  $P_2$ . Ceci indique que la communication ne peut se produire que si  $P_1$  est dans un état  $x_1$  tel que  $k_\alpha(x_1) = \text{vrai}$  et  $P_2$  est dans un état  $x_2$  tel que  $k_{\bar{\alpha}}(x_2) = \text{vrai}$ . Elle consiste alors en la mise à disposition de  $P_2$  par  $P_1$  de la valeur  $o_\alpha(x_1)$  que  $P_2$  utilise avec son état courant pour calculer par  $j_{\bar{\alpha}}$

son prochain état, et symétriquement pour P1. C'est-à-dire si l'action de communication a effectivement lieu P1 passe dans l'état  $j_{\alpha}^{-}(o_{\alpha}^{-}(x2), x1)$ , et P2 dans l'état  $j_{\alpha}^{-}(o_{\alpha}^{-}(x1), x2)$ .

Dans un premier temps on peut considérer que les actions ont une certaine durée. Mais nous faisons l'hypothèse, d'ailleurs clairement exprimée par le formalisme, que le résultat d'une action dépend seulement des états des processus qui l'entreprennent au moment où l'action commence à avoir lieu. C'est-à-dire, il n'y a rien qui puisse interrompre ou influencer le déroulement d'une action déjà entamée. Tout se passe comme si les actions étaient indivisibles. Si l'on exclut de la discussion les actions qui ne se terminent pas cette observation nous permet de faire abstraction de la durée des actions pour considérer qu'elles sont instantanées. Il faut alors supposer aussi qu'entre deux actions successives d'un processus intervient une période de "repos" pour le processus. (On peut remarquer que la durée de cette période de repos doit avoir une borne inférieure positive, pour éviter la situation paradoxal d'un processus qui peut réaliser un nombre infini d'actions dans un temps fini).

A chaque instant s'offre à un processus un ensemble d'actions susceptibles d'avoir lieu. Cet ensemble peut varier avec le temps même sans que le processus ait entrepris une action. Ceci est dû au fait qu'une communication possible peut devenir impossible, ou vice versa, le partenaire ayant entre-temps changé d'état de manière indépendante. Cependant, le sous-ensemble des actions internes réalisables reste inchangé au moins jusqu'à la prochaine action.

Parmi toutes les actions qui s'offrent à un processus aucune n'est privilégiée. De plus il n'y a pas de moyen de savoir si et quand l'une d'entre elles aura lieu. La seule contrainte que nous mettons est qu'une de ces actions aura effectivement lieu dans un délai fini mais non borné si dans le cas où cela ne se produirait pas l'ensemble des actions possibles resterait non-vidé indéfiniment. Autrement dit, si à chaque

instant il y a au moins une action qui peut avoir lieu avec la participation du processus alors le processus participera finalement à une action. Remarquons que d'après cette hypothèse rien n'empêche qu'une action donnée reste possible indéfiniment sans jamais avoir lieu, si, par exemple, d'autres actions sont aussi possibles tout le temps, ou simplement de manière intermittente.

Sans une hypothèse du type de celle que nous venons de poser rien n'obligerait à ce que des actions aient lieu dans le système même dans le cas où il y a des actions possibles. En vérité l'hypothèse choisie est un peu arbitraire, si l'on ne considère que le problème d'assurer l'effective évolution du système. En effet, l'évolution du système serait assurée en supposant simplement que tant qu'il y aura une action possible dans le système, une action aura lieu dans un délai fini mais non borné. Toutefois, cette hypothèse n'est pas acceptable car elle met en cause l'idée sous-jacente fondamentale que l'évolution de chaque processus dépend seulement des actions qui lui sont offertes localement et non pas de ce qui se passe globalement dans le système.

Nous pouvons illustrer les idées que nous sommes en train d'exposer avec une analogie anthropomorphique et sociale des processus et des systèmes de processus. Les processus sont des individus qui coexistent dans une société de processus. La société n'a pas de chef, (il n'y a pas de processus privilégié), et il n'y a pas de normes de conduite imposées extérieurement, (il n'y a pas de règles d'implantation implicites). Le comportement de chaque processus lui est dicté par son "âme" (son programme). On ne peut influencer sur le comportement d'un processus que de manière indirecte, en manipulant des informations qu'il utilisera plus tard.

Les processus sont indépendants les uns des autres mais ils peuvent se regrouper à plusieurs pour réaliser des actions en commun. Il n'y a pas de protocole pour permettre aux processus de se mettre d'accord pour entreprendre une action, car l'établissement de cet accord est lui-même une activité partagée par les processus, requérant un accord préalable

et ainsi de suite. Il est donc impossible d'expliquer comment les processus décident de réaliser une action ensemble. On se limite à constater qu'ils le font. D'ailleurs cette situation n'est pas différente au niveau individuel. Il est aussi impossible de décrire le mécanisme qui mène un processus à décider d'entreprendre une action interne.

Sans maître, sans loi, sans ententes, la société des processus est une société anarchique ! Mais anarchie ne veut pas dire chaos et si les processus ont été bien conçus par leur créateur alors ils coopéreront harmonieusement, de manière effective, dans la poursuite d'un objectif commun.

Tout au long de cette explication intuitive, nous n'avons pas employé le mot "concurrence". Dans notre contexte, la concurrence n'est que le mode de fonctionnement naturel d'un système de processus. En réalité, on peut dire que la concurrence est l'anarchie dans la société de processus. L'adjectif "concurrents" appliqué à un système de processus est donc parfaitement redondant.

#### 4. SEMANTIQUE A PRIORI

Dans la définition formelle de la sémantique d'un système de processus on peut considérer deux étapes. La première concerne la signification individuelle de chaque processus et la seconde le regroupement de ces significations individuelles pour obtenir une description du comportement du système. En fait, quand on prend un processus tout seul, on ne peut pas prévoir quel sera effectivement son comportement individuel. Celui-ci dépend de manière déterminante du système dans lequel s'insère le processus. Tout ce que l'on peut faire au niveau individuel est de définir l'éventail des possibilités d'évolution, sans se soucier du fait qu'elles seront ou non utilisées. Cet aspect de la sémantique est appelé la sémantique a priori du processus.

Comme Francez et al [FHLR79] nous exigeons de la sémantique a priori qu'elle dénote toutes les possibilités de communication du processus, indépendamment de l'environnement où il se trouvera. Ceci signifie que pour un état donné du processus nous devons pouvoir reconnaître toutes les communications possibles à partir de cet état et aussi toutes les évolutions qui empêchent que des communications aient lieu. Remarquons que les communications possibles à partir d'un état donné ne sont pas seulement celles auxquelles le processus peut participer dans cet état. Il faut aussi considérer les communications qui peuvent devenir possibles après une suite quelconque d'actions internes réalisées par le processus à partir de l'état donné. D'autre part, les évolutions internes permettant d'éviter les communications sont les évolutions internes conduisant à la terminaison naturelle du processus et celles qui peuvent continuer indéfiniment.

Nous nous proposons de décrire ces évolutions à l'aide de relations binaires, d'une manière analogue à ce qui a été fait à propos de la commande do - od (cf. chapitre V). Soit P le processus considéré dans le paragraphe VIII.2. Son espace d'états est noté Q et  $\omega \in Q$ . On peut espérer que les évolutions qui mènent à la terminaison soient représentées par des doublets (a,b),  $a \in Q$ ,  $b \in Q$  et  $b \neq \omega$ , et les évolutions infinies par des doublets (a, $\omega$ ),  $a \in Q$ . Pour décrire les évolutions internes auxquelles une communication peut succéder, nous introduisons la notion d'état de communication. L'ensemble C des états de communication du processus P est défini par :

$$C = \{ \langle \alpha, u, v \rangle \mid \alpha \in \text{PORTES}(P) \wedge u \in U_{\alpha} \wedge v \in U_{\alpha}^{-} \rightarrow Q \}$$

D'une certaine manière, un état c,  $c \in C$ ,  $c = \langle \alpha, u, v \rangle$  "anticipe" dans P une communication par la porte  $\alpha$ , u étant la valeur envoyée et v une fonction qui calcule le nouvel état à partir de la valeur reçue. Ainsi, une évolution interne pouvant mener à une communication est représentée par un doublet (a,c),  $a \in A$ ,  $c \in C$ .

Dans ces conditions la sémantique a priori de P est donnée par une relation binaire de  $Q \times (QC)$ , plus l'état initial de P.

#### 4.1 Relation interne

La relation interne, RI, du processus P décrit l'effet d'un pas élémentaire de l'évolution interne de P. Elle correspond à la relation GG considérée à propos de la sémantique de la commande do - od. Nous la définissons par :

$$RI = \bigcup_{i=1}^n \text{gamma} \{ \{ gi \rightarrow ri \} \}$$

#### 4.2 Relation de terminaison et relation de non-terminaison

Considérons, pour simplifier l'écriture, les trois prédicats suivants :

$$\text{cont} = \bigcup_{i=1}^n gi$$

$$\text{comm} = \bigcup_{i=1}^m k_{\alpha i}$$

$$\text{exit} = \neg \text{cont} \wedge \neg \text{comm}$$

Les évolutions internes conduisant le processus à sa terminaison sont celles qui passent (et par conséquent se terminent) dans un état du prédicat exit. Nous les présentons par la relation de terminaison, RT, du processus P :

$$RT = RI^* \circ I_{\text{exit}}$$

De même, les points initiaux des évolutions internes infinies de P sont les états de la plus grande trajectoire de RI. Ces évolutions sont représentées par la relation de non terminaison RNT définie par

$$RNT = \{ (a, \omega) \mid \tilde{wpr}[RI]^x \text{ (cont) } (a) \}$$

#### 4.3 La relation de communication

La relation de communication, RC, du processus P exprime toutes ses possibilités de communication immédiate, i.e, les communications

auxquelles P peut participer sans avoir à entreprendre des actions internes préalables. Pour la définir nous commençons par remarquer que, comme les commandes gardées, les communications gardées sont des entités ayant une signification : les possibilités de communication immédiate qu'elle met à la disposition du processus. Cette signification peut être obtenue à l'aide d'une fonction sémantique kappa qui à chaque communication gardée associe une relation binaire dans  $Q \times (QUC)$  selon la définition suivante :

$$\text{kappa } \llbracket k_{\alpha} : \alpha \rightarrow o_{\alpha}, j_{\alpha} \rrbracket = \{(a, c) \mid k_{\alpha}(a) \wedge c = \langle \alpha, o_{\alpha}(a), \lambda s. j_{\alpha}(s, a) \rangle\}$$

La relation de communication peut maintenant être simplement définie par :

$$RC = \bigcup_{i=1}^m \text{kappa } \llbracket k_{\alpha_i} : \alpha_i \rightarrow o_{\alpha_i}, j_{\alpha_i} \rrbracket$$

#### 4.4 Relation des communications possibles

La relation des communications possibles, RCP, de P enregistre toutes les communications possibles à partir de chaque état :

$$RCP = RI^* \circ RC$$

#### 4.5 Formulation de la sémantique a priori

Nous pouvons définir la relation de sémantique à priori, RSAP, du processus P comme étant l'union de ses trois relations RT, RNT et RCP :

$$RSAP = RT \cup RNT \cup RCP$$

Ainsi, rappelant que X est l'état initial du processus, la sémantique à priori du processus peut être définie par le doublet :

$$\langle X, RSAP \rangle$$

On peut constater que les relations RT, RNT et RCP, et par conséquent RSAP, sont complètement déterminées par RI et RC. Il suffit de voir que

$cont = \lambda a. \{RI(a) \neq \phi\}$

et

$comm = \lambda a. \{RC(a) \neq \phi\}$ .

Dans ces conditions la donnée de  $\langle X, RI, RC \rangle$  peut remplacer la donnée de  $\langle X, RSAP \rangle$  en tant que caractérisation sémantique de P. Nous exploiterons cette possibilité dans le paragraphe suivant.

La présentation de la sémantique formelle que nous venons de faire confirme le fait, implicite lors de la discussion de la sémantique intuitive, que nos systèmes de processus n'ont pas la convention de terminaison distribuée des processus du langage CSP, Hoare [Hoa78b]. Selon cette convention, un processus "en attente" de communication se termine dès que tous les partenaires possibles pour les communications en attente se seront terminés. Dans notre modèle, par contre, un processus ne se termine qu'au moment où il atteint un état de son prédicat exit, ce qui ne peut pas se produire sans la collaboration explicite du processus lui-même.

## 4.6 Exemples

### 4.6.1. Exemple 1

Considérons le processus P1 suivant, où b est une variable booléenne.

```
P1:: b:=true ;
   do b:α → ! l,
           ? s: skip
   [] b → b:= false
   od
```

Pour alléger la présentation des éléments sémantiques importants nous omettons dans les exemples le traitement de l'état  $\omega$ , qui dans ce cas n'est pas très significatif.



Nous avons pour P1 :

- i.  $Q1 = \{\text{vrai}, \text{faux}\}$
- ii.  $\text{PORTES}(P1) = \{\alpha\}$  ;  $U_{\alpha} = \mathbb{N}$  (par exemple)
- iii. l'état initial est vrai
- iv.  $RI = \{(\text{vrai}, \text{faux})\}$
- v.  $RC = \{(\text{vrai}, \langle \alpha, 1, \lambda s. \text{vrai} \rangle)\}$
- vi.  $\text{exit} = \lambda b. \neg b$
- vii.  $RI^* = \{(\text{faux}, \text{faux}), (\text{vrai}, \text{vrai}), (\text{vrai}, \text{faux})\}$
- viii.  $RT = \{(\text{faux}, \text{faux}), (\text{vrai}, \text{faux})\}$
- ix.  $RNT = \emptyset$
- x.  $RCP = \{(\text{vrai}, \langle \alpha, 1, \lambda s. \text{vrai} \rangle)\}$

#### 4.6.2 Exemple 2

Soit le processus P2, où x est une variable entière :

```
P2:: {X ≥ 0}
    x:=X;
    do x>0 :  $\bar{\alpha} \rightarrow ! x,$ 
                ? s:x:=x-s
    od
```

Les éléments sémantiques de P2 sont :

- i.  $Q2 = \mathbb{Z}$
- ii.  $\text{PORTES}(P2) = \{\bar{\alpha}\}$  ;  $U_{\bar{\alpha}} = \mathbb{Z}$
- iii. l'état initial est X
- iv.  $RI = \emptyset$
- v.  $RC = \{(x, \langle \alpha, x, \lambda s. x-s \rangle) \mid x > 0\}$

- vi.         $\text{exit} = \lambda x. x \leq 0$
- vii.       $\text{RI}^* = \{(x, x)\}$
- viii.      $\text{RT} = \{(x, x) \mid x \leq 0\}$
- ix.         $\text{RNT} = \phi$
- x.          $\text{RCP} = \text{RC}$

## 5. SEMANTIQUE DE RESEAUX DE PROCESSUS

### 5.1. Sémantique

Ayant défini la sémantique d'un processus pris isolément, nous voulons maintenant donner la sémantique d'un système de processus. En le faisant, nous obtenons un modèle du fonctionnement concurrent (i.e., anarchique) du système. Il faudra alors vérifier si le modèle traduit bien l'intuition exposée dans le paragraphe VIII.3.

On peut considérer deux approches pour donner une sémantique à un système. La première consiste à définir une opération permettant d'obtenir d'un seul coup le comportement global du système, à partir des sémantiques a priori de chaque composant. C'est la notion de "binding function" trouvée chez Francez et al [FHLR79]. La seconde consiste plutôt à prendre une opération de composition binaire de processus avec laquelle, par applications successives, on aboutit à composer tous les processus du système et à déterminer ainsi son comportement. C'est l'approche de Milner [MiM79] [Mil80], de Francez, Lehmann et Pnueli [FLP80], et aussi la nôtre.

De cette opération binaire nous exigeons trois caractéristiques. D'abord que son résultat soit un processus pouvant remplacer de manière "transparente" le sous-système formé par les deux opérands. Cette exigence exprime la signification intuitive de l'opération. Deuxièmement, il faut qu'elle soit commutative et associative, autrement le comportement global dépendrait de l'ordre d'application de l'opération, ce qui est incorrect. Enfin nous voulons que le résultat de la composition garde toutes les informations sur les évolutions internes de chaque

opérande et des communications entre elles. Cette condition est importante si on veut pouvoir prouver des choses sur le fonctionnement du système.

Notons  $\parallel$  l'opération de composition binaire de processus que nous voulons définir. Soient  $P_1$  et  $P_2$  deux processus avec espaces d'états  $Q_1$  et  $Q_2$  respectivement. Si nous voulons que  $P_1 \parallel P_2$  soit un processus, le premier problème à résoudre est de lui trouver un espace d'états. Nous choisissons, naturellement, le produit cartésien des espaces d'états de  $P_1$  et  $P_2$ , i.e.,  $Q_1 \times Q_2$ <sup>(1)</sup>. En réalité, nous aurions également pu choisir un autre ensemble isomorphe à  $Q_1 \times Q_2$ , par exemple  $Q_2 \times Q_1$ .

Ayant fait le choix d'un espace d'états pour le processus  $P_1 \parallel P_2$  nous devons maintenant indiquer son état initial et ses relations internes et de communication pour que sa sémantique soit précisée, et que l'opération  $\parallel$  soit définie.

Posons  $P = P_1 \parallel P_2$  et  $Q = Q_1 \times Q_2$ . Dans la suite nous indexons les éléments sémantiques par le nom du processus concerné.

La définition de l'état initial de  $P$  ne pose pas de problèmes :

$$X_P = \langle X_{P_1}, X_{P_2} \rangle$$

Regardons maintenant les relations.

Tout d'abord introduisons la notation  $CPORTES(P_1, P_2)$  pour désigner l'ensemble des portes de communication entre  $P_1$  et  $P_2$ , i.e.

$$CPORTES(P_1, P_2) = \{ \alpha \mid \alpha \in PORTES(P_1) \wedge \bar{\alpha} \in PORTES(P_2) \}$$

Pour que  $P$  puisse remplacer  $P_1$  et  $P_2$  il faut qu'il garde toutes les possibilités de communication de  $P_1$  et  $P_2$ , sauf celles qui, étant utilisées pour communiquer entre  $P_1$  et  $P_2$ , ne concernent plus le reste du système. Nous avons donc, pour  $P$  :

---

(1)

Chaque paire de la forme  $(\omega, q_2)$  ou  $(q_1, \omega)$  doit être assimilé à l'élément  $\omega$  de  $Q_1 \times Q_2$ .

$$\text{PORTES}(P) = (\text{PORTES}(P1) \cup \text{PORTES}(P2)) - \text{CPORTES}(P1, P2)$$

et

$$\text{RC}_P = \{(a, c) \mid (a, c) \in \text{RC}_{P1}^0 \cup \text{RC}_{P2}^1 \wedge c+1 \notin \text{CPORTES}(P1, P2)\} \quad (1)$$

où  $\text{RC}_{P1}^0$  et  $\text{RC}_{P2}^1$  sont les "images" de  $\text{RC}_{P1}$  et  $\text{RC}_{P2}$  dans  $Q \times C$  (2), i.e.

$$\begin{aligned} \text{RC}_{P1}^0 = \{ \langle \langle a1, a2 \rangle, \langle \alpha, u, \lambda z. \langle v(z), a2 \rangle \rangle \rangle \mid \\ \langle a1, \langle \alpha, u, v \rangle \rangle \in \text{RC}_{P1}, a2 \in Q2 \} \end{aligned}$$

et

$$\begin{aligned} \text{RC}_{P2}^1 = \{ \langle \langle a1, a2 \rangle, \langle \alpha, u, \lambda z. \langle a1, v(z) \rangle \rangle \rangle \mid \\ \langle a2, \langle \alpha, u, v \rangle \rangle \in \text{RC}_{P2}, a1 \in Q1 \} \end{aligned}$$

Un pas élémentaire dans l'évolution interne de P correspond soit à un pas élémentaire dans P1, soit un pas élémentaire dans P2, soit encore à une communication entre P1 et P2. Par conséquent :

$$\begin{aligned} \text{RI}_P = \text{RI}_{P1}^0 \cup \text{RI}_{P2}^0 \cup \\ \{ \langle \langle a1, a2 \rangle, \langle b1, b2 \rangle \rangle \mid \langle a1, \langle \alpha, u1, v1 \rangle \rangle \in \text{RC}_{P1} \wedge \\ \langle a2, \langle \bar{\alpha}, u2, v2 \rangle \rangle \in \text{RC}_{P2} \wedge \\ b1 = v1(u2) \wedge b2 = v2(u1) \} \end{aligned}$$

où

$$\begin{aligned} \text{RI}_{P1}^0 = \{ \langle \langle a1, a2 \rangle, \langle b1, a2 \rangle \rangle \mid \langle a1, b1 \rangle \in \text{RI}_{P1}, a2 \in Q2 \} \\ \text{RI}_{P2}^0 = \{ \langle \langle a1, a2 \rangle, \langle a1, b2 \rangle \rangle \mid \langle a2, b2 \rangle \in \text{RI}_{P2}, a1 \in Q1 \} \end{aligned}$$

Nous terminons ainsi la définition de l'opération  $\parallel$ . Nous pourrions maintenant démontrer qu'elle est commutative et associative, à un isomorphisme près. Les démonstrations sont directes mais un peu encombrantes, surtout celles de l'associativité. Plus tard, nous rencontrerons un argument simple qui peut remplacer les démonstrations formelles.

(1) La notation  $c+1$  désigne le premier élément du triplet  $c$

(2)  $C$  est l'ensemble d'états de communication de P, implicitement défini à partir de  $Q$  et  $\text{PORTES}(P)$ .

## 5.2 Processus fortement communicants

Dans la définition que nous venons de proposer, l'exécution concurrente des deux processus  $P_1$  et  $P_2$  est expliquée par le non-déterminisme dans le choix de l'action suivante dans le processus composé soit une action interne de  $P_1$  soit une action interne de  $P_2$  soit une communication entre les deux. Si, d'une part cette technique se justifie, les actions étant instantanées et indépendantes les unes des autres, on sait bien qu'elle introduit le problème du "fair merge". Dans le contexte de l'opération  $||$  ce problème traduit le fait que en général on ne peut pas exclure que l'évolution du processus composé consiste entièrement d'une séquence infinie d'actions internes d'un des processus seulement, même si l'autre est tout le temps prêt lui aussi à réaliser une action interne ou à communiquer. Cette possibilité est incompatible avec la description intuitive de comportement des processus faite dans le paragraphe VIII.3. Face à cette constatation, nous devons conclure que le modèle relationnel obtenu avec l'opération  $||$  pour le système de processus, en général n'est pas bon.

Cependant, cette situation désagréable ne pourra pas se produire si les processus composants  $P_1$  et  $P_2$  n'admettent pas d'évolutions internes infinies. Dans ces conditions, une évolution infinie de  $P_1 || P_2$  doit inclure un nombre infini de communications entre  $P_1$  et  $P_2$ . Elle est donc, forcément, "mélangée équitablement".

Un processus qui n'admet pas d'évolutions internes est dit "fortement communicant". Un processus fortement communicant ne peut évoluer indéfiniment sans communiquer avec son environnement. La sémantique a priori permet une caractérisation formelle facile de ce type de processus ce sont ceux pour lesquels on a  $RNT = \{(\omega, \omega)\}$ . (Dans la pratique on dit simplement que la relation RNT est vide).

Un système de processus étant composé d'un nombre arbitraire de processus, pour que le modèle obtenu avec l'opération  $||$  soit bon, il

faut que tout sous-ensemble propre du système soit fortement communicant. La vérification directe de cette propriété peut être très difficile, voire impraticable. Néanmoins, remarquons que si le résultat de la composition de deux processus est fortement communicant, ces deux processus le sont aussi. En particulier, si nous réussissons à montrer que le résultat de la composition de tous les processus d'un système est fortement communicant, (pour un processus qui ne peut pas participer à des communications ceci signifie qu'il se termine toujours, indépendamment de son état initial), alors nous aurons montré en même temps que tous les processus du système et toutes leurs compositions sont fortement communicants. Ainsi, nous aurons garanti a posteriori que le modèle relationnel est applicable. D'ailleurs, dans la pratique, nous pouvons même affaiblir l'exigence de communication forte. Si le système total se termine toujours pour l'état initial donné alors il exhibe un comportement fini, par conséquent forcément "mêlée avec équité", ce qui signifie que dans ce cas encore le modèle est bon.

En conclusion, la sémantique de l'opération  $||$  ne reflète pas toujours le comportement réel d'un système de processus. Dans la plupart des cas, ceci n'est pas trop grave car elle nous en donne quand même une bonne approximation. En effet le modèle relationnel obtenu enregistre encore toutes les évolutions possibles mais peut-être aussi quelques évolutions "mal mêlées" qui n'auront jamais lieu. Autrement dit, si l'ensemble des séquences décrivant le comportement du système est  $B$ , la sémantique relationnelle nous fournit un ensemble  $\hat{B}$  tel que  $B \subseteq \hat{B}$ . Il est évident que toute propriété valable pour tous les éléments de  $B$  est aussi valable pour tous les éléments de  $\hat{B}$ .

### 5.3 Exemple

Rappelons les processus  $P_1$  et  $P_2$  dont la sémantique a priori a été décrite dans la sous-section VIII.4.6. Il s'agit de deux processus

fortement communicants. Posons  $P = P1 || P2$  et calculons les éléments sémantiques de  $P$ :

- i.  $Q = \{\text{vrai}, \text{faux}\} \times Z$
- ii.  $\text{PORTES}(P) = \phi$
- iii. l'état initial est  $\langle \text{vrai}, X \rangle$
- iv.  $RI = \{(\langle \text{vrai}, x \rangle, \langle \text{faux}, x \rangle)\} \cup \{(\langle \text{vrai}, x \rangle, \langle \text{vrai}, x-1 \rangle) \mid x > 0\}$
- v.  $RC = \phi$
- vi.  $\text{exit} = \lambda \langle b, x \rangle. \neg b$
- vii.  $RI^* = I \cup \{(\langle \text{vrai}, x \rangle, \langle \text{faux}, x \rangle)\} \\ \cup \{(\langle \text{vrai}, x \rangle, \langle \text{vrai}, x-k \rangle) \mid x \geq k > 0\} \\ \cup \{(\langle \text{vrai}, x \rangle, \langle \text{faux}, x-k \rangle) \mid x \geq k > 0\}$
- viii.  $RT = \{(\langle \text{faux}, x \rangle, \langle \text{faux}, x \rangle)\} \cup \{(\langle \text{vrai}, x \rangle, \langle \text{faux}, x-k \rangle) \mid x \geq k \geq 0\}$
- ix.  $RNT = \phi$
- x.  $RCP = \phi$

Le processus  $P$  est un processus fermé, qui n'a pas de communications avec son environnement. Son évolution n'est composée que d'actions internes. Le fait que  $RNT = \phi$  montre que  $P$  se termine toujours. Prenant en compte l'état initial, la relation  $RT$  indique que l'ensemble d'états finals atteignables est :

$$RT(\langle \text{vrai}, X \rangle) = \{\langle \text{faux}, X-k \rangle \mid X \geq k \geq 0\}$$

Le seul élément de cet ensemble qui vérifie simultanément  $\text{exit}_{P1}$  et  $\text{exit}_{P2}$  est  $\langle \text{faux}, 0 \rangle$ . Une évolution de  $P$  conduisant à cet état correspond à un comportement du système pour lequel les deux processus se terminent normalement. Par contre, si  $P$  se termine dans un état  $\langle \text{faux}, x \rangle$  tel que  $x > 0$ , ceci signifie que  $P1$  s'est bien terminé mais il n'en est pas de même pour  $P2$ . Il s'agit ici d'une situation de blocage pour  $P2$  qui attendra indéfiniment une communication avec  $P1$ . Une telle communication est impossible,  $P1$  s'étant déjà terminé.

Si on regarde de près la relation RI on s'aperçoit qu'il est très facile d'écrire une commande répétitive dont la relation GG est précisément RI. En ajoutant de manière triviale l'état initial on obtient un programme séquentiel qui représente le processus P :

```
{X≥0}
b, x:= true, X;
do b → b:= false
  [] b and x>0 → x:= x-1
od
```

## 6. RECONSTRUCTION D'UN PROCESSUS COMPOSE

Le programme présenté à la fin de la section précédente suggère qu'il serait agréable d'avoir une méthode directe pour obtenir un programme séquentiel qui représente le comportement d'un système de processus donné. Ceci peut être vu comme un cas particulier du problème consistant à trouver une écriture pour le processus composé à partir de deux processus dans le formalisme utilisé pour écrire les composants. Le formalisme proposé dans ce rapport et la sémantique associée fournissent une solution simple pour ce problème. Elle consiste essentiellement à prendre les règles qui permettent d'obtenir les éléments sémantiques d'un processus, et à les appliquer en sens inverse pour obtenir le texte d'un processus à partir de sa sémantique. Dans le cas qui nous intéresse cette sémantique est la sémantique de la composition des deux processus originaux.

En pratique il n'est pas nécessaire de passer explicitement par la sémantique. Au vu des rapports étroits entre syntaxe et sémantique, nous pouvons définir directement des règles de composition de processus au niveau syntaxique, compatibles avec la sémantique de la composition. Ainsi, pour deux processus P1 et P2 dont les variables sont respectivement x1 et x2, nous pouvons écrire un processus qui représente P1 || P2 utilisant les règles suivantes.



Règles syntaxiques de composition de processus

1. (Etat initial). La commande d'initialisation de  $P1 || P2$  est :

$$x1, x2 := X_{P1}, X_{P2}$$

2. (Relation de communication). Les communications gardées de  $P1 || P2$  sont celles de  $P1$  plus celles de  $P2$  moins celles qui servent à communiquer entre  $P1$  et  $P2$ .

3. (Relation interne). Les commandes gardées de  $P1 || P2$  sont celles de  $P1$ , plus celles de  $P2$ , plus, pour chaque paire de portes  $\{\alpha, \bar{\alpha}\}$  appartenant à  $CPORTES(P1, P2)$  une commande gardée de la forme :

$$k_{\alpha}(x1) \text{ and } k_{\bar{\alpha}}(x2) \rightarrow x1, x2 := j_{\alpha}(o_{\bar{\alpha}}(x2), x1), j_{\bar{\alpha}}(o_{\alpha}(x1), x2)$$

## 7. "PARTITIONNEMENT D'UN ENSEMBLE"

Le premier des deux exemples que nous présentons pour illustrer les propositions de ce chapitre est le problème du "partitionnement d'un ensemble", pour lequel des algorithmes et des preuves existent déjà, Dijkstra [Dij77], Apt, Francez et de Roever [AFR80], Cousot et Cousot [CoC80a]. Ce problème nous donne la possibilité d'utiliser de manière intéressante la communication bidirectionnelle.

L'énoncé du problème est le suivant : étant donné deux ensembles d'entiers  $S_0$  et  $T_0$ , finis et disjoints (donc non-vides) on veut obtenir deux ensembles  $S$  et  $T$  tels que :

- i.  $S \cup T = S_0 \cup T_0$
- ii.  $|S| = |S_0|$  et  $|T| = |T_0|$ .
- iii. tout élément de  $S$  est plus petit que chaque élément de  $T$ , i.e.,  $\max S \leq \min T$ .

Le programme parallèle formé par les processus  $PS$  et  $PT$  présentés ci-dessous résout le problème. Intuitivement le fonctionnement du

programme est le suivant : PS et PT représentent respectivement les ensembles S et T ; PS échange sa plus grande valeur contre la plus petite valeur de PT, jusqu'à ce que la valeur reçue soit inférieure à la valeur par laquelle elle a été échangée, et symétriquement pour PT ; à ce point là, un échange supplémentaire doit être effectué pour rétablir la bonne condition de terminaison, après quoi les deux processus terminent.

Note concernant la syntaxe. L'écriture usuelle des affectations simultanées est difficile à lire quand il y a beaucoup de variables affectées. Ainsi, parfois nous remplaçons la syntaxe  $x,y:= a,b$  par  $x:=a, y:= b$ . Ne pas confondre avec  $x:= a ; y:= b$  dont la signification est en général différente.  $\square$

```

PS :: S:= S0,
      endS, si, so := false, 0, 1 ;
      do si < so :
           $\alpha \rightarrow ! \max S,$ 
          ? s : S:=S-{max S}  $\cup$  {s}, si:=s, so:=max S
       $\square$  non endS and si  $\geq$  so :
           $\beta \rightarrow ! si,$ 
          ? s : S:=S-{si}  $\cup$  {s}, endS:=true
      od.

PT :: T:= T0,
      endT, ti, to := false, 1, 0 ;
      do ti > to :
           $\bar{\alpha} \rightarrow ! \min T,$ 
          ? t : T:=T-{min T}  $\cup$  {t}, ti:=t, to:=min T
       $\square$  non endT and ti  $\leq$  to :
           $\bar{\beta} \rightarrow ! ti,$ 
          ? t : T:=T-{ti}  $\cup$  {t}, endT:=true
      od.

```

On peut remarquer que les deux processus sont parfaitement symétriques, ce qui contraste avec les solutions à communication unidirectionnelle où il y a un processus "maître" et un processus "esclave".

Les processus PS et PT sont fortement communicants. Par conséquent, l'exécution concurrente de PS et PT est décrite par le programme suivant, obtenu selon les règles de la section précédente.

(Pr1)

```

S, T:=S0,T0,
endS, si, so:=false, 0, 1,
endT, ti, to:=false, 1, 0 ;
do si < so and ti > to
    → S:=S-{max S} ∪ {min T}, si:=min T, so:=max S,
    T:=T-{min T} ∪ {max S}, ti:=max S, to:=min T
[] non endS and si ≥ so and non endT and ti ≤ to
    → S:=S-{si} ∪ {ti}, endS:=true,
    T:=T-{ti} ∪ {si} endT:=true
od.
```

Pour prouver la correction du programme parallèle il suffit de prouver la correction du programme Pr1 ci-dessus. Il est trivial de montrer que  $S \cup T = S_0 \cup T_0$ ,  $|S| = |S_0|$ ,  $|T| = |T_0|$ ,  $S \cap T = \emptyset$  sont des invariants de Pr1. Ainsi il nous reste à montrer que le programme se termine et que lors de la terminaison nous avons bien  $\max S \leq \min T$ .

Avant de nous engager dans la preuve, nous transformons informellement Pr1, pour le simplifier. Les variables endS et endT prennent toujours la même valeur. Ceci signifie qu'elles représentent localement la même information globale. La même remarque s'applique à si et to à ti et so.

Cette propriété nous permet de remplacer chaque paire de variables par une nouvelle variable représentant l'entité globale correspondante. Remplaçons alors endS et endT par end, si et to par ss et ti et so par tt. Nous obtenons

(Pr2)

```

S,T:=S0,T0,
end, ss, tt:=false, 0, 1 ;
do ss < tt
  → S:=S-{max S} ∪ {min T}, T:=T-{min T} ∪ {max S},
    ss:=min T, tt:=max S
[] non end and ss ≥ tt
  → S:=S-{ss} ∪ {tt}, T:=T-{tt} ∪ {ss},
    end:=true
od

```

Dans ce programme la deuxième commande gardée n'est exécutée qu'une seule fois, juste avant la terminaison du programme. Ceci indique que Pr2 est équivalent à

(Pr3)

```

S,T:=S0,T0,
ss, tt:=0, 1 ;
do ss < tt
  → S:=S-{max S} ∪ {min T}, T:=T-{min T} ∪ {max S},
    ss:=min T, tt:=max S
od ;
S:=S-{ss} ∪ {tt}, T:=T-{tt} ∪ {ss}.

```

Pour établir la post-condition  $\max S \leq \min T$  il faut garantir que le prédicat

$$(*) \text{ wpr}[S:=S-\{ss\} \cup \{tt\}, T:=T-\{tt\} \cup \{ss\}] (\max S \leq \min T) = \\ \max(S-\{ss\} \cup \{tt\}) \leq \min(T-\{tt\} \cup \{ss\})$$

est vérifié quand la commande répétitive se termine. Par conséquent, nous devons chercher un invariant dont l'intersection avec  $\neg(ss < tt)$  implique (\*). Un tel invariant est le prédicat J défini par :

$$J = \max(S-\{ss\} \cup \{tt\}) \leq \min(T-\{tt\} \cup \{ss\}) \vee ss < tt.$$

La formule VI.2.2.5 confirme que J est un invariant :

$$J \subseteq \underline{ss} \geq \underline{tt}$$

$$\vee \max(S-\{\max S\} \cup \{\min T\}-\{\min T\} \cup \{\max S\}) \leq \\ \min(T-\{\min T\} \cup \{\max S\}-\{\max S\} \cup \{\min T\})$$

$$\vee \min T < \max S$$

ou

$$J \subseteq \underline{ss} \geq \underline{tt} \vee \max S \leq \min T \vee \min T < \max S$$

La partie droite étant toujours vraie l'inclusion est vérifiée, trivialement.

Nous devons encore prouver que la boucle se termine. L'utilisation directe de la technique de la fonction décroissante ne donne pas de bons résultats ici, à cause du dernier échange de valeurs entre les processus qui détruit le comportement monotone de la boucle. Observons, cependant, que la terminaison d'une commande répétitive

$$\underline{\text{do}} \ g \rightarrow r \ \underline{\text{od}}$$

est impliquée par celle de

$$\underline{\text{do}} \ \neg \text{wpr} \ [r] (\neg g) \rightarrow r \ \underline{\text{od}}.$$

Appliquant la transformation suggérée à la boucle de Pr3 nous obtenons

```

do min T < max S
    → S:=S-{max S} ∪ {min T}, T:=T-{min T} ∪ {max S},
    ss:=min T, tt:=max S
od

```

Les variables ss et tt ne sont plus nécessaires. En les éliminant nous arrivons à :

```

do min T < max S
    → S:=S-{max S} ∪ {min T}, T:=T-{min T} ∪ {max S}
od

```

Dans cette boucle simple l'invariant  $S \cap T = \emptyset$  implique que la fonction  $\max S - \min T$  est une fonction décroissante, garantissant par conséquent la terminaison de la boucle et du programme. La preuve est ainsi complète.

## 8. LES CINQ PHILOSOPHES

Notre deuxième exemple est plus substantiel. Le classique problème des cinq philosophes (dû à Dijkstra) est énoncé par Hoare dans [Hoa78] comme suit :

"Five philosophers spend their lives thinking and eating. The philosophers share a common dining room where there is a circular table surrounded by five chairs, each belonging to one philosopher. In the center there is a large bowl of spaghetti, and the table is laid with five forks. On feeling hungry a philosopher enters the dining room, sits in his own chair, and picks up the fork on the left of his place. Unfortunately, the spaghetti is so tangled that he needs to pick up and the fork on his right as well. When he has finished, he puts down both forks, and leaves the room. The room should keep a count of the number

of philosophers in it". (1)

Si l'on essaye de décrire le comportement des philosophes en gardant strictement le formalisme présenté on doit prévoir un nombre considérable de portes et les processus qu'on obtient sont encombrants et trop détaillés. Néanmoins quelques simplifications sont possibles dans le formalisme, rendant l'écriture de ces processus beaucoup plus naturelle. D'abord nous qualifions les noms et les variables des processus de même "type" par des indices, ainsi que les noms de portes similaires par des noms (ou indices) de processus. Il est aussi pratique de fusionner les portes d'un processus qui ont les mêmes gardes et les mêmes fonctions de sortie et d'entrée. De plus si par une porte donnée la seule information transmise est un signal de synchronisation nous nous permettons d'omettre le texte de la fonction de sortie ainsi que la variable d'entrée de la porte complémentaire. Ces simplifications ne sont que des abréviations n'introduisant pas de problèmes sémantiques. Par conséquent les règles de reconstruction du paragraphe VIII.6 peuvent toujours être appliquées, avec quelques adaptations évidentes.

Le programme que nous présentons est purement une traduction dans notre formalisme de la solution de Hoare [Hoa78]. Il est formé par onze processus : cinq philosophes  $ph(1), \dots, ph(5)$ , cinq fourchettes  $f(1), \dots, f(5)$  et une salle à manger  $r$ . Le processus  $r$  a deux portes,

---

(1)

"Cinq philosophes passent leurs vies à penser et à manger. Les philosophes partagent une salle à manger où il existe une table ronde, entourée par cinq chaises, chaque chaise appartenant à l'un des philosophes. Un grand plat avec des spaghetti est posé sur le centre et la table est mise avec cinq fourchettes. Quand un philosophe a envie de manger il entre dans la salle, s'assoit sur sa chaise et prend la fourchette qui se trouve à gauche de sa place. Malheureusement, les spaghetti sont tellement emmêlés que le philosophe doit prendre aussi la fourchette de droite. Dès qu'il termine, il pose les deux fourchettes, et sort de la salle. La salle doit compter le nombre de philosophes qui sont à l'intérieur".

correspondant chacune à la fusion de cinq portes : e (entrée) et x (sortie). Le processus f(i) a deux portes correspondant chacune à la fusion de deux portes : u-f(i) ("pick up" fourchette i) et d-f(i) ("put down" fourchette i). Finalement, le processus ph(i) a six portes :  $\bar{e}$ ,  $\bar{x}$ ,  $\overline{u-f(i)}$ ,  $\overline{u-f(i+1)}$ ,  $\overline{d-f(i)}$  et  $\overline{d-f(i+1)}$ , (le symbole + désignant ici l'addition modulo 5).

Le comportement du philosophe i est décrit par :

```

ph(i) :: p(i) := 1 ;
      do p(i)=1 :  $\bar{e}$            → p(i):=2
      [] p(i)=2 :  $\overline{u-f(i)}$    → p(i):=3
      [] p(i)=3 :  $\overline{u-f(i+1)}$  → p(i):=4
      [] p(i)=4 :  $\overline{d-f(i)}$    → p(i):=5
      [] p(i)=5 :  $\overline{d-f(i+1)}$  → p(i):=6
      [] p(i)=6 :  $\bar{x}$            → p(i):=1
      od.

```

Pour la fourchette i nous avons :

```

f(i) :: free(i) := true ;
      do free(i) : u-f(i)   → free(i) := false
      [] true    : d-f(i)   → free(i) := true
      od.

```

La salle à manger est simplement :

```

r :: n := 0 ;
      do true    : e           → n := n+1
      [] true    : x           → n := n-1
      od.

```



Il est bien connu que ce programme contient une situation de blocage total : il s'agit du cas où tous les philosophes prennent leurs fourchettes gauches, après quoi aucun ne peut continuer car il n'y a plus de fourchette droite disponible. Cette triste situation est évitée si on n'admet pas plus de quatre philosophes simultanément dans la salle à manger, c'est-à-dire, si on remplace le processus  $r$  par :

```

r4 :: n:=0 ;
      do n < 4 : e → n:=n+1
      [] true  : x → n:=n-1
      od.

```

Même si un argument informel peut être suffisamment convainquant pour justifier l'absence de blocage dans ce cas, nous nous proposons de le prouver formellement.

Tout d'abord nous devons remarquer que même si tous les processus considérés sont fortement communicants, ceci n'est pas vrai pour toutes leurs compositions. Prenons, par exemple  $ph(1) \parallel f(1) \parallel f(2) \parallel r4$ . Nous sommes donc dans la situation, discutée dans le paragraphe VIII.5.2., où la sémantique n'est qu'une approximation. De toute façon, il est évident que l'absence de blocage de  $ph(1) \parallel \dots \parallel f(1) \parallel \dots \parallel r4$  implique l'absence de blocage du système "réel".

En appliquant les règles de composition syntaxique nous obtenons le programme suivant, dont la commande do-od a trente commandes gardées. Nous ne présentons que le texte concernant le philosophe  $i$ .

(Pr1)

 $p(1), \dots, p(5) := 1, \dots, 1,$  $free(1), \dots, free(5) := true, \dots, true,$  $n := 0 ;$ do

...

 $\square p(i)=1 \text{ and } n < 4 \quad \rightarrow p(i):=2, n:=n+1$  $\square p(i)=2 \text{ and } free(i) \quad \rightarrow p(i):=3, free(i):=false$  $\square p(i)=3 \text{ and } free(i+1) \rightarrow p(i):=4, free(i+1):=false$  $\square p(i)=4 \quad \rightarrow p(i):=5, free(i):=true$  $\square p(i)=5 \quad \rightarrow p(i):=6, free(i+1):=true$  $\square p(i)=6 \quad \rightarrow p(i):=1, n:=n-1$ 

...

od.

Pour prouver l'absence de blocage il suffit d'exhiber un invariant sans blocage de la boucle de Pr1, qui soit vérifié à l'initialisation.

Nous avons :

$$\begin{aligned} \text{exit} = & (n \geq 4 \vee \bigwedge_1 p(i) \neq 1 \\ & \wedge \bigwedge_1 (free(i) \vee p(i) \neq 2 \wedge p(i-1) \neq 3) \\ & \wedge \bigwedge_1 (p(i) \neq 4 \wedge p(i) \neq 5 \wedge p(i) \neq 6) \end{aligned}$$

Le prédicat qui décrit l'utilisation de la fourchette  $i$  est :

$$F_i = free(i) \vee p(i)=3 \vee p(i)=4 \vee p(i-1)=4 \vee p(i-1)=5.$$

Ces prédicats  $F_i$  sont en fait des invariants comme le confirmerait par exemple la formule VI.2.2.5.

L'occupation de la salle à manger est décrite par les deux prédicats suivants qui sont aussi des invariants :

$$N = n \leq 4$$

$$R = n - \#\{p(i) \mid p(i) \neq 1\}.$$

Le fait que les prédicats  $F_i$ ,  $N$  et  $R$  soient des invariants était indispensable pour que le système de processus décrive convenablement ce que l'on espérait. On peut même constater que ces trois invariants se trouvent déjà, implicitement, dans l'énoncé du problème.

Des calculs plus ou moins immédiats montrent que

$\bigcap_i F_i \cap N \cap R \subseteq \rightarrow \text{exit}$ . Vu que  $\bigcap_i F_i \cap N \cap R$  est un invariant et qu'il est vérifié à l'état initial, il est l'invariant cherché. La preuve d'absence de blocage est ainsi terminée.

## 9. CONCLUSION

Nous avons introduit un formalisme pour décrire des réseaux simples de processus communicants et nous avons vu dans quelle mesure ce formalisme permet de décrire notre intuition du concept de processus communicant. La sémantique proposée fournit des règles de composition de processus au niveau syntaxique, ce qui est très convenable pour la vérification des propriétés du système. Même dans les cas où la sémantique n'est pas applicable exactement à cause du problème du "mélange équitable", la sémantique approchée reste utile, car il y a des propriétés qui ne sont pas affectées par un "mélange non-équitable".

L'exemple des cinq philosophes suggère quelques simplifications du formalisme utilisé. D'autres simplifications et généralisations sont encore possibles, sans toucher aux aspects fondamentaux. Néanmoins nous constatons que le formalisme offre déjà des possibilités intéressantes de description et d'analyse.

Enfin, nous pouvons signaler que, à notre avis, le concept de processus fortement communicant, introduit dans le paragraphe VIII.5.2., est assez fondamental en Programmation. En fait, si un processus n'a pas cette propriété alors il peut s'isoler indéfiniment de son environnement, en exécutant continuellement ses commandes gardées, sans jamais participer à une communication. L'introduction d'un tel processus dans un système de processus serait ainsi, en principe, une erreur de conception.



CHAPITRE IX

UNE EXPERIENCE DE PROGRAMMATION PARALLELE



## 1. INTRODUCTION

Le formalisme présenté dans le chapitre précédent est loin de constituer un langage de programmation parallèle avec un minimum de réalisme. Outre des structures de contrôle plus élaborées, il lui manque notamment des mécanismes de définition de types et de structuration de processus. Le but principal de ce chapitre est précisément de faire quelques suggestions pour combler ces lacunes.

Le langage que nous esquissons utilise le mécanisme de communication directe décrit précédemment. De ce point de vue, il rejoint les langages CSP de Hoare [Hoa78b] et Ada [Ada79] qui utilisent des variantes du même principe, appelé couramment communication par rendez-vous, et il contraste avec un certain nombre de langages parallèles assez utilisés pour lesquels la communication entre processus est établie par l'intermédiaire de moniteur, Hoare [Hoa74]. C'est le cas des langages Concurrent Pascal de Brinch Hansen [Bri75], Modula de Wirth [Wir77] et Pascal Plus de Welsh et Bustard [WeB79].

Quoique le langage proposé soit facilement implémentable selon une technique que nous décrivons, notre souci principal dans cette étude est de définir les bases d'un langage parallèle réaliste, à partir des réflexions sur la sémantique menées dans les chapitres précédents.

Nous commençons par décrire la structure générale du langage. Ensuite nous prenons l'exemple du problème des télégrammes de Handersen et Snowden [HeS72]. Finalement nous expliquons comment ce langage pourrait être implanté et nous montrons le résultat d'une compilation faite à la main.

Note : Dans la suite le langage que nous allons décrire sera simplement appelé "le Langage". □



## 2. STRUCTURE DU LANGAGE

Nous avons signalé qu'une des lacunes du formalisme du chapitre précédent est l'absence d'un mécanisme pour la définition de types. Dans notre approche, la notion de type s'applique à deux classes d'objets : les variables et les portes. Puisque le type d'une porte doit caractériser les valeurs susceptibles d'être émises par cette porte, un seul concept suffit pour les portes et pour les variables.

Dans les langages existants, on trouve un très grand choix de technique de définition de types. Pour des raisons de simplicité, et également raison de facilité de traduction vers un langage répandu, nous prenons l'approche du langage Pascal de Wirth [JEW75]. Concrétisons : chaque processus aura une partie déclarative à la manière d'un programme Pascal. Nous autorisons quand même une plus grande liberté, dans la mesure où il est permis de mêler les définitions de constantes et de types et les déclarations de fonctions et procédures, (mais non de variables). (Pour des raisons de compilation, nous exigerons quand même que chaque objet soit défini ou déclaré avant d'être utilisé). Cette commodité permet de regrouper des déclarations en rapport les unes avec les autres et ainsi de simuler des types abstraits.

L'autre problème à résoudre est de trouver un mécanisme de structuration des processus. Quand on conçoit un programme parallèle, on ne part pas nécessairement des processus élémentaires. Si on suit une approche descendante, il est naturel de considérer d'abord un premier niveau de processus chacun d'eux pouvant alors se décomposer en d'autres processus et ainsi de suite, jusqu'à ce que le degré de décomposition soit jugé suffisant. On aura construit de cette manière une structure hiérarchique, qui peut être assimilée à un arbre. C'est cette structure que nous voulons pouvoir décrire

Pour le faire, deux attitudes sont possibles : on décrit l'arbre soit par niveaux ("breadth first"), soit en profondeur ("depth first"). A première vue la description par niveaux est plus attractive car elle

"supporte" mieux le "paradigme" de la programmation structurée, au sens de Floyd [Flo79]. Pourtant, cette technique pose aussi des problèmes importants de lisibilité de programmes. En effet, nous voulons que les déclarations faites à un certain noeud de l'arbre soient utilisables ("visibles") dans le sous-arbre associé. Or, avec une description par niveaux, la zone de visibilité d'une déclaration donnée est en général disséminée en plusieurs morceaux dans le texte du programme. Dans ces conditions, à moins que l'on introduise des rappels systématiques, il est difficile de savoir pour un processus donné quelles sont les déclarations en vigueur.

La description en profondeur, par contre, n'a pas cet inconvénient, car le texte correspondant à un sous-arbre suit (en fait il peut être inclus dans) le texte du processus qui se trouve à la racine de ce sous-arbre. Ainsi chaque déclaration a une seule zone de validité, bien délimitée, dans le programme. Pour cette raison, nous choisissons cette alternative.

Nous expliquons la structure syntaxique d'un programme à l'aide de la notation BNF, avec les extensions utilisées dans le Rapport Révisé du Langage Pascal, Wirth [JeW75], en particulier la meta-notation { } pour représenter zéro ou plus occurrences de la séquence de symboles entourée par { et }. (Nous laissons les symboles de la grammaire en anglais pour garder la compatibilité avec [JeW75] qui sert de référence pour les non-terminaux que nous ne définissons pas : <identifieur>, <type identifieur>, <constant definition part>, <type definition part>, <procedure and function declaration part> et <variable declaration part>.

Description BNF de la structure du langage.

- i. <program> ::= <process declaration>.
- ii. <process declaration> ::= <process heading> <general declaration part  
<process body> end
- iii. <process heading> ::= system <process name>; |  
process <process name> (<port section> {;<port section>});
- iv. <process name> ::= <identifieur>



La règle viii. suggère que nous ne considérons que des communications unidirectionnelles. D'autres types de communications pourraient être introduits en modifiant cette règle, et, si nécessaire, la règle v.

Chaque processus peut contenir des déclarations de constante, de type, de processus et de fonction dans n'importe quel ordre (règles x. et xi.). Le fait qu'il n'y a pas de variables déclarées dans la partie de déclaration générale (<general declaration part>) garantit que les processus et les fonctions déclarées dans cette partie sont sans effet de bord. Toutes ces déclarations sont en outre visibles dans tous les processus descendants du processus contenant ces déclarations. (En termes syntaxiques, les descendants d'un processus sont ceux déclarés dans son corps de processus ("process body")).

Le corps d'un processus (règle xii.) non-élémentaire consiste en la déclaration des processus qui le composent et la description de la composition utilisée. Celle-ci est faite en trois parties (règle xiv.). D'abord dans la partie network on définit le schéma de connexion du réseau en suivant l'approche expliquée dans Milner [Mil79]. Pour le moment, la seule opération de composition est notée  $\parallel$  et correspond à l'opération (notée par le même symbole) étudiée dans le chapitre précédent. Nous pourrions en considérer d'autres, par exemple celles de Milner et Milner [MiM79] ou de Jorrand [Jor80]. Dans la partie nodes on affecte à chaque noeud du réseau construit un des processus déclarés. Finalement dans la partie ports on spécifie la liaison entre les portes du processus composé et celles des processus composants, en utilisant le réseau construit précédemment. (Cette partie est absente si le processus composé n'a pas de portes). La syntaxe de la partie spécification du réseau est détaillée par les règles suivantes.

Règles BNF de spécification du réseau.

- xv. <network construction> ::= <nodes declaration> : <network expres
- xvi. <nodes declaration> ::= <complete node> {,<complete node>}
- xvii. <complete node> ::= <node name> (<symbolic port>{,<symbolic port
- xviii. <node name> ::= <identifïer>
- xix. <symbolic port> ::= identifïer>|-<identifïer>
- xx. <network expression> ::= <node name>{||<node name>}
- xxi. <node assignment part> ::= <node assignment>{;<node assignment>}
- xxii. <node assignment> ::= <node name> := <process name>
- xxiii. <port assignment part> ::= <port assignment>{;<port assignement>}
- xxiv. <port assignment> ::= <symbolic port> := <port identifïer>

□

A la lumière du chapitre précédent, la signification de ces règles est plus ou moins évidente. Dans xix. le symbole - remplace la barre supérieure pour désigner la porte complémentaire. L'affectation d'un processus à un noeud spécifie implicitement l'association des portes du processus, aux portes symboliques du noeud. Cette association est faite par position. Pour que la composition soit valide, il faut que les portes mises en liais aient le même type et des directions opposées.

Il nous reste à commenter la partie <process body> pour un processus élémentaire. Le fait que des variables ne puissent être déclarées qu'au niveau des processus élémentaires, exprime agréablement l'inexistence de variables partagées entre processus. Quant à la partie des commandes (<command part>), nous gardons, pour le moment, le formalisme du chapitre précédent, avec les simplifications introduites à propos du problème des philosophes. Nous acceptons quand même une petite généralisation qui consiste à admettre dans un processus plusieurs communications gardées avec la même porte, à condition que les gardes respectives soient disjointes deux à deux. La signification d'une telle construction est expliquée de la manière suivante, pour le cas de deux communications gardées, (la généralisation

à plus que deux est triviale) : la paire

$$k1:\alpha \rightarrow o1_\alpha, j1_\alpha, \quad k2:\alpha \rightarrow o2_\alpha, j2_\alpha$$

où  $k1k2 = 1$ , doit être interprété comme

$$k:\alpha \rightarrow o_\alpha, j_\alpha$$

avec  $k = k1k2$ ,  $o_\alpha = \lambda x. \text{ si } k1(x) \text{ alors } o1_\alpha(x) \text{ sinon } o2_\alpha(x)$

et  $j_\alpha = \lambda s. \lambda x. \text{ si } k1(x) \text{ alors } j1_\alpha(s, x) \text{ sinon } j2_\alpha(s, x)$ .

Nous présentons tout de suite un exemple qui illustre l'utilisation du Langage et qui éclairera les points non explicités.

### 3. LE PROBLEME DES TELEGRAMMES

Nous nous proposons maintenant de programmer le "problème des télégrammes" de Henderson et Snowdon [HeS72], dans le Langage décrit dans la section précédente. Ce problème est spécifié par ses auteurs comme suit :

"a program is required to process a stream of telegrams. This stream is available as a sequence of letters, digits and blanks on some device and can be transferred in sections of predetermined size into a buffer area where it is to be processed. The words in the telegram are separated by sequences of blanks and each telegram is delimited by the word ZZZZ. The stream is terminated by the occurrence of the empty telegram i.e. a telegram with no words. Each telegram is to be processed to determine the number of chargeable words and to check for occurrences of overlength words. The words ZZZZ and STOP are not chargeable and words of more than twelve letters are considered overlength. The result of the processing is to be neat listing of the telegrams, each accompanied by the word count and a message indicating the occurrence of an overlength word."<sup>(1)</sup>

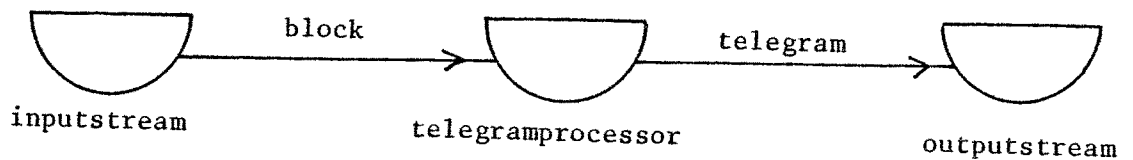
---

(1) "On demande un programme pour traiter un flot de télégrammes. Ce flot est disponible, sous forme d'une séquence de lettres, chiffres et espaces, sur un périphérique et il peut être transféré par sections de taille fixe vers un tampon où il doit être traité. Les mots d'un télégramme sont séparés par des séquences d'espaces et chaque télégramme se termine avec le mot ZZZZ. La fin du flot est signalé par l'occurrence  
.../..."

La programmation de ce problème dans le langage CSP fait l'objet d'un article par McKeag et Milligan [McM80]. Nous suivons leur interprétation de l'énoncé. Ainsi notre "neat listing" consiste simplement, pour chaque télégramme, de la séquence des mots du télégramme, suivi par le nombre de mots comptés. Dans cette séquence, les terminateurs ZZZZ sont omis et les mots sont séparés par un seul espace. (Nous ne prenons pas en compte la longueur de la ligne de l'imprimante; ainsi, il peut y avoir des mots coupés en fin de ligne). Les mots trop longs seront tronqués à douze caractères, et on leur ajoute un astérisque, qui doit être interprété comme le message signalant un mot trop long.

L'intérêt de ce problème réside dans l'existence d'"un conflit de structures" entre les données à l'entrée, (les sections de taille fixe), et les données de sortie (les textes des télégrammes, plus le nombre de mots). Pour obtenir une solution pour le problème, nous cherchons une décomposition du système en essayant d'identifier un certain nombre de "tâches" indépendantes, dont la combinaison réalise le comportement global souhaité. Notre approche est descendante, étant donné qu'une tâche peut très être décomposée en tâches plus élémentaires. Une fois la décomposition choisie, nous programmons les tâches en tant que processus communicants, et nous les assemblons dans un réseau approprié.

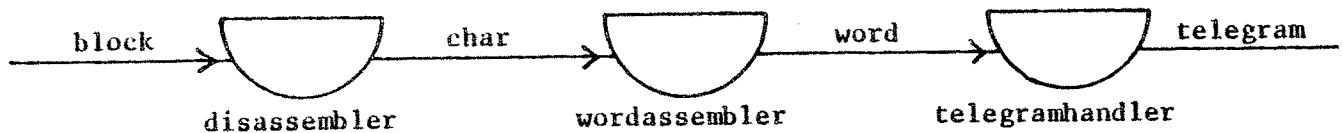
Appelant inputstream le processus qui s'occupe des données à l'entrée, et outputstream celui qui traite les sorties, et notant block les sections de taille fixe, nous avons une première décomposition du système



(suite note) d'un télégramme vide, c'est-à-dire un télégramme sans mot. Le traitement de chaque télégramme consiste à dénombrer les mots à payer et à vérifier les occurrences de mots trop longs. Les mots ZZZZ et STOP sont gratuits et les mots ayant plus de douze lettres sont considérés comme trop longs. Le résultat du traitement est l'édition d'une liste des télégrammes, chaque télégramme étant accompagné du nombre de mots et d'un message indiquant l'occurrence d'un mot trop long".

où telegramprocessor est le processus qui s'occupe de la construction des télégrammes à partir des "blocks". En fait, il faut encore ajouter une "ligne de communication" entre telegramprocessor et outputstream avec laquelle le premier de ces deux processus signale à l'autre la fin du traitement.

Le processus telegramprocessor doit maintenant être décomposé. Un télégramme étant formé par des mots et les mots par caractères, nous proposons



Nous constatons que le réseau obtenu est linéaire. On peut remarquer que ce n'est pas très significatif de ce que l'on peut faire avec des processus communicants. De toute façon, ce qui nous intéresse le plus dans cet exemple, est la conception descendante de la solution en termes de processus communicants, et aussi le fait qu'il nous permet d'illustrer facilement une technique d'implémentation.

Nous devons définir les types block word et telegram. Block sera défini comme un tableau de caractères de taille 40. Quant à word et telegram pour raccourcir le programme nous supposons l'existence d'un type prédéfini string, dont les valeurs sont des chaînes de caractères de longueur arbitraire. A la différence du langage Pascal on admet des variables de type string. Les constantes de ce type sont notées entre guillemets. La chaîne vide est représentée "". L'opération de concaténation est notée +. Il existe une fonction prédéfinie, size, délivrant la longueur d'une chaîne. On peut accéder aux caractères d'une chaîne par indexation. Les indices sont croissants de gauche à droite à partir de 1. Ainsi si  $x = \text{"abc"}$  alors  $\text{size}(x) = 3$ ,  $x(1) = \text{'a'}$ ,  $x(2) = \text{'b'}$ ,  $x(3) = \text{'c'}$  et  $x(4)$  est indéfini. En général pour une expression  $x$  de type str



$x(i)$  n'est défini que si  $0 < i \leq \text{size}(x)$ . La notation  $x(..i)$ ,  $i \geq 0$ , désigne la chaîne formée par les  $i$  premiers éléments de la chaîne  $x$ , si  $i \leq \text{size}(x)$ , et la chaîne  $x$  autrement. Enfin, il y a conversion automatique du type `char` vers le type `string`, de manière triviale.

On suppose encore l'existence d'un autre type prédéfini appelé `signal`, qui a un seul élément pour lequel il n'y a pas de notation.

Nous présentons maintenant le "problème des télégrammes" programmé dans le langage.

```

system telegramsproblem;
  const blocksize=40;
  type block = array [1..blocksize] of char;

  type telegram = record charge : integer;
                    text : string
  end;

  process inputstream (outbk : out block);
    var infile : file of block;
    begin
      reset(infile);
      do not eof(infile) : outbk → ! infile↑, get(infile) od
    end{inputstream};

  process telegramprocessor (inbk : in block; outtg : out telegram;
                            stop : out signal);

    type word = string;

  process disassembler (inbk : in block; outch : out char);
    var b : block;
        i : integer;

```

```

begin
  i:=0;
  do i=0 : inbk → ?x : b, i:=x,1
    □ i≥1 and i≤blocksize : outbk → !b[i], i:=i+1
    □ i=blocksize+1 : outbk → !' ', i:=0
  od
end
end{disassembler};

process wordassembler (inch : in char; outwd : out word);
  var w : word ;
  q : 0..2 ;
  function shorten (w : word) : word;
  begin
    if size>(w) 12 then shorten:=w(..12)+"*"
    else shorten:=w
  end ;

  begin
    w, q:="" ,0;
    do q =0 : inch → ?x:if x≠' ' then w, q:=x,1
    □ q =1 : inch → ?x:if x=' ' then w:=w+x else q:=2
    □ q =2 : outwd → !shorten(w), w, q:="" ,0
  od
end
end{wordassembler};

```

```

process telegramhandler (inwd : in word; outtg : out telegram;
                           stop : out signal);

  var t : telegram;
      input : boolean;
  procedure appendtg (var t : telegram; w : word);
  begin
    if w ≠ "" then
      with t do
        begin
          text:=text+w+" ";
          if w="STOP" then charge:=charge+1
        end
      end;
  end;

  begin
    t.charge, t.text, input:=0, "", true ;
    do input : inwd → ? w:if w="ZZZZ" then input:=false
      else appendtg(t,w)
    [] not input and t;text ≠ "" : outtg → !t:
      t.charge, t.text, input:=0, "", true
    [] not input and t.text = "" : stop → input:=true
    od
  end
end {telegramhandler};

```

network

```
di(a,-b), wa(b,-c), tgh(c,-d,-e) : di||wa||tgh;
```

nodes

```
di:=disassembler;
wa:=wordassembler;
tgh:=telegramhandler ;
```

ports

```
a:=inbk;
-d:=outtg;
-e:=stop
```

```
end {telegramprocessor};
```

```
process outputstream (intg : in telegram; stop : in signal);
```

```
var on : boolean;
```

```
outfile : text;
```

```
procedure writetelegram (t : telemgram);
```

```
begin
```

```
writeln (outfile);
```

```
writeln (outfile, t.text); writeln (outfile, t.charge)
```

```
end;
```

```
begin
```

```
on:=true, rewrite (outfile);
```

```
do on : intg → ?t : writetelegram (t)
```

```
□ on : stop → writeln (outfile, "--end of telegram stream");
```

```
on:=false
```

```
od
```

```
end
```

```
end {outputstream} ;
```

network

ins(-a), tgp (a,-b,-c), outs(b,c) : ins || tgp || outs ;

nodes

ins := inputstream ;  
 tgp := telegramprocessor ;  
 outs := outputstream

end {telegramsproblem}.

#### 4. SUGGESTIONS POUR L'IMPLANTATION

La sémantique relationnelle des processus communicants étudiée dans le chapitre précédent nous enseigne à construire un programme séquentiel (non-déterministe) représentant le fonctionnement d'un système de processus donné. Nous pouvons envisager une technique d'implémentation du Langage, basée sur ces idées, mettant en oeuvre la traduction automatique d'un programme parallèle du Langage dans un programme séquentiel d'un langage répandu. Vu le choix adopté pour la structure déclarative du Langage langage-objet qui mieux s'adapte à ce travail est, naturellement, le langage Pascal.

Notre objectif devient alors de trouver des règles de traduction automatique du Langage vers Pascal. Pour faciliter ce travail nous commençons par transformer le formalisme utilisé dans la partie des commandes, (<command part>), des processus de façon à ce qu'il se rapproche plus du langage Pascal. D'une part les affectations simultanées d'initialisation sont remplacées par des séquences d'affectations simples, ce qui ne pose pas de problèmes car les parties droites sont toujours des constantes. D'autre part dans les commandes gardées g→r les gardes g sont des expressions booléennes Pascal et les commandes r des séquences de commandes ("statements") Pascal. Finalement dans les communications gardées k:α→o,j les k's sont aussi

des expressions booléennes Pascal, les o's sont des expressions Pascal de type approprié, et dans les j's la variable d'entrée est supposée implicitement déclarée avec le type de la porte associée, sa portée étant le texte de la fonction d'entrée. Le "corps" de cette fonction est représenté par une suite de commandes Pascal.

Le programme-objet -ceci est facile à voir- est formé d'une partie déclarative, suivie d'une série d'affectations d'initialisation, suivie d'une commande itérative qui simule la commande do-od du processus composé. A part un certain nombre de points mineurs que nous laissons implicites, il y a quatre aspects requérant une analyse plus détaillée : la gestion des déclarations, la représentation des communications, la traduction des fonctions de sortie et la simulation du non-déterminisme.

#### i. Gestion des déclarations

Les déclarations seront regroupées, dans l'ordre où elles apparaissent, dans des zones de déclarations de constante, de type, de variable et de procédure et fonction. Ces zones sont ensuite juxtaposées dans le programme objet. Il faudra éventuellement renommer certains identificateurs pour éviter des conflits, absents dans la structure arborescente mais qui apparaissent quand on l'aplatit.

L'impossibilité de définir des constantes de type structuré en Pascal est très gênante, d'autant plus que dans le Langage des variables ne peuvent être définies qu'au niveau des processus élémentaires. Pour éviter cette limitation, nous admettons la déclaration de variables au niveau des processus non-élémentaires avec la restriction que ces variables ne peuvent être affectées que dans la zone d'initialisation. Elles se comportent donc comme des vraies constantes.

#### ii. Représentation des communications

Prenons deux communications gardées complémentaires :

$$k_{\alpha}(x1) : \alpha \rightarrow !o_{\alpha}(x1), ?s1 : x1 := j_{\alpha}(s1, x1)$$

dans le processus P1, et

$$k_{\bar{\alpha}}(x2) : \bar{\alpha} \rightarrow !o_{\bar{\alpha}}(x2), ?s2 : x2 := j_{\bar{\alpha}}(s2, x2)$$

dans le processus P2.

Nous avons vu que la communication entre P1 et P2 par la ligne  $\{\alpha, \bar{\alpha}\}$  est représentée par la commande gardée

$$k_{\alpha}(x1) \text{ and } k_{\bar{\alpha}}(x2) \rightarrow x1, x2 := j_{\alpha}(o_{\bar{\alpha}}(x2), x1), j_{\bar{\alpha}}(o_{\alpha}(x1), x2).$$

Dans cette commande gardée l'affectation simultanée ne peut pas être remplacée sans précaution par la composition séquentielle des deux affectations simples. Pour garder la symétrie nous proposons de récupérer les deux variables d'entrée (en renommant éventuellement l'une d'elles), pour mémoriser, dans un premier temps les valeurs échangées et pour ensuite permettre le calcul indépendant des fonctions d'entrée. Ainsi dans notre exemple nous avons

$$k_{\alpha}(x1) \text{ and } k_{\bar{\alpha}}(x2) \rightarrow s1 := o_{\bar{\alpha}}(x2); s2 := o_{\alpha}(x1) ; \\ x1 := j_{\alpha}(s1, x1); x2 := j_{\bar{\alpha}}(s2, x2)$$

Dans le cas unidirectionnel la commande ci-dessus peut se simplifier. Supposons que P1 soit l'émetteur et P2 le récepteur. Ceci signifie, en fait, que la valeur envoyée par P2 est un signal. Dans ces conditions la fonction  $o_{\bar{\alpha}}$  est triviale et la variable s1 peut être éliminée. Nous avons alors :

$$k_{\alpha}(x1) \text{ and } k_{\bar{\alpha}}(x2) \rightarrow s2 := o_{\alpha}(x1) ; \\ x1 := j_{\alpha}(x1) ; x2 := j_{\bar{\alpha}}(s2, x2)$$

Il est maintenant évident que la variable s2 peut aussi être éliminée à condition d'inverser l'ordre des deux dernières affectations. Nous obtenons ainsi finalement :

$$k_{\alpha}(x1) \text{ and } k_{\bar{\alpha}}(x2) \rightarrow x2 := j_{\bar{\alpha}}(o_{\alpha}(x1), x2) ; x1 := j_{\alpha}(x1)$$

iii. Traduction des fonctions de sortie et d'entrée

Les fonctions de sortie sont écrites dans la forme d'expressions Pascal.

Elles ne posent pas de problèmes. Les fonctions d'entrée, dont la mission est de calculer le nouvel état sont traduites par des processus à effet de bord sur les variables définissant l'état du processus. Plus précisément, dans le cas unidirectionnel, la fonction d'entrée du récepteur est représenté par une procédure avec un paramètre du type de la porte complémentaire, passé par valeur. Le nom de la procédure est le nom de la porte correspondante, (ce qui implique peut-être la nécessité de renommer des portes) ; le paramètre formel est la variable d'entrée ; le corps est le texte qui suit les ":". Pour l'émetteur tout se passe de la même manière sauf que la procédure n'a pas de paramètres. Ainsi, la communication qui a illustré le point précédent est traduite par

$$k_{\alpha}(x1) \text{ and } k_{\alpha}(x2) \rightarrow \bar{\alpha}(o_{\alpha}(x1)) ; \alpha$$

les déclarations des procédures  $\alpha$  et  $\bar{\alpha}$  se trouvant ailleurs.

#### iv. Simulation du non-déterminisme

Le non-déterminisme peut être réalisé, par exemple, à l'aide d'un générateur des nombres aléatoires ou de l'horloge de la machine. Nous préférons une implantation déterministe du non-déterminisme. Pour le cas des programmes représentant des systèmes de processus qui se terminent nécessairement ce choix est tout à fait légitime. Nous traduirons la boucle do-od par une boucle while conditionnée par une variable booléenne, exécute, initialisée à vrai. Le corps de while consiste en une série de if then else en cascade, où après le dernier else apparaîtra l'affectation de faux à exécute. Par exemple, la commande

```

do g1 → r1
    □ g2 → r2
od ;

```

sera traduite par

```

execute :=true ;
while execute do
    if g1 then r1
    else if g2 then r2
    else execute :=false ;

```



où les r's sont entourés par les parenthèses begin end, si nécessaire.

L'ordre d'inclusion des commandes gardées dans la cascade des if then else est, en principe sans importance. Néanmoins, des soucis d'efficacité nous conduisent naturellement à privilégier les communications plus fréquentes. Une heuristique simple indique que les communications plus fréquentes correspondent probablement aux transmissions de données de petite taille. Par contre, les signaux ne doivent être utilisés que rarement, pour signaler des situations particulières dans le calcul. Ainsi, nous les laisserons pour la fin. Dans une situation intermédiaire on laisse les commandes gardées des processus élémentaires, s'il y en a. Il est évident que cette question peut être l'objet de nombreuses optimisations.

□

Pour illustrer la technique que nous venons de décrire nous présentons maintenant le programme du problème des télégrammes dans la version du langage avec les conventions Pascal et nous montrons ensuite le résultat d'une compilation faite à la main. Par rapport au programme de la section précédente, notons que nous devons programmer explicitement les types string et signal, et que nous nous efforçons de ne pas avoir de répétition de portes dans un processus afin de ne pas augmenter le nombre de commandes gardées du système complet.

- Notes 1. Nous remplaçons les corps des procédures par des commentaires. On trouve ces corps de procédure dans le programme-objet.
2. Parfois nous utilisons l'appel par référence uniquement pour éviter la recopie des paramètres effectifs.
3. tg.dat et tg.res sont les fichiers physiques d'entrée et de sortie, respectivement.
4. La version du langage Pascal utilisée pour exécuter le programme objet requiert la fermeture explicite des fichiers de sortie, par l'intermédiaire d'une fonction prédéfinie, close.

□

```

system tgproblem ;
    const blocksize=40
    type block=array[1..blocksize] of char ;

    const stringsize=256 ;
    type string=record length : 0..stringsize ;
        text : array[1..stringsize] of char
    end ;
    procedure writestring (var f : text ; var s : string) ;
        {écrit la chaîne s sur le fichier f}

    type telegram=record charge : integer ;
        text : string
    end ;
    procedure emptytg (var t : telegram) ;
        {crée un télégramme vide}
    function isemptytg (var t : telegram) : boolean ;
        {true ssi t est un télégramme vide}

    type signal=(bip) ;

    process inputstream (outbk : out block) ;
        var bkfile : file of block ;
        begin
            reset (bkfile, 'tg.dat') ;
            do not eof (bkfile) : outbk + !bkfile!, get(bkfile) od
        end (inputstream) ;

    process          tgprocessor (inbk : in block ; outtg : out telegram ;
                                stop : out signal)

        const wordsize=16 ;
        type word : record size : 0..wordsize ;
            contents : array [1..wordsize] of char
        end ;
        procedure emptyword (var w : word) ;
            {crée un mot vide}
        function isemptyword (var w : word) : boolean ;
            {true ssi w est le mot vide}

```

```

function equalwords (var w1,w2 : word) : boolean ;
    {true ssi w1=w2}
procedure appendwd (var w : word ; c : char) ;
    {ajoute c à w si la longueur de w est inférieure à
     wordsize ; sinon ne fait rien}
procedure concatenate (var s : string ; w : word) ;
    {concatène w à s ; si le résultat est plus long que
     stringsize seulement les premiers stringsize caracteres
     sont gardés}
var wstop, wzzzz, xspace : word ;
    {il s'agit de constantes de fait}
procedure initconstwords ;
    {initialise les mots constantes ci-dessus}

process disassembler (inbk : in block ; outch : out char) ;
    var bk : block ;
        ibk : integer ;
    function sendchar (i : integer) : char ;
    {calculé le caractère, qui doit être envoyé}
    begin
        ibk :=0 ;
        do ibk=0 : inbk → ?b : bk:=b ; ibk:=i
            □ ibk>0 : outch → !sendchar(ibk),
                ibk:=(ibk+1) mod (blocksize+2)
        od
    end {disassembler} ;

process wordassembler (inch : in char ; outwd : out word) ;
    var wd : word ;
        inputch, inblanks : boolean ;
    procedure shorten (var w : word) ;
        {si w a plus que 12 caracteres on ne garde que les 12
         premiers et on ajoute '*' ; sinon on ne fait rien}
    begin
        emptyword(wd) ;
        inputch :=true ; inblanks :=true ;

```

```

do inputch : inch → ?c :
    if c≠' ' then begin appendwd (wd,c) ;
                        inblanks :=false
                    end
    else
        if not inblanks then
            begin shorten (wd) ;
                    inblanks :=true ;
                    inputch :=false
            end
        [] not inputch : outwd → !wd ;
            emptyword(wd) ; inputch :=true
    od
end {wordassembler} ;

process tghandler(inwd : in word ; outtg : out telegram ;
                 nomore : out signal) ;

var tg : telegram ;
    inputwd : boolean ;
procedure appendtg (var t : telegram ; var w : word) ;
    {ajoute le mot w au telegramme t, en incrémentant le
     nombre de motssi w≠'stop'}

begin
    initconstwords ;
    emptytg(tg) ;
    inputwd :=true ;
do inputwd : inwd → ?w :
    if equalwords (w,wzzzz) then inputwd := false
        else appendtg(tg,w)
    [] not inputwd and not isemptytg(tg) : outtg → !tg,
        emptytg(tg) ; inputwd :=true
    [] not inputwd and isemptytg(tg) : nomore →
        inputwd :=true
    od
end {tgandler} ;

```

```

network {comme dans la section IX.3.} ;
nodes   {idem} ;
ports   {idem}
end {tgprocessor} ;

process outputstream (intg : in telegram ; stop : in signal) ;
  var tgfile : text ;
    onoutput : boolean ;
  procedure writetg (var t : telegram) ;
    {écrit le telegramme sur le fichier tgfile}

  begin
    rewrite (tgfile, 'tg.res') ;
    onoutput :=true ;
    do onoutput : intg →?t : writetg(t)
    □ onoutput : stop →
      writeln(tgfile, '--end of telegram output') ;
      close(tgfile) ;
      onoutput :=false
    od
  end {outputstream} ;

network {comme dans la section IX.3.} ;
nodes   {idem}

end {tgproblem}.

```

Nous présentons de suite le listing du programme-objet ainsi qu'un jeu d'essais.

```

CONST BLOCKSIZE = 40 ;
      STRINGSIZE = 256 ;
      WORDSIZE = 16 ;

TYPE BLOCK = ARRAY [ 1..BLOCKSIZE ] OF CHAR ;
      STRING = RECORD LENGTH : 0..STRINGSIZE ;
                  TEXT : ARRAY [ 1..STRINGSIZE ] OF CHAR
                END ;
      TELEGRAM = RECORD CHARGE : INTEGER ;
                  TEXT : STRING
                END ;
      SIGNAL = ( BIP ) ;
      WORD = RECORD SIZE : 0..WORDSIZE ;
              CONTENTS : ARRAY [ 1..WORDSIZE ] OF CHAR
            END ;

VAR BKFILE : FILE OF BLOCK ;
    TGFILE : TEXT ;
    WSTOP , WZZZ , WSPACE : WORD ;
    BK : BLOCK ;
    IEK : INTEGER ;
    WD : WORD ;
    INPITCH , INBLANKS : BOOLEAN ;
    TG : TELEGRAM ;
    INPUTWD : BOOLEAN ;
    ONOUTPUT : BOOLEAN ;
    EXECUTE : BOOLEAN ;

PROCEDURE WRITESTRING ( VAR S : STRING ) ;
VAR I : INTEGER ;
BEGIN FOR I := 1 TO S.LENGTH DO WRITE ( TGFILE , S.TEXT [ I ] ) END ;

PROCEDURE EMPTYTG ( VAR T : TELEGRAM ) ;
BEGIN T.CHARGE := 0 ; T.TEXT.LENGTH := 0 END ;

FUNCTION ISEMPYTG ( VAR T : TELEGRAM ) : BOOLEAN ;
BEGIN ISEMPYTG := T.TEXT.LENGTH = 0 END ;

PROCEDURE EMPTYWORD ( VAR W : WORD ) ;
BEGIN W.SIZE := 0 END ;

FUNCTION ISEMPYWORD ( VAR W : WORD ) : BOOLEAN ;
BEGIN ISEMPYWORD := W.SIZE = 0 END ;

FUNCTION EQUALWORDS ( VAR W1 , W2 : WORD ) : BOOLEAN ;
VAR I : INTEGER ;
    E : BOOLEAN ;
BEGIN
  I := 0 ;
  E := W1.SIZE = W2.SIZE ;
  WHILE E AND ( I < W1.SIZE ) DO
    BEGIN
      I := I + 1 ;
      E := W1.CONTENTS [ I ] = W2.CONTENTS [ I ]
    END ;
  EQUALWORDS := E
END ;

PROCEDURE APPENDWD ( VAR W : WORD ; C : CHAR ) ;
(* APPENDS C TO W IF THE CURRENT SIZE OF W IS LESS THAN WORDSIZE *)
BEGIN
  WITH W DO
    IF SIZE < WORDSIZE THEN
      BEGIN
        SIZE := SIZE + 1 ;
        CONTENTS [ SIZE ] := C
      END
    END ;
END ;

```

```

PROCEDURE CONCATENATE ( VAR S : STRING ; W : WORD ) ;
(* CONCATENATES W TO S. IF THE RESULT IS LONGER THAN STRINGSIZE
  ONLY THE FIRST STRINGSIZE CHARACTERS ARE RETAINED *)
VAR I : INTEGER ;
BEGIN
  WITH W , S DO
    BEGIN
      IF SIZE > STRINGSIZE - LENGTH THEN SIZE := STRINGSIZE - LENGTH ;
      FOR I := 1 TO SIZE DO TEXT [ LENGTH + I ] := CONTENTS [ I ] ;
      LENGTH := LENGTH + SIZE
    END
  END ;

PROCEDURE INITCONSTWORDS ;
BEGIN
  WITH WSTOP DO
    BEGIN
      SIZE := 4 ;
      CONTENTS [ 1 ] := 'S' ; CONTENTS [ 2 ] := 'T' ;
      CONTENTS [ 3 ] := 'O' ; CONTENTS [ 4 ] := 'P' ;
    END ;
  WITH WZZZZ DO
    BEGIN
      SIZE := 4 ;
      CONTENTS [ 1 ] := 'Z' ; CONTENTS [ 2 ] := 'Z' ;
      CONTENTS [ 3 ] := 'Z' ; CONTENTS [ 4 ] := 'Z' ;
    END ;
  WSPACE.SIZE := 1 ; WSPACE.CONTENTS [ 1 ] := ' ' ;
END ;

FUNCTION SENDCHAR ( I : INTEGER ) : CHAR ;
BEGIN IF I <= BLOCKSIZE THEN SENDCHAR := BK [ I ]
      ELSE SENDCHAR := ' ' ;
END ;

PROCEDURE SHORTEN ( VAR W : WORD ) ;
BEGIN
  WITH W DO
    IF SIZE > 12 THEN BEGIN SIZE := 13 ; CONTENTS [ 13 ] := '*' END
  END ;
END ;

PROCEDURE APPENDTG ( VAR T : TELEGRAM ; VAR W : WORD ) ;
BEGIN
  IF NOT ISEMPTYWORD ( W ) THEN
    WITH T , W DO
      BEGIN
        CONCATENATE ( TEXT , W ) ; CONCATENATE ( TEXT , WSPACE ) ;
        IF NOT EQUALWORDS ( W , WSTOP ) AND
          NOT EQUALWORDS ( W , WZZZZ ) THEN CHARGE := CHARGE + 1
      END
    END ;
END ;

PROCEDURE WRITETG ( VAR T : TELEGRAM ) ;
BEGIN
  WRITELN ( TGFILE ) ;
  WRITESTRING ( T.TEXT ) ; WRITELN ( TGFILE ) ;
  WRITELN ( TGFILE , T.CHARGE ) ; WRITELN ( TGFILE )
END ;

(***** FORT PROCEDURES *****)

PROCEDURE OUTBK ;
BEGIN GET ( BKFILE ) END ;

PROCEDURE INBK ( B : BLOCK ) ;
BEGIN BK := B ; IBK := IBK + 1 END ;

PROCEDURE OUTCH ;
BEGIN IBK := ( IBK + 1 ) MOD ( BLOCKSIZE + 2 ) END ;

```

```

PROCEDURE INCH ( C ; CHAR ) ;
BEGIN
  IF C = ' ' THEN
    BEGIN
      APPENDWD ( WD , C ) ;
      INBLANKS := FALSE
    END
  ELSE
    IF NOT INBLANKS THEN
      BEGIN
        SHORTEN ( WD ) ;
        INBLANKS := TRUE ; INPUTCH := FALSE
      END
    END ;

PROCEDURE OUTWD ;
BEGIN EMPTYWORD ( WD ) ; INPUTCH := TRUE END ;

PROCEDURE INWD ( W ; WORD ) ;
BEGIN IF EQUALWORDS ( W , WZZZ ) THEN INPUTWD := FALSE
      ELSE APPENDTG ( TG , W )
END ;

PROCEDURE OUTTG ;
BEGIN EMPTYTG ( TG ) ; INPUTWD := TRUE END ;

PROCEDURE NOHORE ;
BEGIN INPUTWD := TRUE END ;

PROCEDURE INTG ( T ; TELEGRAM ) ;
BEGIN WRITETG ( T ) END ;

PROCEDURE STOP ;
BEGIN WRITELN ( TGFILE , '--END OF TELEGRAM OUTPUT' ) ; CLOSE ( TGFILE ) ;
      ONOUTPUT := FALSE
END ;

BEGIN
  RESET ( BKFILE , 'TG.DAT' ) ; REWRITE ( TGFILE , 'TG.RES' ) ;
  INITCONSTWORDS ;
  IBK := 0 ;
  EMPTYWORD ( WD ) ;
  INPUTCH := TRUE ; INBLANKS := TRUE ;
  EMPTYTG ( TG ) ;
  INPUTWD := TRUE ;
  ONOUTPUT := TRUE ;

  EXECUTE := TRUE ;

  WHILE EXECUTE DO
    IF ( IBK > 0 ) AND INPUTCH
    THEN
      BEGIN INCH ( SENDCHAR ( IBK ) ) ; OUTCH END
    ELSE IF NOT INPUTCH AND INPUTWD
    THEN
      BEGIN INWD ( WD ) ; OUTWD END
    ELSE IF NOT EOF ( BKFILE ) AND ( IBK = 0 )
    THEN
      BEGIN INBK ( BKFILE ) ; OUTBK END
    ELSE IF NOT INPUTWD AND NOT ISEMPYTG ( TG ) AND ONOUTPUT
    THEN
      BEGIN INTG ( TG ) ; OUTTG END
    ELSE IF NOT INPUTWD AND ISEMPYTG ( TG ) AND ONOUTPUT
    THEN
      BEGIN STOP ; NOHORE END
    ELSE EXECUTE := FALSE
  END.

```



## Donnée

HEUREUX ANNIVERSAIRE TANTE FRANCOISE  
 ONCLE JULES ZZZZ RENTRE MAISON STOP  
 TOUT EST PARDONNE STOP MAMAN MALADE  
 STOP PAPA ZZZZ ENVOYEZ ARGENT SUP PAUL  
 ZZZZ ARRIVE GARE GRENOBLE MARDI  
 PROCHAIN 15 HEURES SILVAIN ZZZZ ACHETE  
 ROYAL PETROLEUM SIR CHARLES ZZZZ BRAVO  
 TA REUSSITE MAMAN PAPA ZZZZ  
 ILOVEYOUILOVEYOU JOHN ZZZZ REI JOS  
 PARABENS PEDRO ZZZZ TREMBLEMENT TERRE  
 ITALIE STOP 3000 MORTS STOP  
 SECOURS URGENT STOP FRANCOLINI  
 ZZZZ PREMIER MINISTRE PORTUGAIS TUE  
 ACCIDENT AVION STOP CAUSES INCONNUES  
 STOP ATTENDRE INFORMATIONS ULTERIEURES  
 STOP SEQUEIRA ZZZZ MADAME LE MINISTRE  
 STOP ENSEIGNANTS REUNIS ASSEMBLEE  
 GENERALE PROTESTENT CONTRE MESURES  
 ATTENTATOIRES LIBERTES DEMOCRATIQUES  
 STOP EXIGENT SUSPENSION DESDITES STOP  
 LE PRESIDENT ASSEMBLEE MARTIN ZZZZ  
 REGRETTE POUVOIR PAS ETRE PRESENT STOP  
 FELICITATIONS MICHEL ZZZZ EANES  
 REMPORTE ELECTIONS PRESIDENCIELLES  
 PORTUGAISES PREMIER TOUR STOP EANES 56  
 POURCENT SOARESCARNEIRO 40 POURCENT STOP  
 SILVA ZZZZ ZZZZ

Résultat

HEUREUX ANNIVERSAIRE TARTE FRANCOISE ONCLE JULES  
2

RENTRE MAISON STOP TOUT EST PARDONNE STOP MAHAM MALADE STOP PAPA  
8

ENVOYEZ ARGENT SUP PAUL  
4

ARRIVE GARE GRENOBLE MARDI PROCHAIN 15 HEURES SILVAIN  
8

ACHETE ROYAL PETROLEUM SIR CHARLES  
5

BRAVO TA REUSSITE MAHAM PAPA  
5

ILOVEYOUILOV\* JOHN  
2

REI JOS PARABENS PEURO  
3

TREMELEMENT TERRE ITALIE STOP 3000 MORTS STOP SECOURS URGENT STOP FRANCOLINI  
8

PREMIER MINISTRE PORTUGAIS TUE ACCIDENT AVION STOP CAUSES INCONNUES STOP ATTENDRE INFORMATIONS ULTERIEURES STOP SEQUEIRA  
12

MADAME LE MINISTRE STOP PHSEIGNANTS REUNIS ASSEMBLEE GENERALE PROTESTENT CONTRE MESURES ATTENTATOIRE\* LIBERTES DEMOCRATIQUE\* STOP I  
IGENT SUSPENSION DESDITES STOP LE PRESIDENT ASSEMBLEE MARTIN  
20

REGRETTE POUVOIR PAS ETRE PRESENT STOP FELICITATION\* MICHEL  
7

EANES RENPORTE ELECTIONS PRESIDENCIEL\* PORTUGAISES PREMIER TOUR STOP EANES 56 POURCENT SOARESCARNEI\* 40 POURCENT STOP SILVA  
14

--END OF TELEGRAM OUTPUT

## 5. CONCLUSION

L'exemple de la programmation du problème des télégrammes et les suggestions d'implémentation montrent que le langage proposé offre déjà des possibilités de description intéressantes. Il garde cependant, quelques restrictions qui s'avèrent très gênantes dès que la "complexité" du système décrit augmente. Pour surmonter ces contraintes nous envisageons certaines facilités, ne trahissant pas l'esprit du langage, mais améliorant nettement la commodité de son utilisation. Par exemple :

i. Imbrication de la structure do od des processus

L'écriture

```
x:=X ;
do gα(x) : α → !oα(x), ?s:=jα(s,x) ;;
      do gβ(x) : β → !oβ(x), ?s:=jβ(s,x)
      od
od
```

sera admise et sa signification est donnée à l'aide d'un boolean supplémentaire b par

```
x,b :=X, true ;
do b and gα(x) : α → !oα(x), ?s : x,b :=jα(s,x), false
  □ not b and gβ(x) : β → !oβ(x), ?s : x:=jβ(s,x)
  □ not b and gβ(x) → b :=true
od
```

ii. Définition d'autres types de communication, comme la communication à plusieurs, et la communication étendue (où la valeur émise par un des processus est fonction aussi de la valeur reçue, cf. le langage Ada [Ada79]).

iii. Définition d'autres types de connexion dans le réseau, en particulier pour permettre la description des communications prévues dans le point précédent.

CHAPITRE X

CONCLUSION



Nous avons présenté un cadre cohérent pour l'étude des programmes qui décrivent des systèmes de processus. Nous avons étudié en détail le cas correspondant aux systèmes formés d'un seul processus, i.e., les programmes séquentiels non-déterministes. Ceci est d'autant plus justifié que, d'après la sémantique présentée au chapitre VIII, un système de plusieurs processus peut être vu, à un certain niveau de description, comme un seul processus.

Nous estimons que le modèle des relations programmables développé dans les chapitres II, III et IV a atteint une forme assez définitive. Il reste cependant beaucoup de travail à faire, notamment en ce qui concerne la comparaison systématique avec d'autres modèles pour le non-déterministe, tel que ceux de Wand [Wan77], Hoare [Hoa78a], Harel [Har79], et Schmidt [Sch79]. Il serait aussi intéressant de définir clairement la liaison avec la logique modale, Manna et Pnueli [MaP79]. De toute façon, nous croyons que la plupart des propriétés intéressantes des relations programmables auront été établies.

Par contre, en ce qui concerne le parallélisme nous sommes conscients que notre approche demeure assez intuitive et informelle. Nous croyons, néanmoins, que les résultats et les exemples présentés montrent que les potentialités de cette approche méritent d'être mieux exploitées. Nous l'avons d'ailleurs déjà utilisé pour faire quelques expériences sur d'autres problèmes soulevés par la programmation parallèle, et les résultats ont été encourageants.

Du point de vue de la sémantique, nous pensons qu'une formalisation plus abstraite, avec des outils mathématiques plus sophistiqués peut être convenable. En effet, l'utilisation de relations binaires devient maladroitée dès que les opérations à définir impliquent un changement fréquent de l'ensemble de base.

Un sujet que nous n'avons pas abordé est la comparaison et la liaison de notre modèle avec le calcul des systèmes communicants (CCS) de Milner [Mil80]. Nous croyons qu'une telle étude aidera à éclairer un certain nombre de points qui restent un peu flous dans notre modèle.

Quant au langage esquissé dans le chapitre IX, nous envisageons son élargissement avec un nombre restreint d'opérations de structuration de processus, pour qu'il devienne un langage de programmation à part entière. Son implémentation séquentielle selon les lignes suggérées aurait alors un intérêt effectif. Nous estimons que celle-ci est maintenant la voie expérimentale à poursuivre de manière prioritaire.

## REFERENCES

## Abréviations

## Périodiques

CACM	Communications of the ACM.
JACM	Journal of the ACM.
JCSS	Journal of Computer and System Sciences.
TCS	Theoretical Computer Science.
TOPLAS	ACM Transactions on Programming Languages and Systems.

## Conferences

ICALP	International Colloquium on Automata Languages and Programming.
MFCS	Mathematical Foundations of Computer Science.

## Série

LNCS	Lecture Notes on Computer Science.
------	------------------------------------

- [Ada81] "The Programming Language Ada". Reference Manual. LNCS 106, Springer (1981).
- [AFR79] K.R. Apt, N. Francez et W.P. de Roever.  
"Semantics for concurrently communicating finite sequential processes, based on predicate transformers".  
Université d'Utrecht, Progress Report (1979).
- [AFR80] K.R. Apt, N. Francez et W.P. de Roever.  
"A proof system for communicating sequential process".  
TOPLAS 2,3 (Juillet 1980), pp. 359-385.
- [AmC78] B. Amy et M. Caplain.  
"Invariances algorithmiques et récurrences".  
Proceedings of the Third International Symposium on Programming 1978, Dunod (Mars 1978).



- [Bes79] E. Best.  
"Weakest preconditions and recovery block semantics".  
Univ. Newcastle upon Tyne Comp. Lab. SRM/221 (Juin 1979)
- [Bli77] A. Blikle.  
"A comparative review of some program verification method  
MFCS 1977, LNCS 53 (1977), pp. 17-33.
- [BPW80] M. Broy, P. Pepper et M. Wirsing.  
"On relations between programs".  
Proceedings of the Fourth International Symposium on  
Programming (1980), LNCS 83, Springer (Avril 1980) pp. 5
- [Bri75] P. Brinch Hansen.  
"The programming language Concurrent Pascal".  
IEEE Trans Software Eng. 1,2 (Juin 1975), pp. 199-207.
- [BrK80] M. Broy et B. Krieg-Brückner.  
"Derivation of invariant assertions during program  
development and transformation". TOPLAS 2,3 (Juillet 1980)  
pp. 321-337.
- [Bur74] R. Burstall.  
"Program proving as hand simulation with a little  
induction". Proceedings IFIP Congress 1974,  
North Holland (1974), pp. 308-312.
- [Cla77] E.M. Clarke.  
"Program invariants as fixed points". 18th Ann. IEEE  
Symp. on Foundation of Computer Science, Providence R.I.  
EUA (Novembre 1977), pp. 18-28.
- [Cla80] E.M. Clarke.  
"Synthesis of resource invariants for concurrent  
programs". TOPLAS 2,3 (Juillet 1980), pp. 338-358.

- [CMPS80] F. De Cindio, G. De Michelis, L. Pomello, C. Simone.  
"A Petri net model of CSP". Istituto de Cibernetica,  
Univ. de Milan (1980).
- [CoC80a] P. Cousot et R. Cousot.  
"Semantic analysis of communicating sequential processes".  
Proceedings ICALP 1980, LNCS 85, Springer (1980) pp. 119-133
- [CoC80b] P. Cousot et R. Cousot.  
"Constructing program invariance proof methods".  
International Workshop on Programming Construction,  
Château de Bonas, France (Sept. 1980), Ed. INRIA.
- [Cod70] E.F. Codd.  
"A relational model for large shared data bases".  
CACM 13, 6 (Juin 1970), pp. 377-387.
- [Cou78] P. Cousot.  
"Méthodes itératives de construction et approximation de  
points fixes d'opérateurs monotones". Thèse d'Etat,  
Univ. Scient. et Méd. de Grenoble (Mars 1978).
- [Dij72] E.W. Dijkstra.  
"Notes on structured programming" in O.J. Dahl,  
E.W. Dijkstra et C.A.R. Hoare "Structured Programming".  
Academic Press (1972).
- [Dij75] E.W. Dijkstra.  
"Guarded commands, nondeterminacy and formal derivation  
of programs". CACM 18, 8 (Août 1975) pp. 453-457.
- [Dij76] E.W. Dijkstra.  
"A Discipline of Programming". Prentice Hall (1976).

- [Dij77] E.W. Dijkstra.  
"A correctness proof for communicating processes—a small exercise". EWD-607 (1977).
- [FHLR79] N. Francez, C.A.R. Hoare, D.J. Lehmann et W.P. de Roever.  
"Semantics of nondeterminism, concurrency and communication". JCSS 19, 3 (Décembre 1979).
- [Flo67] R.W. Floyd.  
"Assigning meaning to programs". Proc. Symp. in Applied Mathematics, vol 19, American Mathematical Society, Providence R.I. EUA (1967), pp. 19-32.
- [Flo79] R.W. Floyd.  
"The paradigms of programming". CACM 22, 8 (Août 1979) pp. 455-460.
- [FLP80] N. Francez, D.J. Lehmann et A. Pnueli.  
"A linear history semantics for distributed languages". 21st Symp on Found. of Comp. Sci, Proceedings IEEE (Octobre 1980), pp. 143-151.
- [FIS78] L. Flon et N. Suzuki.  
"Consistent and complete proof rules for the total correctness of parallel programs". Xerox PARC Report CSL -78-6 (Novembre 1978).
- [Gri79] D. Gries.  
"Is sometime ever better than always?". TOPLAS 1, 2 (Octobre 1979), pp. 258-265.
- [Gue79] P. Guerreiro.  
"Un modèle relationnel pour les programmes non-déterministes". Rapport de D.E.A., Univ. Scient. et Méd. de Grenoble (Juin 1979).

- [Har80] D. Harel.  
"On folk theorems". CACM 23, 7 (Juillet 1980), pp. 379-389.
- [HeS72] P. Henderson et R. Snowdon.  
"An experiment in structured programming", BIT 12 (1972),  
pp. 38-53.
- [HLP81] M. Hennessy, W. Li et G. Plotkin.  
"A first attempt at translating CSP into CCS". Second Conference  
on Distributed Computing Systems 1981, Proceedings IEEE  
(1981), pp. 105-115.
- [Hoa69] C.A.R. Hoare.  
"An axiomatic basis for computer programming". CACM 12, 10  
(Octobre 1969), pp. 576-580, 583.
- [Hoa74] C.A.R. Hoare.  
"Monitors : an operating system structuring concept".  
CACM 17, 10 (Octobre 1974), pp. 459-557.
- [Hoa78a] C.A.R. Hoare.  
"Some properties of predicate transformers".  
JACM, 25, 3 (Juillet 1978), pp. 461-480.
- [Hoa78b] C.A.R. Hoare.  
"Communicating sequential processes". CACM 21, 8  
(Août 1978), pp. 666-677.
- [Hoa79] C.A.R. Hoare.  
"A model for communicating sequential processes".  
Oxford Univ. Comp. Lab. (Juillet 1979).
- [HoL74] C.A.R. Hoare et P.E. Lauer.  
"Consistent and complementary formal theories of the  
semantics of programming languages". Acta Informatica 3  
(1974), pp. 135-153.

- [Hue77] G.Huet.  
"Confluent reductions : abstract properties and applicat  
to term rewriting systems". 18th IEEE Symposium on  
Found. Comp. Sci (1977), pp. 30-45.
- [JeW75] K. Jensen et N. Wirth.  
"Pascal User Manual and Report". Springer (1975).
- [Jor80] P. Jorrand.  
"Specification and analysis of communication protocols".  
IBM Research Report RJ 2853 (Février 1980).
- [Kel76] R.M. Keller.  
"Formal verification of parallel programs". CACM 19, 7  
(Juillet 1976), pp. 371-384.
- [Knu75] D.E. Knuth.  
"The Art of Computer Programming", Vol 1, "Fundamental  
Algorithms". Addison-Wesley (1975).
- [Lon77] R.L. London.  
"Perspectives on program verification", in R.T.Yeh (ed)  
"Current Trends in Programming Methodology".  
Vol II, chapitre 6. Prentice Hall (1977), pp. 151-172.
- [MaP79] Z. Manna et A. Pnueli.  
"The modal logic of programs". Proceedings ICALP 1979,  
LNCS 71, Springer (1979), pp. 385-410.
- [MaW78] Z. Manna et R. Waldinger.  
"Is "sometime" sometimes better than "always" ?"  
CACM 21, 2 (Février 1978), pp. 159-172.

- [Maz74] A. Mazurkiewicz.  
 "Proving properties of processes". *Algorithmy*, XI n° 19 (1974), pp. 5-22.
- [McM80] R.M. McKeag et P. Milligan.  
 "An experiment in parallel program design".  
*Software-Practise and Experience* 10 (1980) pp. 687-696.
- [Mi179] R. Milner.  
 "Flowgraphs and flow algebras". *JACM* 26, 4 (Octobre 1979), pp. 794-818.
- [Mi180] R. Milner.  
 "A Calculus of Communicating Systems". LNCS 92  
 Springer (1980).
- [MiM79] G. Milne et R. Milner.  
 "Concurrent processes and their syntax". *JACM* 26, 2 (Avril 1979), pp. 302-321.
- [MNV73] Z. Manna, S. Ness et J. Vuillemin.  
 "Inductive methods for proving properties of programs".  
*CACM* 16, 8 (Août 1973), pp. 491-502.
- [OwG76] S. Owicki et D. Gries.  
 "Verifying properties of parallel programs : an axiomatic approach". *CACM* 19, 5 (Mai 1976), pp. 279-285.
- [Que80] J.P. Queille.  
 "The CESAR system : an aided design and certification system for distributed applications". *IMAG*, RR 214, Grenoble (Septembre 1980). Aussi *Second Conference on Distributed Computing Systems 1981, Proceedings IEEE (1981)*, pp. 149-16

- [Roe76] W.P. de Roever.  
"Dijkstra's predicate transformer, non-determinism, recursion and termination". MFCS 1976 Proceedings, LNCS 45 Springer (1976) pp. 472-481.
- [San80] J.G. Sanderson.  
"A Relational Theory of Computing". LNCS 82, Springer (1980)
- [Sch79] G. Schmidt.  
"Investigating programs in terms of partial graphs". ICALP 1979 Proceedings, LNCS 71 Springer (1979), pp. 515-519.
- [Sif78] J. Sifakis.  
"Structural properties of Petri nets". MFCS 1978 Proceedings, LNCS 64, Springer (Septembre 1978), pp. 474-481.
- [Sif79a] J. Sifakis.  
"Le contrôle des systèmes asynchrones : concepts, propriétés, analyse statique". Thèse d'Etat, Univ. Scient. et Méd. de Grenoble (Juin 1979).
- [Sif79b] J. Sifakis.  
"A unified approach for studying the properties of transition systems". Rapport de recherche 179, IMAG, Grenoble (Décembre 1979), (à paraître dans TCS).
- [Sif80] J. Sifakis.  
"Deadlocks and livelocks in transition systems". MFCS 1980 Proceedings, LNCS 88, Springer (1980) pp. 587-600.
- [Tar41] A. Tarski.  
"On the calculus of relations". The Journal of Symbolic Logic 6, 3 (Septembre 1941), pp. 73-89

- [Tar55] A. Tarski.  
"A lattice-theoretical fixpoint theorem and its applications"  
Pacific Journal of Mathematics, 5 (1955), pp. 285-309.
- [Wan77] M. Wand.  
"A characterisation of weakest preconditions". JCSS 15  
(1977), pp. 209-212.
- [WeB79] J. Welsh et D.W. Bustard.  
"Pascal Plus - another language for modular multiprogramming"  
Software-Practise and Experience, 9 (1979), pp. 947-957.
- [Wir77] N. Wirth.  
"Modula : a language for modular multiprogramming".  
Software-Practise and Experience, 7 (1977), pp. 3-35.



Dernière page d'une thèse

---

VU

Grenoble, le 12 juin 1981

Le Président de la thèse

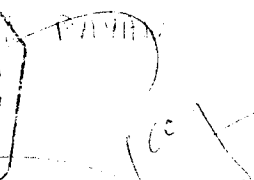


C. DELOBEL

Vu, et permis d'imprimer,

Grenoble, le 15 juin 1981

Le Président de l'Université Scientifique et Médicale



A handwritten signature in ink, appearing to be 'J. C. F.', written over the seal.