# Une perspective relationnelle de la programmation

Ali Mili

# THESE

*présentée à*

**l' Université Scientifique et Médicale de Grenoble**
*et à*
**l' Institut National Polytechnique de Grenoble**

*pour obtenir le grade de*
DOCTEUR ES SCIENCES
«Informatique»

*par*

Ali MILI

## UNE PERSPECTIVE RELATIONNELLE DE LA PROGRAMMATION.

**Thèse soutenue le 28 octobre 1985 devant la commission d'examen.**

| | |
|---|---|
| C. DELOBEL | Président |
| M. CHEILAN<br>M. DIAZ<br>E. GIRARD<br>G. SAUCIER<br>Y. CHIARAMELLA | Examinateurs |
| J.C. ABRIAL<br>W. HATCHER | Rapporteurs |

## UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

**Année universitaire 1982-1983**

**Président de l'Université : M. TANCHE**

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

**(RANG A)**

**SAUF ENSEIGNANTS EN MEDECINE ET PHARMACIE**

### PROFESSEURS DE 1ère CLASSE

| | |
|---|---|
| ARNAUD Paul | Chimie organique |
| ARVIEU Robert | Physique nucléaire I.S.N. |
| AUBERT Guy | Physique C.N.R.S. |
| AYANT Yves | Physique approfondie |
| BARBIER Marie-Jeanne | Electrochimie |
| BARBIER Jean-Claude | Physique expérimentale C.N.R.S. (labo de magnétisme) |
| BARJON Robert | Physique nucléaire I.S.N. |
| BARNOUD Fernand | Biosynthèse de la cellulose-Biologie |
| BARRA Jean-René | Statistiques - Mathématiques appliquées |
| BELORISKY Elie | Physique |
| BENZAKEN Claude (M.) | Mathématiques pures |
| BERNARD Alain | Mathématiques pures |
| BERTRANDIAS Françoise | Mathématiques pures |
| BERTRANDIAS Jean-Paul | Mathématiques pures |
| BILLET Jean | Géographie |
| BONNIER Jean-Marie | Chimie générale |
| BOUCHEZ Robert | Physique nucléaire I.S.N. |
| BRAVARD Yves | Géographie |
| CARLIER Georges | Biologie végétale |
| CAUQUIS Georges | Chimie organique |
| CHIBON Pierre | Biologie animale |
| COLIN DE VERDIERE Yves | Mathématiques pures |
| CRABBE Pierre (détaché) | C.E.R.M.O. |
| CYROT Michel | Physique du solide |
| DAUMAS Max | Géographie |
| DEBELMAS Jacques | Géologie générale |
| DEGRANGE Charles | Zoologie |
| DELOBEL Claude (M.) | M.I.A.G. Mathématiques appliquées |
| DEPORTES Charles | Chimie minérale |
| DESRE Pierre | Electrochimie |
| DOLIQUE Jean-Michel | Physique des plasmas |
| DUCROS Pierre | Cristallographie |
| FONTAINE Jean-Marc | Mathématiques pures |
| GAGNAIRE Didier | Chimie physique |

.../...

| | |
|---|---|
| GASTINEL Noël | Analyse numérique - Mathématiques appliquées |
| GERBER Robert | Mathématiques pures |
| GERMAIN Jean-Pierre | Mécanique |
| GIRAUD Pierre | Géologie |
| IDELMAN Simon | Physiologie animale |
| JANIN Bernard | Géographie |
| JOLY Jean-René | Mathématiques pures |
| JULLIEN Pierre | Mathématiques appliquées |
| KAHANE André (détaché DAFCO) | Physique |
| KAHANE Josette | Physique |
| KOSZUL Jean-Louis | Mathématiques pures |
| KRAKOWIAK Sacha | Mathématiques appliquées |
| KUPTA Yvon | Mathématiques pures |
| LACAZE Albert | Thermodynamique |
| LAJZEROWICZ Jeannine | Physique |
| LAJZEROWICZ Joseph | Physique |
| LAURENT Pierre | Mathématiques appliquées |
| DE LEIRIS Joël | Biologie |
| LLIBOUTRY Louis | Géophysique |
| LOISEAUX Jean-Marie | Sciences nucléaires I.S.N. |
| LOUP Jean | Géographie |
| MACHE Régis | Physiologie végétale |
| MAYNARD Roger | Physique du solide |
| MICHEL Robert | Minéralogie et pétrographie (géologie) |
| MOZIERES Philippe | Spectrométrie - Physique |
| OMONT Alain | Astrophysique |
| OZENDA Paul | Botanique (biologie végétale) |
| PAYAN Jean-Jacques (détaché) | Mathématiques pures |
| PEBAY PEYROULA Jean-Claude | Physique |
| PERRIAUX Jacques | Géologie |
| PERRIER Guy | Géophysique |
| PIERRARD Jean-Marie | Mécanique |
| RASSAT André | Chimie systématique |
| RENARD Michel | Thermodynamique |
| RICHARD Lucien | Biologie végétale |
| RINAUDO Marguerite | Chimie CERMAV |
| SENGEL Philippe | Biologie animale |
| SERGERAERT Francis | Mathématiques pures |
| SOUTIF Michel | Physique |
| VAILLANT François | Zoologie |
| VALENTIN Jacques | Physique nucléaire I.S.N. |
| VAN CUTSEN Bernard | Mathématiques appliquées |
| VAUQUOIS Bernard | Mathématiques appliquées |
| VIALON Pierre | Géologie |

**PROFESSEURS DE 2ème CLASSE**

| | |
|---|---|
| ADIBA Michel | Mathématiques pures |
| ARMAND Gilbert | Géographie |

.../...

| | |
|---|---|
| AURIAULT Jean-Louis | Mécanique |
| BEGUIN Claude (M.) | Chimie organique |
| BOEHLER Jean-Paul | Mécanique |
| BOITET Christian | Mathématiques appliquées |
| BORNAREL Jean | Physique |
| BRUN Gilbert | Biologie |
| CASTAING Bernard | Physique |
| CHARDON Michel | Géographie |
| COHENADDAD Jean-Pierre | Physique |
| DENEUVILLE Alain | Physique |
| DEPASSEL Roger | Mécanique des fluides |
| DOUCE Roland | Physiologie végétale |
| DUFRESNOY Alain | Mathématiques pures |
| GASPARD François | Physique |
| GAUTRON René | Chimie |
| GIDON Maurice | Géologie |
| GIGNOUX Claude (M.) | Sciences nucléaires I.S.N. |
| GUITTON Jacques | Chimie |
| HACQUES Gérard | Mathématiques appliquées |
| HERBIN Jacky | Géographie |
| HICTER Pierre | Chimie |
| JOSELEAU Jean-Paul | Biochimie |
| KERCKOVE Claude (M.) | Géologie |
| LE BRETON Alain | Mathématiques appliquées |
| LONGEQUEUE Nicole | Sciences nucléaires I.S.N. |
| LUCAS Robert | Physiques |
| LUNA Domingo | Mathématiques pures |
| MASCLE Georges | Géologie |
| NEMOZ Alain | Thermodynamique (CNRS - CRTBT) |
| OUDET Bruno | Mathématiques appliquées |
| PELMONT Jean | Biochimie |
| PERRIN Claude (M.) | Sciences nucléaires I.S.N. |
| PFISTER Jean-Claude (détaché) | Physique du solide |
| PIBOULE Michel | Géologie |
| PIERRE Jean-Louis | Chimie organique |
| RAYNAUD Hervé | Mathématiques appliquées |
| ROBERT Gilles | Mathématiques pures |
| ROBERT Jean-Bernard | Chimie physique |
| ROSSI André | Physiologie végétale |
| SAKAROVITCH Michel | Mathématiques appliquées |
| SARROT REYNAUD Jean | Géologie |
| SAXOD Raymond | Biologie animale |
| SOUTIF Jeanne | Physique |
| SCHOOL Pierre-Claude | Mathématiques appliquées |
| STUTZ Pierre | Mécanique |
| SUBRA Robert | Chimie |
| VIDAL Michel | Chimie organique |
| VIVIAN Robert | Géographie |

**INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE**

**Année universitaire 1982-1983**

**Président de l'Université : D. BLOCH**

**Vice-Président : René CARRE**
**Hervé CHERADAME**
**Marcel IVANES**

**PROFESSEURS DES UNIVERSITES :**

| | |
|---|---|
| ANCEAU François | E.N.S.I.M.A.G. |
| BARRAUD Alain | E.N.S.I.E.G. |
| BAUDELET Bernard | E.N.S.I.E.G. |
| BESSON Jean | E.N.S.E.E.G. |
| BLIMAN Samuel | E.N.S.E.R.G. |
| BLOCH Daniel | E.N.S.I.E.G. |
| BOIS Philippe | E.N.S.H.G. |
| BONNETAIN Lucien | E.N.S.E.E.G. |
| BONNIER Etienne | E.N.S.E.E.G. |
| BOUVARD Maurice | E.N.S.H.G. |
| BRISSONNEAU Pierre | E.N.S.I.E.G. |
| BUYLE BODIN Maurice | E.N.S.E.R.G. |
| CAVAIGNAC Jean-François | E.N.S.I.E.G. |
| CHARTIER Germain | E.N.S.I.E.G. |
| CHENEVIER Pierre | E.N.S.E.R.G. |
| CHERADAME Hervé | U.E.R.M.C.P.P. |
| CHERUY Arlette | E.N.S.I.E.G. |
| CHIAVERINA Jean | U.E.R.M.C.P.P. |
| COHEN Joseph | E.N.S.E.R.G. |
| COUMES André | E.N.S.E.R.G. |
| DURAND Francis | E.N.S.E.E.G. |
| DURAND Jean-Louis | E.N.S.I.E.G. |
| FELICI Noël | E.N.S.I.E.G. |
| FOULARD Claude | E.N.S.I.E.G. |
| GENTIL Pierre | E.N.S.E.R.G. |
| GUERIN Bernard | E.N.S.E.R.G. |
| GUYOT Pierre | E.N.S.E.E.G. |
| IVANES Marcel | E.N.S.I.E.G. |
| JAUSSAUD Pierre | E.N.S.I.E.G. |
| JOUBERT Jean-Claude | E.N.S.I.E.G. |
| JOURDAIN Geneviève | E.N.S.I.E.G. |
| LACOUME Jean-Louis | E.N.S.I.E.G. |
| LATOMBE Jean-Claude | E.N.S.I.M.A.G. |

.../...

| | |
|---|---|
| LESSIEUR Marcel | E.N.S.H.G. |
| LESPINARD Georges | E.N.S.H.G. |
| LONGEQUEUE Jean-Pierre | E.N.S.I.E.G. |
| MAZARE Guy | E.N.S.I.M.A.G. |
| MOREAU René | E.N.S.H.G. |
| MORET Roger | E.N.S.I.E.G. |
| MOSSIERE Jacques | E.N.S.I.M.A.G. |
| PARIAUD Jean-Charles | E.N.S.E.E.G. |
| PAUTHENET René | E.N.S.I.E.G. |
| PERRET René | E.N.S.I.E.G. |
| PERRET Robert | E.N.S.I.E.G. |
| PIAU Jean-Michel | E.N.S.H.G. |
| POLOUJADOFF Michel | E.N.S.I.E.G. |
| POUPOT Christian | E.N.S.E.R.G. |
| RAMEAU Jean-Jacques | E.N.S.E.E.G. |
| RENAUD Maurice | U.E.R.M.C.P.P. |
| ROBERT André | U.E.R.M.C.P.P. |
| ROBERT François | E.N.S.I.M.A.G. |
| SABONNADIERE Jean-Claude | E.N.S.I.E.G. |
| SAUCIER Gabrielle | E.N.S.I.M.A.G. |
| SCHLENKER Claire | E.N.S.I.E.G. |
| SCHLENKER Michel | E.N.S.I.E.G. |
| SERMET Pierre | E.N.S.E.R.G. |
| SILVY Jacques | U.E.R.M.C.P.P. |
| SOHM Jean-Claude | E.N.S.E.E.G. |
| SOUQUET Jean-Louis | E.N.S.E.E.G. |
| VEILLON Gérard | E.N.S.I.M.A.G. |
| ZADWORNY François | E.N.S.E.R.G. |

**PROFESSEURS ASSOCIES**

| | |
|---|---|
| BASTIN Georges | E.N.S.H.G. |
| BERRIL John | E.N.S.H.G. |
| CARREAU Pierre | E.N.S.H.G. |
| GANDINI Alessandro | U.E.R.M.C.P.P. |
| HAYASHI Hirashi | E.N.S.I.E.G. |

**PROFESSEURS UNIVERSITE DES SCIENCES SOCIALES (Grenoble II)**

BOLLIET Louis
Chatelin Françoise

**PROFESSEURS E.N.S. Mines de Saint-Etienne**

RIEU Jean
SOUSTELLE Michel

**CHERCHEURS DU C.N.R.S.**

| | |
|---|---|
| FRUCHART Robert | Directeur de Recherche |
| VACHAUD Georges | Directeur de Recherche |

.../...

| | |
|---|---|
| ALLIBERT Michel | Maître de Recherche |
| ANSARA Ibrahim | Maître de Recherche |
| ARMAND Michel | Maître de Recherche |
| BINDER Gilbert | |
| CARRE René | Maître de Recherche |
| DAVID René | Maître de Recherche |
| DEPORTES Jacques | |
| DRIOLE Jean | Maître de Recherche |
| GIGNOUX Damien | |
| GIVORD Dominique | |
| GUELIN Pierre | |
| HOPFINGER Emil | Maître de Recherche |
| JOUD Jean-Charles | Maître de Recherche |
| KAMARINOS Georges | Maître de Recherche |
| KLEITZ Michel | Maître de Recherche |
| LANDAU Ioan-Dore | Maître de Recherche |
| LASJAUNIAS J.C. | |
| MERMET Jean | Maître de Recherche |
| MUNIER Jacques | Maître de Recherche |
| PIAU Monique | |
| PORTESEIL Jean-Louis | |
| THOLENCE Jean-Louis | |
| VERDILLON André | |

**CHERCHEURS du MINISTERE de la RECHERCHE et de la TECHNOLOGIRE (Directeurs et Maîtres de Recherches, ENS Mines de St. Etienne)**

| | |
|---|---|
| LESBATS Pierre | Directeur de Recherche |
| BISCONDI Michel | Maître de Recherche |
| KOBYLANSKI André | Maître de Recherche |
| LE COZE Jean | Maître de Recherche |
| LALAUZE René | Maître de Recherche |
| LANCELOT Francis | Maître de Recherche |
| THEVENOT François | Maître de Recherche |
| TRAN MINH Canh | Maître de Recherche |

**PERSONNALITES HABILITEES à DIRIGER des TRAVAUX de RECHERCHE (Décision du Conseil Scientifique)**

| | |
|---|---|
| ALLIBERT Colette | E.N.S.E.E.G. |
| BERNARD Claude | E.N.S.E.E.G. |
| BONNET Rolland | E.N.S.E.E.G. |
| CAILLET Marcel | E.N.S.E.E.G. |
| CHATILLON Catherine | E.N.S.E.E.G. |
| CHATILLON Christian | E.N.S.E.E.G. |
| COULON Michel | E.N.S.E.E.G. |
| DIARD Jean-Paul | E.N.S.E.E.G. |
| EUSTAPOPOULOS Nicolas | E.N.S.E.E.G. |
| FOSTER Panayotis | E.N.S.E.E.G. |

.../...

| | |
|---|---|
| GALERIE Alain | E.N.S.E.E.G. |
| HAMMOU Abdelkader | E.N.S.E.E.G. |
| MALMEJAC Yves | E.N.S.E.E.G. (CENG) |
| MARTIN GARIN Régina | E.N.S.E.E.G. |
| NGUYEN TRUONG Bernadette | E.N.S.E.E.G. |
| RAVAINE Denis | E.N.S.E.E.G. |
| SAINFORT | E.N.S.E.E.G. (CENG) |
| SARRAZIN Pierre | E.N.S.E.E.G. |
| SIMON Jean-Paul | E.N.S.E.E.G. |
| TOUZAIN Philippe | E.N.S.E.E.G. |
| URBAIN Georges | E.N.S.E.E.G. (Laboratoire des ultra-réfractaires ODEILLON) |
| GUILHOT Bernard | E.N.S. Mines Saint Etienne |
| THOMAS Gérard | E.N.S. Mines Saint Etienne |
| DRIVER Julien | E.N.S. Mines Saint Etienne |
| BARIBAUD Michel | E.N.S.E.R.G. |
| BOREL Joseph | E.N.S.E.R.G. |
| CHOVET Alain | E.N.S.E.R.G. |
| CHEHIKIAN Alain | E.N.S.E.R.G. |
| DOLMAZON Jean-Marc | E.N.S.E.R.G. |
| HERAULT Jeanny | E.N.S.E.R.G. |
| MONLLOR Christian | E.N.S.E.R.G. |
| BORNARD Guy | E.N.S.I.E.G. |
| DESCHIZEAU Pierre | E.N.S.I.E.G. |
| GLANGEAUD François | E.N.S.I.E.G. |
| KOFMAN Walter | E.N.S.I.E.G. |
| LEJEUNE Gérard | E.N.S.I.E.G. |
| MAZUER Jean | E.N.S.I.E.G. |
| PERARD Jacques | E.N.S.I.E.G. |
| REINISCH Raymond | E.N.S.I.E.G. |
| ALEMANY Antoine | E.N.S.H.G. |
| BOIS Daniel | E.N.S.H.G. |
| DARVE Félix | E.N.S.H.G. |
| MICHEL Jean-Marie | E.N.S.H.G. |
| OBLED Charles | E.N.S.H.G. |
| ROWE Alain | E.N.S.H.G. |
| VAUCLIN Michel | E.N.S.H.G. |
| WACK Bernard | E.N.S.H.G. |
| BERT Didier | E.N.S.I.M.A.G. |
| CALMET Jacques | E.N.S.I.M.A.G. |
| COURTIN Jacques | E.N.S.I.M.A.G. |
| COURTOIS Bernard | E.N.S.I.M.A.G. |
| DELLA DORA Jean | E.N.S.I.M.A.G. |
| FONLUPT Jean | E.N.S.I.M.A.G. |
| SIFAKIS Joseph | E.N.S.I.M.A.G. |
| CHARUEL Robert | U.E.R.M.C.P.P. |
| CADET Jean | C.E.N.G. |
| COEURE Philippe | C.E.N.G. (LETI) |

.../...

| | |
|---|---|
| DELHAYE Jean-Marc | C.E.N.G. (STT) |
| DUPUY Michel | C.E.N.G. (LETI) |
| JOUVE Hubert | C.E.N.G. (LETI) |
| NICOLAU Yvan | C.E.N.G. (LETI) |
| NIFENECKER Hervé | C.E.N.G. |
| PERROUD Paul | C.E.N.G. |
| PEUZIN Jean-Claude | C.E.N.G. (LETI) |
| TAIEB Maurice | C.E.N.G. |
| VINCENDON Marc | C.E.N.G. |

**LABORATOIRES EXTERIEURS**

| | |
|---|---|
| DEMOULIN Eric | C.N.E.T. |
| DEVINE | C.N.E.T. (R.A.B.) |
| GERBER Roland | C.N.E.T. |
| MERCKEL Gérard | C.N.E.T. |
| PAULEAU Yves | C.N.E.T. |
| GAUBERT C. | I.N.S.A. Lyon |

Je tiens à exprimer toute ma reconnaissance à Madame Gabrièle SAUCIER, d'avoir bien voulu m'accueillir dans son équipe de recherche.

Je remercie Monsieur Jean Claude ABRIAL et Monsieur le Professeur William HATCHER qui m'ont fait l'honneur de bien vouloir être les rapporteurs de cette thèse, ainsi que Madame GAUDEL, qui a rapporté sur une version antérieure.

Je tiens à remercier :

- Le Professeur Claude DELOBEL, qui a accepté de Présider le Jury,
- Messieurs CHEILAN, DIAZ et GIRARD pour leur participation au jury.

Dédicace

Cet humble travail est dédié à la Tunisie, avec le souhait qu'elle soit sans cesse travailleuse et paisible.

H. M.

# Une Perspective Relationnelle de la Programmation

ECOLE NATIONALE SUPERIEURE DES MINES DE SAINT-ETIENNE

Directeur : Monsieur M. MERMET
Directeur des Etudes et de la formation : Monsieur J. LEVASSEUR
Directeur des recherches : Monsieur J. LEVY
Secrétaire Général : Mademoiselle M. CLERGUE

Professeurs de 1ère Catégorie

| | | |
|---|---|---|
| COINDE | Alexandre | Gestion |
| GOUX | Claude | Métallurgie |
| LEVY | Jacques | Métallurgie |
| LOWYS | Jean-Pierre | Physique |
| MATHON | Albert | Gestion |
| RIEU | Jean | Mécanique - Résistance des matériaux |
| SOUSTELLE | Michel | Chimie |
| FORMERY | Philippe | Mathématiques Appliquées |

Professeurs de 2ème catégorie

| | | |
|---|---|---|
| HABIB | Michel | Informatique |
| PERRIN | Michel | Géologie |
| VERCHERY | Georges | Matériaux |
| TOUCHARD | Bernard | Physique Industrielle |

Directeur de recherche

| | | |
|---|---|---|
| LESBATS | Pierre | Métallurgie |

Maîtres de recherche

| | | |
|---|---|---|
| BISCONDI | Michel | Métallurgie |
| DAVOINE | Philippe | Géologie |
| FOURDEUX | Angeline | Métallurgie |
| KOBYLANSKI | André | Métallurgie |
| LALAUZE | René | Chimie |
| LANCELOT | Francis | Chimie |
| LE COZE | Jean | Métallurgie |
| THEVENOT | François | Chimie |
| TRAN MINH | Canh | Chimie |

Personnalités habilitées à diriger des travaux de recherche

| | | |
|---|---|---|
| DRIVER | Julian | Métallurgie |
| GUILHOT | Bernard | Chimie |
| THOMAS | Gérard | Chimie |

Professeur à l'UER de Sciences de Saint-Etienne

| | | |
|---|---|---|
| VERGNAUD | Jean-Maurice | Chimie des Matériaux & chimie industrielle |

**********

# Introduction

"I do believe that the Calculus of Relations deserves more attention than it receives. For, aside from the fact that the concepts occurring in this calculus possess an objective beauty and are in these times almost indispensable in any scientific discussion, the calculus of relations has an intrinsic charm and beauty which make it a source of delight to all who become acquainted with it".
Alfred Tarski, 1941.

Cette thèse présente une perspective relationnelle à plusieurs aspects de la programmation. L'algèbre relationnelle de Tarski est utilisée pour formuler -et parfois résoudre- des problèmes pertinents à la programmation tels que: la spécification de programmes, l'analyse fonctionnelle de programmes, la vérification de programmes, la conception de programmes et le traitement d'erreurs dans les programmes.

La perspective que nous adoptons dans cette thèse est caractérisée par les prémisses suivantes: les programmes sont spécifiés à l'aide de relations; l'analyse fonctionnelle de programmes se fait par composition de fonctions; la conception de programmes se fait par décomposition de relations.

Notre thèse peut être interprétée comme une généralisation des travaux de H.D.Mills sur la programmation structurée. Elle partage son outil de base -le calcul relationnel- avec d'autres travaux récents de Hatcher, Parnas, Finance, Blikle, Sanderson et de Crespi-Reghizzi.

Sans contribuer de résultat fondamental nouveau, la perspective adoptée ici permet d'éclairer certains mécanismes de la programmation que d'autres approches -basées sur les prédicats ou les fonctions- n'exhibent pas. On s'efforcera au cours de cette thèse de prouver cette affirmation.

Le chapitre 1 présente des éléments de Mathématiques discrètes et d'algèbre relationnelle. En plus des notions usuelles de la théorie des ensembles et relations, ce chapitre inclut des notions nouvelles sur le déterminisme des relations ainsi que sur la quantité d'information portée par des relations.

Le chapitre 2 discute de la spécification de programmes, c'est à dire l'expression de ce qu'un programme doit faire, alors que le chapitre 3 discute discute de l'abstraction fonctionnelle de programmes, c'est à dire l'expression de ce que fait un programme. Ces deux notions sont confrontées dans le chapitre 5 pour définir la notion de cohérence.

Quant au chapitre 4, il traite des fonctions invariantes, et discute leur role dans l'analyse fonctionnelle de programmes itératifs.

Dans le chapitre 6, on étudie une approche relationnelle à la

programmation, qui formalise une décision de conception comme la résolution d'une équation relationnelle -ou encore (plus précisément) comme la résolution d'un problème d'optimisation dans l'algèbre relationnelle. Le chapitre 7 utilise l'approche du chapitre 6 afin de tenter de définir la notion évasive d'algorithme.

Le chapitre 8 prend une approche relationnelle à l'analyse, la vérification et la synthèse de programmes à assertions; ce chapitre tente de formaliser l'utilisation des assertions exécutables dans les programmes. Le chapitre 9 discute des possibilités de recouvrement avant dans les programmes; ce chapitre montre comment on peut utiliser la redondance naturelle des programmes pour des fins de tolérance aux pannes.

Enfin la conclusion fait une synthèse des principaux résultats de cette thèse et présente des directions de recherche futures.

# Chapitre 1

# Eléments d'Algèbre Relationnelle

Ce chapitre présente quelques définitions et conventions de notation relatives aux ensembles, relations et fonctions.

## 1. Ensembles

### 1.1. Définitions

Un ensemble S est une collection d'objets distincts identifiables appelés les éléments de S. Si s est un élément de S, on dit que s appartient à S, et on note s∈S. Si le nombre n d'éléments d'un ensemble S est fini, alors on dit que n est la cardinalité de S, que l'on note |S|. Il existe un seul ensemble de cardinalité zéro, l'ensemble vide, noté ∅. Quand un ensemble comporte un seul élément, on dit que c'est un singleton. Quand deux ensembles S et S' sont tels que tout élément de S' est élément de S, on dit que S' est un sous-ensemble de S, et on note S'⊆S.

Un prédicat sur l'ensemble S est une condition logique que l'on peut évaluer pour tout s dans S et qui ne peut prendre que les valeurs **vrai** ou **faux**. Si p est un prédicat sur S alors on dénote par S|p l'ensemble des éléments de S pour lesquels p est **vrai**. Inversement, si S'⊆S on dénote par ∈S' le prédicat vrai seulement pour les éléments de S'.

### 1.2. Opérations

Les notions d'union, intersection, différence et produit cartésien d'ensembles sont supposées connues; elles sont denotées par **U**, **O**, - et x respectivement. Une partition d'un ensemble S est la définition de sous-ensembles $S_1$, $S_2$, ... $S_n$ de S tels que l'union de tous ces sous-ensembles est S et leurs intersections deux à deux sont vides.

### 1.3. Représentation

Quand un ensemble est fini et de cardinalité faible, il est commun de le représenter par la liste de ses éléments. Par exemple,

feux-d'intersection = {vert, orange, rouge}.

Dans le cadre de ce travail, il nous arrivera souvent de définir nos ensembles commes des sous-ensembles de produits cartésiens d'ensembles connus. Par exemple,

trinomes = {(a,b,c) | a, b, et c sont réels et a≠0}.

Nous représentons cet ensemble de la façon suivante:

```
trinome = ens
    crt
        a, b, c: réel;
    sse
        a≠0
    fin.
```

Les mot-clés **ens**, **crt** et **sse** sont des abbréviations de (resp.) ensemble, produit cartésien et sous-ensemble. Soit t un

élément de l'ensemble trinomes; on dénotera (resp.) par a(t), b(t) et c(t) la première, deuxième et troisième composante de t. Par exemple, pour t=(1,0,-1), on a a(t)=1, b(t)=0 et c(t)=-1. Notez que l'on a toujours t=(a(t),b(t),c(t)). Quand le prédicat de la section sse est vrai, on peut omettre toute la section, ainsi que l'entête crt.

Les conventions de notation introduites ici pour l'exemple trinomes seront adoptées à travers cette thèse sans davantage de définition formelle.

Dans la suite de ce travail, nous supposerons connus les ensembles suivants:
- **naturel**, l'ensemble des entiers naturels (0 compris).
- **entier**, l'ensemble des entiers relatifs.
- **réel**, l'ensemble des nombres réels.
- **booléen**, l'ensemble des valeurs logiques **vrai** et **faux**.

## 1.4. Expressions

L'étude de la représentation des fonctions que nous aborderons dans la section 4.1 de ce chapitre nécessite que l'on définisse la notion d'**expression**; ceci fait l'objet de la présente section. Nous présenterons les expressions à base d'exemples plutôt que par des définition formelles, pour essentiellement deux raisons:
- premièrement, pour alléger les discussions.
- deuxièmement, les définitions données ici sont aisément transportables à d'autres ensembles ayant d'autres opérations.

Soit S l'ensemble **réel** et s le nom d'une variable dans S. Les chaînes s+3, 3*s-1, 3/s+1, 3/(s+2) sont des expressions sur S, en la variable s; une constante, telle que 5/3 peut aussi être considérée comme une expression sur S, bien que s n'y figure pas. Soit E l'expression 3/s+1; la valeur de l'expression E pour l'argument -disons- 2 est le réel obtenu en remplaçant s par 2 dans l'expression E et en effectuant les opérations prescrites; ainsi on trouve 3/2+1=5/2. On dénotera la valeur de l'expression E pour l'argument s par E(s). La valeur de l'expression E ne peut être calculée pour toutes les valeurs de l'argument s; par exemple, 3/0+1 ne peut être calculée. On appelle domaine de définition de E (abbréviation: **def**(E)) l'ensemble des arguments pour lesquels la valeur de l'expression E peut etre calculée; ainsi, pour E=**log**, on trouve **def**(E)={s| s>0}.

On définit trois opérations sur les expressions: la composition, l'alternation et le produit cartésien.

**Composition.** Soient E et E' deux expressions sur S. La composition de E par E' est l'expression (notée par E.E') obtenue en remplaçant chaque occurrence de la variable de E par l'expression E'(s). Par exemple, si

E(s)=3/s+1 et
E'(s)=s+3

alors E.E'(s) = 3/(s+3)+1 et E'.E(s)=3/s+4.

**Alternation.** Soient E et E' deux expressions sur S et soit p un

prédicat sur S. L'alternation de E et E' par rapport à p est l'expression sur S représentée par **sas**(p,E,E') dont la valeur pour tout argument s est définie de la façon suivante: Si p est vraie alors E(s) sinon E'(s).

**Produit Cartésien.** Soit un ensemble S=S1xS2 et soient E1 et E2 des expressions sur S à valeurs dans (resp.) S1 et S2. Le produit cartésien de E1 par E2 est l'expression sur S représentée par (E1,E2) dont la valeur pour l'argument s est (E1(s),E2(s)).

## 2. Relations

Dans cette section on étudiera des relations de multiplicité quelconque, c'est à dire des relations liant un nombre quelconque de variables; dans la section 3 nous étudierons les relations binaires, qui ne lient que deux variables entre elles.

### 2.1. Définitions

Soit $A_1$, $A_2$, ... $A_n$ une séquence ordonée d'ensembles. Une relation n-aire sur ($A_1$, $A_2$, ... $A_n$) est un sous-ensemble du produit cartésien $A_1xA_2x$ ... $A_n$.

**Exemple**
On prend $A_i$ = **réel** pour i =1, 2, 3. et
$R = \{(a,b,c) \mid a\neq 0 \ \& \ b^2-4ac=0\}$.
Cette relation lie tous les triplets (a,b,c) qui constituent un trinôme ayant une racine double.

### 2.2. Représentation

Vu qu'une relation est le sous-ensemble d'un produit catésien, il est naturel de penser à la représenter par la syntaxe que nous avons déjà prévue pour la représentation de sous-ensembles de produits catésiens, à savoir

```
ens
crt
    (*déclarations de variables
    selon la syntaxe de Pascal*)
sse
    (*prédicat définissant le
    sous-ensemble du produit*)
fin.
```

Il peut être génant pour le lecteur que la même représentation peut être interprétée de deux façons si distinctes (comme un ensemble ou comme une relation). En fait ces deux interprétations reflètent si oui ou non on veut mettre en évidence la structure catésienne de la représentation. Ainsi, par exemple, la représentation

```
ens
crt
    a, b, c: réel;
sse
    a≠0 & b²-4ac=0
fin
```

peut être interprétée soit:

- en ignorant sa structure cartésienne, comme un ensemble (de trinômes) dont les éléments se trouvent être représentés par des triplets de nombres réels vérifiant une certaine condition.
- en considérant sa structure cartésienne comme essentielle, comme une relation entre trois variables a, b et c, incluant les valeurs de ces variables qui vérifient la condition $a \neq 0$ & $b^2-4ac=0$.

Donc nous utiliserons la notation ci-haut et l'interprèterons tantôt comme un ensemble, tantôt comme une relation.

## 2.3. Opérations

Outre les opérations ensemblistes d'union, intersection et différence, nous définissons les opérations suivantes sur les relations: Produit cartésien, projection et jointure. Ces opérations sont amplement discutées dans les manuels de bases de données (par exemple, [DATE81]); on les dicutera brièvement ici. Ces opérations nous seront utiles dans le traitement des appels de procédure.

**Produit Cartésien.** Le produit cartésien de deux relations R et R' est la relation obtenue en concaténant les éléments de R avec les éléments de R'. Formellement, si R est une relation sur $A_1$, $A_2$, ... $A_n$ et R' une relation sur $B_1$, $B_2$, ... $B_m$ alors le produit cartésien de R par R' est la relation notée RxR' et définie comme étant l'ensemble des éléments de la forme ($a_1$, $a_2$, ... $a_n$, $b_1$, $b_2$, ... $b_m$) tels que ($a_1$, $a_2$, ... $a_n$)$\in$R et ($b_1$, $b_2$, ... $b_m$)$\in$R'.

**Exemple**

```
R = ens
    crt
       a, b, c:  réel;
    sse
       a≠0 & b²-4ac>0
    fin.
R'= ens
    crt
       x', x":  réel;
    sse
       x'<x"
    fin.
```

La relation RxR' est alors

```
RxR' =
    ens
    crt
       a, b, c:  réel;
       x', x":  réel;
    sse
       a≠0 & b²-4ac>0 & x'<x"
    fin.
```

Un sous-ensemble intéressant de RxR' serait l'ensemble des quintuplets (a,b,c,x',x") tels que x' et x" sont les racines du trinôme (a,b,c).

**Sélection.** Quand on définit un sous-ensemble d'une relation R, on dit que l'on a effectué une sélection sur la relation R. Par exemple, la

relation R' (ci dessous) est obtenue de la relation R (ci-dessous) via l'opération de sélection.

```
R = ens
crt
   a, b, c: réel;
sse
   a≠0 & b²-4ac>0
fin.
R' = ens
crt
   a, b, c: réel;
sse
   |a|>u & b²-4ac>u
fin,
```

pour un certain u plus grand que zéro.

**Projection.** Soit R une relation sur $(A_1, A_2, \ldots A_n)$; la projection de R sur, par exemple, $(A_1, A_2, \ldots A_k)$, pour $k<n$ est l'ensemble des k-uplets $(a_1, a_2, \ldots a_k)$ tels qu'il existe un (n-k)-uplet $(a_{k+1}, a_{k+2}, \ldots a_n)$ vérifiant $(a_1, a_2, \ldots a_n) \in R$. Donc si

```
R = ens
crt
   a1: A1;
   a2: A2;
   ...  ...  ...
   an: An;
sse
   p(a1, a2, ... an)
fin
```

alors la projection de R sur $(A_1, A_2, \ldots A_k)$ est

```
R' = ens
crt
   a1: A1;
   a2: A2;
   ...  ...  ...
   ak: Ak;
sse
   ∃(ak+1, ak+2, ... an): (a1, a2, ... an)∈R
fin.
```

**Jointure.** Soient R une relation sur $(A_1, A_2, \ldots A_k)$ et R' une relation sur $(A_1, A_{i+1}, \ldots A_n)$ où $i \leq k$; la jointure des relations R et R' sur les composantes $(A_1, A_{i+1}, \ldots A_k)$ est l'ensemble des n-uplets $(a_1, a_2, \ldots a_n)$ tels que $(a_1, a_2, \ldots a_k) \in R$ et $(a_1, a_{i+1}, \ldots a_n) \in R'$. Par exemple, si

```
R = ens
crt
   a, b: entier; (A1, A2)
sse
   a≤b
fin
```

et

```
R' = ens
crt
   b, c: entier; (A2, A3)
```

```
      sse
         b≤c
      fin
```

alors la jointure des relations R et R' sur la composante $A_2$ est la relation

```
      R" = ens
      crt
         a, b, c: entier;
      sse
         a≤b≤c
      fin.
```

## 3. Relations Binaires

Dans la section précédente nous nous sommes intéressés aux relations à multiplicité quelconque; celles-ci incluent les relations unaires (des ensembles), les relations binaires, les relations tertiaires, etc... Dans cette section nous nous intéressons uniquement aux relations binaires; de plus, nous nous intéresserons aux relations qui impliquent deux fois le même ensemble ($A_1=A_2$).

### 3.1. Définition

Une relation binaire R sur un ensemble S est un sous-ensemble de SxS. Soit (s,s') un élément de la relation binaire R; on dit que s' est une image de s par R et que s est un argument de s' par R. Dans cette thèse, nous utiliserons les relations binaires bien plus souvent que nous n'utiliserons les relations d'autres multiplicités; pour cette raison, quand le contexte ne prête à aucune confusion, nous utiliserons le terme relation pour désigner une relation binaire. Le domaine de la relation R est le sous-ensemble de S défini par

$$dom(R) = \{s \mid \exists s' : (s,s') \in R\}.$$

Le co-domaine de R est le sous-ensemble de S défini par

$$cod(R) = \{s' \mid \exists s : (s,s') \in R\}.$$

Parmi les relations spéciales sur un ensemble S, on cite: la relation universelle U=SxS, la relation identité $I=\{(s,s') \mid s'=s\}$, la relation vide $\emptyset$ et la relation diversité $V=\{(s,s') \mid s \neq s'\}$. Soit t un prédicat sur S; on note par I(t) la relation $\{(s,s') \mid s'=s \ \& \ t(s)\}$. De même, si S' est un sous-ensemble de S on notera -par abus de notation- par I(S') la relation $\{(s,s') \mid s'=s \ \& \ s \in S'\}$.

Une relationnelle sur S est une relation sur l'ensemble des relations binaires sur S.

### 3.2. Opérations

Les opérations ensemblistes d'union, intersection et différence s'appliquent aux relations et présentent un certain intérêt. En plus, nous définissons les opérations suivantes sur les relations.

**Complément.** Le complément de la relation R sur S est la relation notée $R^-$ et définie par U-R.

**Inversion.** L'inverse de la relation R est la relation notée $R^\smile$ et

définie par

$$\{(s,s') \mid (s',s) \in R\}.$$

**Produit Relatif.** Soient R et R' deux relations sur S. Le produit relatif de R par R' est la relation sur S notée par R*R' et définie par

$$\{(s,s') \mid \exists s'': (s,s'') \in R \ \& \ (s'',s') \in R'\}.$$

Il est simple de vérifier que $(R*R')^{\wedge}=R'^{\wedge}*R^{\wedge}$.

**Puissance relative.** Soit R une relation sur S. La i-ème puissance relative de R, pour $i \geq 0$, est la relation notée $R^i$ et définie par

Si i=0 alors I
sinon $R^{i-1}*R$.

Notez que pour tout $i \geq 1$, $dom(R^i)$ est un sous-ensemble de dom(R).

**Fermeture Transitive.** Soit R une relation sur S. La fermeture transitive de R est la relation notée $R^+$ et définie par

$$R^+ = \{(s,s') \mid \exists i \geq 1: (s,s') \in R^i\}.$$

En d'autres termes, $R^+$ est l'union des puissances relatives positives de R. Notez que l'on a $dom(R^+)=dom(R)$ puisque $dom(R^i) \subseteq dom(R)$ pour tout $i \geq 1$. Si $T=R^+$, on dit que R est un noyau transitif de T. Si aucun sous-ensemble de R n'est un noyau transitif de T, on dit que R est un noyau transitif irréductible de T (traduit de [KAND80]).

**Fermeture Transitive Rélexive.** La fermeture transitive rélexive de la relation R sur S est la relation dénotée par $R^*$ et définie par

$$R^* = I \cup R^+.$$

**Pré-restriction.** Soit R une relation sur S et t un prédicat sur S. La pré-restriction de R à t est le sous-ensemble de R défini par

$$\{(s,s') \mid t(s) \ \& \ (s,s') \in R\}.$$

La pré-restriction de R à t n'est autre, en fait, que I(t)*R.

**Post-restriction.** La post-restriction de R à t est la relation contenant toutes les paires (s,s') de R telles que t(s') est vrai; elle n'est autre que R*I(t). Pour toute relation R et prédicat t, on a

$$I(t)*R \cup I(\sim t)*R = R,$$
$$R*I(t) \cup R*I(\sim t) = R.$$

Ces deux formules sont dites -respectivement- la pré-projection de R sur t et la post-projection de R sur t (ou sur ~t).

**Pré-application.** Soit R une relation sur S et s un élément de S. La pré-application de s à R est l'ensemble noté s.R et défini par

$$s.R = \{s' \mid (s,s') \in R\}.$$

Par extension à cette notation, si T est un sous-ensemble de S, on notera par T.R l'union pour tout t dans T de t.R.

**Post-application.** Soit R une relation sur S et s un élément de S. La post-application de s à R est l'ensemble noté R.s et égal à $s.R^{\wedge}$. Par extension à cette notation, si T est un sous-ensemble de S, on notera par R.T l'ensemble $T.R^{\wedge}$.

Il est utile de faire des remarques concernant les opérateurs de pré- et post- application.

- $s.R = R^{\wedge}.s$, par définition.

- $s.(R*R') = (s.R).R'$.
  Cette pseudo-associativité peut être prouvée de la façon suivante: Soit s' un élément de $s.(R*R')$; alors $(s,s')\in(R*R')$, donc il existe s" tel que $(s,s")\in R$ et $(s",s')\in R'$; s' appartient à s".R' puisque $s"\in s.R$, s' appartient à (s.R).R'. D'autre part, soit s' un élément de (s.R).R'; il existe un élément s" dans s.R tel que $(s",s')\in R'$; vu que $s"\in s.R$, $(s,s")\in R$; de $(s,s")\in R$ et $(s",s')\in R'$ on déduit $(s,s')\in R*R'$, soit $s'\in s.(R*R')$.
- $(R*R').s = R.(R'.s)$.
  La preuve en est analogue à celle de la formule précédente.
- $dom(R)=R.S$.
  Par définition.
- $cod(R)=S.R$.
  Par définition.

Désormais, on prendra les conventions notationnelles suivantes afin d'alléger les expressions relationnelles. L'inversion, la complémentation, la puissance relative et les fermetures transitives (simple et réflexive) ont la plus haute priorité (ils sont appliqués en premier); ils sont suivis par le produit relatif; ensuite viennent les opérations ensemblistes (union, intersection, différence); enfin l'opération d'application (.).

## 3.3. Propriétés des Relations

### 3.3.1. Propriétés d'Equivalence

Soit R une relation sur S. R est dite réflexive si et seulement si $I\subseteq R$. R est dite symétrique si et seulement si $R=R^{\wedge}$. R est dite transitive si et seulement si $R*R\subseteq R$. Relation R est dite une équivalence si et seulement si elle est réflexive, symétrique et transitive. La classe d'équivalence d'un élément s de S modulo une équivalence R est l'ensemble de tous les éléments s' de S tels que $(s,s')\in R$; les classe d'équivalence modulo R forment une partition de S.

### 3.3.2. Propriétés d'Ordre

Soit R une relation sur S. R est dite antisymétrique si et seulement si $R\cap R^{\wedge}\subseteq I$. R est dite asymétrique si et seulement si $R\cap R^{\wedge}=\emptyset$. R est dite connexe si et seulement si $V\subseteq R\cup R^{\wedge}$. R est dite fortement connexe si et seulement si $U=R\cup R^{\wedge}$.

R est dite un ordre partiel si et seulement si elle est antisymétrique et transitive. R est dite un ordre simple (ou un ordre total) si et seulement si elle est un ordre partiel fortement connexe. R est dîte un ordre partiel strict si et seulement si elle est transitive et asymétrique. R est dite un ordre simple (ou total) strict si et seulement si elle est asymétrique, transitive et connexe.

R est dite un ordre bien fondé si et seulement si elle est transitive et que pour tout s dans S il existe un entier positif k (fonction de s) tel que $s.R^{k}=\emptyset$. En d'autres termes, pour tout s il n'existe pas de suite décroissante commençant à s (par R) dont la

longueur est supérieure à k. Les ordres bien fondés interviennent dans l'analyse et la conception de programmes itératifs.

### 3.3.3. Mesurer le rapport sorties/entrées d'une relation

Intuitivement, nous considérons qu'une relation est d'autant moins déterministe qu'elle a un rapport plus grand de sorties pour chaque entrée. On ne saurait mesurer le déterminisme d'une relation par la simple moyenne de son nombre de sorties pour chaque entrée, et ce pour deux raisons: premièrement, ce nombre peut ne pas être défini -dans le cas où la relation n'est pas finie; deuxièmement, une simple moyenne est une information bien trop abstraite pour servir de base à des comparaisons. Pour cela, on se tourne vers une formulation ensembliste.

Soit une relation R sur S. On appelle co-noyau de R la relation $R^{-1};R$. Notez que $R^{-1};R$ est symétrique et que $I(cod(R)) \subseteq R^{-1};R$. Une paire $(s,s') \in R^{-1};R$ si et seulement si il existe s" tel que $(s",s) \in R$ et $(s",s') \in R$; en d'autres termes, $(s,s') \in R^{-1};R$ si et seulement si s et s' sont les images par R d'un argument commun. Une relation est d'autant plus déterministe qu'elle a peu d'images pour chaque argument. Ceci s'écrit plus formellement: R est plus-déterministe que R' si et seulement si $R^{-1};R \subseteq R'^{-1};R'$. Notez qu'il découle de cette définition que $cod(R) \subseteq cod(R')$. La preuve en est la suivante: Soit s un élélment de cod(R); alors il existe s' tel que $(s',s) \in R$ et $(s,s') \in R^{-1}$; alors $(s,s) \in R^{-1};R$, d'où l'on déduit que $(s,s) \in R'^{-1};R'$; donc il existe s" tel que $(s,s") \in R'^{-1}$ et $(s",s) \in R'$; on en conclut que $s \in cod(R)$.

**Exemples**
Dans le but de faciliter l'assimilation des exemples donnés ci-dessous, on représentera les arguments par des caractères alphabetiques et les images par des caractères numériques. De plus, dans un premier temps, on comparera des relations ayant le même co-domaine.

$R = \{(a,0),(b,1),(c,2)\}$,
$R' = \{(a,0),(a,1),(b,1),(b,2),(c,2),(c,0)\}$.
$R^{-1};R = \{(0,0),(1,1),(2,2)\}$.
$R'^{-1};R' = \{(0,0),(0,1),(1,0),(1,1),(0,2),(2,0),(1,2),(2,1),(2,2)\}$
$= \{0,1,2\} \times \{0,1,2\}$.

On considère maintenant des relations ayant des co-domaines distincts.
$R = \{(a,0),(b,1),(c,1)\}$,
$R' = \{(a,0),(b,1),(b,2),(c,2),(c,3)\}$.
$R^{-1};R = \{(0,0),(1,1)\}$.
$R'^{-1};R' = \{(0,0),(1,1),(1,2),(2,1),(2,2),(2,3),(3,2),(3,3)\}$. []

Bien que l'on utilise le terme plus déterministe dans la définition ci-dessus, la relationnelle plus-déterministe n'est pas un ordre strict: elle a la connotation plus-déterministe-ou-aussi-déterministe plutôt que la connotation strictement-plus-déterministe.

Une relation R sur S est dite déterministe si et seulement si elle est plus-déterministe que I. En d'autres termes, R est déterministe si et seulement si $R^{-1};R \subseteq I$. Si R est déterministe, on dit alors que R est une fonction.

**Exemples**

$R = \{(a,0),(b,1),(c,1)\}$,
$R^{-1}*R = \{(0,0),(1,1)\} \subseteq I$.
R est une fonction.
$R' = \{(a,0),(a,1),(b,1),(c,1)\}$,
$R'^{-1}*R' = \{(0,0),(0,1),(1,0),(1,1)\} \not\subseteq I$.
R n'est pas une fonction. []

Pour établir des comparaisons entre relations, la relationnelle "plus-déterministe" ne prête pas attention aux paires individuelles (s,s') des relations concernées; plutôt, elle prête attention à la façon dont les relations concernées répartissent les images (s') en fonction de leurs arguments (s); si bien qu'elle (la relationnelle) ne serait pas en mesure de distinguer, par exemple, la relation

$\{(a,0),(b,1),(c,2)\}$

de la relation

$\{(b,0),(a,1),(a,2)\}$.

Dans la section suivante, on introduit une relationnelle qui prête davantage attention aux paires individuelles des relations concernées.

### 3.3.4. Mesurer la Quantité d'Information d'une Relation

Soit à observer un processus déterministe d'entrée sortie, dont on suppose que les sorties ne peuvent être observées avec précision. Deux agents A et B observent le processus pendant un certain temps et donnent les rapports suivants:

$Ra = \{(a,0),(a,1),(b,3),(c,4),(c,5),(d,2)\}$.
$Rb = \{(a,0),(a,1),(a,2),(b,0),(b,1),(b,2),(b,3)\}$.

En d'autres termes, l'agent A sait que pour l'entrée a, le processus observé retourne 0 ou 1, pour b il retourne 3, pour c il retourne 4 ou 5 et pour d il retourne 2. Par ailleurs, l'agent B sait que pour l'entrée a le processus retourne 0, 1 ou 2, et pour l'entrée b le processus retourne la sortie 0, 1, 2 ou 3. Sachant que les deux sont corrects dans leurs observations, lequel a recueilli plus d'information sur le processus observé? Il s'agit de A, bien entendu. En effet, il a observé plus d'entrées que B, et, pour les entrées que B aussi a observées, il est plus précis dans son estimation de la sortie.

Soient R et R' deux relations sur S. On dit que R est <u>plus définie</u> que R' si et seulement si

$dom(R') \subseteq dom(R)$,
$\forall s \in dom(R')$, $s.R \subseteq s.R'$.

La relationnelle "plus défini" apparaît très souvent dans notre étude de la programmation.

Si R et R' ont le même domaine alors R est plus définie que R' si et seulement si $R \subseteq R'$. Par exemple,

$R = \{(a,0),(b,1)\}$ est plus définie que
$R' = \{(a,0),(a,1),(b,1)(b,2)\}$.

Si f et f' sont des fonctions, alors f est plus définie que f' si et seulement si $f' \subseteq f$. Par exemple, $\{(a,0),(b,1),(c,2)\}$ est

plus-définie que {(a,0),(b,1)}. Le terme plus-définie est inspiré de [MANN74] (chapitre 5) où f' est dite moins-définie (less-defined) que f si et seulement si elle en est un sous-ensemble. La définition donnée ici pour les relations généralise celle de Manna (1974) pour les fonctions.

Si l'on considère la deuxième clause de la définition de la relationnelle "plus-défini",

$$\forall s \in dom(R'),\ s.R \subseteq s.R',$$

on réalise qu'elle a une signification voisine de la définition de la relationnelle "plus-déterministe",

$$R^{*}R \subseteq R'^{*}R'.$$

En effet, les deux définitions tendent à "favoriser" (comme plus-définie ou plus-déterministe) les relations qui ont moins d'images par argument. Le lien exact qui existe entre ces deux définitions est éclairé par la proposition suivante.

**Proposition 1.** Soient R et R' deux relations sur un ensemble S telles que R est plus-définie que R' et que dom(R)=dom(R'). Alors R est plus-déterministe que R'.

**Preuve.** Puisque R est plus définie que R' et que dom(R)=dom(R'), R est une sous-ensemble de R'. Il en découle aisément que R^*R est un sous-ensemble de R'^*R'. CQFD

**Exemple**
R = {(a,0),(b,1),(b,2)},
R'= {(a,0),(a,1),(b,1),(b,2)}.
a.R = {0}, b.R = {1,2}
a.R'= {0,1}, b.R'= {1,2}.
R^*R = {(0,0),(1,1),(2,2),(1,2),(2,1)},
R'^*R' = {(0,0),(1,1),(1,0),(0,1),(1,2),(2,1),(2,2)}. []

Il est bien clair que R peut être plus-déterministe que R' sans pour autant être plus définie. L'exemple ci-dessous le prouve.

**Exemple**
R = {(a,0),(b,1),(c,2)},
R'= {(a,1),(b,2),(c,0),(c,2)}.
R^*R = {(0,0),(1,1),(2,2)}.
R'^*R' = {(0,0),(1,1),(2,2),(0,2),(2,0)}.
R est plus-déterministe que R'. Par ailleurs, vu que a.R={0} n'est pas inclus dans a.R'={1}, on ne peut conclure que R est plus définie que R'. []

Pour clore cette section, rappelons que la relationnelle plus-déterministe compare le rapport (images/argument) de relations alors que la relationnelle plus-défini compare la quantité d'information portée par des relations. La relationnelle plus-défini nous servira dans notre étude de l'analyse et la conception de programmes alors que la relationnelle plus-déterministe nous servira dans notre étude du recouvrement.

## 3.4. Calcul Relationnel

Dans cette section, nous regroupons des équations relationnelles qui nous serviront dans la suite de cette thèse. L'objet de cet exposé n'est pas de faire un développement axiomatique du calcul relationnel; si bien que la liste donnée ci-dessous n'a aucune prétention de complétude ou de minimalité. L'arrangement de cette liste est inspiré de [GRIE81].

On considère l'ensemble des relations sur un ensemble S. Les noms de relations sont R, R' et R"; les relations constantes sont -rappelons le- U, I, V et 0; les opérateurs relationnels sont U, O, -, * et ^.

**Lois Commutatives**

C1. $R \cup R' = R' \cup R$.
C2. $R \cap R' = R' \cap R$.

**Lois Associatives**

A1. $(R \cup R') \cup R'' = R \cup (R' \cup R'')$.
A2. $(R \cap R') \cap R'' = R \cap (R' \cap R'')$.
A3. $(R * R') * R'' = R * (R' * R'')$.

**Lois Distributives**

D1. $R * (R' \cap R'') = R*R' \cap R*R''$.
D2. $R * (R' \cup R'') \subseteq R*R' \cup R*R''$.
D3. $R \cap (R' \cup R'') = (R \cap R') \cup (R \cap R'')$.
D4. $R \cup (R' \cap R'') = (R \cup R') \cap (R \cup R'')$.

**Lois d'Identité**

I1. $R \cap 0 = R$.
I2. $R \cup U = R$.
I3. $R * I = R$.

**Lois de l'Inverse**

V1. $R^{\wedge\wedge} = R$.
V2. $(R*R')^{\wedge} = R'^{\wedge} * R^{\wedge}$.
V3. $I^{\wedge} = I$.
V4. $U^{\wedge} = U$.
V5. $0^{\wedge} = 0$.

**Lois d'Absorption**

S1. $R \cup 0 = 0$.
S2. $R \cap U = U$.
S3. $R * 0 = 0$.

**Lois de Monotonicité**

M1. $R \subseteq R' \Rightarrow R*R'' \subseteq R'*R''$.
M2. $R \subseteq R' \Rightarrow R''*R \subseteq R''*R'$.
M3. $R \subseteq R' \Rightarrow R^{\wedge} \subseteq R'^{\wedge}$.

**Lois de Fermeture Transitive**

T1. $(R^{+})^2 \subseteq R^{+}$.
T2. $I \cup (R^{*})^2 \subseteq R^{*}$.
T3. $R*R \subseteq R \Rightarrow R^{+}=R \ \&\ R^{*}=I \cup R$.

T4. $IUR*R\subseteq R \Rightarrow R^*=R$.

**Lois de Restriction**

R1. $I(dom(R))*R = R*I(cod(R)) = R$.
R2. $dom(R)\subseteq S' \Rightarrow I(S')*R=R$.
R3. $cod(R)\subseteq S' \Rightarrow R*I(S')=R$.
R4. $dom(R)\cap S'=\emptyset \Rightarrow I(S')*R=\emptyset$.

## 4. Fonctions

### 4.1. Définition et Représentation

Une fonction est -rappelons le- une relation déterministe, soit une relation telle que $f^{-1}*f\subseteq I$. Il est facile de se convaincre que pour une fonction f sur S et pour un élément s de S, s.f est un singleton. Pour s'en convaincre, on considèrera deux éléments s' et s" de s.f et on prouvera que s'=s": de $s'\in s.f$ et $s"\in s.f$ on déduit $(s',s)\in f^{-1}$ et $(s,s")\in f$, d'où $(s',s")\in f^{-1}*f$; puisque $f^{-1}*f\subseteq I$, on a s'=s".

Par abus de notation, nous utiliserons la notation s.f pour désigner, non pas le singleton contenant l'image par f de s, mais aussi l'image elle-même; donc s.f représentera tantôt un ensemble, tantôt un élément. Dans ce dernier cas, s.f représente ce qu'il est usuel de noter par f(s).

La fonction f sur l'ensemble **naturel** définie par

$$f = \{(1,2),(2,3),(3,4),(4,5)\}$$

ne saurait être représentée par l'expression E(s)=s+1 car cette expression ne contient pas l'information que dom(f)={1,2,3,4}. Donc nous représenterons cette fonction par la paire $[1\leq s\leq 5,\ s+1]$. Généralement, nous représenterons toute fonction f sur S par la paire [p,E] où p est un prédicat défini pour tout élément de S et E une expression sur S dont le domaine de définition inclut S|p. Cette représentation est dite la <u>pE-formule</u> de la fonction f; sa valeur sémantique est la suivante:

$$[p,E] = \{(s,s') \mid p(s) \ \&\ s'=E(s)\}.$$

### 4.2. Opérations sur les Fonctions

Parmi les opérations relationnelles présentées dans la section 2.2, trois opérations sont d'intérêt pour les fonctions: Il s'agit de l'union, intersection et produit relatif. Nous les présentons ici en utilisant la notation pE.

**Union.** L'union de deux fonctions f=[p,E] et f'=[p',E'] n'est une fonction que si la condition suivante (C) est vérifiée:

$$(\forall s,\ p(s) \ \&\ p'(s) \Rightarrow E(s)=E'(s)).$$

Cette condition signifie: Pour tout s appartenant aux domaines de f et f', les expressions calculées par f et f' pour l'argument s sont identiques. Quand la condition (C) est satisfaite, l'union de f et f' s'écrit:

$$[p,E] \cup [p',E'] = [p \vee p',\ sas(p,E,E')]. \qquad (U)$$

Il est facile de se convaincre que les expressions sas(p,E,E') et

sas(p',E',E) sont interchangeables dans l'équation (U), si la condition (C) est vérifiée.

**Exemple**
S = **entier**;
f = [-5≤s≤2, 3*s²];
f'= [0≤s≤8, s³+2*s];
On vérifie la condition (C):
(C)
<=> (-5≤s≤2 & 0≤s≤8 => 3*s²=s³+2*s)
<=> (0≤s≤2 => 3*s²=s³+2*s).
Le lecteur se convaincra aisément de la validité de l'équation 3*s² = s³+2*s pour toutes les valeurs de s entre 0 et 2, en l'occurrence 0, 1, et 2. L'union de f et f' est alors
[-5≤s≤8, sas(-5≤s≤2, 3*s², s³+2*s)].

**Intersection.** L'intersection de deux fonctions est toujours une fonction. La formule de l'intersection de fonctions est la suivante:
[p,E] ∩ [p',E'] = [p & P' & E(s)=E'(s), E]. (I)
Pour qu'un élément s de S appartienne au domaine de l'intersection, il faut qu'il appartienne au domaine de [p,E] (d'où p(s)) et au domaine de [p',E'] (d'où p'(s)) et que les expressions E et E' donnent la même valeur pour s (d'où E(s)=E'(s)).

**Exemple**
On calcule l'intersection des fonctions f et f' définies sur S=**naturel** par
f = [0≤s≤12, 3*s²],
f'= [0≤s≤5, s³+2*s].
f∩f'
= [0≤s≤12 & 0≤s≤5 & 3*s²=s³+2*s, 3*s²]
= [0≤s≤5 & 3*s²=s³+2*s, 3*s²].
Les expressions 3*s² et s³+2*s ne sont égales que pour 0≤s≤2.
= [0≤s≤5 & 0≤s≤2, 3*s²]
= [0≤s≤2, 3*s²],
ou encore
= [0≤s≤2, s³+2*s]. []

**Produit Relatif.** La formule du produit relatif de deux fonctions f=[p,E] et f'=[p',E'] est la suivante:
[p,E]*[p',E'] = [p(s) & p'(E(s)), E'.E].
En effet, pour que s appartienne au domaine du produit relatif f*f', il faut qu'il appartienne au domaine de f (d'où p(s)) et que son image par f (E(s)) appartienne au domaine de f' (d'où p'(E(s))). Quant à l'image de s par f*f', elle se calcule en appliquant à s l'expression de f (E) suivie de l'expression de f' (E'), soit en fait l'expression composée E'.E. Notez que la <u>composition</u> des <u>expressions</u> s'effectue en ordre inverse du <u>produit relatif</u> des <u>fonctions</u>.

**Exemple**
S = **entier**;
f = [0≤s≤5, s+2],
f'= [0≤s≤5, s-1].
f*f' = [0≤s≤5 & 0≤s+2≤5, s+1]
= [0≤s≤5 & -2≤s≤3, s+1]

$= [0 \leq s \leq 3,\ s+1].$

$f'*f = [0 \leq s \leq 5 \ \&\ 0 \leq s-1 \leq 5,\ s+1]$

$= [1 \leq s \leq 5,\ s+1].$ []

En plus des trois opérations ci-haut, qui opèrent deux fonctions pour générer une troisième, nous étudions ici l'opération de noyau, qui génère une relation (relation d'équivalence, en fait) à partir d'une fonction.

noyau. Soit f une fonction sur S. Le noyau de f est la relation $f*f^{\smile}$; cette relation peut s'écrire comme

$$\{(s,s') \mid f(s)=f(s')\}.$$

Il est simple de vérifier que ceci est une relation d'équivalence sur le domaine de f. Réflexivité: Pour tout s dans le domaine de f, il existe s' tel que $(s,s') \in f$; donc $(s,s) \in f*f^{\smile}$. Symétrie: $(f*f^{\smile})^{\smile} = f^{\smile\smile}*f^{\smile} = f*f^{\smile}$. Transitivité: $(f*f^{\smile})*(f*f^{\smile}) = f*(f^{\smile}*f)*f^{\smile} \subseteq f*I*f^{\smile} = f*f^{\smile}$. Les classes d'équivalence de dom(f) modulo $f*f^{\smile}$ sont appelées les ensembles de niveau de la fonction f (traduction de l'anglais level set, dû à Mills); dans chaque ensemble de niveau tous les éléments ont la même image par f.

## 4.3. Propriétés des Fonctions

Soit une fonction f sur S. Fonction f est dite totale si et seulement si dom(f)=S. Fonction f est dite surjective si et seulement si cod(f)=S.

Dualement à la notion de déterminisme définie dans la section 2.3 pour les relations, nous définissons ici la notion d'injectivité. Nous l'illustrons tout d'abord par un exemple.

**Exemple**
Nous utilisons des lettres pour représenter les arguments et des chiffres pour représenter les images.
$f = \{(a,0),(b,1),(c,2),(d,3),(e,3)\}$,
$f' = \{(a,0),(b,1),(c,1),(d,2),(e,2)\}$.
La fonction f discrimine davantage parmi ses arguments que la fonction f'. []

La fonction f est dite plus injective que la fonction f' si et seulement si $f*f^{\smile} \subseteq f'*f'^{\smile}$. Ceci correspond à l'idée intuitive que l'on a de la notion d'injectivité puisque si l'équivalence $R=f*f^{\smile}$ est un sous-ensemble de l'équivalence $R'=f'*f'^{\smile}$, alors les classes d'équivalence de R sont des sous-ensembles des classes d'équivalence de R'; en d'autres termes, les ensembles de niveau de f sont des sous-ensembles des ensembles de niveau de f'. Notez que l'équation $f*f^{\smile} \subseteq f'*f'^{\smile}$ est équivalente à

$$(\forall s,s' \in dom(f),\ f(s)=f(s') \Rightarrow f'(s)=f'(s'))$$

ou encore

$$(\forall s,s' \in dom(f),\ f'(s) \neq f'(s') \Rightarrow f(s) \neq f(s')).$$

Notez que si $f*f^{\smile} \subseteq f'*f'^{\smile}$ alors $dom(f) \subseteq dom(f')$. Notez aussi que f est plus-injective que f' si et seulement si $f^{\smile}$ est plus-déterministe que $f'^{\smile}$.

**Exemple**

Prenant les fonctions définies dans l'exemple précédent, on calcule
$f*f^ = \{(a,a),(b,b),(c,c),(d,d),(e,d),(d,e),(e,e)\}$,
$f'*f'^= \{(a,a),(b,b),(b,c),(c,b),(c,c),(d,d),(d,e),(e,d),(e,e)\}$.
Le noyau de f est un sous-ensemble du noyau de f'. Les ensembles de niveau de f sont
{a}, {b}, {c}, {d,e};
ceux de f' sont
{a}, {b,c}, {d,e}. []

La relationnelle "plus-injective" est réflexive ($f*f^ \subseteq f*f^$) et transitive (transitivité de l'inclusion). L'exemple ci-dessous montre qu'elle n'est pas symétrique.

**Exemple**
$f = \{(a,0),(b,1),(c,1)\}$,
$f' = \{(a,1),(b,0),(c,0)\}$.
$f*f^ = \{(a,a),(b,b),(b,c),(c,b),(c,c)\}$,
$f'*f'^ = \{(a,a),(b,b),(b,c),(c,b),(c,c)\}$.
Donc $f*f^ \subseteq f'*f'^$ et $f'*f'^ \subseteq f*f^$; pourtant $f \neq f'$. []

Une fonction est dite injective si et seulement si elle est plus-injective que I; en d'autres termes, f est injective si et seulement si $f*f^ \subseteq I$.

**Exemple**
$f = \{(a,0),(b,1),(c,2)\}$; $f*f^=\{(a,a),(b,b),(c,c)\}$; f est injective.
$f' = \{(a,0),(b,1),(c,1)\}$; $f'*f'^=\{(a,a),(b,b),(b,c),(c,b),(c,c)\}$; f' n'est pas injective. []

Notez qu'une fonction f est injective si et seulement si son inverse $f^$ est déterministe.

Les fonctions calculées par des programmes sont généralement très non-injectives. En fait le nom même d'ordinateur suggère la non-injectivité: dans l'idée d'ordre il y a l'idée de plusieurs états initiaux désordonnées qui sont transformés en quelques états finaux ordonnés; d'où l'idée de non-injectivité. Dans le chapitre 9 nous verrons comment cette non-injectivité des programmes d'ordinateurs peut être exploitée à profit pour produire de la tolérance aux erreurs.

# Chapitre 2
# Spécifications de Programmes

La spécification d'un programme est l'expression d'exigences qu'un utlisateur formule au sujet d'un programme qu'il souhaite acquérir. Dans sa forme la plus générale, une specification exprime des exigences fonctionnelles (capabilités fonctionnelles requises du programme) et performancielles (contraintes relatives aux moyens de calcul sur lesquels le programme sera exploité). La spécification sert à double titre: D'abord comme un contrat entre l'utilisateur / spécifieur et le concepteur; ensuite comme le document de base du concepteur. Dans ce chapitre on utilisera les mots utilisateur et spécifieur de façon quasiment interchangeable pour désigner la même personne; On choisira l'un ou l'autre de ces deux termes en fonction de l'aspect de sa fonction que l'on veut mettre en évidence.

## 1. Spécifications: Nature et Définition

On dit que l'on a defini une spécification quand on s'est donné:
- Un ensemble S, appelé l'espace de la spécification,
- Une relation R sur S, appelée la relation de la spécification.

Il est naturel de représenter une spécification par la paire (S,R). Alors tout élément de S est dit un état de la spécification; tout élément de dom(R) est dit un état initial de la spécification; tout élément de cod(R) est dit un éta final de la spécification. Quand S est implicite dans le contexte de nos discussions, on ne le mentionnera pas dans la représentation de la spécification: On la représentera simplement par R.

La sémantique de la spécification (S,R) est définie de la façon suivante:
- Seuls les éléments de dom(R) peuvent être soumis par l'utilisateur; aucun élément en dehors de dom(R) ne sera soumis.
- Pour tout s dans dom(R), le programme candidat doit finir d'exécuter au bout d'un temps fini, dans un état appartenant à l'ensemble s.R .

Nous discutons tour à tour les deux clauses de cette définition.

La **première clause** peut être exprimée de la façon suivante: L'utilisateur doit définir R de telle sorte que dom(R) inclue tous les états qui peuvent être soumis (comme états initiaux) au programme candidat à construire. Par exemple, soit à spécifier un programme pour calculer la racine carrée d'un nombre réel. On définit S=**réel**. Quant à R, on peut le définir de deux façons différentes: Si on est sûr que seuls des réels positifs seront soumis au programme, alors on pose

$$R = \{(s,s') \mid s \geq 0 \ \&\ s' = \sqrt{s}\};$$

si on suspecte que des valeurs négatives peuvent être soumises aussi, alors on pose

$$R' = \{(s,s') \mid s < 0 \ \&\ s' = -1\} \cup \{(s,s') \mid s \geq 0 \ \&\ s' = \sqrt{s}\}.$$

Notez que la relation R' est plus définie que la relation R; cette

remarque sera interprétée dans la section 5. Bien sûr, on considère généralement qu'il est souhaitable pour un programmeur à qui la spécification R est soumise de prévoir le cas où l'état initial se trouve être négatif, en ajoutant à son programme la clause "**si** s<0 **alors** ... "; cette pratique est connue sous le nom de programmation défensive (défensive programming). Bien qué nous apprécions l'intérêt pratique d'une telle approche, nous considérons que d'un point de vue purement théorique, un programme qui n'aurait pas prévu le cas (s<0) n'est guère plus "correct" vis-à-vis la spécification R qu'un programme qui n'aurait pas prévu le cas. Nous pensons qu'il n'appartient pas au programmeur de deviner quels états initiaux peuvent possiblement être soumis; plutôt il appartient au spécifieur de le déterminer.

Quant à la **deuxième clause**, elle exprime que pour tout s dans dom(R), l'ensemble des états finaux considérés corrects est s.R. Puisque R n'est pas déterministe, s.R peut être arbitrairement large. Par exemple, soit à spécifier un programme qui effectue l'addition de deux entiers.

S = ens
a, b: **entier**
**fin**,
R = {(s,s') | a(s')=a(s)+b(s)}.

Vu qu'elle n'impose aucune contrainte sur b(s'), cette relation est "très" non-déterministe. Ainsi, pour s=(3,5), s.R = {s' | a(s')=8}; des exemples d'éléments de s.R sont (8,0), (8,3), (8,5), (8,8), (8,9), etc ... Certains modèles de spécification ([HEHN83], par exemple) supposent que dés lors que le spécifieur n'exprime aucune contrainte sur l'état final correspondant à un état initial s, alors le spécifieur ne s'intéresse pas à ce que le programme-candidat termine pour l'état initial s. Précisons que telle n'est pas l'interprétation de notre modèle: Si le spécifieur n'exprime pas de contraintes sur l'état final correspondant à l'état initial s, alors s.R=S; si le spécifieur ne s'intéresse pas à ce que le programme-candidat termine pour l'état initial s alors s∉dom(R), autrement dit s.R=∅. Il s'agit là de deux conditions distinctes.

**Spécifications Spéciales** sur un espace S: La spécification vide ∅ est satisfaite par tout programme; la spécification totale U est satisfaite par tout programme qui termine pour tout s dans S.

## 2. Spécifications: Génération

A partir d'une idée abstraite qu'il a, le spécifieur doit générer une spécification sous la forme (S,R). Si la spécification à laquelle il pense est complexe, le spécifieur affrontera sa complexité de façon progressive en générant des sous-spécifications simples pour ensuite les combiner ensemble et composer une spécification complexe. Si on analyse la démarche de l'esprit qu'un spécifieur adopte dans cette tâche, on remarque qu'elle est la combinaison de deux méthodes orthogonales, que nous étudions ici: La méthode intersection et la méthode union.

### 2.1. Intersection: Composer des propriétés vagues

Cette méthode consiste à formuler des relations R1, R2, ... Rk ayant le même domaine; ensuite on définit R comme étant l'intersection de toutes ces relations.

Chaque relation Ri est arbitrairement non-déterministe (ayant un rapport images/argument arbitrairement grand); car chaque Ri exprime une propriété arbitrairement vague que le spécifieur souhaiterait voir entre les états initiaux et les états finaux. Une fois que le spécifieur a pensé avoir exprimé dans les Ri toutes les propriétés qu'il imaginait, alors il en fait l'intersection pour obtenir R.

Cette méthode est adaptée aux cas où la propriété que l'on aimerait exprimer entre l'état initial et l'état final est mieux perçue comme une conjonction de propriétés plus vagues; des exemples d'une telle situation sont donnés ci-bas.

**Exemple**
On appelle T l'ensemble des tableaux de nombres réels; on définit les fonctions suivantes:
- **tai**, une fonction de T vers **naturel**, définie par
  **tai**(t) = taille du tableau t.
- **ord**, un prédicat sur T, défini par
  **ord**(t) = t est prdonné.
- **prm** un prédicat sur TxT, défini par
  **prm**(t,t') = t' est une permutation de t.

Nous nous proposons de spécifier un programme de tri. Une séquence "réaliste" de relations Ri serait

R1 = {(t,t') | **ord**(t')},
R2 = {(s,s') | **tai**(t)=**tai**(t')},
R3 = {(s,s') | **prm**(t,t')}.

Etant convaincus que nous avons ainsi exprimé toutes les relations que nous voulions, nous posons

R = R1 ○ R2 ○ R3.

Constatant que R3 est un sous-ensemble de R2, nous posons

R = R1 ○ R3. □

**Exemple**
Soit S l'ensemble défini par

```
S = ens
crt
   a, b:  entier;
sse
   a>0 & b>0
fin.
```

On se propose de spécifier un programme de plus grand commun diviseur, qui met le résultat dans a et préserve b. On écrira trois relations.
R1 exprime que la valeur finale de a divise (la valeur initiale de) a et b:
R1 = {(s,s') | a(s) **mod** a(s') = 0 & b(s) **mod** a(s') = 0}.
R2 exprime que la valeur finale de a est le <u>plus grand</u> diviseur de a et b:
R2 = {(s,s') | ∀x, x>a(s') => (a(s) **mod** x ≠ 0 v b(s) **mod** x ≠ 0)}.
R3 exprime que b est préservé:

R3 = {(s,s') | b(s')=b(s)}. []

Pour conclure cette section, nous attirons l'attention du lecteur sur un lien intéressant qui existe entre les relations Ri et la relation R = R1 O R2 ... O Rk. Souvenons-nous que les relations Ri ont le même domaine, que l'on appellera D. Supposons que dom(R) ne soit pas égal à D (mais qu'il en soit un sous-ensemble); alors il existe s dans D tel que pour tout i, s.Ri≠ø alors que s.R=ø. Cette situation "anormale" émerge quand les relations Ri sont incohérentes les unes avec les autres. Excluant cette situation, nous posons

dom(R)=dom(Ri). (a)

Par ailleurs, nous avons -par définition de R- que R est un sou-ensemble de Ri; d'où l'on déduit que pour tout s,

s.R⊆s.Ri. (b)

Des équations (a) et (b) on déduit que R est plus définie que Ri; cette assertion sera commentée dans la section 5 de ce chapitre.

## 2.2. Union: Composer des sous-domaines

Cette méthode consiste à formuler des relations R1, R2, ... Rk dont les domaines D1, D2, ... Dk sont disjoints deux à deux; ensuite on définit R comme étant l'union de toutes ces relations.

Chaque relation Ri a un domaine arbitrairement petit (par rapport à l'inclusion); donc chaque relation Ri porte arbitrairement peu d'information et est arbitrairement facile à générer. Quand on est convaincu que l'union des domaines Di couvre l'ensemble de tous les états initiaux possibles, alors on pose R = R1 U R2 U ... Rk.

Cette méthode est adaptée aux cas où la propriété que l'on aimerait exprimer entre les états initiaux et les états finaux est mieux formulée par une analyse de cas sur le domaine des états initiaux; des exemples d'une telle situation sont donnés ci-dessous.

**Exemple**
Soit S=**réel**; on aimerait écrire la spécification d'un programme "robuste" de racine carrée, c'est à dire un programme qui réagit convenablement même quand un nombre négatif lui est soumis.

R1 = {(s,s') | s≥0 & s'=√s},
R2 = {(s,s') | s<0 & s'=-1}. []

**Exemple**
Soit S = **ens**
**crt**
a, b, c: **réel**;
x', x": **réel** U {*};
**sse**
a≠0
**fin.**

On aimerait écrire un programme qui calcule les racines du trinôme (a,b,c) et les met dans (x',x"). On génère les relations suivantes:

R1 = {(s,s') | d(s)≥0 & x(s')=(-b(s)-√d(s))/(2*a(s))
& x(s")=(-b(s)+√d(s))/(2*a(s))},
R2 = {(s,s') | d(s)<0 & x'(s')=* & x"(s')=*}. []

Avant de conclure cette section, nous attirons l'attention du lecteur sur un lien intéressant qui existe entre les relations Ri et la relation R. Pour tout i, nous avons

$$dom(Ri) \subseteq dom(R), \qquad (a)$$

puisque $Ri \subseteq R$. De plus, par définition de R, on a $I(Di)*R = I(Di)*(R1 \cup R2 \cup ...Rk)$; puisque les domaines des Di sont disjoint deux à deux, on peut écrire $Ri=I(Di)*R$ (car $i \neq j \Rightarrow I(Di)*Rj=\emptyset$); d'où l'on déduit que pour tout s dans Di=dom(Ri), on a

$$\begin{aligned} s.Ri &= s.I(Di)*Ri \\ &= (s.I(Di)).Ri \\ &\qquad \text{puisque } s \in Di,\ s.I(Di)=s, \text{ donc} \\ &= s.R. \end{aligned} \qquad (b)$$

Des équations (a) et (b) on déduit que R est plus-défini que Ri. Cette remarque sera commentée dans la section 5 de ce chapitre.

## 3. Spécifications: Validation

L'activité de validation de la spécification (S,R) consiste à formuler des propriétés dont on pense que la specification (S,R) doit satisfaire, ensuite à confronter ces proprietés contre la spécification.

Dans cette section nous tentons de résoudre deux questions: premièrement, quelle forme une telle propriété prend-elle? deuxièmement, comment confronte-t-on une propriété à une spécification?

**Définir une propriété.** Une expression informelle d'une propriété serait la suivante: Pour une telle classe d'états initiaux, la spécification exprime que les états finaux verifient telle condition. Il serait raisonnable de représenter une telle propriété par une relation contenant toutes les paires (s,s') telles que s appartient à la classe d'états initiaux citée et s' appartient à l'ensemble des états finaux vérifiant la condition citée.

**Confronter une Propriété à une Spécification.** Soit R une spécification et V une propriété. V ne concerne qu'une classe limitée d'états initiaux. De plus, elle n'exprime qu'une parmi les exigences fonctionnelles exprimées par R; donc elle admet plus d'états finaux pour chaque état initial que ne le fait R. Ceci s'écrit formellement

$$dom(V) \subseteq dom(R)$$
$$\forall s \in dom(V),\ s.R \subseteq s.V.$$

Si bien que la question de savoir si la spécification R vérifie la propriété V revient à la question de savoir si la relation R est plus-définie que la relation V. En fait ceci n'est pas surprenant quand on pense à l'interprétation de la relationnelle "plus-définie": valider la spécification R contre la propriété V consiste simplement à vérifier que R contient plus d'information que V (qui ne reflète qu'une propriété de R).

**Exemple**

S = **réel**;

$R = \{(s,s') \mid s<0 \ \&\ s'=-1\} \cup \{(s,s') \mid s \geq 0 \ \&\ s' \backslash / s\}$.

On peut vouloir vérifier plusieurs propriétés.

- Que pour tout etat initial negatif, la spécification R exprime que

l'état final est négatif:
$V1 = \{(s,s') \mid s<0 \ \& \ s'<0\}$.
- Que pour tout état initial plus grand que 4, la spécification R exprime que l'état final est plus petit que l'état initial:
$V2 = \{(s,s') \mid s>4 \ \& \ s'<s\}$.
- Que pour tout état initial entre 1 et 6 inclus, la spécification R exprime que l'état final est la racine carrée de l'état initial:
$V3 = \{(s,s') \mid 1\leq s\leq 6 \ \& \ s'=\sqrt{s}\}$.
- Que pour tout état initial entre -1 et 1 inclus, la spécification R exprime que l'état final est plus petit ou égal à 1 en valeur absolue:
$V4 = \{(s,s') \mid -1\leq s\leq 1 \ \& \ |s'|\leq 1\}$.
- Que la spécification R est définie pour tout état initial positif:
$V5 = \{(s,s') \mid s>0\}$.
- Que la spécification est définie pour tout état initial:
$V6 = \{(s,s') \mid \text{vrai}\} = S\times S$.

On valide R contre chacune des propriétés Vi.

V1. $dom(V1) = \{s \mid s<0\} \subseteq dom(R)=S$.
$\forall s<0$, $s.R = \{-1\} \subseteq \{s \mid s<0\} = s.V1$.

V2. $dom(V2) = \{s \mid s>4\} \subseteq dom(R)=S$.
$\forall s>4$, $s.R = \{\sqrt{s}\} \subseteq s.V2 = \{s' \mid s'<s\}$.

V3. $dom(V3) = \{s \mid 1\leq s\leq 6\} \subseteq dom(R)=S$.
$\forall s: 1\leq s\leq 6$, $s.R = \{\sqrt{s}\} \subseteq s.V3 = \{\sqrt{s}\}$.

V4. $dom(V4) = \{s \mid -1\leq s\leq 1\} \subseteq dom(R)=S$.
$\forall s: -1\leq s\leq 1$,
si $-1\leq s<0$ alors $s.R=\{1\} \subseteq s.V4=\{s' \mid |s'|\leq 1\}$,
si $0\leq s\leq 1$ alors $s.R=\{\sqrt{s}\} \subseteq s.V4=\{s \mid |s'|\leq 1\}$.

V5. $dom(V5) = \{s \mid \neg>0\} \subseteq dom(R)=S$.
$\forall s: s>0$, $s.R=\{\sqrt{s}\} \subseteq s.V5=S$.

V6. $dom(V6)=S \subseteq dom(R)=S$.
$\forall s$, $s.V6=S \Rightarrow \forall s$, $s.R \subseteq s.V6$. []

## 4. Spécifications à Base de Prédicats

Dans la tradition Hoarienne ([FLOY67], [HOAR69]), les spécifications sont représentées par des paires de prédicats, à savoir le prédicat d'entrée et le prédicat de sortie. Nous nous efforçons, dans cette section, de définir le lien qui existe entre notre mode de représentation des spécifications -à base de relations- et le mode de représentation à base de prédicats.

On dit que l'on a défini une <u>spécification à base de prédicats</u> quand on s'est donné:
- Un ensemble S, dit <u>espace</u> de la spécification.
- Un prédicat p sur S, dit <u>prédicat d'entrée</u> de la spécification.
- Un prédicat q sur S, dit <u>prédicat de sortie</u> de la spécification.

Pour refléter cette définition, nous représenterons une spécification à base de prédicats par le triplet (S,p,q); quand S est implicite, on écrira simplement (p,q). La sémantique de la spécification (S,p,q) est définie de la façon suivante: (S,p,q) est une représentation de la spécification (S,R) où $R = \{(s,s') \mid p(s) \ \& \ q(s')\}$. Quand R est ainsi défini, on dit que (S,p,q) et (S,R) sont

équivalentes.

Comment transforme-t-on une spécification écrite sous une forme (paire de prédicats ou relation) en une spécification équivalente écrite sous l'autre forme?

**Transformation: Prédicats -> Relation.** La spécification (S,p,q) est transformée en (S,R) où R = {(s,s') | p(s) & q(s')}. Cette transformation doit sa validité à la définition sémantique de (S,p,q).

**Transformation: Relation -> Prédicats.** La spécification (S,R) est transformée en (S,p,q) où

p(s) = s∈dom(R) & s=s0,
q(s) = (s0,s)∈R.

Si on calcule la valeur sémantique de (S,p,q), on trouve $(S,R_{p,q})$ où

$R_{p,q}$ = {(s,s') | s∈dom(R) & s=s0 & (s0,s')∈R}
= {(s,s') | s∈dom(R) & (s,s')∈R}
= {(s,s') | (s,s')∈R}.

D'où l'on déduit la validité de la transformation proposée.

**Exemple**

```
S = ens
    crt
       a, b: entier;
    sse
       a>0 & b>0
    fin.
```

On dénotera le plus grand commun diviseur de a(s) et b(s) par pgcd(s).

p(s) = (s=s0)
q(s) = (a(s)=pgcd(s0)).

Alors on transforme (S,p,q) en (S,R) où

R = {(s,s') | a(s')=pgcd(s)}.

Inversement, soit la spécification (S,R) où

R = {(s,s') | b(s')=a(s)+b(s)}.

On transforme cette spécification en (S,p,q) où

p(s) = (s=s0,)
q(s) = (b(s)=a(s0)+b(s0)). []

## 5. Simplicité d'une Spécification: Génération

En tentant d'analyser la démarche de l'esprit dans l'activité de générer une spécification, on peut déterminer que, face à la complexité d'une relation, on la décompose en des relations moins-définies. Ceci suggère que, pour ce qui est de l'activité de génération de spécifications sous forme de relations, être moins-défini est synonyme de être plus simple. Nous verrons dans les chapitres suivants que ce rapprochement peut etre fait dans plusieurs autres contextes que la génération de spécification.

# Chapitre 3

## Analyse Fonctionnelle de Programmes

Dans ce chapitre nous étudions comment on peut générer la fonction calculée par un programme. L'approche que nous prenons à cet effet est une approche déductive: La fonction d'un programme complexe est calculée à partir des fonctions de ses composantes.

## 1. Abstraction Fonctionnelle: Définitions

Soit P un programme sur les variables a, b, c de types A, B, C. Si on ne s'intéresse qu'aux valeurs des variables a, b, c vérifiant une condition p(a,b,c) alors on définit l'espace du programme P comme étant l'ensemble

```
S = ens
crt
   a: A;
   b: B;
   c: C;
sse
   p(a,b,c)
fin.
```

Des éléments de S sont appelés les états du programme P; quand le programme P commence son exécution avec des valeurs initiales a0 pour a, b0 pour b et c0 pour c, on dit que s0=(a0,b0,c0) est un état initial du programme; quand le programme P finit son exécution avec des valeurs $a_f$ pour a, $b_f$ pour b et $c_f$ pour c, on dit que $s_f=(a_f,b_f,c_f)$ est un état final du programme.

Soit P un programme sur l'espace S. Quand P est exécuté sur un état initial s, une parmi plusieurs conditions peut se produire:

- P termine normalement au bout d'un temps (nombre d'étapes ou de changements d'etats) fini dans un état défini s'.
- P se lance dans une boucle infinie -de sorte qu'il ne génère pas d'état final défini.
- P invoque une opération impossible telle que le logarithme d'un nombre négatif, la division par zéro, la lecture au delà de la fin d'un fichier, etc ...
- P cause un évènement inattendu tel que un débordement, une erreur de parité, etc ...

Nous dirons que l'exécution de P dans le premier cas se passe dans des conditions normales alors dans les autres cas elle se déroule dans des conditions exceptionnelles. Dans ce chapitre, nous supposerons que l'arithmétique de notre machine est parfaite, de sorte que les seules conditions exceptionnelles dont nous nous préoccupons sont le fait de boucler indéfiniment et le fait d'invoquer une opération indéfinie. Un modèle de traitement d'erreurs englobant

toutes les conditions exceptionnelles sera proposé dans le chapitre 9.

L'abstraction fonctionnelle du programme P est la fonction notée [P] et définie par:
[P] = {(s,s') | Si P exécute à partir de l'état initial s alors il termine normalement dans l'état final s'}.
Si f est l'abstraction fonctionnelle de P, on dit que P calcule la fonction f.

Il résulte de la définition de [P] que le domaine de [P] est:
dom([P]) = {s| Si P exécute à partir de l'état initial s alors il termine normalement}.

**Exemple**
Nous donnons -sans preuve- quelques exemples d'abstractions fonctionnelles.
S = **réel**; P = **début** s:=1/(1-s); s:=log(s) **fin**.
[P] = [s<1, -log(1-s)]; dom([P]) = {s| s<1}.
S = **réel**; P = **début si** s<4 **alors** s:=s+6 **sinon** s:=s-5;
**si** s<6 **alors** s:=s+2 **sinon** s:=s-1 **fin**.
[P] = [s<0, s+8] U [0≤s<4, s+5] U [4≤s<11, s-3] U [s≥11, s-6].
dom([P]) = S.
S = **ens**
a, b: **entier**
**fin**;
P = **tantque** b≠0 **faire début** a:=a+1; b:=b-1 **fin**.
[P] = [b(s)≥0, (a(s)+b(s),0)]; dom([P]) = {s| b(s)≥0}. []

## 2. Abstraction Fonctionnelle: Génération

Dans cette section, nous présentons un ensemble d'axiomes pour la génération d'abstractions fonctionnelles de programmes Pascal. En particulier, on s'intéressera aux figures suivantes de la programmation:

- L'affectation, ayant la forme <variable> := <expression>,
- La séquence, ayant la forme <instruction>; <instruction>,
- L'alternation, ayant la forme **si** <expression> **alors** <instruction> **sinon** <instruction>,
- L'itération, ayant la forme **tantque** <expression> **faire** <instruction>,
- La simultanéité, ayant la forme **codébut** <instruction> , <instruction> **cofin**,
- L'appel de procédure,
- Le passage de paramètres,
- La récursion.

Pour chacune des figures de programmation mentionnées ci-dessus, on présentera un axiome qui permet d'en calculer l'abstraction fonctionnelle; le système d'axiomes ainsi obtenu sera adopté comme notre définition formelle de la sémantiques des figures de programmation concernées; nous nous efforcerons de convaincre le lecteur de la cohérence des axiomes présentés avec la sémantique

associée aux figures concernées.

## 2.1. Affectation

L'instruction d'affectation a la forme s:=E(s), où s est l'état global du programme et E est une expression sur tout l'espace du programme concerné; la sémantique de l'affectation est définie par l'

**Axiome de l'Affectation**
[s:=E(s)] = [s∈def(E), E(s)].

En effet, pour que l'instruction (s:=E(s)) puisse s'exécuter dans des conditions normales, il faut que l'état initial appartienne au domaine de définition de l'expression E; de plus l'état final généré par l'exécution de cet énoncé sur s est E(s).

**Exemple**
Soit S = **ens**
a, b: **réel**
**fin**
et P = **début** b:=log(a+b) **fin.**
Alors, on interprète cette instruction comme ayant la forme s:=E(s) où s = (a(s),b(s)) et E(s) = (a(s), log(a(s)+b(s))); on a def(E) = {s/ a(s)+b(s)>0}; donc
[P] = [a(s)+b(s)>0, (a(s), log(a(s)+b(s)))]. □

## 2.2. La Séquence

La sémantique de la séquence est définie par l'

**Axiome de la séquence**
[P1; P2] = [P1]*[P2].

En effet, une paire (s,s') appartient à [P1;P2] si et seulement si il existe un état intermédiaire s" tel que (s,s")∈[P1] et (s",s')∈[P2].

**Exemples**
S = **réel**; P = **début** s:=s+1; s:=s*s **fin.**
[P] = [vrai, s+1] * [vrai, s*s]
= [vrai, $(s+1)^2$].
S = **réel**; P = **début** s:=1/(1-s); s:=log(s) **fin.**
[P] = [s≠1, 1/(1-s)] * [s>0, log(s)]
= [s≠1 & (1-s)>0, -log(1-s)]
= [s<1, -log(1-s)].
S = **réel**; P = **début** s:=s*s+4; s:=log(3-s) **fin.**
[P] = [vrai, $s^2+4$] * [3-s>0, log(3-s)]
= [vrai & $3-s^2-4>0$, $\log(3-s^2-4)$]
= ∅.
Le programme P n'est défini pour aucun état initial. □

## 2.3. L'Alternation

La sémantique de l'alternation est définie par un axiome,

**L'axiome de l'alternation**
**[si** t **alors** ca **sinon** cs] = I(t)*[ca] U I(~t)*[cs].

(ca est une abbréviation de clause alors et cs une abbérviation de clause sinon). En effet, l'ensemble des paires (s,s') qui constitue la fonction **[si** t **alors** ca **sinon** cs] est l'union de deux ensembles: L'ensemble des paires (s,s') telles que t(s) est vrai, qui est représenté pat I(t)*[ca]; l'ensemble des paires (s,s') telles que ~t(s) est vrai, qui est représenté par I(~t)*[cs].

De L'axiome de l'alternation on peut déduire le cas particulier de

**L'axiome de l'instruction conditionnelle**
**[si** t **alors** ca] = I(t)*[ca] U I(~t).

Cet axiome se déduit du précédent si on remarque que **(si** t **alors** ca) est équivalent à **(si** t **alors** ca **sinon** ) et que l'abstraction fonctionnelle de l'instruction vide est la fonction identité.

**Exemple.**
Cet exemple combine l'axiome de l'alternation avec ceux de l'affectation et de la séquence; notez que la fonction obtenue est difficile à obtenir sans le secours des axiomes mentionnés.
S = **réel; p = début si** s<4 **alors** s:=s+6 **sinon** s:=s-5;
**si** s<6 **alors** s:=s+2 **sinon** s:=s-1 **fin.**

[p] = ([s<4, s+6] U [s≥4, s-5]) * ([s<6, s+2] U [s≥6, s-1])
= [s<4 & s+6<6, s+8] U [s<4 & s+6≥6, s+5] U
[s≥4 & s-5<6, s-3] U [s≥ & s-5≥6, s-6]
= [s<0, s+8] U [0≤s<4, s+5] U [4≤s<11, s-3] U [s≥11, s-6]. []

## 2.4. Itération

La sémantique de l'itération est définie par un axiome,

**L,axiome de l'itération**
**[tantque** t **faire** b] = (I(t)*[b])**I(~t).

Il n'est pas possible de donner une preuve rigoureuse de cet axiome vu que nous n'avons pas préalablement défini la sémantique de l'itération. Ce que nous ferons, par contre, est de prouver la validité de cet axiome par rapport à trois hypothèse simples concernant le comportement des instructions tantque; le lecteur se convaincra aisément du bien fondé de ces hypothèses par rapport au comportement opérationnel qu'il associe à cette instruction. On dénotera par w l'instruction (**tantque** t **faire** b) et par f la fonction

$(I(t)*[b])^{**}I(\sim t)$.

H0. Pour tout s tel que $\sim t(s)$, l'instruction tantque termine normalement sans modifier son état; formellement, ceci s'écrit
$\sim t(s) \Rightarrow [w](s)=s$.

H1. Pour tout s tel que $t(s)$, l'instruction tantque est équivalente -quand elle termine- à l'exécution d'un nombre fini de tests de t suivis du corps de boucle b; formellement, $t(s) \;\&\; s\in dom([w]) \Rightarrow \exists i>0: s.[w]=s.(I(t)*[b])^i$.

H2. Soit i l'entier défini ci-dessus; alors $\sim t(s.(I(t)*[b])^i)$.

**Preuve.** Pour prouver que $[w]=f$, il suffit de prouver trois lemmes:
L1. f est une fonction.
L2. $[w]\subseteq f$.
L3. $dom(f)\subseteq dom([w])$.
Il est facile de se convaincre que la conjonction de ces trois lemmes entraine $[w]=f$: Soit $(s,s')\in f$; $s\in dom(f)$, donc (L3) $s\in dom([w])$. Soit $s"=s.[w]$; on a $(s,s")\in[w]$, donc (L2) $(s,s")\in f$. De $(s,s')\in f$ et $(s,s")\in f$ on déduit (L1) $s'=s"$, d'où $(s,s')\in[w]$. On a déduit $(s,s')\in[w]$ de $(s,s')\in f$ donc $f\subseteq[w]$; ceci, en conjonction avec L2 conduit au résultat désiré.

**Lemme L1.** Soient $(s,s')$ et $(s,s")$ deux éléments de f; on prouvera que $s'=s"$.
$(s,s')\in f \Rightarrow \exists x:\ (s,x)\in(I(t)*[b])^* \;\&\; (x,s')\in I(\sim t)$
$\Rightarrow (s,s')\in(I(t)*[b])^* \;\&\; \sim t(s')$
$\Rightarrow \exists i\geq 0: (s,s')\in(I(t)*[b])^i \;\&\; \sim t(s')$.
De même,
$(s,s")\in f \Rightarrow \exists j\geq 0: (s,s")\in(I(t)*[b])^j \;\&\; \sim t(s')$.
Pour prouver $s'=s"$, on prouvera en fait un résultat plus fort, à savoir $i=j$. Supposons que $i>j$; alors
$(s",s') \in ((I(t)*[b])^j)^\sim * (I(t)*[b])^j * (I(t)*[b])^{i-j}$.
$I(t)*[b]$ est -par définition- une fonction; donc $(I(t)*[b])^j$ est aussi une fonction; donc $((I(t)*[b])^j)^\sim * (I(t)*[b])^j$ est un sous-ensemble de la relation identité. On en déduit
$(s",s')\in(I(t)*[b])^{i-j}$.
On en déduit $s" \in dom((I(t)*[b])^{i-j})$
$\Rightarrow$ $s"\in dom(I(t)*[b])$
$\Rightarrow$ $s"\in dom(I(t))$
$\Rightarrow$ $t(s")$,
ce qui est contradiction avec l'hypothèse $(s,s")\in f$.

**Lemme L2.** Soit $(s,s')\in[w]$. Si $\sim t(s)$ alors -d'après H0- $s'=s$; on en déduit que $(s,s')\in I(\sim t)$, d'où $(s,s')\in f$. Si $t(s)$ alors -d'après H1- il existe $i>0$ tel que $(s,s')\in(I(t)*[b])^i$. Puisque $s'=s.(I(t)*[b])^i$, on a -d'après H2- $\sim t(s')$. De $(s,s')\in(I(t)*[b])^i$ et $(s',s')\in I(\sim t)$ on déduit $(s,s')\in(I(t)*[b])^{**}I(\sim t)$.

**Lemme L3.** Soit s un élément de dom(f). Si $\sim t(s)$ alors (H0) w termine normalement donc $s\in dom([w])$. Si $t(s)$ alors il existe $i>0$ tel que $s\in dom((I(t)*[b])^i*I(\sim t))$, donc -d'après H1 et H2- w termine normalement; d'où $s\in dom([w])$. CQFD

Bien qu'elle soit constructive, cette définition de la fonction d'une instruction tantque est peu pratique car le calcul de la fermeture transitive d'une relation est -potentiellement- fort complexe. Nous proposons ci-dessous une définition alternative de la fonction d'une boucle. Bien qu'elle ne soit pas constructive, cette définition est d'usage facile -une fois que l'on a deviné la fonction de la boucle.

> **Théorème de l'Instruction Tantque** (version originale dûe à H.D. Mills). Soit w = (**tantque** t **faire** b) un programme sur S et soit f une fonction sur S. Alors [w]=f si et seulement si
> p0: I(~t)*f = I(~t),
> p1: I(t)*f = I(t)*[b]*f,
> p2: dom(f)=dom([w]).

**Preuve.** La preuve de nécessité est très simple: Si [w]=f alors p0 et p1 peuvent être vérifiés en utilisant l'axiome de l'itération et p2 est une tautologie. Quant à la suffisance des prémisses p0, p1 et p2, on peut la prouver ainsi: Soit s un élément de dom([w]); d'après p2, s appartient au domaine de f; on aimerait prouver que s.f=s.[w]. Si s vérifie ~t(s) alors (s.f=s.[w]) résulte de p0 et H0. Supposons que s vérifie t(s); alors d'après H1 (et puisque s∈dom([w])), il existe i>0 tel que s.[w] = s.(I(t)*[b])$^i$.
On applique I(~t) de chaque coté de l'égalité
s.[w].I(~t) = s.(I(t)*[b])$^i$*I(~t).
Or s.[w].I(~t) = s.([w]*I(~t)); d'autre part, [w]*I(~t)=[w] puisque cod([w])⊆S!~t. Donc
s.[w] = s.(I(t)[b])$^i$*I(~t).
D'après la prémisse p0, on peut écrire
s.[w] = s.(I(t)*[b])$^i$*I(~t)*f.
Puisque (H2) ~t(s.(I(t)*[b])$^i$), on a s.(I(t)*[b])$^i$*I(~t) = s.(I(t)*[b])$^i$; d'où
s.[w] = s.(I(t)*[b])$^i$*f.
On factorise I(t)*[b]:
s.[w] = s.(I(t)*[b])$^{i-1}$*(I(t)*[b]*f).
D'après la prémisse p1, on a
s.[w] = s.(I(t)*[b])$^{i-1}$*I(t)*f.
Par définition de i, s∈dom((I(t)*[b])$^i$); donc s.(I(t)*[b])$^{i-1}$ ∈ dom(I(t)*[b]); or, dom(I(t)*[b]) ⊆ dom(I(t)) = S!t. Donc s.(I(t)*[b])$^{i-1}$ vérifie t; on en déduit
s.[w] = s.(I(t)*[b])$^{i-1}$*f.
Factorisant un à un les (i-1) facteurs I(t)*[b] comme nous venons de le faire, nous obtenons
s.[w] = s.f. QED

Etant donnée une instruction w = (**tantque** t **faire** b), nous pouvons déterminer sa fonction [w] par l'une des deux formules suivantes:
- la formule [w] = (I(t)*[b])*I(~t),
- la formule [w] = la fonction qui vérifie p0, p1 et p2.

Dans la suite de ce travail nous utiliserons l'une ou l'autre de ces deux formules de façon interchangeable, au gré de nos besoins.

Nous donnons ci-dessous deux exemples simples d'application de

ces deux formules.

**Exemple**
S = **ens**
    x, y, z: **entier**
**fin,**
w = **tantque** y≠0 **faire début** y:=y-1; z:=z+x **fin.**
On utilise la formule [w] = (I(t)*[b])**I(~t).
I(t) = {(s,s') | y(s)≠0 & s'=s},
[b] = {(s,s') | x(s')=x(s) & y(s')=y(s)-1 & z(s')=z(s)+x(s)}.
I(t)*[b]
= {(s,s') | y(s)≠0 & x(s')=x(s) & y(s')=y(s)-1 & z(s')=z(s)+x(s)}.
La fermeture transitive de cette relation est
(I(t)*[b])+
= {(s,s') | ∃i>0: ((∀k, 0≤k<i => y(s)-k≠0) & x(s')=x(s) &
y(s')=y(s)-i & z(s')=z(s)+i*x(s))}.
(I(t)*[b])+*I(~t)
= {(s,s') | ∃i>0: ((∀k, 0≤k<i => y(s)-k≠0) & x(s')=x(s) &
y(s')=y(s)-i & z(s')=z(s)+i*x(s) & y(s')=0}.
= {(s,s') | (∀k, 0≤k<y(s) => y(s)-k≠0) & x(s')=x(s) & y(s')=0 &
z(s')=z(s)+y(s)*x(s)}
= {(s,s') | y(s)>0 & x(s')=x(s) & y(s')=0 & z(s')=z(s)+y(s)*x(s)}.
Notant que (I(t)*[b])**I(~t) = (I(t)*[b])+*I(~t) U I(~t), on a
[w]
= {(s,s') | y(s)=0 & s'=s}
  U
  {(s,s') | y(s)>0 & x(s')=x(s) & y(s')=0 & z(s')=z(s)+y(s)*x(s)}
= {(s,s') | y(s)=0 & x(s')=x(s) & y(s')=0 & z(s')=z(s)+y(s)*x(s)}
  U
  {(s,s') | y(s)>0 & x(s')=x(s) & y(s')=0 & z(s')=z(s)+y(s)*x(s)}
= {(s,s') | y(s)≥0 & x(s')-x(s) & y(s')=0 & z(s')=z(s)+y(s)*x(s)}.
Interprétons ce résultat: w n'est défini que pour y non-négatif (en effet, pour y négatif w ne termine pas); quand w est exécuté sur s, il préserve x(s), annule y(s) et ajoute le produit y(s)*x(s) à z(s).

Notez qu'en dépit de la simplicité et de l'uniformité de ce programme, le calcul de sa fonction s'avère difficile, en particulier au niveau du calcul de la fermeture transitive. Dans l'exemple suivant, l'application de la méthode constructive s'avèrerait quasiment impossible; on aura recours alors à la caractérisation de [w] via les prémisses p0, p1 et p2. []

**Exemple**
S = **ens**
**crt**
  x, y: **entier;**
**sse**
  x>0 & y>0
**fin.**
w = **tantque** x≠y **faire si** x>y **alors** x:=x-y **sinon** y:=y-x.
On notera par pgcd(s) le plus grand commun diviseur de x(s) et y(s). On pose f = [vrai, (pgcd(s),pgcd(s))] et on vérifie que f satisfait les prémisses p0, p1 et p2 du théorème de l'itération; on en déduira alors f=[w].

p0: $I(x(s)=y(s)) * f = [x(s)=y(s), (pgcd(s),pgcd(s))]$.
Il est une propriété de la fonction pgcd que si $x(s)=y(s)$ alors $pgcd(s) = x(s) = y(s)$. Donc
$[x(s)=y(s), (pgcd(s),pgcd(s))]$
peut s'écrire
$= [x(s)=y(s), (x(s),y(s))]$
$= [x(s)=y(s), s]$
$= I(x(s)=y(s))$.

p1: $I(x(s)\neq y(s))*[b]*f$
$= [x(s)\neq y(s), s]$
$* ([x(s)>y(s),(x(s)-y(s),y(s))] \cup [x(s)\leq y(s),(x(s),y(s)-x(s))])$
$* f$
$= [x(s)\neq y(s),s] * [x(s)>y(s),(x(s)-y(s),y(s))] * f$
$\cup$
$[x(s)\neq y(s),s] * [x(s)<y(s),(x(s),y(s)-x(s))] * f$
$= [x(s)>y(s), (pgcd(x(s)-y(s),y(s)),pgcd(x(s)-y(s),y(s)))]$
$\cup$
$[x(s)<y(s), (pgcd(x(s),y(s)-x(s)),pgcd(x(s),y(s)-x(s)))]$
En utilisant les identités connues sur la fonction pgcd, on peut écrire
$= [x(s)>y(s),(pgcd(s),pgcd(s))] \cup [x(s)<y(s),(pgcd(s),pgcd(s))]$
$= [x(s)\neq y(s), (pgcd(s),pgcd(s))]$
$= I(x(s)\neq y(s)) * f$.

p2: $dom(f)=S$; or, w termine normalement pour tout s, donc $dom([w])=S$.

Notez qu'il aurait été très difficile de tenter de calculer $(I(t)*[b])^*$ dans le cas de ce programme. []

## 2.5. Simultanéité

Dans le cadre de cette thèse, on se contentera d'étudier trois figures simples de la programmation parallèle, à savoir

- Le parallélisme entre deux processus qui n'interfèrent pas l'un avec l'autre. Le but d'étudier cette figure est essentiellement théorique: On fera le lien entre cette figure de programmation et une opération relationnelle simple.

- La synchronisation de deux processus via une variable booléenne partagée; à cet effet on utilisera la structure de contrôle (**attendre ... puis**) qui est traduite de [OWIC76]: **await ... then**. On suppose que sa sémantique exacte est connue, mais en donnons une description sommaire ici: Le processus exécutant (**attendre** t **puis** b) attends que la condition t soit vérifiée (par une modification appropriée de la part d'un autre processus partageant la variable) ensuite il exécute le segment b de façon indivisible.

- **La communication** entre deux processus à l'aide d'un mécanisme simple par lequel l'émetteur envoie une donnée au récepteur, attend que celui-ci l'utilise, ensuite reprends son exécution.

**Parallélisme sans Interférence.** L'étude du parallélisme nécessite que l'on généralise la notion d'abstraction fonctionnelle. En effet, dans un contexte mono-processus, l'abstraction fonctionnelle d'un programme est une fonction (déterministe) car le processus contrôle l'état de toutes les variables, même celles qu'il n'affecte pas (et qui restent alors inchangées). Par contre, dans un contexte multi-processus, chaque processus ne contrôle que certaines variables qui lui sont propres et ne peut avoir d'effet sur les variables affectées à d'autres processus; donc son abstraction fonctionnelle est une relation potentiellement non-déterministe. Ainsi, soit

```
S = ens
      x, y: entier
    fin
```

et

p = (x:=x+1).

Dans un contexte mono-processus, l'abstraction fonctionnelle de p est la relation déterministe (fonction)

$[p]_1 = \{(s,s') \mid x(s')=x(s)+1 \ \& \ y(s')=y(s)\}$

alors que dans un contexte multi-processus (où il existe un processus p' qui contôle y), l'abstraction fonctionnelle de p est la relation (non-déterministe)

$[p]_2 = \{(s,s') \mid x(s')=x(s)+1\}$.

On définit formellement l'abstraction fonctionnelle dans le contexte multi-processus de la façon suivante: On considère le processus p sur l'espace S et on appelle S1 l'espace défini par les variables affectées à p; il existe S2 tel que S = S1xS2; l'abstraction fonctionnelle de p est la relation (potentiellement non-déterministe)

[p] = {((s1,s2),(s1',s2')) | Si l'exécution de p commence dans un état où s1 est la valeur des variables affectées à p alors il exécute normalement et finit dans un état où s1' est la valeur des variables affectées à p}.

La sémantique du parallélisme sans interférence est définie par un axiome,

**L'axiome du Parallélisme**

[cobegin p1, p2 coend] = [p1] ⊗ [p2].

Si p1 contrôle un ensemble de variables S1 et p2 contrôle un ensemble de variables S2 alors l'exécution simultanée (et indépendante, c-a-d sans interférence) de p1 et de p2 sur l'espace S=S1xS2 donne un effet qui est la conjonction des effets de p1 et p2. L'exemple simple suivant illustre cet axiome.

**Exemple**

```
S = ens
      x, y:  entier
    fin,
p1 = (x:=x+1),
p2 = (y:=y-1).
```

Alors on a

$[p1] = \{(s,s') \mid x(s')=x(s)+1\}$; $[p2] = \{(s,s') \mid y(s')=y(s)-1\}$.

[cobegin p1, p2 coend] = [p1] O [p2]
= {(s,s') | x(s')=x(s)+1 & y(s')=y(s)-1}. []

**Synchronisation.** On considère deux processus p1 et p2 opérant sur les espaces S1 et S2 (respectivement) et on pose S=S1xS2. On a

```
p1 = début                          p2 = début

     p11;
     b:=vrai;                             attendre b puis;
                                             p22
     fin,                                 fin,
```

où b est une variable booléenne partagée, initialement fausse; le contrôle de l'accès à la variable b est assuré par la sémantique de l'instruction (**attendre ... puis**). Le segment p11 n'accède à et ne modifie que des variables de S1; le segment p22 accède non seulement aux variables de S2 mais aussi à celles de S1, et il ne modifie que les variables de S2. L'objet de cette synchronisation est de s'assurer que p22 ne lit les variables de S1 que quand p11 les a affectées.

• La sémantique de la synchronisation est définie par l'axiome suivant:

**L'axiome de synchronisation**

[**codébut**
**début p11; b:=vrai fin, début attendre b puis; p22 fin**
**cofin**]
= ([p11]OI'(S2)) * (I'(S1)O[p22]),
où I'(S1) et I'(S2) sont les relations sur S=S1xS2 définies par
I'(S1) = {((s1,s2),(s1',s2')) | s1=s1'}.
I'(S2) = {((s1,s2),(s1',s2')) | s2=s2'}.

En effet I'(S1) -une pseudo-identité sur S1- est l'abstraction fonctionnelle de l'instruction vide qui se trouve entre b:=vrai et **fin** dans p1; de même I'(S2) est l'abstraction fonctionnelle de l'instruction vide qui apparaît entre **début** et **attendre** dans p2.

**Exemple**

```
S1 = entier;  S2 = entier;
S = ens
   x:  S1;
   y:  S2
fin.
```

```
p1 :  début                          p2 = début
      x:=3*x+1
      b:= vrai;                            attendre b puis ;
                                           y:=y+x-1
      fin.                                 fin.
```

D'après l'axiome de synchronisation, on a

[codébut p1, p2 cofin]
= ([x:=3*x+1]OI (S2)) * (I'(S1)O[y:=y+x-1])
= ({(s,s') | x(s')=3*x(s)+1}O{(s,s') | y(s')=y(s)})
  * ({(s,s') | x(s')=x(s)}O{(s,s') | y(s')=y(s)+x(s)-1})
= {(s,s') | x(s')=3*x(s)+1 & y(s')=y(s)}
  * {(s,s') | x(s')=x(s) & y(s')=y(s)+x(s)-1}
= {(s,s') | ∃s": x(s")=3*x(s)+1 & y(s")=y(s) &

$$x(s')=x(s'') \ \&\ y(s')=y(s'')+x(s'')-1\}$$
$$= \{(s,s') \mid x(s')=3*x(s)+1 \ \&\ y(s')=y(s)+3*x(s)\} \qquad []$$

Communication. On considère deux processus p1 et p2 opérant sur les espaces S1 et S2 (respectivement) et on pose S=S1xS2.

```
p1: début                          p2: début
    p11;                               p21;
    envoi:=vrai;                       attendre envoi puis ;
                                       l2;
    attendre retour puis;              retour:=vrai;
    p12                                p22
    fin.                               fin.
```

où envoi et retour sont des variables booléennes initialisées à faux. Les segments p11 et p12 n'accèdent à et ne modifient que S1; les segments p21 et p22 n'accèdent à et ne modifient que S2; le segment l2 (lecture) accède à S1 et S2 mais ne modifie que S2.

**Axiome de Communication.**

```
[codébut
    début p11; envoi:=vrai; attendre retour puis ; p12 fin,
    début p21; attendre envoi puis ; l2; retour:=vrai; p22
                                                    cofin]
= ([p11]O[p21])*(I'(S1)O[l2])*([p12]O[p22]).
```

**Exemple**

```
S1 = entier; S2 = entier;
S = ens
   x:  S1;
   y:  S2
fin.
```

```
p1:  début                         p2:  début
     x:=4;                              y:=y-2;
     envoi:=vrai;                       attendre envoi puis;
                                        y:=2*y+x;
     attendre retour puis               retour:=vrai;
     x:=2*x                             y:= y mod 2
     fin.                               fin.
```

On a

```
[codébut p1, p2 cofin]
= ([x:=4]O[y:=y-2])*(I'(S1)O[y:=2*y+x])*([x:=2*x]O[y:=ymod2])
= {(s,s') | x(s')=4 & y(s')=y(s)-2}
            * {(s,s') | x(s')=x(s) & y(s')=2*y(s)+x(s)}
                          * {(s,s') | x(s')=2*x(s) & y(s')=y(s)mod2}
= {(s,s') | x(s')=4 & y(s')=2*y(s)}
                          * {(s,s') | x(s')=2*x(s) & y(s')=y(s)mod2}
= {(s,s') | x(s')=8 & y(s')=0}.                                  []
```

Pour conclure cette section sur la simultanéité, rappelons-en les principaux résultats:

- En mode multi-processus, l'abstraction fonctionnelle d'une instruction Pascal est une relation potentiellement non-déterministe.
- Le parallélisme peut être formalisé par l'opération relationnelle d'intersection.

- La synchronisation et la communication peuvent être formalisées en utilisant les opérations relationnelles de produit relatif et intersection.

## 2.6. Appel de Procédure

Dans cette section, nous étudions l'appel de procédure sans paramètres. L'étude du passage de paramètres se fera dans la section suivante.

On considère un programme P sur l'espace S et une procédure déclarée dans P sous la forme suivante:

```
procédure p;
type
   T = <déclaration de type>;
var
   t: T;
début
   <code impliquant des variables de S
    et des variables de T>
fin.
```

Soit c le corps de procédure de la procédure p (le code inclus entre **début** et **fin**); l'espace de c est SxT où S et T sont considérés (non pas comme de simples ensembles mais) comme des relations (à multiplicité quelconque) et x représente le produit cartésien de relations (défini au chapitre 1); [p] est une fonction sur S alors que [c] est une fonction sur SxT. La sémantique de l'appel de procédure est définie par un axiome,

**L'axiome de l'appel de procédure**
Soit la procédure
**procédure p; var t:T; début c fin.**
Alors
$[p] = \{(s,s') \mid \exists t,t' : ((s,t),(s',t')) \in [c]\}$.

Quand la procédure p est invoquée, l'état du programme (s) est augmenté de la composante t (initialement indéterminée); une opération de produit cartésien est effectuée sur l'espace S. L'exécution de c transforme (s,t) en (s',t'); enfin le retour de procédure applique une opération de projection (au sens relationnel: voir chapitre 1) de SxT sur la composante S.

Exemple

```
S = ens
   a, b, c: entier
fin,
procédure puissances;
var t: entier;
début t:=a; b:=t*t; t:=b; c:=t*t fin.
```

On a
$[c] = \{((s,t),(s',t')) \mid a(s')=a(s) \ \&\ b(s')=t' \ \&\ c(s')=t'^2 \ \&\ t'=a(s)^2\}$.

Donc

$$[p] = \{(s,s') \mid \exists t,t' :\ a(s')=a(s)\ \&\ b(s')=t'\ \&\ c(s')=t'^2\ \&\ t'=a(s)^2\}$$
$$= \{(s,s') \mid a(s')=a(s)\ \&\ b(s')=a(s)^2\ \&\ c(s')=a(s)^4\ \&\ \exists t,t' :\ t'=a(s)^2\}$$
$$= \{(s,s') \mid a(s')=a(s)\ \&\ b(s')=a(s)^2\ \&\ c(s')=a(s)^4\}.$$ []

## 2.7. Passage de Paramètres

### 2.7.1. Appel par Valeur

On considère un programme P sur l'espace S et une procédure déclarée dans P sous la forme suivante:

```
procédure p(t: T);
type
   T=<déclaration de type>;
var
   t:T;
début
c
fin.
```

Dans un mode d'appel par valeur, la procédure p initialise la (les) variable(s) t' à partir de l'état s∈S; au cours de son exécution, la procédure p est susceptible d'affecter l'état du programme P; quand la procédure p finit d'exécuter, les états $\underline{t}$ et t sont simplement ignorés. En termes relationnels, l'invocation de la procédure a pour effet d'affecter à $\underline{T}'$ une projection de S (définie par les paramètres réels) ensuite d'effectuer le produit cartésien $S\times\underline{T}'\times T$; l'exécution de c a pour effet d'appliquer la fonction [c] sur l'état $(s,\underline{t}',t)$ de l'espace $S\times\underline{T}'\times T$; le retour de la procédure a pour effet d'effectuer une projection de $S\times\underline{T}'\times T$ sur S.

**L'axiome du Passage de Paramètres par Valeur**

Soit la procédure

```
procédure p(t:T);
type T = <type>; var t:T; début c fin.
```

Alors

$$[p(r(s))] = \{(s,s') \mid \exists \underline{t},t,\underline{t}',t' :\ \underline{t}=r(s)\ \&\ ((s,\underline{t},t),(s',\underline{t}',t')) \in [c]\},$$

où r(s) est la (séquence de) composante(s) de s correspondant aux paramètres réels.

**Exemple**

```
S = ens
   a, b, c: entier
fin,
procédure moyenne(x,y: entier);
var k1, k2:  réel;
début k1:=2.1; k2:=1.9; c:=(k1*x+k2*y)/(k1+k2) fin.
```

On calculera [p(a,b)]

$= \{(s,s') \mid \exists (x,y),(k1,k2),(x',y'),(k1',k2'):\ x=a(s)\ \&\ y=b(s)\ \&\ ((s,x,y,k1,k2),(s',x',y',k1',k2')) \in [c]\}$

$= \{(s,s') \mid \exists (x,y),(k1,k2),(x',y'),(k1',k2'):\ x=a(s)\ \&\ y=b(s)\ \&\ a(s')=a(s)\ \&\ b(s')=b(s)\ \&\ c(s')=(k1'*x+k2'*y)/4\ \&\ x'=x\ \&\ y'=y\ \&\ k1'=2.1\ \&\ k2'=1.9\}$

$= \{(s,s') \mid a(s')=a(s)\ \&\ b(s')=b(s)\ \&\ c(s')=(2.1*a(s)+1.9*b(s))/4\ \&\ \exists (x,y),(k1,k2),(x',y'),(k1',k2'):\ x=a(s)\ \&\ y=b(s)\ \&\ x'=x\ \&\ y'=y\ \&\ k1'=2.1\ \&\ k2'=1.9\}$

$= \{(s,s') \mid a(s')=a(s)\ \&\ b(s')=b(s)\ \&\ c(s')=(2.1*a(s)+1.9*b(s))/4\}.$

[]

## 2.7.2. Appel par Valeur-Nom

On considère un programme sur l'espace S et une procédure déclarée dans P sous la forme suivante:

**procédure** p(**var** t:T);
**type** T = <type>; **var** t:T;
**début** c **fin.**

Dans un modèle d'appel par valeur-nom, la procédure p initialise la (les) variable(s) t à partir des paramètres réels pris dans l'état $s \in S$ et noue une identité de nom entre les paramètres formels et les paramètres réels; au cours de son exécution, la procédure p affecte l'état $s \in S$ soit par des références explicites aux variables de s, soit par des références implicites -via l'identité de nom qui est établie entre les paramètres réels et les paramètres formels; quand la procédure p finit d'exécuter, t est ignoré sans laisser de trace alors que t est ignoré en laissant sa valeur dans les paramètres réels avec lesquels p était invoquée.

Le modèle relationnel que nous proposons peut rendre compte de tous les mécanismes de l'appel par valeur-nom sauf l'identité de nom que nous avons mentionnée ci-dessus; celle-ci sera prise en compte de façon artificielle lors de l'évaluation de [c]. L'invocation de la procédure p effectue une jointure entre S et T dont les composantes sont les paramètres formels pour S et réels pour T, suivie d'un produit cartésien par T, pour donner l'espace (S **join** T) x T; l'exécution de p applique [c] sur l'espace global ainsi obtenu et le retour de procédure effectue une projection de cet espace global sur S.

**L'axiome du Passage de Paramètres par Valeur-Nom**
Soit la procédure
**procédure** p(**var** t:T);
**type** T = <type>; **var** t:T; **début** c **fin.**
Alors on a
$[p(r(s))] = \{(s,s') \mid \exists t,t':\ ((s,t),(s',t')) \in [c(\hat{}t,!r(s))]$,
où $c(\hat{}t,!r(s))$ est le programme obtenu en remplaçant dans c tous les paramètres formels (t) par des paramètres réels (r(s)).

**Exemple**

```
    S = ens
       x, y, z: entier
    fin.
    procédure tri(var x,y: entier);
    var t:entier;
    début si x>y alors début t:=x; x:=y; y:=t fin fin.
```

Soit à calculer [tri(x,z)]. On a

```
    c(^(x,y),!(x,z)) = si x>z alors début t:=x; x:=z; z:=t fin.
```

On en déduit

```
    [c(^(x,y),!(x,z))]
    = {((s,t),(s',t')) |  x(s)≤z(s) & s'=s & t'=t}
      U
      {((s,t),(s',t')) |  x(s)>z(s) & x(s')=z(s) & y(s')=y(s) &
                          z(s')=x(s) & t'=x(s)}
```

Alors,

```
    [tri(x,z)]
    = {(s,s') |  ∃t,t': ((s,t),(s',t'))∈[c(^(x,y),!(x,z))]}
    = {(s,s') |  x(s)≤z(s) & s'=s}
      U
      {(s,s') |  x(s)>z(s) & x(s')=z(s) & y(s)=y(s) & z(s')=x(s)}.    []
```

## 2.8. Récursion

Dans cette section nous montrons brièvement comment on peut définir la sémantique d'une **fonction** (au sens de Pascal) récursive. Notez que si une fonction (au sens mathématique) a tous ses arguments dans un ensemble A et toutes ses images dans un ensemble B, on peut supposer qu'elle a tous ses arguments et images dans un ensemble unique, en l'ocurrence S=AUB. Donc c'est sans perte de généralité que nous restreignons notre domaine d'étude aux **fonctions** Pascal dont l'entête s'écrit comme suit:

**fonction** f (s:S): S.

De plus, nous adoptons le format suivant pour le corps de procédure:

**début si** t **alors** f:=E(s) **sinon** f:=K(f)(s) **fin**,

où E(s) est une expression sur S et K une fonctionnelle sur S (donc K(f) est une fonction sur S). Nous transformons la déclaration de fonction récursive ci-dessus en l'équation fonctionnelle suivante:

$$f = I(t)*e \cup I(\neg t)*K(f),$$

où e=[∈def(E),E]. Si on pose

$$H = \{(f,f') \mid f' = I(t)*e \cup I(\neg t)*K(f)\}$$

alors cette équation se réduit simplement à

$$f=H(f).$$

**Axiome de la Récursion**

Soit f la **fonction** récursive définie par

**fonction** f(s:S):S; **début si** t **alors** f:=E(s) **sinon** K(f)(s) **fin.**

Alors

[f] = La fonction la moins-définie qui vérifie f=H(f), si elle existe.

Notez que sur l'ensemble des fonctions, <u>moins-défini</u> est synonyme de <u>inclus</u>. Donc l'axiome ci-dessus peut être reformulé comme: la plus

petite fonction vérifiant f=H(f). Par abus de notation, on utilisera le même nom (f) pour représenter la fonction Pascal à traiter et l'inconnue de l'équation f=H(f).

L'application de cet axiome est très peu pratique, même pour des fonctions simples; il est commun d'utiliser des méthodes inductives pour analyser les propriétés des fonctions récursives.

**Exemple**

```
fonction f(s:entier):entier;
début si faux alors f:=0 sinon f:=f(s) fin.
```

Equation récursive dérivée:

$$f = I(faux)*[vrai,0] \cup I(vrai)*f$$

<=>

$$f=f.$$

Tout f est solution de cette équation; le moins-défini est Ø. []

**Exemple**

```
fonction f(s:entier):entier;
début si s=0 alors f:=1 sinon f:=f(s) fin.
```

Equation dérivée:

$$f = I(s=0)*[vrai,1] \cup I(s\neq 0)*f$$

<=>

$$f = \{(0,1)\} \cup I(s\neq 0)*f.$$

Toute fonction contenant (0,1) est solution de cette équation; la moins-définie parmi toutes ces solutions est la fonction {(0,1)}. []

## 3. Abstraction Fonctionnelle: Un exemple

Pour illustrer l'usage des axiomes présentés dans la section précédente, nous donnons ici un exemple d'application. Il s'agit d'un programme qui détermine si deux tableaux a et b ont un élément en commun.

```
programme intersection (input, output);
var
   a: tableau [1..n] de item;
   b: tableau [0..m] de item;
   i: 1..n; x: item; u: booléen (*trouvé*);
   procédure chercher (y: item);
   var j:0..m;
   début j:=m; b[0]:=y;
   tantque b[j]≠y faire j:=j-1;
   u:= u v (j≠0)
   fin;
début
i:=1; u:=faux;
tantque i≤n faire
   début
   x:=a[i];
   chercher(x);
   i:=i+1
   fin
fin.
```

Nous représentons par a[i..j] la séquence (a[i] .. a[j]) si i≤j ou la séquence vide si i>j. De plus, si x est un item et a[i..j] un sous-tableau, on représente par x.a[i..j] la concaténation de x au sous-tableau a[i..j].

On commence par calculer l'abstraction fonctionnelle du corps de la procédure chercher. L'espace de cette procédure est

```
S = ens
   a: tableau [1..n] de item;
   b: tableau [0..m] de item;
   i: 1..n; j:0..m;
   x,y: item; u: booléen
fin.
```

```
[j:=m; b[0]:=y]
= {(s,s'): a(s')=a(s) & b(s')=y.b(s)[1..m] & i(s')=i(s) & j(s')=m &
          x(s')=x(s) & y(s')=y(s) & u(s')=u(s)}.
= [vrai, (a(s),y.b(s)[1..m],i(s),m,x(s),y(s),u(s))].
```

Pour calculer l'abstraction fonctionnelle de la boucle tantque, il faut proposer une fonction candidate f, ensuite vérifier les prémisses p0, p1 et p2 du théorème de l'instruction tantque. On pose

```
f = {(s,s'): (0≤j(s)≤m & (∃k: 0≤k≤j(s) & b(s)[k]=y(s))) &
             a(s')=a(s) & b(s')=b(s) & i(s')=i(s) & j(s')=h(s) &
             x(s')=x(s) & y(s')=y(s) & u(s')=u(s)},
```

où h(s) est le plus grand indice compris entre 0 et j(s) (inclus) tel que b(s)[h(s)]=y(s). La première clause (entre parentèses) définit le domaine de la fonction f (pour une interprétation de cette clause, voir la preuve de p2); les autres clauses expriment que j prend la valeur h(s) alors que toutes les autres variables restent inchangées.

**prémisse p0.** Soit s un état vérifiant b(s)[j(s)]=y(s). Il est simple de vérifier que l'on a alors (s,s)∈f.

**prémisse p1.** Soit s un état vérifiant b(s)[j(s)]≠y(s). On doit montrer que ses images par f et par [j:=j-1]*f sont identiques. Ceci résulte immédiatement de la définition de h(s) et de l'hypothèse b(s)[j(s)]≠y(s): Si b(s)[j(s)]≠y(s) alors le plus grand indice h(s) entre 0 et j(s) tel que b(s)[h(s)]=y(s) est identique au plus grand indice entre 0 et j(s)-1 tel que b(s)[h(s)]=y(s).

**prémisse p2.** Le domaine de f est
{s| 0≤j(s)≤m & (∃k: 0≤k≤j(s) & b(s)[k]=y(s))}.
En effet, la boucle tantque ne termine normalement que si j(s) est un indice légal (0≤ ≤m) et que la condition d'arrêt est garantie d'être vérifiée après un nombre fini d'itérations.

Nous avons donc
[**tantque** b[j]≠y **faire** j:=j-1]
= {(s,s')| 0≤j(s)≤m & (∃k: 0≤k≤j(s) & b(s)[k]=y(s)) &
a(s')=a(s) & b(s')=b(s) & i(s')=i(s) & j(s')=h(s) &
x(s')=x(s) & y(s')=y(s) & u(s')=u(s)},
que nous écrivons encore sous forme pE comme
= [0≤j(s)≤m & (∃k: 0≤k≤j(s) & b(s)[k]=y(s)),
(a(s), b(s), i(s), h(s), x(s), y(s), u(s))].

Il résulte de l'axiome de la séquence que
[j:=m; b[0]:=y; **tantque** b[j]≠y **faire** j:=j-1]
= [vrai, (a(s), y(s).b(s)[1..m], i(s), m, x(s), y(s), u(s))]
* [0≤j(s)≤m & (∃k: 0≤k≤j(s) & b(s)[k]=y(s)),
(a(s), b(s), i(s), h(s), x(s), y(s), u(s))].
On utilise la formule du produit relatif:
= [vrai & 0≤m≤m & (∃k: 0≤k≤m & (y(s).b(s)[1..m])[k]=y(s)),
(a(s), y(s).b(s)[1..m], i(s), H(s), x(s), y(s), u(s))],
où H(s) est le plus grand indice entre 0 et m tel que b(s)[k]=y(s). Si on considère que (0≤m≤m) est vrai et que k=0 vérifie la condition (0≤0≤m & (y(s).b(s)[1..m])[0]=y(s)), on écrit:
= [vrai, (a(s), y(s).b(s)[1..m], i(s), H(s), x(s), y(s), u(s))].
Par ailleurs, on a
[u:= (u v j≠0)]
= [vrai, (a(s), b(s), i(s), j(s), x(s), y(s), u(s) v (j(s)≠0))].
On en déduit, d'après la règle de la séquence et la formule du produit relatif:
[j:=m; b[0]:=y; **tantque** b[j]≠y **faire** j:=j-1; u:=uv(j≠0)]
= [vrai,
(a(s), y(s).b(s)[1..m], i(s), H(s), x(s), y(s), u(s)v(H(s)≠0)].
L'expression H(s)≠0 s'interprète comme suit: le plus grand indice entre 0 et m tel que b(s)[H(s)]=y(s) est différent de zéro. Ceci est équivalent à y(s)∈b(s)[1..m]. Si bien que, si on appelle c le corps

de la procédure chercher, on a

```
[c]
=  [vrai,(a(s),y(s).b(s)[1..m], i(s), H(s), x(s), y(s),
                                          u(s) v y(s)∈b(s)[1..m])].
```

ou encore,

```
= {(s,s') |  a(s')=a(s) & b(s')=y(s).b(s)[1..m] & i(s')=i(s) &
             j(s')=H(s) & x(s')=x(s) & y(s')=y(s) &
             u(s') = u(s) v y(s)∈b(s)[1..m]}.
```

Il nous appartient présentement de calculer l'abstraction fonctionnelle de l'appel de procédure chercher(x), avec passage de paramètres par valeur. L'état du programme intersection dans lequel cet appel de procédure s'effectue est:

```
S = ens
      a:  tableau [1..n] de item;
      b:  tableau [0..m] de item;
      i:  1..n; x: item; u: booléen;
    fin.
```

Notez que cette définition de S entraine une modification dans l'interprétation de la variable s utilisée jusques là: On ne pourra plus parler, désormais, de j(s) ni de y(s); conformément aux notations introduites dans l'axiome du passage de paramètres par valeur, y(s) sera représenté par t et j(s) par $\underline{t}$. On a

```
[chercher(x)]
= {(s,s') |  ∃t,t,t',t':  x(s)=t & ((s,t,t),(s',t',t'))∈[c]}
= {(s,s') |  ∃t,t,t',t':  t=x(s) & a(s')=a(s) & b(s')=t.b(s)[1..m] &
                i(s')=i(s) & t'=H(s) & x(s')=x(s) & t=t' &
                u(s')= u(s) v t∈b(s)[1..m]}
= {(s,s') |  a(s')=a(s) & b(s')=x(s).b(s)[1..m] & i(s')=i(s) &
             x(s')=x(s) & u(s')= u(s) v x(s)∈b(s)[1..m] &
             (∃t,t,t',t':  t=x(s) & t'=H(s) & t=t')}
= {(s,s') |  a(s')=a(s) & b(s')=x(s).b(s)[1..m] & i(s')=i(s) &
             x(s')=x(s) & u(s') = u(s) v x(s)∈b(s)[1..m])}.
```

Sous forme pE, ceci s'écrit:

```
= [vrai, (a(s), x(s).b(s)[1..m], i(s), x(s), u(s)vx(s)∈b(s)[1..m])].
```

D'autre part,

```
[x:=a[i]] = [vrai, (a(s), b(s), i(s), a(s)[i(s)], u(s))]
et
[i:=i+1]  = [vrai, (a(s), b(s), i(s)+1, x(s), u(s))].
```

Donc

```
[x:=a[i]; chercher(x); i:=i+1]
= [vrai, (a(s), b(s), i(s), a(s)[i(s)], u(s))]
* [vrai, (a(s), x(s).b(s)[1..m], i(s), x(s), u(s)vx(s)∈b(s)[1..m])]
                                * [vrai, (a(s), b(s), i(s)+1, x(s), u(s))]
= [vrai, (a(s), a(s)[i(s)].b(s)[1..m], i(s), a(s)[i(s)],
                                          u(s)va(s)[i(s)]∈b(s)[1..m])]
                                * [vrai, (a(s), b(s), i(s)+1, x(s), u(s))]
= [vrai, (a(s), a(s)[i(s)].b(s)[1..m], i(s)+1, a(s)[i(s)],
                                        u(s)va(s)[i(s)]∈b(s)[1..m])].
```

Pour calculer l'abstraction fonctionnelle de la boucle tantque,

on doit proposer une fonction candidate pour laquelle on vérifiera les prémisses p0, p1 et p2. On propose

$$f = \{(s,s') \mid 0 \le i(s) \le n \ \&\ a(s')=a(s)\ \&\ b(s')=a(s)[n].b(s)[1..m]\ \&\ i(s')=n+1\ \&\ x(s')=a(s)[n]\ \&\ u(s') = u(s)\ v\ (\exists k:\ i(s) \le k \le n\ \&\ a(s)[k] \in b(s)[1..m])\}$$
$$\cup$$
$$\{(s,s') \mid i(s)>n\ \&\ s'=s\}.$$

On écrit la formule pE de cette fonction comme suit

$$f = [0 \le i(s) \le n,\ (a(s),\ a(s)[n].b(s)[1..m],\ n+1,\ a(s)[n],\ u(s)\ v\ (\exists k:\ i(s) \le k \le n\ \&\ a(s)[k] \in b(s)[1..m])]$$
$$\cup$$
$$[i(s)>n,\ s].$$

**prémisse p0.** Il est clair que $I(i(s)>n)*f = I(i(s)>n)$.

**prémisse p1.** On doit prouver que

$$I(i(s) \le n)*[x:=a[i];\ \text{chercher}(x);\ i:=i+1]*f = I(i(s) \le n)*f.$$

On a

$$I(i(s) \le n) * [x:=a[i];\ \text{chercher}(x);\ i:=i+1] * f$$
$$= [i(s) \le n,\ (a(s),\ a(s)[i(s)].b(s)[1..m],\ i(s)+1,\ a(s)[i(s)],\ u(s) v a(s)[i(s)] \in b(s)[1..m])]$$
$$* [0 \le i(s) \le m,\ (a(s),\ a(s)[n].b(s)[1..m],\ n+1,\ a(s)[n],\ u(s)\ v\ (\exists k:\ i(s) \le k \le n\ \&\ a(s)[k] \in b(s)[1..m])]$$
$$\cup$$
$$[i(s) \le n\ \&\ i(s)+1>n,\ (a(s),\ a(s)[i(s)].b(s)[1..m],\ i(s)+1,\ a(s)[i(s)],\ u(s)\ v\ a(s)[i(s)] \in b(s)[1..m])]$$
$$= [i(s) \le n\ \&\ 0 \le i(s)+1 \le n,\ (a(s),\ a(s)[n].b(s)[1..m],\ n+1,\ a(s)[n],\ u(s)\ v\ a(s)[i(s)] \in b(s)[1..m]\ v\ (\exists k:\ i(s)+1 \le k \le n\ \&\ a(s)[k] \in d(s)[1..m])]$$
$$\cup$$
$$[i(s)=n,\ (a(s),\ a(s)[n].b(s)[1..m],\ n+1,\ a(s)[n],\ u(s)\ v\ a(s)[n] \in b(s)[1..m])]$$
$$= [0 \le i(s) \le n-1,\ (a(s),\ a(s)[n].b(s)[1..m],\ n+1,\ a(s)[n],\ u(s)\ v\ (\exists k:\ i(s) \le k \le n\ \&\ a(s)[k] \in b(s)[1..m])]$$
$$\cup$$
$$[i(s)=n,\ (a(s),\ a(s)[n].b(s)[1..m],\ n+1,\ a(s)[n],\ u(s)\ v\ (\exists k:\ i(s) \le k \le n\ \&\ a(s)[k] \in b(s)[1..m])]$$
$$= [0 \le i(s) \le n,\ (a(s),\ a(s)[n].b(s)[1..m],\ n+1,\ a(s)[n],\ u(s)\ v\ (\exists k:\ i(s) \le k \le n\ \&\ a(s)[k] \in b(s)[1..m])]$$
$$= I(i(s) \le n)*f.$$

**prémisse p2.** Le domaine de f est S; par ailleurs, il est clair que la boucle tantque termine normalement pour tout état initial.

On considère le segment d'initialisation:

$$[i:=1;\ u:=\text{faux}]$$
$$= [\text{vrai},\ (a(s),\ b(s),\ 1,\ x(s),\ \text{faux})].$$

On applique l'axiome de la séquence pour calculer l'abstraction fonctionnelle du programme en entier.

$$[\text{vrai},\ (a(s),\ b(s),\ 1,\ x(s),\ \text{faux})]$$

$$* [0 \leq i(s) \leq n,\ (a(s),\ a(s)[n].b(s)[1..m],\ n+1,\ a(s)[n],\ u(s) \vee (\exists k:\ i(s) \leq k \leq n \ \&\ a(s)[k] \in b(s)[1..m]]$$

$$\cup$$

$$[\textbf{vrai},\ (a(s),\ b(s),\ 1,\ x(s),\ \textbf{faux})] * [i(s) > n,\ s]$$

$$= [0 \leq i \leq n,\ (a(s),\ a(s)[n].b(s)[1..m],\ n+1,\ a(s)[n],\ (\exists k:\ 1 \leq k \leq n \ \&\ a(s)[k] \in b(s)[1..m])]$$

$$\cup$$

$$[i > n,\ (a(s),\ b(s),\ 1,\ x(s),\ \textbf{faux})]$$

Puisque l'on suppose que n est plus grand ou égal à 1, on a

$$= [\textbf{vrai},\ (a(s),\ a(s)[n].b(s)[1..m],\ n+1,\ a(s)[n],\ (\exists k:\ 1 \leq k \leq n \ \&\ a(s)[k] \in b(s)[1..m])]$$

L'expression de la veleur finale de u peut être écrite plus élégamment, comme suit:

$$= [\textbf{vrai},\ (a(s),\ a(s)[n].b(s)[1..m],\ n+1,\ a(s)[n],\ a(s) \cap b(s)[1..m] \neq \emptyset)]$$

En termes ensemblistes, cette fonction s'écrit:

$$= \{(s,s') \mid a(s')=a(s) \ \&\ b(s')=a(s)[n].b(s)[1..m] \ \&\ i(s')=n+1 \ \&\ x(s')=a(s)[n] \ \&\ u(s') = (a(s) \cap b(s)[1..m] \neq \emptyset)\}.$$

En utilisant l'une ou l'autre de ces représentations, le lecteur se convaincra que cette fonction est effectivement la fonction du programme donné. []

Bien que relativement simple, ce programme a nécessité plusieurs pages de calcul. Parcequ'elle tente d'exhiber <u>toutes</u> les propriétés fonctionnelles d'un programme, cette méthode est d'usage difficile.

# Chapitre 4

## Fonctions Invariantes Plus Fortes: Analyse des Programmes Itératifs

Depuis leur introduction par Hoare en 1969, les invariants de boucle n'ont cessé de jouer un rôle central dans l'analyse -et la synthèse- de boucles tantque. Dans ce chapitre, nous présentons un autre outil d'analyse des boucles tantque: Les fonctions invariantes. La notion de fonction invariante fut introduite dans [MILI83p] et étudiée en détail dans [MILI85a]; elle est en quelque sorte une extension d'un effort par M. Caplain de générer des invariants de boucles généraux.

### 1. Fonctions Invariantes: Définition

Soit w = (**tantque** t **faire** b) un programme itératif sur un espace S. Nous allons prouver que l'on peut sans perte de généralité supposer que dom([w])=S. On fera cela à l'aide de deux propositions (dues à [MILI83i]).

**proposition 1.** Si w est un programme itératif sur S alors cod([w])$\subseteq$dom([w]).

**Preuve.** Soit s$\in$cod([w]); alors ~t(s), donc l'exécution de w sur s termine normalement (très rapidement, en fait); donc s$\in$dom([w]). CQFD

**Proposition 2.** Soit w un programme itératif sur S et soit s un état dans dom([w]). Alors tous les états générés à partir de s par les itérations consécutives de w sont dans dom([w]).

**Preuve.** On prouvera que si s$\in$dom([w]) et t(s) alors [b](s)$\in$dom([w]). Si s$\in$dom([w]) et t(s) alors s$\in$dom(I(t)*[w]). Or, si nous appliquons le théorème de l'instruction tantque (chapitre 3) en prenant [w] pour fonction f, nous déduisons d'après la clause pi que I(t)*[w] = I(t)*[b]*[w]. Il en résulte que si s$\in$dom(I(t)*[w]) alors [b](s)$\in$dom([w]). CQFD

Par définition de dom([w]), tous les états initiaux de w se trouvent dans dom([w]); la proposition 1 montre que tous les états finaux de w sont dans dom([w]) et la proposition 2 montre que tous les états intermédiares de w sont aussi dans dom([w]). Donc c'est sans perte de généralité que nous limiterons notre étude de w à dom([w]); autrement dit, nous poserons S=dom([w]), où S est l'espace sur lequel on étudie w. Quand on étudie une boucle tantque w sur un espace S, les deux ensembles qui nous intéressent sont alors S qui est dom([w]) et S|~t qui est cod([w]).

**Définition 1.** Une fonction invariante de w = (tantque t faire b)

est une fonction f sur S telle que:
a) dom(f)=S,
b) I(t)*f = I(t)*[b]*f.

Une fonction constante sur S est une fonction invariante pour tout programme itératif, mais n'a aucun intérêt. Une fonction invariante peut être arbitrairement insignifiante, c'est à dire porter arbitrairement peu d'information sur le programme itératif en question. Une fonction invariante est d'autant plus intéressante qu'elle discrimine plus entre ses arguments, c'est à dire qu'elle est plus-injective. D'où la définition suivante.

**Définition 2.** Soit w = (**tantque** t **faire** b) un programme itératif sur un espace S tel que dom([w])=S et soit f une fonction sur S. On dit que f est une plus forte fonction invariant (abbréviation: pfi) si et seulement si
a) f est une fonction invariante pour w,
b) si f' est une fonction invariante pour w alors I(t)*f est plus-injective que I(t)*f'.

Notez que l'on ne compare l'injectivité des fonctions f et f' que sur S!t. Vu que dom(f) et dom(f') sont tous deux égaux à S, les domaines de I(t)*f et I(t)*f' sont tous deux égaux à S!t.

Dans la section suivante on étudiera la génération automatique de pfi's et dans la section 3 on discutera la relation qui existe entre la pfi d'un programme itératif et son abstraction fonctionnelle.

## 2. Fonctions Invariantes: Leur Génération

On présente des formules de génération automatique de fonctions invariantes plus fortes à partir d'une analyse systématique des paramètres du programme itératif en question, à savoir son espace (S), son corps de boucle (b) et sa condition (t). Toutes les formules présentées ici concernent des espaces qui sont soit **réel** soit une puissance cartésienne de **réel**. Vu que les fonctions invariantes ont un domaine égal à S, leur pE-formule est de la forme [vrai,E]: on se contentera de les représenter par leur expression.

Dans les formules qui vont suivre nous avons besoin d'une fonction que l'on notera **fr** et que l'on définira pour tout x dans **réel** par **fr**(x)=x-e(x), où e(x) est le plus grand entier plus petit ou égal à x; ainsi, pour tout x positif, **fr**(x) est la partie fractionnaire de x, d'où son nom. On notera l'ensemble des réels positifs par R0 et l'ensemble des réels plus grands que 1 par R1.

### 2.1. Additionner Une Constante

**Proposition 3.** On considère w = (**tantque** s<a **faire** s:=s+c) sur l'espace S=**réel**, où c>0. Alors dom([w])=S et la fonction

$$f(s) = \mathbf{fr}(s/c)$$

est une plus forte fonction invariante pour w.

**Preuve.** Il est simple de vérifier que dom([w])=S et que f est une

fonction invariante. Pour prouver qu'elle est la plus forte, on se donne une fonction invariante f' et on prouve que I(s<a)*f est plus-injective que I(s<a)*f'. Soient deux états tels que s<a et s'<a et que f(s)=f(s'); alors

**fr**(s/c) = **fr**(s'/c)
=> s/c = s'/c + k
pour un certain entier k, que l'on supposera négatif ou nul, sans perte de généralité.
=> s = s' + k*c
=> f'(s) = f'(s'+k*c)
puisque k≤0 et s'<a alors s'+k*c<a; puisque f' est une fonction invariante, on peut déduire:
=> f'(s) = f'(s' + (k+1)*c)
on continue ainsi jusqu'à obtenir:
=> f'(s) = f'(s'). CQFD

**Exemple**
On prend S=**naturel** et w = (**tantque** s<21 **faire** s:=s+5). Alors

f(s) = **fr**(s/5)

est une plus forte fonction invariante pour w. Puisque S=**naturel**, on peut en fait définir une autre pfi plus simple, en l'ocurrence

f'(s) = s **mod** 5. []

## 2.2. Multiplier par une Constante

**Proposition 4.** On considère l'énoncé w = (**tantque** s<a **faire** s:=l*s) sur l'espace S=R0, où a>0 et l>1. Alors dom([w])=S et

f(s) = **fr**(log(s)/log(l))

est une plus forte fonction invariante.

**Preuve.** On utilise la proposition 3 en effectuant le changement de variable s:=log(s) et c:=log(l). CQFD

**Exemple**
On prends w = (**tantque** s<101 **faire** s:=2*s); alors **fr**(log(s)) est une plus forte fonction invariante. CQFD

## 2.3. Elever à la Puissance

**Proposition 5.** On pose S=R1 et w = (**tantque** s<a **faire** s:=s$^p$, où p>1 et s>0. Alors dom([w])=S et

f(s) = **fr**(log(log(s))/log(p))

est une plus forte fonction invariante.

**Preuve.** On utilise la proposition 4 avec le changement de variable s:=log(s). CQFD

**Exemple**
On prends w = (**tantque** s<100 **faire** s:=s$^4$) sur l'ensemble S=R1. Alors **fr**(log(log(s))/2) est une plus forte fonction invariante. On vérifie brièvement que f est une fonction invariante: Soit s<100,

f(s$^4$) = **fr**(log(log(s$^4$))/2)
= **fr**(log(4*log(s))/2)
= **fr**(log(4)/2+log(log(s))/2)

$= fr(1+\log(\log(s))/2)$
$= fr(\log(\log(s))/2).$ []

## 2.4. Forme Linéaire

**Proposition 6.** On pose S=R0 et w = (**tantque** s<a **faire** s:=l*s+c), où a>0, c>0 et l>1. Alors dom([w])=S et
$$f(s) = fr(\log(s+c/(l-1))/\log(l))$$
est une plus forte fonction invariante de f.

**Preuve.** On utilise la proposition 4 avec le changement de variable s:=s+c/(l-1), dû à Caplain [CAPL75]. CQFD

**Exemple**
On considère le programme w = (**tantque** s<120 **faire** s:=2*s+1). Alors fr(log(s+1)) est une plus forte fonction invariante. On se contentera de prouver que c'est une fonction invariante.
$f(2*s+1) = fr(\log(2*s+2))$
$= fr(\log(2*(s+1))$
$= fr(1+\log(s+1))$
$= fr(\log(s+1))$ []

## 2.5. Forme Monomiale

**Proposition 7.** Si on considère S=R1 et w = (**tantque** s<a **faire** $s:=l*s^p$), où l>1 et p>1, alors on a dom([w])=S et
$$f(s) = fr(\log(\log(s)+\log(l)/(p-1))/\log(p))$$
est une plus forte fonction invariante de w.

**Preuve.** On utilise la proposition 6 avec le changement de variable s:=log(s), p:=l et c:=log(l). CQFD

**Exemple**
On considère le programme w = (**tantque** s<120 **faire** $s:=4*s^2$). Alors
$$fr(\log(\log(s)+2))$$
est une plus forte fonction invariante de w. On prouvera seulement qu'elle est invariante.
$f(4*s^2) = fr(\log(\log(4*s^2)+2))$
$= fr(\log(\log(4)+2*\log(s)+2))$
$= fr(\log(2*(\log(s)+2)))$
$= fr(1+\log(\log(s)+2))$
$= fr(\log(\log(s)+2))$ []

## 2.6. Forme Linéaire Diagonale

**Proposition 8.** Soit S=R1 et
w = **tantque** $s_1$<a **faire**
**début**
$s_1 := l_1*s_1$;
$s_2 := l_2*s_2$;
... ... ...
$s_k := l_k*s_k$
**fin**,
où $l_i$>1 pour tout i. Alors dom([w])=S et

$f(s) = (fr(\log(s_1)/\log(l_1)),$
$\log(s_1)*\log(l_2)-\log(s_2)*\log(l_1),$
$\log(s_1)*\log(l_3)-\log(s_3)*\log(l_1),$
$\ldots \quad \ldots \quad \ldots,$
$\log(s_1)*\log(l_k)-\log(s_k)*\log(l_1))$
est une plus forte fonction invariante de w.

**Preuve.** Ce résultat ne sera pas prouvé formellement ici car la preuve en est lourde et peu intéressante. Le fait que f est invariante est facile à prouver; le fait qu'elle est plus forte est plus difficile. Cette formule est confirmée par des résultats donnés dans [CAPL75]. Voir aussi l'exemple suivant.

**Exemple**
On prend

**S = ens**
**crt**
a, b: **entier**;
**sse**
$b \geq 0$
**fin.**

w = (**tantque** b>0 **faire début** a:=a+1; b:=b-1 **fin.**
On fait le changement de variables $x:=2^a$ et $y:=2^{-b}$, ce qui donne

**S' = ens**
**crt**
x, y: **réel**;
**sse**
$x>0$ & $0<y\leq 1$
**fin.**

w' = (**tantque** y<1 **faire début** x:=2*x; y:=2*y **fin**).
La plus forte fonction invariante de w' est alors
f'(s') = (**fr**(log(x(s'))), log(x(s'))-log(y(s')))
= (**fr**(log(x(s'))), log(x(s')/y(s'))).
Nous retournons au système de variables (a,b):
f(s) = (**fr**(a(s)), (a(s)+b(s)))
puisque a(s)∈**naturel**, on simplifie
f(s) = (0,(a(s)+b(s))). []

Remarquez que des formules qui ne sont pas strictement diagonales telles que

$s_i := l_i + s_i$,
$s_i := s_i ** p_i$ (** désigne l'exponentiation)
$s_i := l_i * s_i ** p_i$, et
$s_i := l_i * s_i + c_i$

peuvent être rameneés à des formules linéaires diagonales moyennant des changements de variable simples. De plus, si le corps de boucle a la forme

$s := L*s$

où s est $(s_1, s_2, s_3, \ldots s_k)$ et L une matrice diagonalisable dont les valeurs propres sont plus grandes que 1 alors on peut effectuer le changement de variable

$x := M*s$

en utilisant la matrice de diagonalisation M et appliquer la proposition 8 au programme obtenu. Ainsi il est raisonnable de penser que la proposition 8 a potentiellement un domaine d'application vaste.

## 3. Fonctions Invariantes: Leur Usage

La première question qu'il nous appartient de clarifier est la relation qui existe entre les fonctions invariantes (dans le sens défini ici) et les invariants de boucle (dans le sens de la logique Floyd-Hoare). Un invariant de boucle n'est pas, comme on serait tenté de croire, une fonction invariante qui se trouve être un prédicat; plutôt, c'est un prédicat tel que, s'il est vrai avant l'exécution de b et que t est vrai alors il est vrai après. Ainsi un invariant de boucle peut être faux avant l'exécution de b mais devenir vrai après. Considérons, par exemple, le prédicat p(s)=(s>0) et le corps de boucle b=(s:=s+2); p(s) est un invariant de boucle, pourtant p(-1) -qui est faux- n'est pas égal à p([b](-1)) qui est vrai; donc p n'est pas invariant dans le sens entendu ici. Ceci suggèrerait que l'adjectif invariant n'est pas approprié pour décrire le comportement des invariants de boucle: On devrait plutôt les appeler des prédicats croissants (sur la base que **faux**<**vrai**) et utiliser l'adjectif invariant pour les fonctions (et les prédicats) qui sont véritablement invariants. Dans le but de clarifier notre présentation, nous introduisons les deux définitions suivantes:

> **Définition 3.** Soit w = (**tantque** t **faire** b) une instruction tantque sur un espace S et soit p un prédicat sur S. On dit que p est un prédicat croissant si et seulement si
> $$p(s) \;\&\; t(s) \Rightarrow p([b](s)).$$

Ceci est l'invariant de boucle au sens traditionnel.

> **Définition 4.** Soit w = (**tantque** t **faire** b) une instruction tantque sur un espace S et soit p un prédicat sur S. On dit que p est un prédicat invariant si et seulement si
> $$t(s) \Rightarrow p(s)=p([b](s)).$$

Ceci est une fonction invariante (à une fonction injective près) qui se trouve être un prédicat. Notez que si un prédicat est invariant, alors il est croissant.

> **Proposition 9.** Si f est une fonction invariante pour w = (**tantque** t **faire** b) et s0 un état quelconque de S alors
> $$p(s) = (f(s)=f(s0))$$
> est un prédicat invariant pour w.

**Preuve.** Soit s tel que t(s). Alors

$$\begin{aligned} p([b](s)) &= (f([b](s))=f(s0)) \\ &= (f(s)=f(s0)) \\ &= p(s). \end{aligned}$$

CQFD

Le prédicat invariant dérivé d'une fonction constante (qui est invariante) est p(s)=vrai; il est peu intéressant. A l'autre extrême se trouvent les prédicats invariants dérivés de plus fortes fonctions invariantes; on admettra sans preuve (la preuve en sera partiellement donnée dans le théorème 1) que le prédicat invariant dérivé de la

fonction invariante la plus forte a le privilège qu'il permet de prouver (par la méthode de Hoare) les propriétés les plus fortes de la boucle.

Etant un prédicat invariant, p(s) = (f(s)=f(s0)) est aussi un prédicat croissant, c'est à dire un invariant de boucle. La proposition ci-dessus affirme que, étant donnée une fonction invariante, on peut en générer un invariant de boucle. Il est intéressant de se demander si pour tout invariant de boucle p, il existe une fonction f et un état s0 tel que p(s) = (f(s)=f(s0)). Deux éléments de réponse sont donnés à cette question.

**Proposition 10.** Soit p un prédicat croissant différent de **faux**. Si p est invariant alors il existe une fonction invariante f et un état s0 tel que p(s) = (f(s)=f(s0)).

**Preuve.** On eput écrire le prédicat p comme p(s) = (p(s)=**vrai**); vu que p≠**faux**, il existe s0 tel que p(s0)=**vrai**; alors on écrit p(s) comme (p(s)=p(s0)). Puisque p est invariant, on a t(s) => (p(s)=p([b](s))). Soient $s_\vee$ et $s_\neq$ deux éléments de S et soit k la fonction

$$k = \{(\textbf{vrai}, s_\vee), (\textbf{faux}, s_\neq)\};$$

on pose f = p*k. Alors on a: premièrement f est une fonction sur S (C'est à dire dont les arguments et les images sont dans S); deuxièmement, f est une fonction invariante pour w; troisièmement, p(s) = (f(s)=f(s0)). CQFD

**Proposition 11.** Soit p un prédicat croissant. Si p n'est pas invariant, alors il n'existe pas de fonction invariante f telle que, pour un certain s0, p(s) = (f(s)=f(s0)).

**Preuve.** Supposons qu'il existe un tel s0 et une fonction f. Puisque p est croissant mais n'est pas invariant, il existe un état s1 dans Sit tel que

$$\sim p(s1) \;\&\; p([b](s1))$$

<=>

$$f(s) \neq f(s0) \;\&\; f([b](s)) = f(s0).$$

On en déduit f(s1)≠f([b](s1)), ce qui contredit l'hypothèse que t(s1) et que f est invariante. CQFD

Bien que nous ne l'ayons pas prouvé, nous avons l'intuition que tout invariant de boucle (au sens de Floyd-Hoare) peut en fait s'écrire comme la conjonction d'un prédicat invariant avec un prédicat croissant (l'un ou l'autre pouvant être le prédicat **vrai**- donc ne pas figurer explicitement). On trouve ci-dessous un exemple d'illustration.

**Exemple**

```
S = ens
crt
   a, b, c: entier;
sse
   vrai
fin.
w = (tantque b>0 faire
   début
```

```
        b:=b-1;
        c:=c+a
        fin.
```

Invariant de boucle:

$p(s) = (b(s) \geq 0 \ \& \ c(s)+a(s)*b(s) = c(s0)+a(s0)*b(s0))$.

Prédicat croissant:

$b(s) \geq 0$.

Prédicat invariant:

$c(s)+a(s)*b(s) = c(s0)+a(s0)*b(s0)$.

Notez que le prédicat invariant peut s'écrire $f(s)=f(s0)$ où $f$ est -par exemple- la fonction invariante [vrai, (0,0,c(s)+a(s)*b(s))]. Quant au prédicat croissant, il ne semble pas qu'il puisse s'écrire pareillement; nous pensons qu'il ne peut pas. []

On étudie maintenant la relation entre les fonctions invariantes plus fortes et l'abstraction fonctionnelle d'une instruction tantque.

**Proposition 12.** Soit w une instruction tantque. La fonction [w] est une fonction invariante pour w.

**Preuve.** Cette proposition résulte immédiatement de la clause p1 du théorème de l'instruction tantque (chapitre 3) avec f=[w]. CQFD

Cette proposition suggère -bien entendu- la question de savoir si [w] est une fonction invariante plus forte.

**Proposition 13.** Soit w une instruction tantque. Alors [w] est une fonction invariante plus forte pour w.

**Preuve.** Soit f une fonction invariante pour w; on doit prouver que $I(t)*[w]$ est plus injective que $I(t)*f$. En d'autres termes, si s et s' sont deux états dans S!t tels que [w](s)=[w](s'), on doit montrer que l'on a f(s)=f(s'). On a

[w](s) = [w](s')

=> f([w](s)) = f([w](s'))

=> $f((I(t)*[b])^i(s)) = f((I(t)*[b])^j(s'))$,

pour un certain i et j positifs. Parceque f est une fonction invariante, on peut faire décroître les indices i et j jusqu'à obtenir

=> f(s) = f(s'). CQFD

La proposition ci-dessus énonce que l'abstraction fonctionnelle d'une instruction itérative est une plus forte fonction invariante. Donc si on a obtenu une plus forte fonction invariante f de w, on sait que f et [w] sont également injectives on S!t, c'est à dire qu'elles définissent les mêmes ensembles de niveaux sur S!t. Vu que sur S!t le comportement de [w] est connu (voir la clause p0 du théorème de l'instruction tantque), la connaissance d'une plus forte fonction invariante semble suffir pour déterminer l'abstraction fonctionnelle d'une boucle tantque. En effet,

**Théorème 1.** Soit w = (**tantque** t **faire** b) une boucle tantque sur l'espace S et soit f une plus forte fonction invariante de w. Alors

$$[w] = \{(s,s') \mid \neg t(s) \ \&\ s'=s\} \cup \{(s,s') \mid t(s) \ \&\ f(s)=f(s') \ \&\ \neg t(s') \ \&\ prem(s')\},$$

où $prem(s') = (\exists s'': t(s'') \ \&\ s'=[b](s''))$; le mnémonique prem (premier) signifie que s' est le premier état qui vérifie ~t: son antécédent s" vérifie t.

**Preuve.** Par souci de lisibilité, nous donnons ici une preuve informelle de ce théorème. La preuve formelle en est donnée en appendice à ce chapitre.

Soit s un état dans S. Si ~t(s) alors l'image par [w] de s est s'=s. Si t(s) alors l'image par [w] de s peut être charactérisée comme suit: premièrement, s' vérifie ~t (d'où ~t(s')); deuxièmement, dans la chaîne d'applications de I(t)*[b] qui commence à s et finit à s', s' est le premier état qui vérifie ~t (d'où prem(s')); troisièmement, si s" est l'état précédant s' dans la chaîne citée ci-dessus, alors s et s" ont la même image par [w] (dû à la sémantique de l'instruction tantque) donc ils ont la même image par f car f et [w] sont également injectives sur S|t (d'où l'on déduit f(s)=f(s"), ou, puisque s'=[b](s"), t(s") est vrai et que f est invariante, f(s)=f(s')). CQFD

Le théorème de plus forte fonction invariante permet de générer l'abstraction fonctionnelle d'une boucle à partir d'une fonction invariante plus forte d'une boucle. Nous en donnons deux exemples d'application.

**Exemple**
S = **naturel**; w = (**tantque** s<21 **faire** s:=s+5). Il est clair que dom([w])=S; nous avons vu qu'une pfi de cette boucle est

$$f(s') = s \bmod 5.$$

Alors

$$[w] = [s \geq 21,\ s] \cup \{(s,s') \mid s<21 \ \&\ smod5=s'mod5 \ \&\ s' \geq 21 \ \&\ (\exists s'': s''<21 \ \&\ s'=s''+5)\}$$
$$= [s \geq 21, s] \cup \{(s,s') \mid s<21 \ \&\ smod5=s'mod5 \ \&\ 21 \leq s<26\} \qquad []$$

**Exemple**

```
S = ens
crt
   a, b: entier;
sse
   b≥0
fin.
w = (tantque b>0 faire début a:=a+1; b:=b-1 fin).
```

D'après l'exemple traité dans la sous-section 2.6, la fonction f(s)=(0,a(s)+b(s)) est une pfi pour w. On en déduit

$$[w] = [\neg(b(s)>0),\ s] \cup \{(s,s') \mid b(s)>0 \ \&\ a(s)+b(s)=a(s')+b(s') \ \&\ \neg(b(s')>0) \ \&\ (\exists s'': b(s'')>0 \ \&\ a(s')=a(s'')+1 \ \&\ b(s')=b(s'')-1)\}.$$
$$= [b(s)=0,\ s] \cup \{(s,s') \mid b(s)>0 \ \&\ a(s)+b(s)=a(s')+b(s') \ \&\ b(s')=0\}$$

$= [b(s)=0,\ s] \cup [b(s)>0,\ (a(s)+b(s),0)]$
$= [b(s)=0,\ (a(s)+b(s),0)] \cup [b(s)>0,\ (a(s)+b(s),0)]$
$= [vrai,\ (a(s)+b(s),0)].$ □

## 4. Discussions

Le présent chapitre a été consacré à l'étude des plus fortes fonctions invariantes. Nous en avons étudié la signification théorique, en particulier leur relation avec les invariants de boucle et avec les abstractions fonctionnelles d'instructions tantque. D'autre part, nous avons vu que sous certaines conditions portant sur l'espace, le corps de boucle et le prédicat de sortie d'une boucle tantque, il est possible de générer une pfi de façon systématique à l'aide de formules.

Ce travail est actuellement poursuivi dans deux directions, l'une théorique, l'autre pratique.

**Extensions théoriques.** Nous étudions la relation qui existe entre les fonctions invariantes des programmes itératifs et les points fixes des programmes récursifs; en particulier, on s'intéresse à la relation qui existe entre une pfi de programme itératif et un plus petit point fixe de programme récursif. Il y a beaucoup d'arguments à l'effet que les notions de fonction invariante et de point fixe sont liées puisqu'en fait une fonction invariante est un point fixe pour l'équation récursive donnée dans la prémisse p1 du théorème de l'instruction tantque dans le chapitre 3; de plus il y a des arguments intuitifs à l'effet que le plus petit point fixe et la plus forte fonction invariante sont liées puisque le plus petit point fixe d'un programme récursif est l'abstraction fonctionnelle du programme alors qu'une pfi d'un programme itératif est égale à l'abstraction fonctionnelle de ce programme à une fonction injective près. Il est intriguant de constater qu'un programme récursif a généralement un plus petit point fixe unique alors qu'un programme itératif a généralement plusieurs plus fortes fonctions invariantes.

**Extensions pratiques.** Nous tentons de trouver d'autres formules de génération de plus fortes fonctions invariantes du même genre que celles qui sont présentées dans la section 2. De plus, nous étudions les moyens de transférer les formules existantes vers des espaces non-numériques mais qui auraient la structure minimale requise par les formules: remarquez par exemple que les formules données dans la section 2 n'utilisent que les propriétés des groupes (réel,+) et (réel-{0},*) et leurs homomorphismes (log et exp); il est donc concevable que ces formules puissent être appliquées à des espaces isomorphes à ces groupes là. L'exemple suivant montre comment des programmes non-numériques peuvent être manipulés en vue d'être préparés à l'application des formules données.

**Exemple**
On considère l'instruction itérative
**tantque** b<5 **faire début** b:=b+1; **écrire**('x') **fin**,
où b est une variable entière. On suppose qu'un caractère est représenté par une séquence de m bits et on s'intéresse au nombre

représenté par cette séquence dans la numération binaire. Le fichier de sortie est aussi considéré comme un entier, f; l'exécution de l'instruction **écrire**('x') a pour effet de multiplier f par $2^m$ (décaler f de m positions) ensuite de lui ajouter c, le code de x. On suppose que c>0 et on pose $q=2^m$. On pose

**S = ens**
**crt**
b, f: **entier**;
**sse**
$f \geq 0$
**fin**,
w = (**tantque** b<5 **faire début** b:=b+1; f:=q*f+c **fin**).

En appliquant les formules de forme linéaire diagonale, on obtient la plus forte fonction invariante suivante:

$$F(s) = (0,\ (f(s)+c/(q-1))/q^{b(s)}).$$

En vertu du théorème 1, on obtient

[w] =
[b(s)≥5, s]
∪
{(s,s') | b(s)<5 & b(s')=5 & f(s') = $f(s)*q^{5-b(s)}+c*(q^{5-b(s)}-1)/(q-1)$}.

Si on s'intéresse à l'expression de f(s'), on note que $f(s)*q^{5-b(s)}$ représente le décalage de f -en tant que représentation de nombre- (5-b(s)) fois alors que $c*(q^{5-b(s)}-1)/(q-1)$ est le nombre dont la représentation en base q est (ccc...c): (5-b(s)) fois. Décodant le contenu du fichier en caractères, on trouve

$$\underline{f}xxx...x$$

où $\underline{f}$ est le fichier initial et la chaine xxx...x est une répétition de (5-b(s)) fois x. []

Il est souvent reproché à des efforts d'analyse de programmes tel que celui-ci qu'ils vont à l'encontre de la vogue en programmation: Il est mondain de penser que le programmeur est supposé connaître l'abstraction fonctionnelle de w et/ou l'invariant de boucle de w <u>avant</u> de concevoir w. Le chapitre 6 de cette thèse -ainsi que plusieurs approches relationnelles à la programmation proposées récemment- montre que l'on peut fort bien concevoir un programme itératif sans jamais connaître ni sa fonction ni ses invariants: on ne fait que manipuler ses spécifications; le non-déterminisme de la spécification originale est transmis à travers toutes les décisions de conception à mesure que la spécification originale est raffinée. Une fois que le programme est conçu, il est parfaitement légitime de vouloir en évaluer l'abstraction fonctionnelle ou les invariants de boucle. C'est ce que ce chapitre a tenté de faire.

## Appendice:
## Preuve du Théorème des Plus Fortes Fonctions Invariantes.

On utilisera le théorème de l'instruction tantque. On pose

v = {(s,s') | f(s)=f(s') & t(s') & prem(s')}

et

u = I(¬t) ∪ I(t)*v.

Il nous appartient tout d'abord de prouver que u (qui est la fonction proposée par le théorème pour [w]) est effectivement une fonction.

Pour cela, on calculera le produit relatif de son inverse par elle-même et on prouvera que c'est un sous-ensemble de I.

$(I(\check{}t) \cup I(t)*v)\hat{} * (I(\check{}t) \cup I(t)*v)$
$= (I(\check{}t) \cup v\hat{}*I(t)) * (I(\check{}t) \cup I(t)*v)$
$= I(\check{}t) \cup I(\check{}t)*I(t)*v \cup v\hat{}*I(t)*I(\check{}t) \cup v\hat{}*I(t)*v$
$= I(\check{}t) \cup v\hat{}*I(t)*v$

Il est clair que $I(\check{}t) \subseteq I$. Il reste à prouver que $v\hat{}*I(t)*v \subseteq I$.

$v\hat{}*I(t)*v$
$= \{(s,s') \mid \exists x: (s,x) \in v\hat{} \ \&\ t(x) \ \&\ (x,s') \in v\}$
$= \{(s,s') \mid \exists x: (x,s) \in v \ \&\ t(x) \ \&\ (x,s') \in v\}$
$= \{(s,s') \mid \exists x: f(x)=f(s) \ \&\ \check{}t(s) \ \&\ prem(s) \ \&\ t(x) \ \&\ f(x)=f(s') \ \&\ \check{}t(s') \ \&\ prem(s')\}$
$\subseteq \{(s,s') \mid f(s)=f(s') \ \&\ \check{}t(s) \ \&\ \check{}t(s') \ \&\ prem(s) \ \&\ prem(s')\}$
$= \{(s,s') \mid \exists x,x': f(s)=f(s') \ \&\ \check{}t(s) \ \&\ \check{}t(s') \ \&\ s=[b](x) \ \&\ t(x) \ \&\ s'=[b](x') \ \&\ t(x')\}.$

Parceque $s=[b](x)$ et $s'=[b](x')$, on a $f([b](x))=f([b](x'))$; parceque $t(x)$ et $t(x')$ et que f est une fonction invariante, on a $f(x)=f(x')$; parceque $t(x)$ et $t(x')$ et que f et $[w]$ sont egalement injectives, on a $[w](x)=[w](x')$; parceque $t(x)$ et $t(x')$ et que $s=[b](x)$ et $s'=[b](x')$, on a $[w](s)=[w](s')$; parceque $\check{}t(s)$ et $\check{}t(s')$, on a $[w](s)=s$ et $[w](s')=s'$; on en déduit $s=s'$; donc

$\subseteq \{(s,s') \mid s'=s\}$
$= I.$

**Maintenant on s'assure que la fonction u vérifie les prémisses p0, p1 et p2 du théorème de l'instruction tantque (chapitre 3).**

**Prémisse p0.** $I(\check{}t)*u = I(\check{}t)*I(\check{}t) \cup I(\check{}t)*I(t)*v = I(\check{}t)*I(\check{}t) = I(\check{}t).$

**Prémisse.**

$I(t)*[b]*u$
$= I(t)*[b]*I(\check{}t) \cup I(t)*[b]*I(t)*v$
$= I(t) * (I(t)*[b]*I(\check{}t) \cup I(t)*[b]*I(t)*v)$

On a les deux identités suivantes (prouvées plus loin):
$I(t)*[b]*I(\check{}t) \subseteq I(t)*v,$
$I(t)*[b]*I(t)*v = I(t)*v.$
Donc

$= I(t)*(I(t)*v).$

Puisque $u = I(\check{}t) \cup I(t)*[v]$, on a $I(t)*v=I(t)*u.$

$= I(t)*I(t)*u$
$= I(t)*u.$

**On prouve maintenant les deux propositions supposées ci-dessus.**

$I(t)*[b]*I(\check{}t) \subseteq I(t)*v?$

Soit $(s,s') \in I(t)*[b]*I(\check{}t)$; alors $t(s) \ \&\ s'=[b](s) \ \&\ \check{}t(s')$; on en déduit:

$t(s) \ \&\ f(s)=f(s') \ \&\ \check{}t(s') \ \&\ prem(s').$

prem(s') est vrai vu que s' a un antécédent par [b] -à savoir s- qui vérifie la condition t; donc

$(s,s') \in I(t)*v.$

**L'autre condition était**

$I(t)*[b]*I(t)*v = I(t)*v?$

On a

$I(t)*[b]*I(t)*v$

$= \{(s,s') \mid \exists x:\ t(s)\ \&\ x=[b](s)\ \&\ t(x)\ \&\ f(x)=f(s')\ \&\ \sim t(s')\ \&\ prem(s')\}$

$= \{(s,s') \mid \exists x:\ x=[b](s)\ \&\ t(x)\ \&\ t(s)\ \&\ f(s)=f(s')\ \&\ \sim t(s')\ \&\ prem(s')\}.$

On remplace x par [b](s) dans f(x)=f(s') ensuite on remplace f([b](s)) par f(s) car f est une fonction invariante et que t(s) est vraie.

$= \{(s,s') \mid \exists x:\ t(s)\ \&\ x=[b](s)\ \&\ t(x)\}\ \mathbf{0}\ I(t)*v.$

La relation écrite entre les accolades est en fait l'ensemble des paires (s,s') telles que t(s); on peut l'écrire comme I(t)*U, et l'on obtient alors

$= I(t)*U\ \mathbf{0}\ I(t)*v$

$= I(t)\ *\ (U\ \mathbf{0}\ v)$

$= I(t)*v.$

La preuve de la prémisse p1 est terminée.

**Prémisse p2.** Cette prémisse découle tout naturellement du fait que dom(u)=S (d'après la définition de u) et dom([w])=S (par hypothèse).

**Le théorème de l'instruction tantque permet d'écrire: [w]=u.** CQFD

# Chapitre 5

# Cohérence de Programmes

Maintenant que nous avons vu ce qu'est l'abstraction fonctionnelle d'un programme et ce qu'est une spécification, nous discutons comment on peut confronter un programme à une spécification afin de déterminer si le programme est cohérent par rapport à la spécification.

## 1. La Cohérence: Une Illustration

Soit l'espace S = {a,b,c,d,e,f} et soit la relation R définie sur S par

R = {(a,a),(a,b),(a,c), (b,b),(b,c),(b,d), (c,c),(c,d),(c,e), (d,d),(d,e),(d,f)}.

Soit p un programme sur l'espace S tel que

[p] = {(a,c),(b,c),(c,c),(d,f),(e,f),(f,f)}.

La fonction [p] est définie pour tous les éléments de dom(R); de plus, pour chaque s dans dom(R), [p](s) est l'un des éléments recommandés par R comme un état final correct. On dit alors que p est <u>totalement cohérent</u> (ou encore <u>totalement correct</u>) par rapport à R.

Soit p' un programme sur l'espace S tel que

[p'] = {(a,a),(b,b),(c,c),(f,f)}.

On constate que [p'] n'est pas défini pour tout s dans dom(R) puisqu'il n'est pas défini pour l'état initial d; toutefois, toutes les fois que [p'] est défini, il se comporte tel que recommandé par R. On dit alors que p' est <u>partiellement cohérent</u> par rapport à R. La notion de cohérence partielle est très faible puisqu'un programme p' dont l'abstraction fonctionnelle [p'] est vide (qui ne termine normalement pour aucun état initial dans S) est partiellement correct par rapport à n'importe quelle spécification.

Soit p" un programme sur l'espace S tel que

[p"] = {(a,a),(b,a),(c,c),(d,c),(e,e),(f,e)}.

On constate que [p"] est défini pour tout élément s dans dom(R); [p"](s) peut (comme dans le cas de a) ou peut ne pas (comme dans le cas de b) appartenir à s.R. On dit alors que p" est <u>défini</u> par rapport à R ou qu'il <u>termine</u> par rapport à R.

## 2. La Cohérence: Présentation Formelle

Les définitions mathématiques données ci-dessous reflètent les explications verbales de la section précédente.

**Définition 1: Cohérence Totale.** On dit qu'un programme p est <u>totalement cohérent</u> (ou <u>totalement correct</u>) par rapport à une spécification R si et seulement si

$$(\forall s,\ s \in dom(R) \Rightarrow s \in dom([p]) \ \&\ (s,[p](s)) \in R).$$

Quand le contexte ne prête pas à confusion, on pourra des fois ne pas mentionner l'adverbe totalement. La définition proposée ci-dessus peut être décomposée en deux clauses:

dom(R)⊆dom([p])

∀s∈dom(R), s.[p]⊆s.R.

En fait on peut dire que le programme p est cohérent par rapport à la spécification R si et seulement si la relation (fonction, en fait) [p] est plus-définie que la relation R. Il n'est pas surprenant que la relation (au sens litéraire du terme) que doit vérifier un programme vis-à-vis d'une spécification est identique à la relation que doit vérifier une spécification vis-à-vis d'une propriété (voir validation de spécifications). cette identité entre les activités de vérification de cohérence et de vérification de validité est facile à comprendre si l'on pense que la relation plus-défini signifie: porte plus d'information.

**Définition 2: Cohérence Partielle.** On dit qu'un programme p est partiellement cohérent par rapport à une spécification R si et seulement si

(∀s, s∈dom(R) & s∈dom([p]) => (s,[p](s))∈R).

En fait, par rapport à la définition 1, la clause s∈dom([p]) a simplement changé de côté par rapport au signe =>.

**Définition 3: Terminaison.** On dit qu'un programme p est défini (ou qu'il termine) par rapport à une spécification R si et seulement si

(∀s, s∈dom(R) => s∈dom([p])).

Les formules de cohérence et cohérence partielle présentées ici sont faciles à comprendre et à interpréter mais difficiles à utiliser; pour cette raison, on introduit ci-dessous des formules alternatives, dont on prouvera qu'elles sont équivalentes aux définitions.

**Proposition 1 (Formule de Mills).** Un programme p est correct par rapport à R si et seulement si

dom(R∩[p]) = dom(R).

**Preuve.** Vu que dom(R∩[p])⊆dom(R) est une tautologie, on se contentera de prouver, en fait, l'équivalence entre

(∀s, s∈dom(R) => s∈dom([p]) & (s,[p](s))∈R) (a)

et

dom(R) ⊆ dom(R∩[p]). (b)

Preuve de a=>b. Soit s dans dom(R). D'après (a), s∈dom([p]) et (s,[p](s))∈R; or, par définition d'une fonction, (s,[p](s))∈[p]. Donc (s,[p](s))∈R∩[p], et s∈dom(R∩[p]).

Preuve de b=>a. Soit s dans dom(R). D'après (b), s∈dom(R∩[p]); donc il existe s' tel que (s,s')∈R∩[p]. Ce s' est précisément [p](s) car (s,s')∈[p] et [p] est déterministe. De s'=[p](s) on déduit que s∈dom([p]) et que (s,[p](s))∈R. CQFD

**Proposition 2.** Un programme p est partiellement correct par rapport à R si et seulement si

dom(R∩[p]) = dom(R) ∩ dom([p]).

**Preuve.** Il est clair que dom(R○[p])⊆dom(R)○dom([p]); donc on se contentera de prouver l'équivalence entre
(a) (∀s, s∈dom(R) & dom([p]) => (s,[p](s))∈R)
et
(b) dom(R)○dom([p]) ⊆ dom(R○[p]).
Preuve de a=>b. Soit s un élément de dom(R)○dom([p]); d'après (a), (s,[p](s))∈R; comme par définition (et parceque s∈dom([p])) on a (s,[p](s))∈[p], on déduit que s∈dom(R○[p]).
Preuve de b=>a. Soit s un élément appartenant à dom(R) et dom([p]); d'après (b), s∈dom(R○[p]), donc il existe s' -en l'occurrence [p](s)- tel que (s,s')∈R○[p]; d'où (s,[p](s))∈R. CQFD

**Proposition 3.** Un programme p est défini par rapport à R si et seulement si

dom(R)○dom([p]) = dom(R).

**Preuve.** En fait cette condition est équivalente à dom(R)⊆dom([p]), qui est elle même une paraphrase de la définition 3. CQFD

La formule que nous avons adoptée pour la proposition 3 n'est pas fortuite: en fait elle est choisie de façon à montrer pourquoi un programme qui est partiellement correct et défini est correct.

## 3. Preuve de Cohérence: Exécution Symbolique

Pour prouver la cohérence du programme p par rapport à la relation R, la méthode d'exécution symbolique consiste à calculer l'abstraction fonctionnelle de p, ensuite à la confronter contre la spécification donnée. A titre d'exemple d'application, nous considérons le programme étudié dans la section 3 du chapitre 3; on en prouve la cohérence par rapport à la spécification

R = {(s,s') | u(s') = (a(s)○b(s)[1..m]≠ø)}.

Cette spécification vérifie si le programme intersection mat dans la variable u la valeur logique (a(s)○b(s)[1..m]≠ø); elle (la spécification) ne s'intéresse pas aux autres aspects fonctionnels du programme. On utilisera la proposition 1.

dom(R)=S.
dom(R○[p])
= dom({(s,s') | a(s')=a(s) & b(s')=a(s)[n].b(s)[1..m] & i(s')=n+1 &
x(s')=a(s)[n] & u(s')=(a(s)○b(s)[1..m]≠ø)}).
= S.

On considère une autre spécification:

R = {(s,s') | a(s)[1]=b(s)[i] & u(s')=vrai}.

Cette spécification exprime que pour tout état initial s tel que a(s)[1]=b(s)[1], le programme intersection doit mettre la variable u à **vrai**.

dom(R') = {s | a(s)[1]=b(s)[1]}.
dom(R'○[p])
= dom({(s,s') | a(s')=a(s) & b(s')=a(s)[n].b(s)[1..m] & i(s')=n+1 &
x(s')=a(s)[n] & u(s')=(a(s)○b(s)[1..m]≠ø) &
a(s)[1]=b(s)[1] & u(s')=vrai})
= dom({(s,s') | a(s)[1]=b(s)[1] & a(s')=a(s) & b(s')=a(s)[n].b(s)[1..m]
& i(s')=n+1 & x(s')=a(s)[n] & u(s')=vrai})

$= \{s \mid a(s)[1]=b(s)[1]\}$.
En fait on aurait pu prouver la cohérence de p par rapport à R' simplement en constatant que R est plus-défini que R' (donc si [p] est plus-défini que R alors il est plus défini que R').

Soit à prouver la cohérence de p par rapport à la spécification
$R'' = \{(s,s') \mid u(s')=\textbf{faux}\}$.
$dom(R'') = S$.
$dom(R'' \circ [p])$
$= dom(\{(s,s') \mid a(s')=a(s) \;\&\; b(s')=a(s)[n].b(s)[1..m] \;\&\; i(s')=n+1 \;\&\; x(s')=a(s)[n] \;\&\; u(s')=(a(s)Ob(s)[1..m]\neq\emptyset \;\&\; u(s')=\textbf{faux}\}$
$= \{s \mid a(s)Ob(s)[1..m]=\emptyset\}$.

La méthode de preuve de conhérence par exécution symbolique est conceptuellement simple, mais elle présente deux faiblesses majeures:
- Il arrive souvent que l'on veuille prouver la cohérence d'un programme p par rapport à une spécification R qui est très peu-définie, donc qui n'exerce que certains aspects fonctionnels de p. Or, la méthode d'exécution symbolique exige que l'on calcule [p] quelque soit la spécification, donc nous force à nous concerner avec toutes les proprietés fonctionnelles de p même si on n'en prouve que quelques unes.
- Il est parfois très difficile d'exprimer l'abstraction fonctionnelle de programmes, même de simples programmes: Considerez, par exemple, un programme de recherche binaire ou un programme de tri par permutation.

Les méthodes de preuve de cohérence par induction puisent leur intérêt dans les faiblesses mentionnées ci-dessus. Plusieurs méthodes inductives ont été présentées pendant les quinze dernières années: [BASU75], [FLOY67], [HOAR69], [MANN74], [MORR77], pour n'en mentionner que quelques unes. Une classification de ces méthodes est présentée dans [MILI84o], et sert de base pour un cours en vérification de programmes [MILI84v].

Dans la section suivante, nous introduisons une nouvelle méthode inductive pour la vérification de programmes itératifs. Cette méthode se veut l'équivalent itératif de la méthode introduite par Manna [MANN74] pour la vérification de programmes récursifs, sous le nom: Computational Induction Method.

La méthode que nous présentons ici a un double intérêt:
- Sa caractérisation de cohérence est une équation relationnelle, n'impliquant pas des états individuels mais plutôt des relations.
- Elle donne une perspective intéressante sur la sémantique de l'instruction tantque.

L'interêt pratique de cette méthode n'a pas été étudié quoiqu'elle semble être aussi puissante -en pratique- que la méthode d'induction par sous-but (subgoal induction).

## 4. Preuve de Cohérence: Induction Calculatoire

La méthode d'induction calculatoire (computational induction) pour la

vérification de programmes récursifs est présentée dans [MANN74]. Elle consiste à construire une séquence infinie de fonctions F0, F1, F2, ... qui converge vers la fonction du programme récursif, F. Pour prouver une propriété q sur F, on prouve par induction que q est vrai pour tout Fi, pour en déduire la validité de q pour F. Dans cette section on prendra une approche similaire à la vérification d'une instruction itérative telle que

w = **tantque** t **faire** b.

On étudie cette instruction sur l'espace S = dom([w]).

## 4.1. Construction de la Chaine

Soit $F_0$, $F_1$, $F_2$, $F_3$ ... une séquence infinie de fonctions. On dit que $(F_i)$ est une chaine si pour tout $i \geq 0$, $F_i \subseteq F_{i+1}$. On introduit ci-dessous une chaîne de fonctions dont on prouvera qu'elle converge vers [w]:

W0 = NT,
W1 = TB*W0 U NT
W2 = TB*W1 U NT
... ... ...

où NT=I(~t) et TB=I(t)*[b]. On prouve deux résultats sur la séquence $(W_i)$.

**Proposition 4.** La séquence $(W_i)$ est une chaîne.

**Preuve.** Pour prouver cette proposition, on prouvera en fait un résultat plus fort, à savoir
$(\forall i \geq 0,\ W_i = (TB^0 \cup TB^1 \cup TB^2 \cup \ldots \cup TB^i) * NT.$
On fera une preuve par induction.

$W_0 = TB^0 * NT$
$= NT.$

$W_{i+1} = TB * W_i \cup NT$
$= TB * (TB^0 \cup TB^1 \cup TB^2 \cup \ldots \cup TB^i) * NT \cup NT$
$= (TB^0 \cup TB^1 \cup TB^2 \cup \ldots \cup TB^{i+1}) * NT$

CQFD

En effet, $W_i$ est la pré-restriction de [w] à l'ensemble de tous les états qui exigent au plus i itérations pour être traités par [w]; il semble que, à mesure que i tend l'infini, $W_i$ tend vers [w]. La définition et proposition ci-dessous confirment cette intuition.

**Définition 4.** Soit $(W_i)$ une chaîne de fonctions. On dit que W est la plus petite borne supérieure de $(W_i)$ si W est la plus petite fonction telle que pour tout i, $W_i \subseteq W$.

On notera la plus petite borne supérieure d'une chaîne $(W_i)$ par pbs$(W_i)$; l'existence d'une plus petite borne supérieure pour toute chaîne est prouvée dans [MANN74].

**Proposition 5.** La fonction W = TB**NT est la plus petite borne supérieure de la chaîne $(W_i)$.

**Preuve.** Il est clair (preuve de la proposition 4) que pour tout i, $W_i \subseteq W$. Soit W' une autre fonction qui contient tous les $W_i$. On montrera que $W \subseteq W'$. Soit $(s,s') \in W$; alors il existe $i \geq 0$ tel que

$(s,s')\in BT^{i}*NT$; alors $(s,s')\in W_i$; donc $(s,s')\in W'$. CQFD

On considère la fonctionnelle K définie sur S par K(f) = TB*fUNT. Alors la chaîne $(W_i)$ peut être re-définie comme suit:

$W_0 = NT$,

$\forall i \geq 1,\ W_i = K(W_{i-1})$;

ou encore, de façon équivalente

$W_0 = NT$

$\forall i \geq 1,\ W_i = K^{i}(NT)$.

Deux propriétés intéressantes des fonctionnelles sont la monotonicité et la continuité. La fonctionnelle K est monotone si $F \subseteq F' \Rightarrow K(F) \subseteq K(F')$. La fonctionnelle K est dite continue si pour toute chaîne $(W_i)$, $K(pbs(W_i)) = pbs(K(W_i))$.

**Proposition 6.** La fonctionnelle K définie par K(f)=TB*fUNT est continue.

**Preuve.** De toute évidence K est monotone puisque $f \subseteq f' \Rightarrow TB*f \subseteq Tb*f'$. Afin de prouver que K est aussi continue, on écrit K(f) comme K1(K2(f)) où

K2(f) = TB*f, et

K1(f) = fUNT.

Or, K1 est continue puisque $K1(pbs(G_i)) = pbs(G_i) \cup NT = pbs(G_i \cup NT) = pbs(K1(G_i))$. D'autre part, d'après le théorème de la fonctionnelle continue ([MANN74]: Chapitre 5), la fonctionnelle K2 est aussi continue (produit relatif d'une constante -TB- par une variable -f-). Donc

$K(pbs(W_i))$

$= K1(K2(pbs(W_i)))$

$= K1(pbs(K2(W_i)))$

$= pbs(K1(K2(W_i)))$

$= pbs(K(W_i))$. CQFD

Soit q un prédicat sur les fonctions sur S. Le prédicat q est dit admissible si pour toute fonctionnelle K et pour toute fonction f0, si $q(K^{i}(f0))$ est vrai pour tout $i \geq 0$ alors $q(pbs(K^{i}(f0)))$ est aussi vrai.

Dans la sous-section suivante on montre que la propriété de cohérence est admissible, ensuite on présente le théorème de cohérence par induction calculatoire.

## 4.2. Théorème de Cohérence

Soit w une instruction tantque sur S=dom([w]) et soit R une relation sur S. On prouve que c'est sans perte de généralité que l'on peut poser S=dom(R): Soit R' la relation définie par

$$R' = \{(s,s') \mid s \in dom(R) \Rightarrow (s,s') \in R\}.$$

Alors R' présente deux propriétés intéressantes, à savoir:

- dom(R')=S, puisque pour tout s dans S il existe s' tel que $(s \in dom(R) \Rightarrow (s,s') \in R)$ est vrai.
- w est cohérent par rapport à R si et seulement si w est cohérent par rapport à R'. La cohérence de w par rapport à R' s'écrit

$$s \in dom(R') \Rightarrow s \in dom([p]) \ \&\ (s,[p](s));$$

d'après l'hypothèse dom(R')=S, s∈dom(R') est vrai; d'après l'hypothèse S=dom([w]), s∈dom([w]) est vrai; donc on a
vrai => vrai & (s,[p](s))∈R'
=
(s,[p](s))∈R'
=
s∈dom(R) => (s,[p](s))∈R
=
s∈dom(R) => vrai & (s,[p](s))∈R
=
s∈dom(R) => s∈dom([p]) & (s,[p](s))∈R
=
w est cohérent par rapport à R.

Donc en plus de l'hypothèse dom([w])=S, on prend l'hypothèse dom(R)=S. Alors la cohérence de w par rapport à R peut s'écrire comme

$$\forall s,\ (s,[p](s))\in R$$

ou encore, en termes relationnels,

$$[w]\subseteq R.$$

Soit la fonction W=TB**NT; d'après l'axiome de l'itération (chapitre 3), W est l'abstraction fonctionnelle de W. D'après la sous-section précédente, W=pbs($W_i$). On définit le prédicat q sur les fonctions sur S par q(W)=W⊆R; la cohérence de w par rapport à R s'écrit comme q(W) vu que W=[w]. Pour prouver q(W), on prouvera q($W_i$) pour tout $i\geq 0$ pour en déduire q(W) si q est admissible.

**Proposition 7.** Le prédicat q(W)=(W⊆R) est admissible.

**Preuve.** Si pour tout i, $W_i\subseteq R$ alors R est une borne supérieure de $W_i$; donc la plus petite borne supérieure W de $W_i$ est un sous-ensemble de R. Ceci s'écrit: $W_i\subseteq R$. CQFD

**Théorème 1 (Induction Calculatoire pour l'Instruction Itérative).** Soit w = (**tantque** t **faire** b) une instruction itérative telle que dom([w]) et soit R une relation telle que dom(R)=S. Si

$$(\forall \text{fonction } f,\ f\subseteq R \Rightarrow (TB*f\cup NT)\subseteq R) \qquad (c)$$

alors w est correct par rapport à R.

**Preuve.** Si on écrit la condition (c) avec f=∅ ensuite avec $f=W_i$, on obtient respectivement la base d'induction et l'étape d'induction d'une preuve par induction à l'effet que pour tout i, q($W_i$) est vrai. Vu que q est admissible, on en déduit que q(W) est vrai; autrement dit que q([w]) est vrai. CQFD

Ce théorème présente deux intérêts théoriques:
- Il donne une condition de cohérence qui est purement relationnelle, n'impliquant que des variables relationnelles, des constantes relationnelles et des opérateurs relationnels. cette condition de cohérence peut être perçue comme une caractérisation de spécifications R par rapport auxquelles w est correct.
- Il donne une certaine perspective sur la sémantique de l'instruction tantque. Soit F une fonction sur S telle que S=dom(F); la cohérence de w par rapport à F s'écrit comme [w]⊆F; toutefois, comme dom([w])=S=dom(F), ceci s'écrit encore comme [w]=F.

Le théorème s'énonce alors:
Si $(\forall f,\ f \underline{C} F \Rightarrow (TB * f UNT) \underline{C} F)$
Alors $[w]=F$.
L'équation $(\forall f,\ f \underline{C} F \Rightarrow (TB * f UNT) \underline{C} F)$ peut être considérée comme une équation caractérisant l'abstraction fonctionnelle F de w.

Un exemple simple d'application pratique de ce théorème est donné ci-dessous.

**Exemple**

**S = ens**
**crt**
a, b, c: **entier;**
**sse**
b>0
**fin;**
w = **tantque** b≠0 **faire début** b:=b−1; c:=c+a **ens;**
R = {(s,s') | c(s')=c(s)+a(s)*b(s) & b(s')=0}.

Il est clair que dom([w])=S et que dom(R)=S. Soit f une fonction incluse dans R.

$f \underline{C} R \Rightarrow$
$(TB * f \cup NT)$
$\underline{C}\ (TB * R \cup NT)$
$= \{(s,s') \mid b(s)\neq 0 \ \&\ b(s')=b(s)-1 \ \&\ c(s')=c(s)+a(s)\}$
$\quad * \{(s,s') \mid c(s')=c(s)+a(s)*b(s) \ \&\ b(s')=0\}$
$\quad \cup$
$\quad \{(s,s') \mid s'=s \ \&\ b(s)\neq 0\}$
$= \{(s,s') \mid \exists s'':\ b(s)\neq 0 \ \&\ a(s'')=a(s) \ \&\ b(s'')=b(s)-1 \ \&$
$\qquad c(s'')=c(s)+a(s) \ \&\ c(s')=c(s'')+a(s'')*b(s'') \ \&\ b(s')=0\}$
$\quad \cup$
$\quad \{(s,s') \mid s'=s \ \&\ b(s)=0\}$
$= \{(s,s') \mid b(s)\neq 0 \ \&\ c(s')=c(s)+a(s)*b(s) \ \&\ b(s')=0\}$
$\quad \cup$
$\quad \{(s,s') \mid b(s)=0 \ \&\ s'=s \ \&\ b(s')=0\}$
$\underline{C}\ \{(s,s') \mid c(s')=c(s)+a(s)*b(s) \ \&\ b(s')=0\}$
$= R.$ []

# Chapitre 6

## Conception de Programmes

On abordera dans ce chapitre le problème de conception de programmes. On prendra une approche relationnelle à ce problème en formalisant chaque décision de conception comme la résolution d'équations relationnelles.

## 1. Modèle de Conception

L'activité de conception de programmes consiste à transformer une relation (spécification) R en un programme. Une démarche naturelle de l'esprit consiste à aborder cette activité de la façon suivante:

**Processus de Conception:** Si R est assez simple alors
trouver un programme correct par rapport à R
Sinon
décomposer R en relations plus simples et leur appliquer le processus de conception.

Ce processus décrit ci-dessus soulève immédiatement deux questions cruciales, à savoir: Premièrement, comment définit-on la simplicité d'une relation? en particulier, quand dit-on qu'une relation est assez simple et quand dit-on qu'une relation est plus simple qu'une autre? Deuxièmement, comment décompose-t-on une relation en relations plus simples? Ces deux questions font l'objet des deux sections suivantes.

## 2. Simplicité d'une relation

La notion de simplicité d'une relation (interprétée comme une spécification) est difficile à cerner formellement. Il est raisonnable de percevoir la propriété de simplicité comme la disjonction de plusieurs critères indépendants de simplicité. Nous avons réussi à cerner deux de ces critères; ils font l'objet des deux sous-sections suivantes.

### 2.1. Simplicité: Critère Intrinsèque

Dans cette sous-section on présente un critère de simplicité qui ne dépend pas de la façon dont la relation est représentée; plutôt, il dépend des propriétés ensemblistes de la relation. Soit S = {a,b,c,d,e,}, où a, b, c, d, et e sont des noms abstraits d'éléments; ceci veut dire, par exemple, qu'il n'existe pas de relation successeur qui lie a à b, b à c, c à d, etc... . On considère deux spécifications R et R' définies comme suit:

R = {(a,b),(b,c),(c,d),(d,e),(e,a)}, et
R'= {(a,b),(a,c),(a,d), (b,c),(b,d),(b,e), (c,d),(c,e),(c,a), (d,e),(d,a),(d,b)}.

Si un programmeur avait le choix entre résoudre (trouver un programme

correct par rapport à) R ou résoudre R', lequel devrait-il choisir? Il devrait -bien entendu- choisir ... R' vu que:
- R' présente moins d'états initiaux (dont il faut se soucier) que R; formellement, ceci s'écrit dom(R')⊆dom(R).
- Pour tout s dans le domaine de R', R offre moins d'options possibles pour les états finaux que ne le fait R'; formellement, ceci s'écrit ∀s∈dom(R'), s.R⊆s.R'.

D'où la définition:

> **Définition 1.** On dit que la relation R' est intrinsèquement plus simple que la relation R si et seulement si R' est moins-définie que R.

Dans le chapitre 2, nous avons montré qu'une relation (spécification) est d'autant plus simple à générer qu'elle est moins-définie. D'après cette définition (et la discussion qui l'a motivée), une relation est d'autant plus simple à résoudre qu'elle est moins-définie.

La proposition suivante a pour but de fournir une justification additionnelle pour la définition ci-dessus. Elle montre qu'une spécification est d'autant plus intrinsèquement simple qu'elle a plus de solutions (programmes corrects).

> **Proposition 1.** Si R' est intrinsèquement plus simple que R alors tout programme correct par rapport à R est correct par rapport à R'.

**Preuve.** Soit un programme p correct par rapport à R; alors [p] est plus-défini que R. Vu que R' est intrinsèquement plus simple que R, alors R est plus-définie que R'. Donc [p] est plus-définie que R'. CQFD

Les deux propositions suivantes montrent des configurations particulières où une relation est intrinsèquement plus-simple qu'une autre; ces propositions sont invoquées dans la section 3.

> **Proposition 2.** Soient R et R' deux relations telles que R'⊆R et dom(R')∩dom(R-R')=ø. Alors R' est intrinsèquement plus simple que R.

**Preuve.** De R'⊆R on déduit aisément que dom(R')⊆dom(R). Soit s dans dom(R'); d'après dom(R')∩dom(R-R')=ø, il résulte que s∉dom(R-R'). Donc

$$s.R = s.(R' \cup (R-R')) = s.R'.$$

On déduit que R' est moins-définie que R. CQFD

> **Proposition 3.** Soient R et R' deux relations telles que R⊆R' et dom(R)=dom(R'). Alors R' est intrinsèquement plus simple que R.

**Preuve.** De dom(R')=dom(R) on déduit dom(R')⊆dom(R). De R⊆R' on déduit que pour tout s dans dom(R'), s.R⊆s.R'. CQFD

## 2.2. Simplicité: Critère Représentationnel

Il existe une mesure de la simplicité d'une relation qui échappe à une formalisation ensembliste; elle est liée plutôt aux notations qui ont été choisies pour représenter les relations: C'est ce que nous appelons un critère représentationnel.

Ce critère ne se prête pas à une définition formelle; donc on se contentera de l'illustrer par des exemples simples.

**Exemple**
On considère un univers où la seule opération arithmétique connue est l'incrémentation par 1. Soient R et R' les deux relations suivantes sur S=**naturel**.

R = {(0,2),(1,3),(2,4),(3,5),...}
R'= {(0,1),(1,2),(2,3),(3,4),...}.

En utilisant l'arithmétique à notre disposition, on peut représenter R' comme

R'={(s,s') | s'=Inc(s)}, où Inc(s)=s+1.

Pour représenter R, on peut écrire R=R'*R'.
Il est raisonnable de convenir que dans le mode de représentation choisi R' est représentationnellement plus simple que R. []

**Exemple**
On considère ces mêmes relations dans un univers où les seules opérations arithmétiques connues sont l'incrémentation par 2 et la décrémentation par 1. En utilisant cette arithmétique, on peut représenter R comme suit:

R = {(s,s') | s'=Inc2(s)}.

Pour représenter R', il faut introduire une relation D définie par

D = {(s,s') | s'=dec(s)}, où dec(s)=s-1.

On a alors R'=R*D.
Il est raisonnable de convenir alors que dans le mode de représentation choisi, c'est R qui est représentationnellement plus simple que R'. []

De par les opérations (arithmétiques, logiques ou autres) qu'il offre, un langage de programmation affecte le processus de conception de programmes via la mesure de simplicité représentationnelle qu'il définit.

## 3. Règles de Conception

Parmi les règles de conception, on distingue entre la règle de l'affectation qui transforme une spécification en une instruction, les règles de décomposition qui transforment une spécification (complexe) en une ou plusieurs spécifications (représentationnellement ou intrinsèquement) plus simples, et la règle de généralisation qui transforme une spécification (simple) en une spécification (intrinsèquement) plus complexe (l'utilité de cette règle paradoxale sera discutée plus tard).

### 3.1. Règle de l'Affectation

**Règle.** Etant donnée une relation R sur S, trouvez une expression E sur S telle que [∈def(E),E] est plus défini que R.

**Proposition 4.** Si E est ainsi choisi alors le programme (s:=E(s)) est correct par rapport à R.

**Preuve.** Ceci résulte simplement de la définition sémantique de l'affectation et de la définition de cohérence. CQFD

**Exemple**

S = **entier**;
R = {(s,s') | $0 \leq s \leq 9$ & $s-1 \leq s' \leq s+2$}
**U**
{(s,s') | $10 \leq s \leq 19$ & $s+1 \leq s' \leq s+4$}.

On pose E(s) = s+2; alors
f = [∈def(E),E]
= [vrai,s+2]
= {(s,s') | s'=s+2}.
R◦f
= {(s,s') | $0 \leq s \leq 9$ & $s-1 \leq s' \leq s+2$ & s'=s+2}
**U**
{(s,s') | $10 \leq s \leq 19$ & $s+1 \leq s' \leq s+4$ & s'=s+2}
= {(s,s') | $0 \leq s \leq 9$ & s'=s+2} **U** {(s,s') | $10 \leq s \leq 19$ & s'=s+2}
= {(s,s') | $0 \leq s \leq 19$ & s'=s+2}.
Donc dom(R◦f) = {s | $0 \leq s \leq 19$}.
Par ailleurs, dom(R) = {s | $0 \leq s \leq 19$}. []

## 3.2. Règles de Décomposition

Ces règles consistent à tranformer une spécification (complexe) en des spécifications plus simples. Les spécifications générées peuvent être soit intrinsèquement plus simples, soit représentationnellement plus simples que la spécification d'origine; en introduisant ces règles, nous les marquerons de (i) ou de (r) en super-indice -afin d'indiquer si les relations générées par ces règles sont (respectivement) intrinsèquement ou représentationnellement plus simples que la relation d'origine.

A. Règle de la Séquence (ou du Produit Relatif).$^{(r)}$

Cette règle transforme une relation R en un produit relatif de deux relations représentationnellement plus simples R1 et R2.

**Règle.** Etant donnée une relation R sur S, trouver des relations R1 et R2 telles que
i) R = R1*R2,
ii) cod(R1)$\subseteq$dom(R2).

**Proposition 5.** Si p1 est correct par rapport à R1 et p2 est correct par rapport à R2 alors (p1; p2) est correct par rapport à R.

**Preuve.** Soit s un élément de dom(R); en vertu de (i), s∈dom(R1);

puisque p1 est correct par rapport à R1, on a

$$s \in dom([p1]) \qquad (a)$$

et

$$(s,[p1](s)) \in R1. \qquad (a')$$

De (a') on voit que $[p1](s) \in cod(R1)$, de (ii) on déduit que $[p1](s) \in dom(R2)$; de la cohérence de p2 par rapport à R2 on déduit

$$[p1](s) \in dom([p2]) \qquad (b)$$

et

$$([p1](s),[p2]([p1](s))) \in R2. \qquad (b')$$

De (a) et (b) on déduit que

$$s \in dom([p1]*[p2]). \qquad (c)$$

De (a') et (b') on déduit que

$$(s,[p1]*[p2](s)) \in R1*R2. \qquad (c')$$

D'après la définition sémantique de la séquence, on déduit de (c) que

$$s \in dom([p1;p2]), \qquad (d)$$

et on déduit de (c') et de (i) que

$$(s,[p1;p2](s)) \in R. \qquad (d')$$

On a pu déduire (d) et (d') de l'hypothèse $s \in dom(R)$; donc p = (p1;p2) est cohérent par rapport à R. CQFD

**Exemple**

$S$ = **réel**;

$R = \{(s,s') \mid \log(s^2+1) \leq s' \leq 2*\log(s^2+1)\}$.

On pose

$R1 = \{(s,s') \mid s' = s^2+1\}$,

$R2 = \{(s,s') \mid s>0 \ \& \ \log(s) \leq s' \leq 2*\log(s)\}$.

On a

$R1*R2$

$= \{(s,s') \mid \exists s'': s''=s^2+1 \ \& \ s''>0 \ \& \ \log(s'') \leq s' \leq 2*\log(s'')\}$

$= \{(s,s') \mid s^2+1>0 \ \& \ \log(s^2+1) \leq s' \leq 2*\log(s^2+1) \ \& \ \exists s'': s''=s^2+1\}$

$= \{(s,s') \mid \log(s^2+1) \leq s' \leq 2*\log(s^2+1)\}$

$= R$.

D'autre part, on a

$cod(R1) = \{s \mid s \geq 1\}$,

$dom(R2) = \{s \mid s>0\}$,

donc $cod(R1) \subseteq dom(R2)$. Une autre décomposition acceptable serait:

$R1 = \{(s,s') \mid s^2+1 \leq s' \leq (s^2+1)^2\}$,

$R2 = \{(s,s') \mid s>0 \ \& \ s'=\log(s)\}$.

Il est simple de vérifier que R=R1*R2. On vérifie la clause $cod(R1) \subseteq dom(R2)$:

$cod(R1) = \{s \mid \exists x: x^2+1 \leq s \leq (x^2+1)^2\}$

$= \{s \mid \exists x>1: x \leq s \leq x^2\}$.

$dom(R2) = \{s \mid s>0\}$.

Il est clair que $cod(R1) \subseteq dom(R2)$. []

## B. Règle de l'Alternation (ou de l'union)[1].

Cette règle décompose une relation (complexe) en l'union de deux relations intrisèquement plus simples, CA et CS (Clause Alors et Clause Sinon).

**Règle.** Etant donnée R, trouver des relations CA et CS telles que

i) $R = CA \cup CS$,

ii) $dom(CA) \cap dom(CS) = \emptyset$.

**Proposition 6.** Si ca est correct par rapport à CA et cs est correct par rapport à CS alors

**p = si t alors ca sinon cs**

est correct par rapport à R, où t est le prédicat ∂dom(CA).

**Preuve.** On prouvera que dom(RO[p])=dom(R). Pour cela on analysera R'=RO[p]. On considère la pré-projection de cette relation par rapport à t:

$$R' = I(t)*(RO[p]) \cup I(\sim t)*(RO[p]). \qquad (a)$$

En vertu de (i) et (ii) et de la définition de t, on a

$I(t)*R = CA$

$I(\sim t)*R = CS.$

En vertu de la défintion sémantique de l'alternation, on a

$I(t)*[p] = I(t)*[ca],$

$I(\sim t)*[p] = I(\sim t)*[cs].$

Donc L'équation (a) peut être réécrite comme

$$R' = CA \circ I(t)*[ca] \cup CS \circ I(\sim t)*[cs]. \qquad (b)$$

Par définition de t, ceci peut s'écrire

$$R' = I(t)*CA \circ I(t)*[ca] \cup I(\sim t)*CS \circ I(\sim t)*[cs],$$

ou encore

$$R' = I(t)*(CAO[ca]) \cup I(\sim t)*(CSO[cs]).$$

Le domaine de (CAO[ca]) est un sous-ensemble du domaine de CA, qui est SIt; donc $I(t)*(CAO[ca]) = CAO[ca]$. De même, $I(\sim t)*(CSO[cs]) = CSO[cs]$. Donc

$$R' = CAO[ca] \cup CSO[cs]. \qquad (c)$$

dom(RO[p])

= dom(R') par définition de R'.

= dom(CAO[ca])∪dom(CSO[cs]) en vertu de (c)

= dom(CA) ∪ dom(CS) en vertu de la cohérence de ca et cs

= dom(R) en vertu de (i)

Donc p est correct par rapport à R. CQFD

**Remarque.** On prouve que CA est intrinsèquement plus simple que R. Pour cela, on utilise la proposition 2 de ce chapitre. Les hypothèses de cette proposition sont

(a) $CA \subseteq R$,

(b) $dom(CA) \cap dom(R-CA)=\emptyset$.

L'hypothèse (a) découle naturellement de la clause (i) de la règle. On peut déduire (b) de (ii) pourvu que l'on prouve que CS=R-CA. Puisque R=CA∪CS, $CS \subseteq R-CA$; d'autre part, puisque $dom(CA) \cap dom(CS)=\emptyset$, on a $CA \cap CS=\emptyset$, donc $R-CA \subseteq CS$.

Quanbd on applique la règle de l'alternation à une spécification R, on est assuré de générer des relations CA et CS qui sont intrinsèquement plus simples que R.

**Exemple**

```
S = ens
    a, b, c: entier
fin,
R = {(s,s') | c(s')=max(a(s),b(s))}.
```

On prends

$CA = \{(s,s') \mid a(s) > b(s) \ \&\ c(s')=a(s)\},$

$CS = \{(s,s') \mid a(s) \leq b(s) \ \&\ c(s')=b(s)\}.$

On a

$CA \cup CS$
$= \{(s,s') \mid a(s) > b(s) \ \&\ c(s') = a(s) \ \vee\ a(s) \leq b(s) \ \&\ c(s') = b(s)\}$
$= \{(s,s') \mid c(s') = \max(a(s),b(s))\}$
$= R.$

De plus

$dom(CA) = \{s \mid a(s) > b(s)\},$
$dom(CS) = \{s \mid a(s) \leq b(s)\}.$

Donc $dom(CA) \cap dom(CS)$ est effectivement vide. []

**Remarque.** Bien que la décomposition par la règle de l'alternation génère toujours des relations intrisèquement plus simples, elle (la décomposition) peut ne pas être judicieuse; considérons par exemple la décomposition suivante de la relation R ci-dessus.

$CA = \{(s,s') \mid a(s)$ est pair $\&\ c(s') = \max(a(s),b(s))\},$
$CS = \{(s,s') \mid a(s)$ est impair $\&\ c(s') = \max(a(s),b(s))\}.$

En fait il doit intervenir une composante représentationnelle dans l'application de cette règle. []

## C. Règle de l'Itération (ou de la fermeture transitive) 'r'.

Cette règle décompose une relation (complexe) en la fermeture transitive réflexive d'une relation représentationnellement plus simple. Elle ne s'applique que quand la spécification donnée vérifie une certaine condition de faisabilité.

**Règle.** Etant donnée une relation R sur S telle que $dom(R)=S$. Si R vérifie la condition de faisabilité

$$I(cod(R)) * R = I(cod(R)) \qquad (F)$$

alors déterminer la relation B telle que

i) $dom(B) = S - cod(R)$,
ii) $B^+$ est un ordre bien fondé,
iii) $R = B^* * I(\neg \in dom(B))$.

**Proposition 7.** a) Si R ne vérifie pas la condition (F) alors il n'existe pas de relation B vérifiant (i), (ii) et (iii).
b) Si R vérifie la condition (F) alors il existe une relation B vérifiant (i), (ii) et (iii) et pour tout énoncé b correct par rapport à B, l'énoncé

**w = tantque t faire b**

est correct par rapport à R, où $t(s) = s \in dom(B)$.

**Preuve.** Tout d'abord mentionnons que, en vertu des arguments présentés au chapitre 4, l'hypothèse $dom(R)=S$ n'affecte pas la généralité de notre étude; remarquons aussi que cette hypothèse implique, en vertu de (i), que $\neg t(s) \Rightarrow s \in cod(R)$. Pour prouver la clause (a) de la proposition, on prouvera que s'il existe B vérifiant les conditions (i), (ii) et (iii) alors F est vérifiée.

$I(\neg t) * R$
$= I(\neg t) * B^* * I(\neg t) \cup I(\neg t) * I(\neg t)$ en vertu de (iii)
$= \emptyset * I(\neg t) \cup I(\neg t)$ en vertu de (i)
$= I(\neg t).$

Soit R une relation vérifiant F. On pose $B = I(t) * R$ et on vérifie tour à tour toutes les clauses (i), (ii) et (iii) pour cette relation.

Clause (i). On a dom(B)=dom(I(t)*R)=S¦t vu que dom(R)=S. Par ailleurs, cod(R)=S¦~t -par définition de t. Donc dom(B)=S-cod(R).

Clause (ii). Pour calculer $B^+$, on commencera par calculer $B^2$; on a

$B^2 = I(t)*R*I(t)*R$
$= I(t)*R*I(cod(R))*I(t)*R$
$= I(t)*R*I(\sim t)*I(t)*R$
$= \emptyset$.

Donc $B^+ = B = I(t)*R$. Ceci est bien un ordre bien fondé vu que $B^2 = \emptyset \subseteq B$ et que pour tout s, $s \in dom(B^2)$.

Clause (iii). On a

| | |
|---|---|
| $B^**I(\sim t)$ | |
| $= B^+*I(\sim t) \cup I(\sim t)$ | par définition de $B^*$ |
| $= B*I(\sim t) \cup I(\sim t)$ | car $B^+=B$ |
| $= I(t)*R*I(\sim t) \cup I(\sim t)$ | car $B=I(t)*R$ |
| $= I(t)*R \cup I(\sim t)$ | car cod(R)=S¦~t |
| $= I(t)*R \cup I(\sim t)*R$ | condition F |
| $= R$ | Pré-projection de R sur t. |

Soit b un programme correct par rapport à B. Pour prouver que w = **(tantque t faire b)** est correct par rapport à R, on utilisera le théorème d'induction par sous-but (Subgoal Induction Theorem, [MORR77]). Une version modifiée de ce théorème (due à [MILI83i]) est donnée ci-dessous.

> Soit w = **(tantque t faire b)** un programme et soit R une relation sur S telle que dom(R)=S; si
> a) dom([w]) = S,
> b) ~t(s) => (s,s)∈R,
> c) t(s) & ([b](s),s*)∈R => (s,s*)∈R
> alors w est correct par rapport à R.

Pour prouver (a), on doit prouver que w termine pour tout s dans S: ceci résulte de la condition (ii) et de l'hypothèse que b est correct par rapport à B.

Pour prouver (b), il suffit de constater: ~t(s) => (s,s)∈I(~t) => (s,s)∈R.

Preuve de (c): Nous devons déduire (s,s*)∈R des hypothèses suivantes:

$$t(s) \qquad (0)$$
$$([b](s),s^*) \in R \qquad (1)$$
$$s \in dom(B) \Rightarrow s \in dom([b]) \;\&\; (s,[b](s)) \in B. \qquad (2)$$

A partir de (0) et (2) et de la convention que t(s) = (s∈dom(B)), on déduit

$$s \in dom([b]) \;\&\; (s,[b](s)) \in B. \qquad (3)$$

On combine (1) et (3) pour déduire

$$(s,s^*) \in B*R.$$

Or, $B*R$
$= B*B^**I(\sim t)$
$= B^+*I(\sim t)$
$\subseteq R$.

Donc (s,s*)∈R. CQFD

Nous donnons trois exemples d'application de cette règle, tous

les trois étant fort simples; on les utilise uniquement pour illustrer les mécanismes de la règle de l'itération. Un exemple plus élaboré sera donné à la section 4.

**Exemple**

S = ens
crt
a, b, c: entier;
sse
$b \geq 0$
**fin.**
R = {(s,s') | a(s')=a(s) & b(s')=0 & c(s')=c(s)+a(s)*b(s)}.

On vérifie la condition de faisabilité pour R: cod(R) = {s | b(s)=0}; alors

I(cod(R))*R
= {(s,s') | b(s)≠0 & a(s')=a(s) & b(s')=b(s) & c(s')=c(s)}
= I(cod(R)).

On est assuré de l'existence d'une relation B. On considère les équations (i), (ii) et (iii) et on essaie de trouver une relation B qui vérifie ces conditions. On propose

B={(s,s') | b(s)≠0 & a(s')=a(s) & b(s')=b(s)-1 & c(s')=c(s)+a(s)}.

On vérifie tour à tour les clauses (i), (ii) et (iii).

(i): dom(B) = {s | b(s)≠0} = S-{s | b(s)=0} = S-cod(R).

(ii): $B^+$
= {(s,s') | ∃i>0: b(s)≠0 & a(s')=a(s) & b(s')=b(s)-i &
c(s')=c(s)+i*a(s)}
$\subseteq$ {(s,s') | ∃i>0: b(s')=b(s)-i}
= {(s,s') | b(s')<b(s)}.

Sur l'ensemble S, ceci est un ordre bien fondé; donc $B^+$ qui en est un surensemble est aussi un ordre bien fondé.

(iii): $B^+$
= {(s,s') | b(s)≠0 & a(s')=a(s) &
∃i>0: b(s')=b(s)-i & c(s')=c(s)+i*a(s)}
= {(s,s') | b(s)≠0 & a(s')=a(s) & b(s')<b(s) &
c(s')=c(s)+(b(s)-b(s'))*a(s)}.
$B^+$*I(~t)
= {(s,s') | b(s)≠0 & a(s')=a(s) & 0<b(s) & c(s')=c(s)+a(s)*b(s) &
b(s')=0}.

Dans S, b(s)≠0 et b(s)>0 sont équivalents; on a alors

$B^+$*I(~t)
= {(s,s') | b(s)≠0 & a(s')=a(s) & b(s')=0 & c(s')=c(s)+a(s)*b(s)}.

Quant à I(~t), on peut l'écrire

I(~t)
= {(s,s') | b(s)=0 & a(s')=a(s) & b(s')=b(s) & c(s')=c(s)}

ou encore

= {(s,s') | b(s)=0 & a(s')=a(s) & b(s')=0 & c(s')=c(s)+a(s)*b(s)}.

Donc $B^+$*I(~t) $\cup$ I(~t)
= {(s,s') | a(s')=a(s) & b(s')=0 & c(s')=c(s)+a(s)*b(s)}
= R. []

**Exemple**

S = {0,1,2,3,4,5,6}.
R = {(6,0),(5,1),(4,0),(3,1),(2,0),(1,1),(0,0)}.

On a dom(R)=S et cod(R)={0,1}; il est clair que la condition F est vérifiée. On choisit B = {(6,4),(5,3),(4,2),(3,1),(2,0)}; alors

dom(B) = {2,3,4,5,6} et la condition (i) est vérifiée. D'autre part,
B* = {(6,6),(6,4),(6,2),(6,0),
(5,5),(5,3),(5,1),
(4,4),(4,2),(4,0),
(3,3),(3,1),
(2,2),(2,0),
(1,1),
(0,0)}.
Alors on a B**I(~t) = {(6,0),(5,1),(4,0),(3,1),(2,0),(1,1),(0,0)} = R.
[]

**Exemple**

S = naturel;
R = {(s,s')| s>4 & s'≤4) U {(s,s')| s≤4 & s'=s}.

Remarquez que cette relation n'est pas déterministe. On a dom(R)=S; d'autre part, cod(R) = {s| s≤4} et il est clair que la condition de faisabilité est vérifiée par R puisque

I(cod(R)) * {(s,s')| s>4 & s'≤4} = Ø

et que

I(cod(R)) * {(s,s')| s≤4 & s'=s} = I(cod(R)).

On choisit B = {(s,s')| s>4 & s-4≤s'≤s-1}. Alors dom(B) = {s| s>4}, donc la condition (i) est vérifiée. De plus, on a

B+ = {(s,s')| s>4 & s'<s} et
B+*I(~t) = {(s,s')| s>4 & s'≤4} et
B**I(~t) = {(s,s')| s>4 & s'≤4} U {(s,s')| s≤4 & s'=s}
= R.

Interprétation: R exprime que si s≤4 alors s'=s sinon on doit trouver s'≤4; B exprime que si s>4 alors s' peut être obtenu à partir de s en faisant des décrémentations successives de longueur soit 1, soit 2, soit 3 soit 4. []

**Remarque.** On peut imaginer que la recherche d'une relation B vérifiant les clauses (i), (ii) et (iii) s'effectue en deux étapes.

**Etape 1.** Chercher une relation X telle que
i) dom(X) = S-cod(R),
ii) X est un ordre bien-fondé,
iii) R = X*I(~t) U I(~t).

**Etape 2.** Chercher un noyau transitif B de X.

Les compromis qui interviennent dans le choix de B une fois que X est choisi sont les suivants:

a) On peut choisir pour B une racine transitive irréductible de X. tel a été le choix fait dans les deux premiers exemples étudiés ci-dessus.
b) On peut choisir pour B une racine transitive de X, arbitrairement grande (donc -puisque son domaine est fixé- arbitrairement non-définie). Tel a été le choix fait dans le troisième exemple ci-dessus.

Avant de comparer ces options, il y lieu de faire la remarque suivante: Une fois que X est choisi, le concepteur n'a aucune latitude quant au choix du domaine de B, puisque dom(B)=dom(X) (car X=B+); le seul choix qui lui revient est de savoir quelle(s) image(s)

associer à chaque élément de dom(X). Plus la relation B est petite (par inclusion) et plus elle définie (donc difficile à résoudre dans les étapes ultérieures).

Dans l'option (a), la relation affectée à B tend à être très définie, donc très intrinsèquement complexe; c'est pour des raisons de simplicité représentationnelle qu'un concepteur choisirait une racine transitive irréductible de X. Pour illustrer ce cas, nous considérons le premier exemple étudié ci-dessus; dans cet exemple,

$X = \{(s,s') \mid b(s)\neq 0 \ \&\ a(s')=a(s) \ \&\ b(s')<b(s) \ \&\ c(s')=c(s)+a(s)*(b(s)-b(s'))\}.$

La relation B qui a été choisie dans cet exemple est un noyau transitif irréductible de X; un noyau transitif non-irréductible serait, par exemple,

$B' = \{(s,s') \mid b(s)>0 \ \&\ a(s')=a(s) \ \&\ b(s')=b(s)-1 \ \&\ c(s')=c(s)+a(s)\}$
$\cup$
$\{(s,s') \mid b(s)>1 \ \&\ a(s')=a(s) \ \&\ b(s')=b(s)-2 \ \&\ c(s')=c(s)+2*a(s)\}.$

B' est intrinsèquement plus simple que B mais -dans notre mode de représentation- B est représentationnellement plus simple que B'. En choisissant B dans l'exemple, nous avons préféré la simplicité représentationnelle à la simplicité intrinsèque.

L'option (b) quant à elle, tend à générer des relations intrinsèquement simples. Une telle option laisse beaucoup de liberté au concepteur dans les étapes ultérieures, mais elle enregistre peu de progrès dans le processus de conception; il suffit de noter que la relation $B=I(t)*R$ est la relation la plus intrinsèquement simple pour B; il est clair qu'un tel choix pour B ne fait pratiquement pas progresser le processus de conception. []

## D. Règle de la Concurrence (ou de l'intersection)[4]

Cette règle décompose une relation (complexe) en l'intersection de deux relations intrinsèquement plus simples. Avant de présenter la règle, nous introduisons deux lemmes.

Soient S1 et S2 deux ensembles et soit S leur produit cartésien; on dénote un élément de S par $s=(s1,s2)$. Soient R1 et R2 deux relations sur S ayant la forme

$R1 = \{((s1,s2),(s1',s2')) \mid q1(s1,s1')\},$
$R2 = \{((s1,s2),(s1',s2')) \mid q2(s2,s2')\}.$

On dira que R1 et R2 sont espace-disjoints. Alors on a

**Lemme 1.** $dom(R1) \cap dom(R2) = dom(R1 \cap R2)$.

**Preuve.** Notez que $dom(R1 \cap R2) \subseteq dom(R1) \cap dom(R2)$ est une tautologie; donc on se contentera de prouver $dom(R1) \cap dom(R2) \subseteq dom(R1 \cap R2)$. Soit $s=(s1,s2)$ un élément de $dom(R1) \cap dom(R2)$; alors il existe $s'=(s1',s2')$ tel que $(s,s') \in R1$ et $s''=(s1'',s2'')$ tel que $(s,s'') \in R2$. Si l'on définit $s^* = (s1',s2'')$, il est simple de prouver que $s^* \in s.R1 \cap R2$; donc $s \in dom(R1 \cap R2)$. CQFD

**Lemme 2.** $s.R1 \cap s.R2 = s.(R1 \cap R2)$.

**Preuve.** Notez que $s.(R1 \cap R2) \subseteq s.R1 \cap s.R2$ est une tautologie; donc on se

contentera de prouver s.R1Os.R2⊆s.(R1OR2). Soit s'=(s1',s2') un élément de s.R1Os.R2. De (s,s')∈R1 on déduit q1(s1,s1'); de (s,s')∈R2 on déduit q2(s2,s2'); de q1(s1,s1') et q2(s2,s2') on déduit (s,s')∈R1OR2. Donc s'∈s.(R1OR2).

**Règle.** Etant donnée une relation R sur S, trouver deux relations espace-disjointes R1 et R2 telles que
i) R = R1OR2,
ii) dom(R) = dom(R1) = dom(R2).

**Proposition 8.** Si p1 est correct par rapport à R1 et p2 est correct par rapport à R2 alors p = **cobegin** p1, p2 **coend** est correct par rapport à R.

**Preuve.** Soient S1 et S2 les espaces associés à p1 (et R1) et p2 (et R2). Souvenons-nous que les abstractions fonctionnelles de p1 et p2 sont (dans un contexte de programmation concurrente) des relations non nécessairement déterministes. Donc la cohérence de p1 par rapport à R1 s'écrit

$$s\in dom(R1) \Rightarrow s\in dom([p1]) \;\&\; s.[p1]\subseteq s.R1. \qquad (a)$$

De même, la cohérence de p2 par rapport à R2 s'écrit

$$s\in dom(R2) \Rightarrow s\in dom([p2]) \;\&\; s.[p2]\subseteq s.R2. \qquad (b)$$

Soit s un élément de dom(R). En vertu de (ii), s∈dom(R1) et s∈dom(R2); en vertu de (a) et (b),

$$s\in dom([p1])\cap dom[p2]) \qquad (d1)$$

et

$$s.[p1]\subseteq s.R1 \;\&\; s.[p2]\subseteq s.R2. \qquad (r1)$$

En vertu de (d1) et du lemme 1, on a

$$s\in dom([p1]\cap[p2]). \qquad (d2)$$

En vertu de la sémantque de la concurrence on déduit

$$s\in dom([\textbf{cobegin } p1,\ p2\ \textbf{coend}]). \qquad (d3)$$

En vertu de (r1) on déduit

$$s.[p1]\circ s.[p2] \subseteq s.R1\circ s.R2. \qquad (r2)$$

Puisque $s.[p1]\cap[p2]\subseteq s.[p1]\circ s.[p2]$ est une tautologie, on déduit de (r2):

$$s.([p1]\cap[p2]) \subseteq s.R1\circ s.R2. \qquad (r3)$$

En vertu du lemme 2, on déduit

$$s.([p1]\cap[p2]) \subseteq s.(R1\circ R2). \qquad (r4)$$

En vertu de la sémantique de la concurrence et de (i), on déduit

$$s.([\textbf{cobegin } p1,\ p2\ \textbf{coend}]) \subseteq s.R. \qquad (r5)$$

On a déduit (d3) et (r5) de l'hypothèse s∈dom(R). CQFD

**Remarque.** R est plus-définie que R1 ( et que R2) puisuqe
a) d'après (ii), dom(R1)⊆dom(R),
b) d'après (i), ∀s∈dom(R1), s.R⊆s.R1.
Donc R1 (ainsi que R2) est intrinsèquement plus simple que R. □

**Exemple**

```
S = ens
      a, b, c, d: entier
fin,
R = {(s,s') | a(s')=min(a(s),b(s)) & c(s')=min(c(s),d(s))}.
```

On pose

```
S1 = ens
      a, b: entier
```

**fin**, et
S2 = **ens**
c, d: **entier**
**fin**.
Aussi, on propose
R1 = {((s1,s2),(s1',s2'))| a(s1')=min(a(s1),b(s1))},
R2 = {((s1,s2),(s1',s2'))| c(s2')=min(c(s2),d(s2))}. []

## 3.3. Règle de Généralisation

A partir d'une relation donnée, cette règle génère une relation intrinsèquement plus complexe. Il existe une motivation pour cette règle paradoxale, laquelle motivation nous introduisons ci-dessous par un exemple simple.

**Exemple**
S = {0,1,2,3,4,5,6}.
R = {(6,0),(5,1),(4,0),(3,1)}.
Il est clair que l'instruction itérative w = **tantque** s>1 **faire** s:=s-2 est correcte par rapport R; il est alors raisonnable de penser qu'en appliquant la règle de l'itération à R on puisse aboutir à l'instruction w. Mais R ne vérifie pas la condition de faisabilité de la règle de l'itération puisque
I(cod(R)) = {(0,0),(1,1)},
I(cod(R))*R = R.
La règle de la généralisation servirait dans ce cas précis pour transformer R en R' = {(6,0),(5,1),(4,0),(3,1),(2,0),(1,1),(0,0)}, qui vérifie la condition de faisabilité. []

**Règle.** Etant donnée une relation R, trouver une relation R' plus-définie que R.

**Proposition 9.** Si p est correct par rapport à R' alors il est correct par rapport à R.

Preuve. Si p est correct par rapport à R' alors [p] est plus-défini que R', donc il est plus-défini que R. CQFD

## 4. Exemple d'Application

L'approche présentée dans ce chapitre n'a pas été utilisée dans des contextes industriels de conception de programmes. Dans cette section, nous présentons un exemple d'application de cette approche; sans être véritablement "réaliste", l'exemple en question est assez complexe pour exercer toutes les règles du système, et en montrer l'usage et les possibilités.

**Exemple**

```
S = ens
    a: tableau [1..n] de réel;
    sm, my: réel;
    i: 1..n+1
fin,
```

où n est un entier plus grand ou égal à 1.

$R = \{(s,s') \mid my(s') = (|a(s)[1]| + |a(s)[2]| + \ldots + |a(s)[n]|)/n\}$;

R exprime que my(s') doit contenir la moyenne des valeurs absolues des composantes de a. On dénote par **sigma**(i,j) la quantité suivante: Si $i \leq j$ alors $(|a(s)[i]| + |a(s)[i+1]| + \ldots + |a(s)[j]|)$ Sinon 0.

On applique la règle de la <u>séquence</u> à R:

$R = R1 * R2$

où

$R1 = \{(s,s') \mid sm(s') = \mathbf{sigma}(1,n)\}$,

$R2 = \{(s,s') \mid my(s') = sm(s)/n\}$.

On vérifie les prémisses de cette règle:

$R1 * R2$

$= \{(s,s') \mid \exists s'':\ (s,s'') \in R1 \ \&\ (s'',s') \in R2\}$

$= \{(s,s') \mid \exists s'':\ sm(s'') = \mathbf{sigma}(1,n) \ \&\ my(s') = sm(s'')/n\}$

$= \{(s,s') \mid my(s') = \mathbf{sigma}(1,n)/n \ \&\ \exists s'':\ my(s') = sm(s'')/n\}$

$= \{(s,s') \mid my(s') = \mathbf{sigma}(1,n)/n\}$

$= R.$

Donc la prémisse (i) est vérifiée. Quant à la prémisse (ii), elle résulte immédiatement de la constatation que dom(R2)=S.

La relation R2 est <u>assez simple</u>; elle sera traitée plus tard par la règle de l'affectation.

On applique la règle de la <u>séquence</u> à R1:

$R1 = R11 * R12$,

où

$R11 = \{(s,s') \mid sm(s') = 0 \ \&\ i(s') = 1\}$,

$R12 = \{(s,s') \mid sm(s') = sm(s) + \mathbf{sigma}(i(s),n)\}$.

On appliquera la règle de la concurrence à la relation R11. Quant à la relation R2, elle ne se prête pas, dans sa forme présente, à l'application de la règle de l'itération vu que cod(R12)=S et qu'il est clair que I(S)*R12 n'est égal à I(S) (qui est I).

On applique la règle de la <u>généralisation</u> à R12:

$R12' = \{(s,s') \mid a(s') = a(s) \ \&\ sm(s') = sm(s) + \mathbf{sigma}(i(s),n) \ \&\ my(s') = my(s) \ \&\ i(s') = n+1\}$.

Il est clair que R12 et R12' ont le même domaine; de plus, à cause des

clauses additionnelles de R12', R12'$\subseteq$R12. Donc R12' est plus-définie que R.

On applique la règle de l'itération à R12': Tout d'abord, on vérifie la condition de faisabilité. On a cod(R12') = {s| i(s)=n+1}. Donc, si on dénote par I' la relation I(cod(R12)), on a I' = {(s,s')| s'=s & i(s)=n+1}. On calcule

I'*R = {(s,s')| i(s')=n+1 & a(s')=a(s) & sm(s')=sm(s)+**sigma**(i(s),n) & my(s')=my(s) & i(s')=n+1}
= {(s,s')| i(s)=n+1 & a(s')=a(s) & sm(s')=sm(s) & my(s')=my(s) & i(s')=i(s)}
= {(s,s')| i(s)=n+1 & s'=s}.

Maintenant on doit déterminer X tel que
(i) dom(X) = S-cod(R)
(ii) X est un ordre bien fondé,
(iii) R = X*I' $\cup$ I'.

On propose

X = {(s,s')| i(s)≠n+1 & a(s')=a(s) & sm(s')=sm(s)+**sigma**(i(s),i(s')-1) & my(s')=my(s) & i(s)<i(s')}.

Notez qu'il est implicite que s et s' sont dans S, donc i(s) et i(s') sont dans 1..n+1. La premisse (i) se déduit de la clause i(s)≠n+1 et la prémisse (ii) se déduit de la clause i(s)<i(s') dans la description de la relation X. On vérifie la prémisse (iii):

X*I'
= {(s,s')| i(s)≠n+1 & a(s')=a(s) & my(s')=my(s) & i(s)<i(s') & sm(s')=sm(s)+**sigma**(i(s),i(s')-1) & i(s')=n+1}.
= {(s,s')| i(s)≠n+1 & a(s')=a(s) & my(s')=my(s) & i(s)<i(s') & sm(s')=sm(s)+**sigma**(i(s),i(s')-1) & i(s')=n+1}
= {(s,s')| i(s)≠n+1 & a(s')=a(s) & sm(s')=sm(s)+**sigma**(i(s),n) & my(s')=my(s) & i(s')=n+1}.

Par ailleurs, on peut écrire I' de la façon suivante

I'
= {(s,s')| i(s)=n+1 & a(s')=a(s) & sm(s')=sm(s)+**sigma**(i(s),n) & my(s')=my(s) & i(s')=n+1}.

Donc X*I' $\cup$ I'
= {(s,s')| a(s')=a(s) & sm(s')=sm(s)+**sigma**((i(s),n) & my(s')=my(s) & i(s')=n+1}.

Maintenant il convient de choisir une racine transitive de X. On pose

B = {(s,s')| i(s)≠n+1 & a(s')=a(s) & sm(s')=sm(s)+|a(s)[i(s)]| & my(s')=my(s) & i(s')=i(s)+1}.

Il est facile de se convaincre que X=B$^+$, si l'on réalise que pour tout k>n, B$^k$=∅.

On applique la règle de l'alternation à la relation B. On propose la décomposition suivante.

B1 = {(s,s')| i(s)≠n+1 & a(s)[i(s)]≥0 & a(s')=a(s) & sm(s')=sm(s)+a(s)[i(s)] & my(s')=my(s) & i(s')=i(s)+1}.
B2 = {(s,s')| i(s)≠n+1 & a(s)[i(s)]<0 & a(s')=a(s) & sm(s')=sm(s)-a(s)[i(s)] & my(s')=my(s) & i(s')=i(s)+1}.

Il est clair que B1$\cup$B2=B et que dom(B1)$\cap$dom(B2)=∅.

On considère la relation R11 et on lui applique la règle de l'intersection. On propose

R111 = {(s,s')| sm(s')=0},
R112 = {(s,s')| i(s')=1}.

Les prémisses de cette règle sont trivialement vérifiées; on peut poser, par exemple, que sm et a forment l'espace S1 alors que i et my forment l'espace S2. Avant d'appliquer la règle de l'affectation aux relations R111, R112, B1, B2 et R2, nous récapitulons les diverses étapes de ce processus de conception dans un tableau à trois colonnes: dans la première colonne on indiquera les règles appliquées; dans la deuxième colonne on indiquera la décomposition relationnelle obtenue; dans la troisième colonne on indiquera la structure qui en résulte au niveau du programme en construction.

| **Règle** | **Décomposition** | **Structure** |
|---|---|---|
| séquence(R) | R1*R2 | p1; p2 |
| séquence(R1) | R11*R12*R2 | p11; p12; p2 |
| généralis.(R12) | R11*R12'*R2 | p11; p12'; p2 |
| Itération(R12') | R11*B**I'*R2 | p11;<br>**tantque** Ɛdom(B) **faire** b;<br>p2. |
| Alternation(B) | R11*(B1UB2)**I'*R2 | p11;<br>**tantque** Ɛdom(B) **faire**<br>  **si** Ɛdom(B1) **alors** b1<br>  **sinon** b2;<br>p2. |
| Concurrence(R11) | (R111OR112)<br>* (B1UB2)*<br>* I'*R2 | **codébut** p111, p112 **cofin**;<br>**tantque** Ɛdom(B) **faire**<br>  **si** Ɛdom(B1) **alors** b1<br>  **sinon** b2;<br>  p2. |

En appliquant la règle de l'affectation à R111, R112, B2 et R2, on trouve, respectivement,

```
p111:  sm:=0
p112:  i:=1
b1:    sm:=sm+a[i]
b2:    sm:=sm-a[i]
p2:    my:=sm/n.
```

D'où le programme

```
p: début
codébut sm:=0; i:=1 cofin;
tantque i≠n+1 faire
   si a[i]>0 alors sm:=sm+a[i]
   sinon sm:=sm-a[i];
my:=sm/n
fin.
```

[]

## 5. Conclusion

Dans ce chapitre nous avons proposé une approche relationnelle à la conception rigoureuse de programmes. L'idée de base de cette approche

est que, étant donné qu'une spécification est une relation, la conception de programmes se fait par décomposition de relations: Des relations complexes sont décomposées -selon des règles formelles- en des relations plus simples. Trois leçons peuvent être tirées de cette approche; nous les discutons tour à tour ci-dessous.

Une décision de conception peut être formalisée comme la résolution d'une équation relationnelle. En effet, chaque règle de décomposition peut être comprise comme un système de une ou plusieurs (in-) équations à une ou plusieurs inconnues relationnelles. Des méthodes algébriques sont présentement étudiées dans le but d'amorcer une approche -sinon formelle- du moins rigoureuse pour la résolution de ces systèmes d'équations (ou inéquations). En particulier, nous nous intéressons au problème de recherche de la racine transitive d'une relation donnée, et des compromis qui interviennent dans le choix de cette racine.

Cette approche relationnelle montre l'analogie qui existe entre le calcul relationnel d'une part et les instructions de langages de programmation d'autre part: Une séquence correspond à un produit relatif; une alternation correspond à une union; une itération correspond à une fermeture transitive et une concurrence correspond à une intersection. Dr Mills prêche souvent que l'on dispose d'un excellent langage de programmation, à savoir le langage des mathématiques. Faisant écho à sa remarque, on peut faire un pas de plus: le langage de programmation (Pascal) est lui-même un langage des mathématiques -plus précisément un langage de calcul relationnel.

Cette approche nous permet d'exhiber les paramètres qui interviennent dans la formulation progressive d'une solution à un problème de programmation. En effet, chaque règle de conception ne fait que définir le contour des solutions acceptables; le choix judicieux d'une solution parmi tous les candidats implique plusieurs paramètres. Parmi ces paramètres, on mentionnera: Comment partitionner le domaine de R dans la règle de l'alternation; comment choisir X dans la règle de l'itération, ensuite comment choisir sa racine transitive; comment définir la simplicité représentationnelle pour chaque instanciation de la règle de la séquence; comment délimiter les espaces S1 et S2 dans la règle de la concurrence. Un programmeur humain fixerait ces paramètres sur la base de ses connaissances sur la programmation, et de la perspicacité que ces connaissances lui confèrent. Un programmeur automatique doit être pourvu de suffisamment de connaissances sur la programmations pour pouvoir prendre des décisions du type mentionné ci-dessus. Présentement, nous étudions des approches à la programmation basées sur la connaissance (knowledge-based programming); le succès de telles approches repose de façon critique sur une identification précise de ce qui constitue les connaissances de la programmation.

Dans un article publié en 1975, Dr Mills [MILL75] montre comment chaque décision de conception par décomposition fonctionnelle remonte au choix d'un paramètre mathématique simple; nous tentons présentement de faire le même travail pour la décomposition relationnelle. La détermination de ces paramètres pour le contexte relationnel serait utile dans la conception d'un système de programmation à base de

connaissances.

> If software Engineers do not use Artificial Intelligence technology, they will have to reinvent it.
> H Simon, Keynote speaker at ICSE-7.

# Chapitre 7

## Algorithmes: Une Tentative de Définition

Dans tout modèle de conception formelle de programmes, une question qui vient naturellement à l'esprit est celle de savoir comment le processus de conception est aiguillé vers un algorithme ou un autre. En d'autres termes, à quel moment dans le processus de conception et comment un algorithme est-il décidé. Dans ce chapitre nous tentons de répondre à cette question en présentant une définition formelle de la notion d'algorithmes; nous justifions cette définition au moyen du modèle de conception de programmes présenté au chapitre précédent.

### 1. Algorithmes: Définition Formelle

Le mot algorithme vient du nom du mathematicien Abu Jaâfar Muhammad Ibnu Mûssa Al Khawarizmi (780-850), originaire de la ville de Khiva (présentement en Union Soviétique), qui fit ses recherches dans la ville de Bassorah (présentement en Irak) et fut l'auteur du livre d'algèbre "Kitabu Al Jabri Wa Al Mukabalah": Le livre de l'intégration et de l'équation. A partir du mot Al Khuwarizmi est né le mot algorisme qui était utilisé pour désigner les procédures arithmétiques de la numération arabe (de même qu'à partir du mot Al Jabr dans le titre du livre est né le mot Algèbre; on pense aussi que l'usage de x comme inconnue en mathématiques a ses origines dans ce livre). L'orthographe du mot devint algorithme et sa sémantique fut étendue pour prendre le sens qui lui est associé de nos jours:

> "L'ensemble des règles opératoires dont l'application permet de résoudre un problème énoncé au moyen d'un nombre fini d'opérations".
> **Grand Dictionnaire Encyclopédique Larousse, 1982.**

La définition qu'on en donne ci-dessous ressemble peu à celle de Larousse, mais elle se veut cohérente avec elle.

**Définition 1.** Un algorithme est un ordre bien fondé.

L'intérêt de cette définition n'est pas d'ordre pratique; cette définition n'est pas utile au programmeur au même titre qu'un éditeur de programmes est utile au programmeur. Plutôt, l'intérêt de cette définition émane de la lumière qu'elle jette sur les mécanismes intimes de la programmation. Dans un article de CACM, H. D. Mills [MILL75] montre comment chaque règle de conception de programme peut être perçue comme un processus à deux étapes:

- Tout d'abord, le choix heuristique d'un paramètre mathématique simple (tel que la partition d'un ensemble, le raffinement d'une relation d'équivalence, le choix de graphes arborescents sur un ensemble, ...).
- Ensuite l'enregistrement automatique de tous les détails cléricals

qui découlent de ce choix.

La définition que nous donnons ici ainsi que les discussions qui l'entourent se veulent contribuer un résultat semblable à celui de Mills: distinguer ce qui constitute l'essence d'une décision de conception de ce qui constitue des activités cléricales d'entérination de cette décision.

Dans la suite de ce chapitre nous tentons de justifier notre définition d'algorithmes: D'abord par des arguments formels (section 2) ensuite par des arguments informels (section 3).

## 2. Algorithmes: Justification Formelle

Notre définition trouve sa justification formelle dans les trois prémisses que énonçons, puis que nous tentons de prouver:

r0: De la façon dont elle est généralement perçue, la notion d'algorithme sous-entend toujours l'idée d'itération; c'est donc dans la règle de conception de l'itération que nous tentons de puiser nos justifications.

r1: Si l'on analyse la règle (de conception) de l'itération, on s'aperçoit que la décision la plus fondamentale dans l'application de cette règle revient au choix d'un ordre bien fondé.

r2: Une séquence d'ordres bien fondés est un ordre bien fondé.

Justification de r0: L'idée de base d'un algorithme est toujours la formulation d'un processus itératif. Les mentions de "nombre fini d'opérations" dans Larousse et de "répétition d'une étape" dans Encyclopaedia Britannica font écho à cette association d'idées. Bien entendu l'application des règles de séquence, alternation et concurrence a un impact sur la formulation de l'algorithme final mais cet impact nous semble bien moins important que celui de la règle de l'itération.

Justification de r1. On formule la règle de l'itération comme suit: Etant donnée une relation R sur S telle que dom(R)=S. Si

$$I(cod(R))*R = I(cod(R))$$

alors on lui applique la décomposition suivante: Trouver un ordre bien fondé X tel que

i) $dom(X) = S-cod(R)$,

ii) $R = (X \cup I)*I(cod(R))$

puis trouver B telle que $B^+=X$.

Une fois que X est choisi, le choix de B est essentiellement automatique; B est généralement choisie de façon à optimiser la simplicité représentationnelle. En fait X définit la décision importante de comment on évolue de l'état initial à l'état final, alors que B définit la décision bien moins importante de savoir à quelle vitesse (nombre d'itérations) on évolue.

Justification de r2. Soient X1, X2, ... Xk des ordres bien fondés sur un ensemble S. Alors la relation X définie comme étant l'ordre

lexicographique construit sur X1, X2, ... Xk est aussi un ordre bien fondé (résultat prouvé par Manna, [MANN74]). Si au cours de la conception d'un programme, la règle de l'itération a été appliquée k fois donnant lieu à X1, X2, ... Xk, on définit l'algorithme du programme comme étant l'ordre bien fondé X construit comme indiqué ci-dessus.

## 3. Algorithmes: Justification Informelle

On considère par exemple le problème de trier un tableau de réels; on définit l'espace suivant.

```
S = ens
    a: tableau [1..n] de réel;
    i: 0..n+1
fin.
```

Le tri par insertion peut être caractérisé par l'ordre bien fondé suivant:

X = {(s,s') | a(s') est une permutation de a(s) &
i(s)<i(s') &
a(s')[1..i(s')] est une permutation de a(s)[1..i(s')] &
a(s)[1..i(s)] est trié &
a(s')[1..i(s')] est trié}.

Le tri par sélection peut être caractérisé par l'ordre bien fondé suivant:

X = {(s,s') | a(s') est une permutation de a(s) &
i(s)<i(s') &
a(s)[1..i(s)] est trié et tous ses éléments sont plus petits que ceux de a(s)[i(s)+1..n] &
a(s')[1..i(s')] est trié et tous ses éléments sont plus petits que ceux de a(s')[i(s')+1..n]}.

Nous voyons que les algorithmes de tri par insertion et par sélection peuvent être définis au moyen d'ordre bien fondés.

On donne un dernier argument d'ordre général liant algorithmes et ordres bien fondés. Un algorithme dicte comment on peut déduire le traitement d'éléments complexes à partir du traitement d'éléments moins complexes .. jusqu'à ce que l'on obtienne un élément assez simple dont le traitement est trivial; la notion d'algorithme porte donc en elle les deux idées de base d'un ordre bien fondé, à savoir l'idée d'ordre ("moins complexe") et la finitude de toute séquence ordonnée (.. "jusqu'à ce que" ..).

# Chapitre 8

## Analyse des Assertions Dans les Programmes Structurés

L'utilisation des assertions exécutables dans le but d'améliorer la fiabilité des programmes est à la fois naturelle et courante: [RAND75], [ANDR79], [ANDE81]. Le but de ce chapitre est d'étudier l'analyse, la vérification et la synthèse de programmes comportant des assertions exécutables.

Le problème d'analyse des assertions se pose en les termes suivants: Soit un programme P comportant des assertions a1, a2, ... ak dispersées à travers son texte et soit s un état initial sur lequel p est exécuté; si le programme p termine dans l'état final s' et que toutes les assertions a1, a2, ... ak se sont avérées vraies quand elles ont été testées, quelle assertion A peut-on affirmer entre s et s'? En quelque sorte, il s'agit de déduire une assertion globale A des assertions locales a1, a2, ... ak.

Le problème de vérification se pose en les termes suivants: Etant donné un programme comportant des assertions a1, a2, .. ak, et étant donnée une assertion globale A (entre états initiaux et états finaux) peut-on dire que: à chaque fois que le programme est exécuté sur un état initial s, que toutes les assertions testées se trouvent être vérifiées et qu'il termine dans un état final s' alors l'assertion globale A est vraie pour la paire (s,s')?

Le problème de synthèse des assertions exécutables se pose en les termes suivants: Soit à écrire un programme p dont on veut vérifier, pour chaque état initial s, que l'état final s' est tel que l'assertion A(s,s') est vérifiée. Sachant qu'il n'est pas pratique de tester A(s,s') à la fin du programme, comment peut-on disperser des assertions a1, a2, ... ak à travers le programme de telle sorte que, à chaque fois que le programme p est exécuté et que toutes ses assertions s'avèrent vérifiées quand elles sont testées, on puisse dire que l'assertion A lie l'état initial à l'etat final.

Une approche prédicative à ces problèmes est prise dans [MILI81d] et [MILI82c]. Dans ce chapitre, nous prendrons une approche relationnelle à ce même problème. La section suivante discute l'architecture des programmes à assertions et introduit les notions nécessaires à leur étude; dans les trois sections qui la suivent, on reformule les problèmes d'analyse, vérification et conception de programmes à assertions et on tente de les résoudre.

## 1. Architecture des Programmes à Assertions

### 1.1. Nature d'une Assertion

Soit p un programme sur l'espace S et soit b un block de code dans ce programme. Soit à tester le comportement du block b par une assertion

a; on écrira alors

    b;
    si non a alors message,

où message est une procédure qui signale que a n'est pas vérifé en envoyant un message à un fichier dédié. Si l'assertion a ne fait référence qu'à l'état présent du programme, alors elle est fort limitée quant aux détails fonctionnels qu'elle peut vérifier; en effet, le plus qu'elle peut vérifier est alors de savoir si s∈dom([b]). Dans sa forme la plus générale, une assertion testant le comportement de b doit faire référence, non seulement à l'état courant s généré par b, mais aussi à l'état initial $\underline{s}$ (avant l'exécution de b); alors il devient possible de vérifier tous les détails fonctionnels pertinents à b, y compris ($\underline{s}$,s)∈[b]. Pour illustrer que l'assertion a réfère potentiellement à $\underline{s}$ et à s, on la représentera par a($\underline{s}$,s); ainsi l'apparatus qui doit être mis en place pour tester l'assertion a à propos du segment de code b est le suivant:

    $\underline{s}$:=s;
    b (*modifie s mais garde $\underline{s}$ intact*);
    si non a($\underline{s}$,s) alors message.

Une telle assertion définit tout naturellement la relation R = {(s,s') | a(s,s')}.

## 1.2. Force d'une Assertion

Dans cette section nous tentons de définir la notion intuitive de force d'une assertion. Si l'on s'intéressait à des assertions à une seule variable, disons a(s), alors il aurait été naturel de dire que l'assertion a(s) est plus forte que l'assertion a'(s) si et seulement si a(s)=>a'(s). Or nous avons affaire en général à des assertions à deux variables, de la forme a($\underline{s}$,s).

Pour évaluer la force d'une assertion à deux variables, telle que a($\underline{s}$,s), on se réfèrera à la relation R définie par cette assertion -en l'occurrence R = {(s,s') | a(s,s')}- et on mesurera la force de a par la quantité d'information que porte R. Plus précisément, on dira que l'assertion a est plus-forte que l'assertion a' si et seulement si la relation R = {(s,s') | a(s,s')} est plus-définie que la relation R' = {(s,s') | a'(s,s')}. En termes relationnels, ceci s'écrit comme

    dom(R') $\subseteq$ dom(R),
    ∀s∈dom(R'), s.R$\subseteq$s.R'.

En termes assertionnels, ceci s'écrit comme

    ∃s':a'(s,s') => ∃s':a(s,s')
    ∃s':a'(s,s') => (∀s', a(s,s')=>a'(s,s')).

Le fait qu'il est quasiment impossible de deviner ces formulations assertionnelles sans avoir au préalable interprété les assertions a et a' en termes relationnels montre l'utilité des relations pour ce type de manipulations.

## 1.3. Usage des Assertions

Un bloc élémentaire à assertions est un block de code Pascal qui a la structure suivante:

    $\underline{s}$:=s;

```
    b;
    si non a(s̲,s) alors message.
```

L'instruction (s̲:=s) représente la sauvegarde des variables de l'état s dans des variables homologues de l'état s̲; il se peut qu'en pratique il ne soit pas nécessaire de sauvegarder toutes les variables de s mais seulement quelques unes (par exemple, seulement celles qui sont modifiées par b); afin de simplifier nos discussions, nous continuerons de supposer que tout l'état s est sauvegardé.

**Exemple**

Nous donnons des exemples de blocs élémentaires à assertions sur l'espace S = **réel**.

```
bea1:  s̲:=s;
       s:=s*s;
       si non (s=s̲²) alors message.
bea2:  s̲:=s;
       s:=s*s;
       si non (s̲>1 => s>s̲) alors message.
bea3:  s̲:=s;
       s:=s*s;
       si non (s>0) alors message.                              []
```

Des blocs élémentaires à assertions peuvent être combinés ensembles au moyen des diverses structures de contrôle de Pascal pour former ce que l'on appelle des programmes à assertions (asserted programs); ainsi si bea1 et bea2 sont des blocs élémentaires à assertions alors **début** bea1;bea2 **fin**, **si** t **alors** bea1 **sinon** bea2, et **tantque** t **faire** bea1 sont des programmes à assertions.

**Exemple.**

Nous donnons un exemple schématique de programme à assertions sur un espace S.

```
pa:  début
     s̲:=s;
     b1;
     L1: si non a1(s̲,s) alors message(1);
     s̲:=s;
     tantque t faire
        début
        b2;
        L2: si non a2(s̲,s) alors message(2);
        s̲:=s;
        fin;
     b3;
     L3: si non a3(s̲,s) alors message(3);
     s̲:=s;
     si u alors
        début
        b4;
        L4: si non a4(s̲,s) alors message;
        s̲:=s;
        fin
     sinon
        début
        b5;
```

```
        L5: si non a5(s,s) alors message(5);
        s:=s;
        fin
    fin.                                                    []
```

Notez que tout bloc de code peut être -à la limite- considéré comme un bloc élémentaire à assertions, avec l'assertion triviale **vrai**:

```
    s:=s;
    b;
    L: si non vrai alors message.
```

Donc si un programme à assertions pa contient des blocs élémentaires à assertions et des blocs de code sans assertions, on peut par généralisation considérer qu'il est composé exclusivement de blocs élémentaires à assertions.

## 1.4. Programmes Auto-Vérifiants.

Les programmes à assertions sont susceptibles d'avoir certaines propriétés intéressantes que nous définissons ici.

Soit pa un programme à assertions sur un espace S. On définit par nm(s) le prédicat qui est vrai si et seulement si l'exécution de pa sur l'état initial s ne produit pas de messages.

On dit que le programme pa est auto-vérifiant par rapport à la relation (spécification) R si et seulement si

$$s \in dom(R) \;\&\; s \in dom([pa]) \;\&\; nm(s) \Rightarrow (s,[pa](s)) \in R.$$

Informellement, ceci s'exprime comme suit: Pour tout état initial s vérifiant la condition d'entrée ($s \in dom(R)$), si le programme pa termine normalement ($s \in dom([pa])$ sans générer de messages (nm(s)) alors l'état final ([pa](s)) vérifie avec l'état initial s la condition $(s,[pa](s)) \in R$.

Par abus de langage, la propriété d'être auto-vérifiant sera appelée la propriété d'auto-vérification. Il est intéressant de noter la présence de la clause ($s \in dom([pa])$) à gauche du signe => dans la définition de l'auto-vérification; ceci indique que la notion d'auto-vérification s'apparente plutôt avec la cohérence partielle qu'avec la cohérence totale. La parenté de l'auto-vérification avec la cohérence partielle s'explique ainsi: Une assertion dans un programme à assertions n'est utile que dans la mesure où le contrôle de l'exécution lui arrive; elle peut alors s'assurer que le bloc qu'elle contrôle a exécuté correctement mais ne peut rien faire pour s'assurer que le bloc en question termine puisque s'il ne termine pas l'assertion ne sera même pas exécutée!

De la définition d'auto-vérification découlent trois propositions simples:

- Si pa est partiellement correct par rapport à R alors il est auto-vérifiant par rapport à R: La définition d'auto-vérification présente une clause supplémentaire (nm(s)) à gauche du signe =>.
- Un programme auto-vérifiant dont toutes les assertions sont (le prédicat constant) **vrai** est partiellement correct, puisqu'alors nm(s) est vrai.
- Un programme dont les assertions sont toutes égales à **faux** est

auto-vérifiant, puisqu'alors nm(s) est faux.

## 2. Analyse de Programmes à Assertions

L'analyse des programmes à assertions se fera de façon ascendante, à l'image de l'analyse fonctionnelle des programmes, exposée dans le chapitre 3.

### 2.1. Analyse d'un Bloc Elémentaire à Assertions

**Proposition 1.** Le bloc élémentaire à assertions

```
s:=s;
b;
L: si non a(s,s) alors message
```

est auto-vérifiant par rapport à la relation $R=\{(s,s') \mid a(s,s')\}$.

Avant de procéder à la preuve de cette proposition, on fera deux remarques.

**Remarque.** Dans un bloc élémentaire à assertions tel que celui présenté ci-dessus, la propriété d'auto-vérification est indépendante de b; elle ne dépend que de la sauvegarde de l'état (s:=s;) et du test de l'assertion (**si non a alors** message;). Si bien que l'on peut changer b sans pour autant affecter la propriété d'auto-vérification du bloc; ce que l'on change, plutôt, est la fréquence d'apparition des messages.

**Remarque.** Soit à calculer l'abstraction fonctionnelle du bloc élémentaire à assertions

```
bea: s:=s;
     b;
     L: si non a(s,s) alors message;
```

si on ne s'intéresse qu'à l'évolution de l'état s, alors on peut écrire [bea]=[b] vu que l'instruction (s:=s;) ne modifie que les variables de s et que l'instruction (**si non a alors** message;) ne modifie que le fichier de messages.

**Preuve.** Soit s un élément de dom(R) tel que $s \in dom([b])$ et nm(s); puisque $s \in dom([b])$, l'exécution de b termine et l'étiquette L est atteinte par le contrôle de l'exécution, dans un état final s; puisque nm(s) est vrai, la branche **alors** n'est pas prise, donc a(s,s) est vrai; d'où $(s,s) \in R$. CQFD

Quant à savoir si la relation définie par a(s,s) est la relation la plus définie par rapport à laquelle bea est auto-vérifiant, il n'est pas possible de donner une réponse totalement affirmative (à notre surprise!). Les deux propositions suivantes donnent des réponses partielles à cette question.

**Proposition 1.A.** Parmi toutes les relations ayant un domaine plus petit ou égal à celui de R, il n'existe pas de relation (strictement) plus définie que R par rapport à laquelle bea est auto-vérifiant.

**Preuve.** Une relation R' dont le domaine est strictement plus petit que celui de R ne saurait être plus définie que R. Soit R' une relation dont le domaine est égal à celui de R et qui est (strictement) plus-définie que R: Il existe une paire (s,s') qui appartient à R et qui n'appartient pas à R'. On a que s∈dom(R); de plus, on peut choisir b tel que s∈dom([b]) et s'=[b](s), de sorte à avoir aussi s∈dom([b]) & nm(s); pourtant, par construction (s,[b](s))∈R'. Donc bea n'est pas auto-vérifiant par rapport à R'. CQFD

**Proposition 1.B.** Parmi toutes les relations, il existe des relations (strictement) plus définies que R par rapport auxquelles bea est auto-vérifiant.

**Preuve.** On prouvera cette proposition par un exemple; notez que dans cet exemple on n'explicitera pas b, car il n'a aucune importance. On prend

```
    S = entier
```

et

```
    bea: s:=s;
         b;
         si non (s>2 & s=s+3) alors message;
```

la relation définie par cette assertion est R ={(s,s')| s>2 & s'=s+3). On considère la relation R' = {(s,s')| s'=s+3}; elle est plus-définie que R. Néanmoins, bea est auto-vérifiant par rapport à R'. La preuve en est donnée ici: Soit s un élément de dom(R') tel que s∈dom([b]) et nm(s); puisque nm(s) alors (s,[b](s))∈R, donc (s,[b](s))∈R'. CQFD

La proposition 1.A. admet un corollaire intéressant que nous présentons ici.

**Corollaire.** Si dom(R)=S alors il n'existe pas de relation (strictement) plus-définie que R par à laquelle bea est auto-vérifiant.

**Preuve.** Si R' est plus-définie que R, alors dom(R)⊆dom(R'); vu que dom(R) est lui-même S, dom(R')=S. Alors R et R' ont le même domaine, si bien que si bea est auto-vérifiant par rapport à R' alors R' n'est pas (strictement) plus-définie que R. CQFD

La question de savoir si la condition dom(R)=S imposée sur toutes les relations définies par les blocs élémentaires à assertions constitue une perte de généralité est présentement à l'étude. Il semblerait que non, à la lumière de la manipulation qui est illustrée dans l'exemple ci-dessous.

Exemple

```
    S = entier;
    pa:  s:=s;
         b1;
         L1: si non a1(s,s) alors message(1);
         s:=s;
         b2;
         L2: si non (s>2 & s=s+3) alors message(2);
```

La relation attachée à l'étiquette L2 est R2 = {(s,s') | s>2 & s'=s+2}; son domaine est {s| s>2}. On propose de transformer ce programme à assertions de la façon suivante:

```
pa: s:=s;
    b1;
    L1: si non a1(s,s) & (s>2) alors message(1);
    s:=s;
    b2;
    L2: si non (s=s+3) alors message(2);
```

Alors on dom(R2)=S. Peut-on toujours faire cette manipulation? cette question est présentement à l'étude. []

## 2.2. Analyse d'une séquence

**Proposition 2.** Si p1 est auto-vérifiant par rapport à R1 et p2 est auto-vérifiant par rapport à R2 et que cod(R1)⊆dom(R2) alors p = (p1;p2) est auto-vérifiant par rapport à R = R1*R2.

**Preuve.** Au programme à assertions p1 est associé la prédicat nm1(s) indiquant l'absence de messages et au programme à assertions p2 est associé le prédicat nm2(s); il est clair que le prédicat associé au programme p = (p1; p2) est nm(s) = nm1(s) & nm2([p1](s)). Soit s un élément dans dom(R) tel que s∈dom([p]) et nm(s); on se propose de prouver que (s,[p](s))∈R.

De s∈dom(R) et R=R1*R2 on déduit s∈dom(R1); de s∈dom([p]) et p=(p1;p2) on déduit s∈dom([p1]); de nm(s) et de la formule de nm on déduit nm1(s). De s∈dom(R1) & s∈dom([p1]) & nm1(s) et de l'auto-vérification de p1 par rapport à R1 on déduit (s,[p](s))∈R1; d'où l'on déduit aussi [p1](s)∈cod(R1), et, en vertu des hypothèses, [p1](s)∈dom(R2). De s∈dom([p1;p2]) on déduit [p1](s)∈dom([p2]); de nm(s) on déduit nm2([p1](s)). De la conjonction de ces trois prémisses et de l'auto-vérification de p2 par rapport à R2 on déduit ([p1](s),[p2]([p1](s)))∈R2. Cette prémisse, en conjonction avec (s,[p1](s))∈R1, [p]=[p1]*[p2], et R=R1*R2 nous conduit à la conclusion (s,[p](s))∈R. CQFD

Il semble raisonnable de penser que si R1 et R2 sont les relations les plus-définies par rapport auxquelles p1 et p2 sont auto-vérifiantes alors R est la relations la plus-définie par rapport à laquelle p est auto-vérifiante. Cette question est présentement à l'étude.

## 2.3. Analyse d'une Alternation

**Proposition 3.** Si p1 est auto-vérifiant par rapport à R1 et p2 est auto-vérifiant par rapport à R2 alors p = (si t alors p1 sinon p2) est auto-vérifiant par rapport à R = I(t)*R1 ∪ I(~t)*R2.

**Preuve.** Il est clair que si on associe le prédicat nm1 à p1 et le prédicat nm2 à p2 alors le prédicat associé à p est nm = t&nm1 v ~t&nm2. Soit s un élément de dom(R) tel que s∈dom([p1]) et nm(s). On considère deux cas, selon la valeur logique de t(s).

Cas 1. t(s) est vrai. Alors d'après la formule de R, s∈dom(R1). D'autre part, de s∈dom([p]) et de t(s) on déduit s∈dom([p1]).

Finalement, de nm(s) et de t(s) on déduit nm1(s). De s∈dom(R1) & s∈dom([p1]) & nm1(s) et de l'auto-vérification de p1 par rapport à R1 on déduit (s,[p](s))∈R1; d'où l'on déduit, à cause de t(s), que (s,I(t)*[p1](s))∈I(t)*R1. A cause de la sémantique de l'instruction (si alors sinon) et de la formule de R, ceci s'écrit (s,[p](s))∈R.
Cas 2. ~t(s) est vrai. La preuve est identique à la précédente.
CQFD

Il semblerait que R est la relation la plus-définie par rapport à laquelle le programme p est auto-vérifiant, si R1 et R2 sont les relations par rapport auxquelles p1 et p2 sont auto-vérifiants.

## 2.4. Analyse d'une Itération

**Proposition 4.** Si b est auto-vérifiant par rapport à B alors
p = tantque t faire b
est auto-vérifiant par rapport à R = (I(t)*[b])**I(~t).

**Preuve.** Si nm' est le prédicat associé à b pour indiquer l'absence de messages alors le prédicat associé à p est
nm(s) = ~t(s) v (∀j, 0≤j≤k-1, nm'(s.(I(t)*[b])ʲ)),
où k est le nombre d'itérations requises pour traiter s.
Soit s un élément de dom(R) tel que s∈dom([p]) et nm(s). Si ~t(s) est vrai alors [p](s)=s et (s,[p](s))∈I(t)⊆R. Si t(s) est vrai, on considère le nombre d'itérations requises pour traiter s, que l'on appelle k; ce nombre existe puisque s∈dom([p]). Soit $(x_j)$ la séquence définie comme suit
$x_j = s.(I(t)*[b])^j$, pour $0 \le j \le k$.
On a alors $x_0=s$ et $x_k=[p](s)$. Soit j un indice quelconque entre 0 et k-1 inclus. Sans perte de généralité (voir chapitre 4), on pose dom(B)=S, de sorte que s∈dom(B). De plus, parceque s∈dom([p]), on a $x_j$∈dom([b]). Finalement, de nm(s) on déduit nm'($x_j$). De $x_j$∈dom(R') & $x_j$∈dom([b]) & nm'($x_j$) on déduit, en vertu de l'auto-vérification de b par rapport à B que $(x_j,x_{j+1})$∈B; puisque t($x_j$) est vraie (voir définition de $x_j$), on peut écrire $(x_j,x_{j+1})$∈I(t)*B. Par définition de la séquence $(x_j)$, on a
(s,[p](s))∈(I(t)*B)^k;
puisque ~t([p](s)) on a
(s,[p](s))∈(I(t)*B)^k**I(~t)
⊆ (I(t)*B)**I(~t)
= R.
CQFD

Il est raisonnable de penser que si B est la relation la plus définie par rapport à laquelle b est auto-vérifiant, alors R est la relation la plus-définie par rapport à laquelle p est auto-vérifiant. Cette question est présentement à l'étude.

## 2.5. Exemples d'Application

On donnera deux exemples d'application des propositions 1 à 4 données ci-dessus. Dans les programmes que nous présentons, nous marquerons d'un signe (**) toutes les instructions qui interviennet dans la

formulation de la propriété d'auto-vérification; les instructions qui ne sont pas marquées de ce signe peuvent être changées à volonté sans affecter la propriété d'auto-vérification.

**Exemple**

Cet exemple est très simple; en plus d'être illustratif, cet exemple présente l'avantage qu'il nous permet de montrer de façon concrète le lien qui existe entre la force d'une assertion locale et la force de l'assertion globale qui en résulte.

```
S = ens
      x, y: entier
fin,
p =
début
(**) y:=y;
(**) tantque y≠0 faire
      début
      x:=x+1;
      y:=y-1;
      (**) L:  si non (y=y-1) alors message;
      (**) y:=y
      fin
fin.
```

D'après la proposition 4, le programme p est auto-vérifiant par rapport à

$$R = (I(y(s)\neq 0) * \{(s,s') \mid y(s')=y(s)-1\})^* * I(y(s)=0))$$
$$= \{(s,s') \mid y(s)\neq 0 \ \&\ y(s')=y(s)-1\}^{+} * I(y(s)=0) \cup I(y(s)=0)$$
$$= \{(s,s') \mid y(s)\neq 0 \ \&\ y(s')<y(s) \ \&\ y(s')=0\} \cup \{(s,s') \mid s'=s \ \&\ y(s)=0\}$$
$$= \{(s,s') \mid y(s)>0 \ \&\ y(s')=0\} \cup \{(s,s') \mid s'=s \ \&\ y(s)=0\}.$$

Interprétant cette relation, on peut dire: A chaque fois que p est exécuté sur S, qu'il termine et qu'il n'envoie pas de messages, alors les faits suivants peuvent être affirmés sur l'exécution courante du programme:

- La valeur initiale de y était non-négative et la valeur finale est nulle,
- Si la valeur initiale de y était nulle alors l'état final est identique à l'état initial.

La première clause ne porte strictement aucune information vu que $y(s)\geq 0$ peut être conclu dès que le programme a terminé et que $y(s')=0$ peut être déduit de la sémantique de l'itération. Quant à la deuxième clause, elle se déduit simplement de la sémantique de l'itération. Donc l'assertion globale R ne porte aucune information sur le comportement de p; en d'autres termes, tester l'assertion (y=y-1) est simplement une perte de temps. A notre surprise, ceci n'est pas le cas de l'assertion (x=x+1), qui lui semble pourtant symétrique.

Si, au lieu de tester a = (y=y-1) dans le corps de la boucle on testait plutôt a' = (x=x+1) -et qu'on remplaçait (y:=y) par (x:=x)- on obtiendrait l'assertion globale suivante par rapport à laquelle p est auto-vérifiant:

$$R' = \{(s,s') \mid y(s)\neq 0 \ \&\ x(s')=x(s)+1\}^{+} * I(y(s)=0) \cup I(y(s)=0)$$
$$= \{(s,s') \mid y(s)\neq 0 \ \&\ x(s')>x(s) \ \&\ y(s')=0\}$$
$$\cup$$
$$\{(s,s') \mid s'=s \ \&\ y(s)=0\}.$$

A chaque fois que p est exécuté sur s, qu'il termine et qu'il n'envoie pas de message, on peut -d'après la relation R'- affirmer que

- La valeur finale de y est toujours nulle
- Si la valeur initiale de y est nulle alors s est inchangé
- Si la valeur initiale de y est non-nulle alors x a augmenté.

Bien qu'elle est d'apparence aussi forte que l'assertion a, l'assertion a' donne des résultats plus substantifs; quant à la question de savoir si le coût d'exécuter l'assertion a' vaut l'information donnée dans R', il semblerait que non, à la lumière de l'assertion a" proposée ci-dessous.

Si au lieu de tester a' on testait a" = (x=$\underline{x}$+1 & y=$\underline{y}$-1) et qu'on remplaçait ($\underline{x}$:=x) par ($\underline{x}$:=x; $\underline{y}$:=y), on trouverait que le programme obtenu est auto-vérifiant par rapport à

R" = {(s,s') | y(s)≥0 & a(s')=a(s)+b(s) & b(s')=0}.

La relation R" teste toute la fonctionalité du programme p; elle résulte de la synergie des assertions a et a' qui -individuellement- ne fournissaient que des propriétés triviales. Quant à la question de savoir si le coût de tester a" plutôt que a' vaut l'avantage de R" sur R', il semblerait que oui -quoique l'exemple est trop simple pour qu'il soit possible de tirer des conclusions substantielles.

**Exemple**

Le programme que nous traitons ici est un programme de division entière que nous avons emprunté à [WIRT73]. Les assertions que nous y mettons ne testent que certains aspects fonctionnels du programme; en conséquence, la relation obtenue est moins-définie que la fonction du programme.

```
S = ens
crt
   dd, dr, q: entier (*dividende, diviseur, quotient*);
   w: entier;
sse
   dr>0
fin,
p =
début
q:=0; w:=dr;
tantque w≤dd faire w:=2*w;
(**) L1: si non (q=0) alors message(1);
(**) q:=q; w:=w; dd:=dd;
tantque w≠dr faire
   début
   q:=2*q;
   w:= w div 2;
   si w≤dd alors
      début
      dd:= dd-w;
      q:=q+1
      fin;
   (**) L2: si non (q*w+dd=q*w+dd) alors message(2);
   (**) q:=q; w:=w; dd:=dd;
   fin
fin.
```

La boucle tantque (w≠dr) est auto-vérifiante par rapport à

R2 = {(s,s') | w(s)≠dr(s) & q(s')*w(s')+dd(s')=q(s)*w(s)+dd(s)}* * I(w(s)=dr(s)).

Dans cette expression, la relation dont on veut calculer la fermeture

transitive réflexive est transitive, donc sa fermeture transitive réflexive est égale à son union avec I. Donc:

$$R2 = \{(s,s') \mid w(s)\neq dr(s) \;\&\; a(s')*w(s')+dd(s')=a(s)*w(s)+dd(s) \;\&\; w(s')=dr(s')\} \cup \{(s,s') \mid s'=s \;\&\; w(s)=dr(s)\}.$$

Le bloc élémentaire à assertions qui se situe en haut du programme est auto-vérifiant par rapport à la relation

$$R1 = \{(s,s') \mid q(s')=0\}.$$

Il est simple de vérifier que $cod(R1) \subseteq dom(R2)$, puisque $dom(R2)=S$. Alors le programme p est auto-vérifiant par rapport à

$$\begin{aligned}
&R1*R2 \\
&= \{(s,s') \mid q(s')=0\} * \\
&\quad \{(s,s') \mid w(s)\neq dr(s) \;\&\; q(s')*w(s')+dd(s')=q(s)*w(s)+dd(s) \;\&\; w(s')=dr(s')\} \\
&\quad \cup \\
&\quad \{(s,s') \mid q(s')=0\} * \{(s,s') \mid s'=s \;\&\; w(s)=dr(s)\} \\
&= \{(s,s') \mid \exists s'': q(s'')=0 \;\&\; w(s'')\neq dr(s'') \;\&\; \\
&\qquad q(s')*w(s')+dd(s')=q(s'')*w(s'')+dd(s'') \;\&\; w(s')=dr(s')\} \\
&\quad \cup \\
&\quad \{(s,s') \mid \exists s'': q(s'')=0 \;\&\; s'=s'' \;\&\; w(s'')=dr(s'')\} \\
&= \{(s,s') \mid w(s')=dr(s')\} \\
&\quad \cup \\
&\quad \{(s,s') \mid w(s')=dr(s') \;\&\; q(s')=0\} \\
&= \{(s,s') \mid w(s')=dr(s')\}.
\end{aligned}$$

La relation obtenue est très peu-définie (elle porte peu d'information). Ceci est dû au fait que l'assertion a1 attachée à L1 est très faible: Elle ne teste que la valeur finale de q; elle ne se préoccupe pas des autres variables-clés w, dd et dr. Si bien que, en dépit de la force de l'assertion a2 attachée à L2, la relation résultante est trés peu-définie. Chaque assertion individuelle dans un programme à assertions est un goulot d'étranglement potentiel pour la relation globale obtenue; d'où la nécessité d'uniformiser la force des assertions que l'on inclut dans un programme. []

## 3. Vérification de Programmes à Assertions

Etant donné un programme à assertions p contenant les assertions a1, a2, ... ak, et une relation (spécification) R. Pour vérifier si le programme p est auto-vérifiant par rapport à la relation R, on calcule la relation A (la plus-définie?) par rapport à laquelle p est auto-vérifiant et on vérifie si A est plus-défini que R. Cette méthode trouve sa justification dans la proposition suivante.

**Proposition 5.** Si p est auto-vérifiant par rapport à A et que A est plus-défini que R alors p est auto-vérifiant par rapport à R.

Preuve. Par hypothèses,

$$s \in dom(A) \;\&\; s \in dom([p]) \;\&\; nm(s) \Rightarrow (s,[p](s)) \in A.$$

Soit s un élément de dom(R); puisque A est plus-défini que R, $s \in dom(A)$; donc on peut exploiter l'auto-vérification de p par rapport à A. Alors on déduit de $(s,[p](s)) \in A$ que $(s,[p](s)) \in R$. CQFD

Exemple

```
S = ens
crt
   a, b, c: entier (*multiplicateur, multiplicande, produit*);
   w: entier (*auxiliaire*);
sse
   b≥0 & a>0
fin,
p =
début
(**) a:=a; b:=b; c:=c;
(**) tantque b≠0 faire
   début
   b:=b-1;
   w:=a;
   tantque w≠0 faire
      début
      c:=c+1;
      w:=w-1
      fin;
   (**) L: si non (b=b-1 & c=c+a & a=a) alors message;
   (**) a:=a; b:=b; c:=c;
   fin
fin.
R = {(s,s') | c(s')=a(s)*b(s)}.
```

Le programme est auto-vérifiant par rapport à

A = {(s,s') | b(s)≠0 & a(s')=a(s) & b(s')=b(s)-1 & c(s')=c(s)+a(s)}*
    * I(b(s)=0).
  = {(s,s') | b(s)≠0 & a(s')=a(s) & b(s')<b(s) &
              c(s')=c(s)+(b(s')-b(s))*a(s)} * I(b(s)=0)
    U
    I(b(s)=0)
  = {(s,s') | b(s)>0 & a(s')=a(s) & b(s')=0 & c(s')=c(s)+a(s)*b(s)}
    U
    {(s,s') | b(s)=0 & a(s')=a(s) & b(s')=0 & c(s')=c(s)+a(s)*b(s)}
  = {(s,s') | b(s)≥0 & a(s')=a(s) & b(s')=0 & c(s')=c(s)+a(s)*b(s)}.

Alors p est auto-vérifiant par rapport à toute relation qui est moins définie que A. La relation R donnée ci-dessus n'est pas moins définie que A puisqu'il existe s = (3,0,2,0,) dans le domaine de R (=S) tel que s.A n'est pas un sous-ensemble de s.R. En effet,

    s.A = {s' | a(s')=3 & b(s')=0 & c(s')=17}

alors que

    s.R = {s' | c(s')=15}.

Donc on ne peut utiliser la proposition 5 pour juger de l'auto-vérification de p par rapport à R. En fait, pour ce même s = (3,0,2,0), le programme p exécute, termine et n'émet pas de messages sans pour autant que l'on puisse dire (s,[p](s))∈R; donc p n'est pas auto-vérifiant par rapport à R.

Considérons R' = {(s,s') | c(s)=0 & c(s')=a(s)*b(s)}. Alors on a dom(R')⊆dom(A). De plus, pour tout s dans dom(R'), on a

    s.A = {s' | a(s')=a(s) & b(s')=0 & c(s')=a(s)*b(s)},
    s.R'= {s' | c(s')=a(s)*b(s)}.

Donc R' est plus-défini que A. []

## 4. Conception de Programmes à Assertions

Considérons les scénario suivant: Soit à écrire un programme p correct par rapport à R. Etant donné qu'il s'est trouvé très difficile de prouver que toutes les étapes dans la conception de p sont valides, on n'a pu affirmer avec certitude que p est correct par rapport à R; toutefois, on a pu se convaincre que les n premières décompositions de R sont valides, et on peut en fournir des preuves relativement simples. Soient R1, R2, ... Rk les relations obtenues au bout des n premières décompositions; pour chacun des $R_j$ nous allons écrire un bloc élémentaire à assertions de la forme

$\underline{s}$:=s;
$b_j$;
$L_j$: **si non** ($\underline{s}$,s)$\in R_j$ **alors** message(j).

Alors le programme ainsi obtenu est -par construction- auto-vérifiant par rapport à R. Selon que les $b_j$ sont ou ne sont pas tous corrects par rapport aux $R_j$ (respectivement), le programme p sera ou ne sera pas correct. Mais, parceque toutes les étapes de conception en amont des $R_j$ sont prouvées valides, le programme p est prouvé auto-vérifiant par rapport à R.

Un scénario plus élaboré serait le suivant: On se donne une relation R et une relation R' qui est moins-définie que R. On veut écrire un programme correct par rapport à R; vu que cela se trouve être difficile, on ne peut se convaincre que le programme obtenu est effectivement correct par rapport à R. Alors on veut en même temps s'assurer que le programme est au moins auto-vérifiant par rapport à R'. Pour ce faire, on applique parallèlement les mêmes règles de conception aux relations R et R' jusqu'à ce que l'on obtienne des dérivés de R' qui sont assez simples pour figurer dans un bloc élémentaire à assertions. Soient R'1, R'2, ... R'k les relations obtenues par décomposition de R' et soient R1, R2, ... Rk leurs homologues dans la décomposition de R. Alors pour chaque j entre 1 et k on génère un bloc de code dont on pense (est sûr?) qu'il est correct par rapport à $R_j$ et on "protégera" ce bloc par la relation $R'_j$:

$\underline{s}$:=s;
$b_j$ (*correct par rapport à $R_j$*);
$L_j$: **si non** ($\underline{s}$,s)$\in R'_j$ **alors** message(j).

Parceque R' est moins-définie que R (se décompose plus facilement que R) et que R' se décompose moins que R (R' s'arrête après n étapes alors que R continue) on est plus sûr de l'auto-vérification de p par rapport à R' que de la cohérence de p par rapport à R. Une paire typique (R,R') serait, dans l'exemple d'un programme de tri:

. $R = \{(s,s') \mid perm(s,s') \ \&\ ord(s')\}$
$R' = \{(s,s') \mid ord(s')\}$

où

perm(s,s') veut dire que le tableau final est une permutation du tableau initial, et
ord(s') veut dire que le tableau final est trié.

# Chapitre 9

# Recouvrement Avant Dans les Programmes

Ce chapitre présente deux thèses: la première est que l'état d'un programme peut être contaminé (malencontreusement modifié par un événement imprévu) au cours de son exécution et être quand même considéré correct; la deuxième est qu'un programme peut, après contamination de son état, recouvrer spontanément en générant un état correct à partir d'un état contaminé. Ce dernier point de vue contraste avec la philosophie de recouvrement par retour arrière (backward recovery) qui consiste à reprendre l'exécution à partir d'un état correct précédemment sauvegardé.

## 1. Introduction: Motivation et contexte

Il est d'usage de penser que dès que l'état d'un programme est contaminé par un événement imprévu quelconque (par exemple: une condition exceptionnelle [BEST81]) alors une action de recouvrement est nécessaire; aussi, il est d'usage de penser que la meilleure façon de recouvrer dans ce cas est de faire un retour arrière à un état précédemment sauvegardé et de reprendre l'exécution à partir de cet état (voir backward recovery dans [ANDE81], ou recovery blocks dans [RAND75]). Ce chapitre tente de tempérer (modérer) ces deux points de vue en montrant (respectivement) que: A une étape donnée dans l'exécution d'un programme, un état peut être contaminé et demeurer quand même correct (dans un certain sens); d'autre part, un programme peut -sous certaines conditions- transformer un état contaminé en un état correct.

L'objectif de ce chapitre n'est pas de présenter un mécanisme de recouvrement avant (forward recovery, [ANDE81]); plutôt, on tentera de sensibiliser le lecteur aux possibilités (potentiellement grandes) de recouvrement avant. Les applications pratiques de ces possibilités seront brièvement discutées dans la section 6, toutefois.

Il peut sembler difficile d'imaginer comment un état contaminé peut être transformé en un état correct. La raison pour laquelle cela est possible est la suivante. Quand on observe l'état d'un programme à un certain moment au cours de son exécution, on peut distinguer deux types d'information:

- L'information qui reflète l'état initial sur lequel le programme est executé,
- L'information qui reflète la fonction qui a été appliquée à l'état initial (indiquant, en particulier, que l'état courant appartient au codomaine de la fonction appliquée).

Au moment de la conception du programme, l'information du deuxième type est connue, donc il est possible de concevoir des routines de recouvrement qui la regénèrent quand elle est perdue.

## 2. Niveaux de Cohérence et Recouvrement

### 2.1. Modèle de Fautes

Soit p un programme Pascal. Le programme p est prévu pour être exécuté sur une machine Pascal impliquant un compilateur, un système d'exploitation, un matériel, etc ... . Le programme p est écrit sur la base d'une certaine définition sémantique de la machine Pascal, et son abstraction fonctionnelle peut être calculée à partir d'une analyse du texte de p, laquelle analyse se base sur cette définition sémantique. Quand un événement imprévu force la machine Pascal à se comporter d'une façon incohérente avec sa définition sémantique, alors la fonction calculée par p peut être différente de [p]: On dit alors que l'on a affaire à une faute dans la machine Pascal. Une faute peut causer l'état de p à une certaine étape de son exécution d'être différent de l'état dans lequel il devrait être à cette étape: On dit alors que l'on a affaire à une contamination, (erreur) ou que l'état courant est contaminé. Le modèle de faute que nous définissons ainsi est un modèle orienté vers les fautes de la machine Pascal plutôt que vers les erreurs de programmation (bugs) qui se trouvent dans le programme p; dans la section 6 on étudiera comment les résultats de ce chapitre peuvent être utilisés dans le contexte plus large où on met en cause la cohérence du programme p (en plus de mettre en cause la cohérence de la machine Pascal).

Les résultats que nous présentons dans ce chapitre s'appliquent aux fautes anticipées autant qu'aux fautes non anticipées (classification généralement faite entre les fautes: [ANDE81], [BANA80], et [RAND75]), donc nous ne ferons pas de distinction entre ces deux types de fautes. De plus, on ne s'intéressera pas à étudier la cause des fautes (qui apparaissent dans la machine Pascal) ni comment celles-ci sont détectées et reparées; on se contentera d'étudier les actions à prendre une fois que la faute est réparée et la machine Pascal prête à repartir.

### 2.2. Les bornes

Soit p un programme sur S qui est écrit pour satisfaire la spécification R et soit L une étiquette attachée quelque part sur le texte de p. Soit s0 un état initial sur lequel p est executé; on aimerait observer un état intermédiare s à l'étiquette L. Vu que L peut être visitée plus d'une fois au cours d'une même execution, on ne peut parler de l'état du programme à l'étiquette L. On définit alors la notion de borne (comme dans borne kilométrique): Une borne est une paire de la forme (v,L) où L est une étiquette et v un entier positif;

la borne br=(v,L) représente la v-ième visite de l'étiquette L.

En relation avec la borne br = (v,L), il y a lieu de définir deux fonctions qui nous seront utiles dans nos discussions:

- La fonction passé (abrév. ps) qui est l'abstraction fonctionnelle de la partie de code executée en amont de (avant) br,

- La fonction futur (abrév. ft) qui est l'abstraction fonctionnelle de la partie de code qui reste à executer en aval de br.

**Exemple 1**

p = (p1; L: p2). Alors ps=[p1] et ft=[p2]; notez qu'il existe une seule borne à l'étiquette L, la borne (1,L).

p = (L: **tantque** t **faire** b). A la borne (v,L), où v>1, on a
ps = (I(t)*[b])$^{v-1}$, et
ft = [p];

Notez que ft ne dépends pas de v. []

La spécification R est potentiellement non-déterministe; de plus, la fonction ft est potentiellement non-injective, donc sa transposée est non-déterministe. La combinaison de ces deux propriétés fait qu'à la borne br il existe potentiellement plusieurs états contaminés que l'on peut considérer corrects et plusieurs états contaminés qui portent assez d'information correcte pour être repêchables (recouvrables).

Dans les deux sections suivantes on définira les niveaux de cohérence et les niveaux de recouvrabilité (respectivement). On prendra les conventions suivantes: L'état initial est noté s0 et l'etat final [p](s0) est noté s#.

## 2.3. Niveaux de cohérence

Il se peut qu'un état soit contaminé à une borne donnée (c'est à dire qu'il soit distinct de la valeur qu'il doit avoir à cete borne) mais qu'il puisse quand même être considéré correct.

**Définition 1.** L'état s est dit strictement cohérent (s-coherent) à la borne br ssi s=ps(s0).

Pour un état initial s0, il existe un seul état strictement cohérent; l'état cohérent est le seul état non contaminé. Les deux définitions suivantes montrent que cette définition de cohérence est inutilement étroite.

**Définition 2.** L'état s est dit largement cohérent (l-cohérent) à la borne br ssi ft(s)=s#.

En d'autres termes, s est largement cohérent si et seulement si s∈(s#).ft^; il existe d'autant plus d'états largement cohérents à la borne br que ft^ est moins déterministe, ou que ft est moins injective (même valeur pour des arguments différents). Un état largement cohérent peut ne pas être strictement cohérent, indiquant qu'une erreur s'est produite en amont; toutefois l'exécution de ft sur cet état masquera l'erreur. Il est clair qu'un état s-cohérent est l-cohérent puisque s0.ps*ft=s0.[p]=s#.

**Définition 3.** L'état s est dit sp-cohérent à la borne br ssi ft(s) ∈ s0.R.

La définition de la sp-cohérence fait intervenir la spécification de p, d'où son nom. Cette définition n'exige pas que l'application de ft à s génère s# (comme le fait la définition de cohérence large): Il suffit que ft génère un état parmi l'ensemble des états que R considère corrects pour l'état initial s0, à savoir s0.R. L'ensemble des états sp-corrects à la borne br est s0.R*ft^. Le non-déterminisme de R et de ft^ se combinent pour faire que cet ensemble soit potentiellement très grand. Il est clair qu'un état l-cohérent est sp-cohérent puisque s0.[p]∈s0.R (en vertu de la cohérence de p par rapport à R).

## 2.4. Niveaux de recouvrabilité

Il se peut qu'un état ne soit ni s- ni l- ni sp- correct mais qu'il contienne quand même assez d'information pour générer un état (au moins) sp-correct.

**Définition 4.** Un état s est dit strictement recouvrable (s-recouvrable) à la borne br ssi il existe un programme r (indépendant de s0, bien entendu) tel que [r](s) est stritement correct à la borne br.

La définition de recouvrabilité stricte présente peu d'intérêt en pratique; elle n'est introduite ici que par souci de symétrie. L'ensemble des états strictement recouvrables est s0.ps*[r]^; il est d'autant plus large que [r] est non-injective mais le fait qu'il n'existe qu'un seul état strictement cohérent à la borne br limite l'intérêt pratique de la recouvrabilité stricte. Il est clair qu'un état s-cohérent est s-recouvrable (avec r = le programme vide).

**Définition 5.** Un état s est dit largement recouvrable (l-recouvrable) à la borne br s'il existe un programme r tel que [r](s) est largement cohérent à la borne br.

Cette notion est très utile en pratique, comme le montrent les discussions des sections 3 et 4. L'ensemble des états largement recouvrables à la borne br est (s#).(ft^)*[r]^. Cet ensemble est d'autant plus grand que ft^ et [r]^ sont non-déterministes, ou que ft et r sont non-injectives. Il est clair qu'un état l-cohérent est l-recouvrable; de plus si un état est s-recouvrable alors il est l-recouvrable.

**Définition 6.** Un état s est dit sp-recouvrable à la borne br s'il existe un programme r tel que [r](s) est sp-cohérent à la borne br.

La définition de sp-recouvrabilité n'a pas été exploitée en pratique jusqu'ici, mais elle présente un potentiel remarquable. L'ensemble des états sp-recouvrables à la borne br est s0.(R*ft^*[r]^). Il est potentiellement très large puisqu'il est d'autant plus large que R est non-déterministe et que ft et [r] sont non-injectives. Il est clair que si un état est l-recouvrable ou sp-cohérent alors il est sp-recouvrable.

Le treillis expliquant les relations d'implication logique qui existent entre les divers niveaux de cohérence et recouvrabilité est donné dans l'appendice A.

## 3. Recouvrement dans les boucles

### 3.1. Paramètres de la boucle

Soit F une fonction sur un ensemble S telle que dom(F)=S (et cod(F)$\subseteq$S). On suppose que l'on a une boucle sous la forme

w = (**tantque** t **faire** b)

qui implémente la fonction F sur un sous-ensemble D de S; en d'autres termes,

$$s \in D \Rightarrow F(s)=[w](s). \qquad (0)$$

De plus, on suppose que w est fermée sur D, c'est-à-dire que

$$s \in D \;\&\; t(s) \Rightarrow [b](s) \in D.$$

Afin d'utiliser w pour implémenter F sur tout S, on doit trouver un programme Pascal i (initialisation) qui vérifie les équations suivantes

a) dom([i]) = S,
b) cod([i]) $\subseteq$ D,
c) F(s) = F([i](s)).

**Proposition 1.** Si un tel programme existe alors le programme f défini par

f = (i; **tantque** t **faire** b)

implémente F, c'est-à-dire que l'on a [f]=F.

**Preuve.** D'après les clauses (a) et (b), on a que dom([f])=S, donc dom([f])=dom(F). De plus, soit s un élément de S; on a

[f](s) = [w]([i](s)).
En vertu de (b), on sait que [i](s)$\in$D; en vertu de (0), on a
= F([i](s)).
D'après la condition (c), on a
= F(s).

Les fonctions F et [f] ont le même domaine (S) et pour tout s dans ce domaine elles prennent la même valeur; donc [f]=F. CQFD

Les équations (a), (b) et (c) sont dites les équations d'initialisation du programme f; puisqu'elles caractérisent les segments d'initialisation. Un segment d'initialisation a pour but de ramener l'état dans un domaine (D) où le comportement de w est connu (équation (0)) sans perturber le résultat final du programme (F(s)=F([i](s))).

**Exemple 2**
On prend

```
S = ens
crt
   k: entier;
   a: tableau [1..n] de réel;
sse
   1≤k≤n+1
fin,
```

où n est un entier plus grand ou égal à 1.

```
F(s) = (n+1, trié(a(s))),
```

où trié(a') désigne la permutation triée du tableau a'.
On définit D comme étant l'ensemble suivant:

```
D = ens
crt
   k: entier;
   a: tableau [1..n] de réel;
sse
   1≤k≤n+1 & a[1]≤a[2]≤ ... ≤a[k-1]
fin.
```

Il est clair que D⊆S. On définit le programme itératif suivant

w = (tantque k≤n faire inser)

où inser insère a[k] dans sa place appropriée dans le sous-tableau a[1..k] puis incrémente k. Il est clair que si w est exécuté sur un élément de D (qui est partiellement trié) alors il achève de trier a(s) et retourne n+1 dans k; donc on admettra sans (davantage de) preuve que

s∈D => F(s)=[w](s);

on admet aussi que w est fermé sur D. Il s'agit maintenant de déterminer le segment d'initialisation (i) correspondant à w et F; on prend

i = (k:=2)

et on vérifie aisément les trois équations d'initialisation.

a) i = (k:=2) peut être appliquée à n'importe quel état.
b) 0≤2≤n+1 & a[1]≤...a[2-1].
c) F(s) = (n+1, trié(a(s))) = F((2,a(s))) = F([i](s)).

Alors F peut être implémentée par le programme

f = (k:=2; tantque k≤n faire inser)

qui n'est autre que le programme de tri par insertion [WIRT76]. []

## 3.2. Recouvrement spontané dans les boucles

On considère le programme

f = (i; L: tantque t faire b)

et on aimerait caractériser les états s-cohérents, l-cohérents et l-recouvrables à la borne br = (v,L). Notez que l'étiquette L est visitée au début de chaque itération du corps de boucle, avant le test de la condition t.

**Proposition 2** (s-cohérence, boucles). L'état s à la borne br = (v,L) est s-cohérent par rapport à s0 si et seulement si

$$s = s0 \,.\, [i] * (I(t)*[b])^{v-1}.$$

**Preuve.** Cette proposition résulte immédiatement du fait que, à la borne br = (v,L), ps = $[i]*(I(t)*[b])^{v-1}$. []

**Proposition 3** (l-cohérence, boucles). L'état s à la borne br = (v,L) est l-cohérent par rapport à s0 si

$$s \in D \;\&\; F(s)=F(s0).$$

**Preuve.** On doit prouver que ft(s)=s#. Tout d'abord, notons que ft=[w]. De plus, puisque s∈D, [w](s)=F(s); par hypothèse, F(s)=F(s0), donc ft(s)=F(s0)=s#. CQFD

Cette proposition est en fait confirmée par un théorème donnée par Basu et Misra dans [BASU75] selon lequel l'assertion s∈D & F(s)=F(s0) est un invariant de boucle (donc doit être vraie à chaque visite de l'étiquette L).

**Proposition 4** (l-recouvrabilité, boucles). L'état s à la borne br=(v,L) est recouvrable par rapport à s0 si

$$F(s) = F(s0).$$

**Preuve.** Il faut prouver l'existence d'un programme r (indépendant de s0, bien entendu) tel que [r](s) est l-cohérent. Pour ce faire, on prouve d'abord le lemme suivant:

**Lemme.** S'il existe un programme r tel que
a) dom([r]) = S,
b) cod([r]) $\subseteq$ D,
c) F(s) = F([r](s)),
alors r est une routine de recouvrement valide (voulant dire que si F(s)=F(s0) alors [r](s) est l-cohérent).

Soit r un programme vérifiant (a), (b) et (c); en vertu de (a), on peut appliquer [r] à s; en vertu de (b), [r](s)∈D, donc [w]([r](s))=F([r](s)); en vertu de (c) [w]([r](s))=F(s); en vertu de l'hypothèse, [w]([r](s))=F(s0); en vertu de la définition de ft, ([r](s)).ft = F(s0); donc [r](s) est l-cohérent, ce qu'il fallait prouver.

Quant à la question de savoir si un tel programme existe, la réponse est clairement oui, vu que le segment de code i vérifie ces conditions. CQFD

Plusieurs remarques s'imposent ici concernant la proposition 4 et sa preuve.

**Remarque 1.** La proposition 4 et sa preuve montrent que F(s)=F(s0) est

une condition suffisante de 1-recouvrabilité et que toute routine vérifiant les équations d'initialisation est une routine de recouvrement. []

**Remarque 2.** Pour une fonction F donnée et un programme w donné, i et r sont sujet aux mêmes exigences fonctionnelles exprimées par les équations d'initialisation. Toutefois, ils sont sujet à des exigences performancielles distinctes vu que i est invoqué à chaque invocation de f alors que r n'est invoqué qu'en cas d'erreur. La différence d'exigences performancielles fait qu'en général il est raisonnable de concevoir r différent de i; voir exemple 3. []

**Remarque 3.** Supposons que r est choisi identique à i. Ceci ne veut pas dire que l'application de r à un état 1-recouvrable remet l'exécution du programme dans son état initial: l'état initial est [i](s0) alors que l'état recouvré est [i](s). Il est très important de saisir cette distinction; voir exemple 3. []

**Remarque 4.** Les équations d'initialisation peuvent être comprises comme une spécification formelle de la routine de recouvrement r. Soit à générer une spécification sous forme d'une paire de prédicats (prédicat d'entrée, prédicat de sortie); le prédicat d'entrée doit indiquer que l'état d'entrée est 1-recouvrable et le prédicat de sortie doit indiquer que l'état de sortie est 1-cohérent. On a alors

$p(s) = (F(s)=F(s0))$,
$q(s) = (s \in D \ \& \ F(s)=F(s0))$. []

**Remarque 5.** L'équation (F(s)=F(s0)) peut être écrite comme (F(s)=s#) et interprétée comme suit: s contient une quantité d'information suffisante pour calculer s#; n'est-il pas naturel qu'une telle condition suffise pour la 1-recouvrabilité?! []

**Exemple 3**

On continue l'étude du programme introduit dans l'exemple 2. D'après la proposition 3, un état est 1-cohérent à la borne br = (v,L) si

$$s \in D \ \& \ F(s)=F(s0)$$

<=>

$$(1 \le k(s) \le n+1 \ \& \ a(s)[1] \le a(s)[2] \le \ldots \le a(s)[k(s)-1])$$
$$\& \ (n+1, trié(a(s))) = (n+1, trié(a(s0))).$$

Le premier terme de cette conjonction peut être interprété comme: a(s) est partiellement trié entre 1 et k(s)-1 et $(1 \le k(s) \le n+1)$; on le représentera par pt(s).

Le deuxième terme se réduit à: trié(a(s))=trié(a(s0)), ce qui veut dire que a(s) est une permutation de a(s0); on le représentera par perm(s0,s).

Donc la caractérisation devient

$$pt(s) \ \& \ perm(s0,s).$$

Beaucoup de contaminations peuvent se produire sans violer cette assertion: k est décrémenté plutôt qu'incrémenté; k est remis à 2; k est incrémenté de plus que 1 à une certaine itération mais le sous-tableau sur lequel il a "sauté" est trié, etc...

D'après la proposition 4, un état est 1-recouvrable si

$$perm(s0,s).$$

Aussi longtemps que a(s) est une permutation de a(s0) le recouvrement

est possible indépendamment de la valeur de k et de l'arrangement des éléments de a(s). La routine

r = (k:=2)

est une routine de recouvrement possible. Elle remet k(s) à sa valeur initiale mais ne remet pas a(s) à sa valeur initiale a(s0). Une routine de recouvrement plus efficace que r serait, par exemple:

r'= (k:=2; **tantque** k<n **and** a[k-1]≤a[k] **faire** k:=k+1).

[]

**Exemple 4**

On présente brièvement un autre exemple de programme dont les variables sont fort redondantes; il présente alors un grand potentiel de recouvrabilité.

S = ens
crt
n, f, k: entier;
sse
n≥0 & f≥0 & k≥0
**fin**,
F(s) = (n(s),n(s)!,n(s)),
où n'! est la factorielle de n'.
w = (**tantque** k<n **faire début** k:=k+1; f:=f*k **fin**)
D = {s| f(s)=k(s)! & k(s)≤n(s)},
i = (k:=0; f:=1).

On admet sans preuve que w implémente F sur D, que w est fermé sur D et que i vérifie les équations d'initialisation construites à partir de F et D. Un état est 1-cohérent si

f(s)=k(s)! & k(s)≤n(s) & n(s)=n(s0).

Un état est 1-recouvrable si

n(s)=n(s0).

Aussi longtemps que n est intact, le recouvrement est possible: Deux variables parmi trois peuvent être totalement détruites sans menacer la survie du programme! []

Voir l'appendice B pour une description graphique des résultats de la section 3.

## 4. Recouvrement spontané dans les séquences de programmes

Après avoir étudié comment une (ou plusieurs) itération (s) d'un corps de boucle peut corriger le dommage causé par une itération précédente, on considère maintenant comment un programme peut corriger le dommage causé par un programme qui le précède dans une séquence (;) en Pascal.

### 4.1. Paramètres de la séquence

On se donne une fonction F dont le domaine est S. On suppose que F peut se décomposer comme suit: F = F1*F2, avec dom(F1)=S et dom(F2)=S.

On suppose qu'il existe des programmes p1 et p2 et un sous-ensemble D2 de S qui vérifient

$[p1] = F1$,
$s \in D2 \Rightarrow [p2](s)=F2(s)$,
$cod([p1]) \subseteq D2$.

Alors F peut être implémentée par le programme suivant:

$p = (p1;\ L:\ p2)$.

On suppose qu'il existe un programme r2 sur S vérifiant les propriétés suivantes:

$dom([r2]) = S$,
$cod([r2]) \subseteq D2$,
$F2(s) = F2([r2](s))$.

**Exemple 5**

$S =$ **réel**,
$F(s) =$ **sqrt**$(s^2+1)$,
où **sqrt**$(s')$ est la racine carrée de $s'$.
$F1(s) = s^2+1$,
$F2(s) =$ **sqrt**$(|s|)$,
où $|s'|$ est la valeur absolue de $s'$.

Alors on a

$dom(F) = S$,
$dom(F1) = S$,
$dom(F2) = S$, .
$F = F1*F2$.

On définit D2, p1 et p2 comme suit

$D2 = \{s \mid s \geq 0\}$,
$p1 = (s:=s^2+1)$,
$p2 = (s:=$**sqrt**$(s))$.

Alors on a

$[p1] = F1$,
$s \geq 0 \Rightarrow [p2](s)=F2(s)$,
$cod([p1]) = \{s \mid s \geq 1\} \subseteq D2$,

et F peut être implémentée par

$p = (s:=s^2+1;\ L:\ s:=$**sqrt**$(s))$.

Soit r2 le programme Pascal suivant:

$r2 =$ $(s:=$**abs**$(s))$.

Alors on a

$dom([r2]) = S$,
$cod([r2]) \subseteq D2$,
$F2(s) = F2(|s|)$. []

## 4.2. Recouvrement spontané dans les séquences

**Proposition 5** (s-cohérence, séquences). L'état s à la borne br = (1,L) est s-cohérant si et seulement si

$s = [p1](s0)$.

**Preuve.** Car ps = [p1]. CQFD

**Proposition 6** (1-cohérence, séquences). L'état s est 1-cohérent à

la borne br = (1,L) si
$$s \in D2 \ \& \ F2(s)=F(s0).$$

Preuve. $s \in D2 \ \& \ F2(s)=F(s0) \Rightarrow [p2](s)=F(s0) \Rightarrow ft(s)=F(s0)$. CQFD

Proposition 7 (1-recouvrabilité, séquences). L'état s est 1-recouvrable à la borne br = (1,L) si
$$F2(s) = F(s0).$$

Preuve. En fait r2 est une routine de recouvrement, puisque:
$s.[r2]*ft = s.[r2]*[p2] = s.[r2]*F2 = F2(s) = F(s0)$. CQFD

**Exemple 6**
On considère à nouveau le programme étudié dans l'exemple 5. L'état s est s-cohérent à la borne br = (1,L) si et seulement si
$$s = s0^2+1.$$
L'état s est 1-cohérent à la borne br si
$$s \geq 0 \ \& \ \mathbf{sqrt}(|s|)=\mathbf{sqrt}(s0^2+1).$$
Ceci se simplifie pour devenir
$$s = s0^2+1.$$
Il n'existe qu'un seul état 1-cohérent à la borne br: l'état s-cohérent.
L'état s est 1-recouvrable à la borne br si
$$\mathbf{sqrt}(|s|) = \mathbf{sqrt}(s0^2+1)$$
<=>
$$|s| = s0^2+1.$$
Il existe deux états 1-recouvrables à la borne br: $(s0^2+1)$ qui est 1- (et s-) cohérent, et l'état $-(s0^2+1)$ que r2 transforme en un (le) état 1-cohérent. A cette borne, on peut se permettre de perdre le signe de s et recouvrer quand même, mais on ne peut se permettre de perdre sa valeur absolue. []

Voir l'appendice C pour une description graphique des résultats de cette section.

## 5. Information critique et information non-critique

Les discussions des sections 3 et 4 ont montré que les états 1-cohérents peuvent être caractérisés par la conjonction de deux prédicats:
- Un prédicat c(s0,s) implicant s0 (à savoir F(s)=F(s0) pour les boucles et F2(s)=F(s0) pour les programmes de séquence),
- Un prédicat nc(s) n'impliquant pas s0 (à savoir $s \in D$ pour les boucles et $s \in D2$ pour les programmes de séquence).

Dans les deux cas (itératif et séquentiel), on a prouvé que c(s0,s) est une condition suffisante de recouvrement. On dit que ce prédicat porte l'information critique de l'état du programme; il semble que toute contamination qui rend ce prédicat faux constitue une perte irréparable d'information. Ce prédicat porte l'information du

"premier type" présentée dans l'introduction. L'information critique des exemples 3, 4 et 6 est (respectivement) perm(s0,s), n(s)=n(s0) et |s|=s0²+1.

Aussi, les discussions des sections 3 et 4 ont montré que le prédicat nc(s) n'est pas important pour la survie de l'exécution du programme puisqu'il ne figure pas dans la condition de recouvrement. On dit que ce prédicat porte l'information non-critique de l'état du programme: ce prédicat peut -à la suite d'une contamination- devenir faux sans menacer la survie du programme; on dit que toute violation de ce prédicat constitue une perte réparable d'information. Ce prédicat porte l'information du "deuxième type" mentionnée dans l'introduction. L'information non-critique dans les exemples 3, 4 et 6 est (respectivement) pt(s), k(s)!=f(s) & k(s)≤n(s), et s≥0.

## 6. Applications pratiques

Dans un premier temps on discutera des applications pratiques des idées présentées ci-dessus, dans le cadre du modèle de faute donné dans la section 2. Ce modèle tient compte des aléas du temps d'exécution (run-time) mais suppose que le programme est cohérent. Après chaque itération du corps de boucle (dans le cas séquentiel: après exécution de p1), le programme p se renseigne auprès de la machine Pascal afin de voir si un événement susceptible de contaminer l'état de p s'est produit au cours de la dernière itération (dans le cas séquentiel: au cours de l'exécution de p1). Si tel est le cas, le programme p se doit alors (théoriquement) d'évaluer l'étendue de la contamination: Il doit déterminer si l'état qu'il a est 1-recouvrable, 1-cohérent, etc ... . Vu que cela exige la connaissance de s0, il est impossible. Alors la philosophie la plus raisonnable consiste à:

- Prendre autant de mesures que possible pour protéger l'information critique (emmagasiner les variables dans des mémoires sûres [SCHL80], faire des tests extensifs avant de les modifier, etc ...),

- Appliquer la routine de recouvrement "à l'aveuglette" à chaque fois qu'une contamination est suspectée: Si l'état est 1-cohérent, l'activation de la routine serait inutile mais inoffensive; si l'état est 1-recouvrable sans être 1-cohérent alors l'activation de la routine serait appropriée; si l'état n'est pas 1-recouvrable alors la routine sert au moins à le transformer en un état dans D (D2), donc un état qui va converger (voir la figure B4 dans l'appendice B).

Bien que les discussions de ce chapitre soient présentées sous un modèle qui ne met en cause que la machine Pascal sur laquelle p exécute, elles peuvent être généralisées au cas où p est lui-même mis en cause; la seule différence est qu'alors on ne jugera plus le comportement de p par rapport à [p] (obtenu en analysant le texte de p) mais par rapport à la fonction que p est supposé implémenter

(d'après le concepteur de p). Ceci ouvre des horizons intéressants sur le plan pratique: On pourrait envisager une méthode hybride de validation de programmes selon laquelle on peut prouver formellement que p ne détruit pas son information critique et donner à p les moyens de recouvrer son information non-critique quand celle-ci est détruite. Cette méthode est d'autant plus intéressante qu'il existe des données expérimentales et des raisons intuitives à l'effet qu'il est généralement très facile de prouver l'invariance de l'information critique dans une boucle.

- Données expérimentales: Dans l'exemple 3, il s'agit simplement de prouver qu'aucune cellule de a n'est détruite par le programme; dans l'exemple 4 il s'agit de prouver que n n'est pas modifié par le programme.

- Raisons intuitives: F(s)=F(s0) exprime ce qui ne doit pas être détruit dans le programme; c'est une mesure conservatrice (statique) de l'évolution de l'état alors que s∈D mesure le progrès (dynamique) du programme vers la terminaison (l'évolution des indices, la coordination du tri, etc ... ).

## 7. Conclusion

Ce chapitre utilise une approche relationnelle à l'étude du recouvrement avant d'erreurs. Il tente de convaincre le lecteur de deux thèses:
- Un état peut être contaminé sans être incorrect,
- Un programme peut transformer un état (très, mais pas trop) contaminé en un état (assez, quoique pas exactement) correct.

Les deux conclusions majeures de ce chapitre s'identifient à ces deux thèses:
- Dans la littérature concernant la validation de programmes par assertions ([ANDE78], [ANDR79], [BEST81], [MILI81d] et [RAND75]) les assertions utilisées ne doivent pas tester la s-cohérence de l'état; elles doivent plutôt en tester la sp-cohérence.
- Le recouvrement avant est non seulement possible, il est aussi très simple puisqu'il n'est pas plus complexe que la génération d'un segment d'initialisation (chose très connue de tous les programmeurs). Ceci exige une certaine compréhension du programme qu'on écrit, mais n'est-on pas supposé comprendre les programmes qu'on écrit?

Parmi les autres résultats de ce chapitre, on peut mentionner
- Une classification des divers niveaux de contamination dans l'état d'un programme et une indication -pour certains d'entre eux- de l'action à entreprendre pour y remédier.
- Une formulation rigoureuse des notions intuitives d'information critique et d'information non-critique.
- Une mise en évidence d'un lien qui existe entre un segment d'initialisation et une routine de recouvrement.
- Un certain aperçu sur la nature de l'itération, en particulier la

capacité d'un programme itératif de "guérir" spontanément des contaminations infligées à son état (dans [MILI81c] ce phénomène fut appelé the healing power of loops).

- Une certaine perspective de la redondance: Il est d'usage d'assimiler la redondance avec la duplication, triplication (TMR, [ANDE81]) ou multiplication de l'état d'un programme; ce chapitre tente d'exploiter la redondance naturelle qui existe entre les variables d'un programme (un cas frappant en est montré dans l'exemple 4).

Le leçon de base qui peut être tirée de ce chapitre est qu'un programme en exécution a beaucoup de potentiel de recouvrement et que si seulement on prenait la peine de faire une analyse fonctionnelle détaillée de son programme, on pourrait tirer profit de ce potentiel.

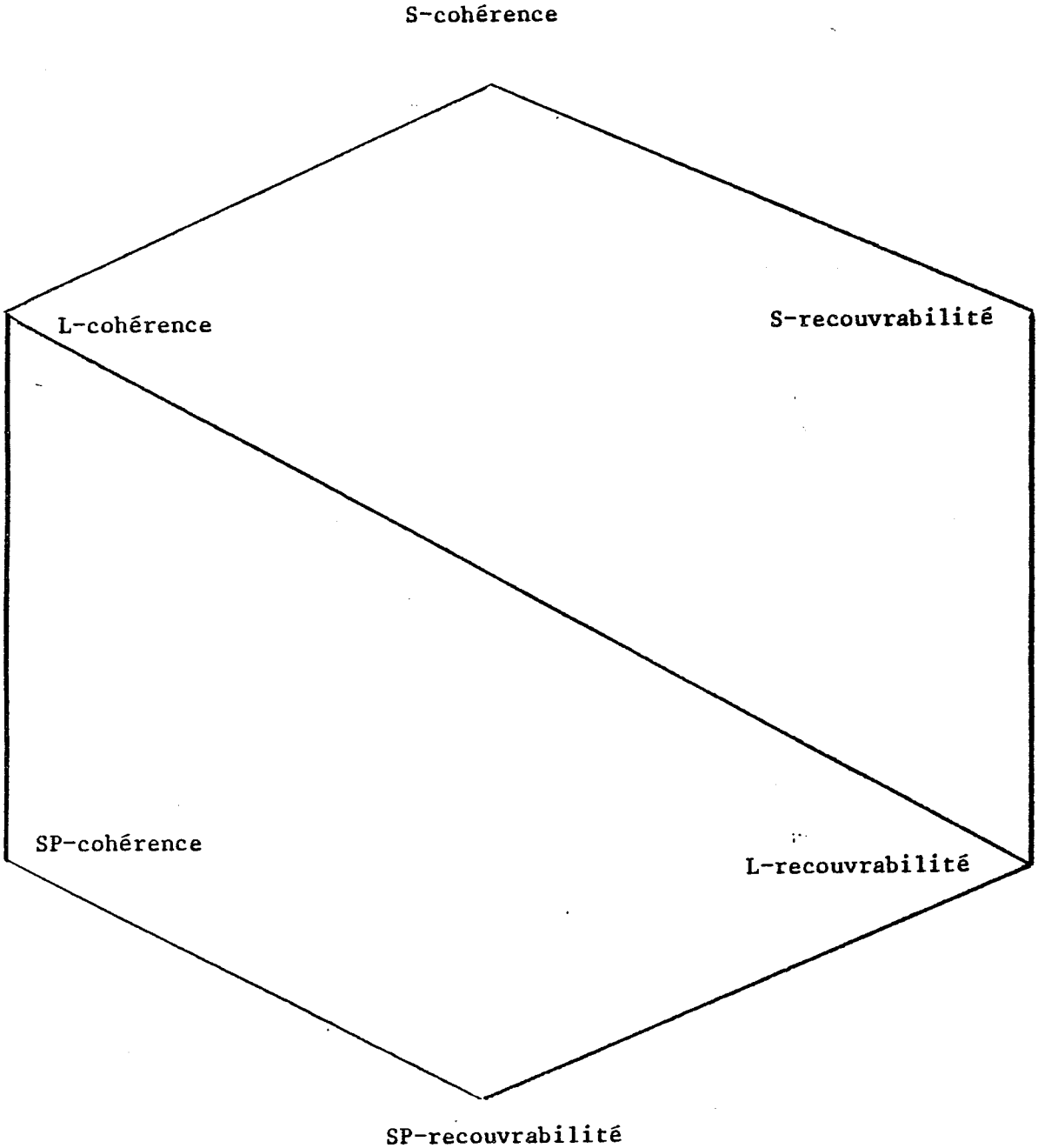
Les recherches présentées ici sont préliminaires. Plusieurs extensions en sont envisagées:

- Premièrement, exploiter les autres niveaux de cohérence et recouvrabilité (sp-cohérence, sp-recouvrabilité, etc...)
- Deuxièmement, les caractérisations que nous avons données de la 1-cohérence et de la 1-recouvrabilité sont des conditions suffisantes; il s'agit de voir si elles sont nécessaires.
- Finalement, il s'agit d'étudier en profondeur l'application pratique de ces idées afin de voir si elles peuvent déboucher sur une méthode intégrée de recouvrement avant; cela paraît difficile, mais prometteur à cause des limitations économiques du recouvrement arrière.

## Appendice A

## Le treillis des niveaux de cohérence et recouvrabilité

Les Proprietés les plus fortes sont représentées plus haut dans le treillis.



## Appendice B
## S-cohérence, L-cohérence et L-recouvrabilité: Boucles.

On définit la relation binaire EF sur S par {(s,s') | F(s)=F(s')}. Ceci est une relation d'équivalence. On note par C(s) la classe d'équivalence de s. La condition de l-recouvrabilité s'écrit alors

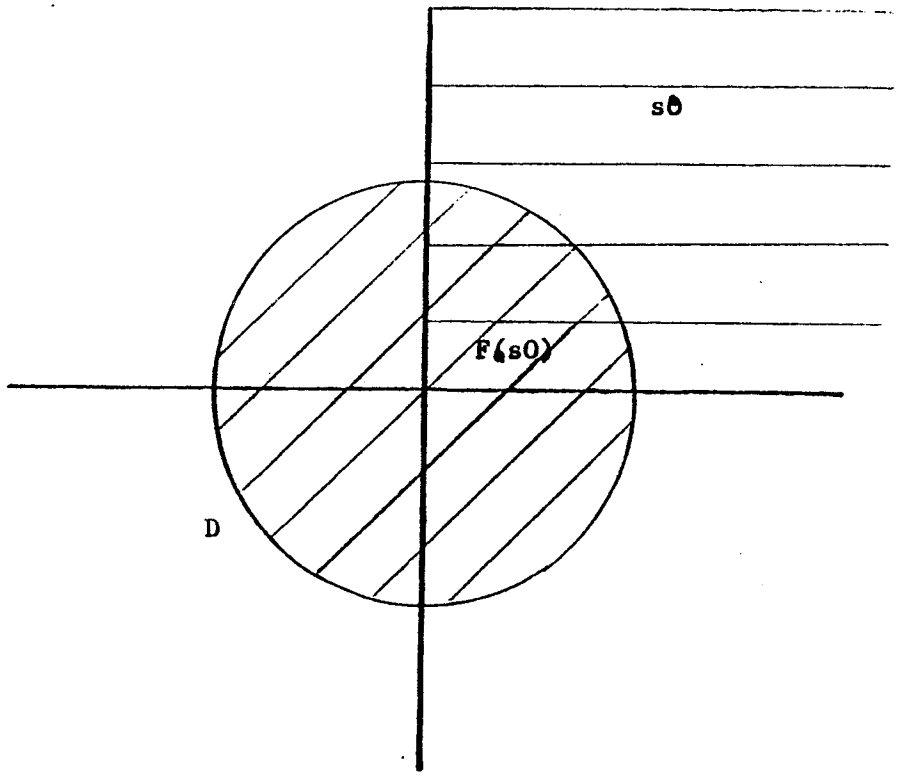
$$s \in C(s0)$$

et la condition de l-cohérence s'écrit

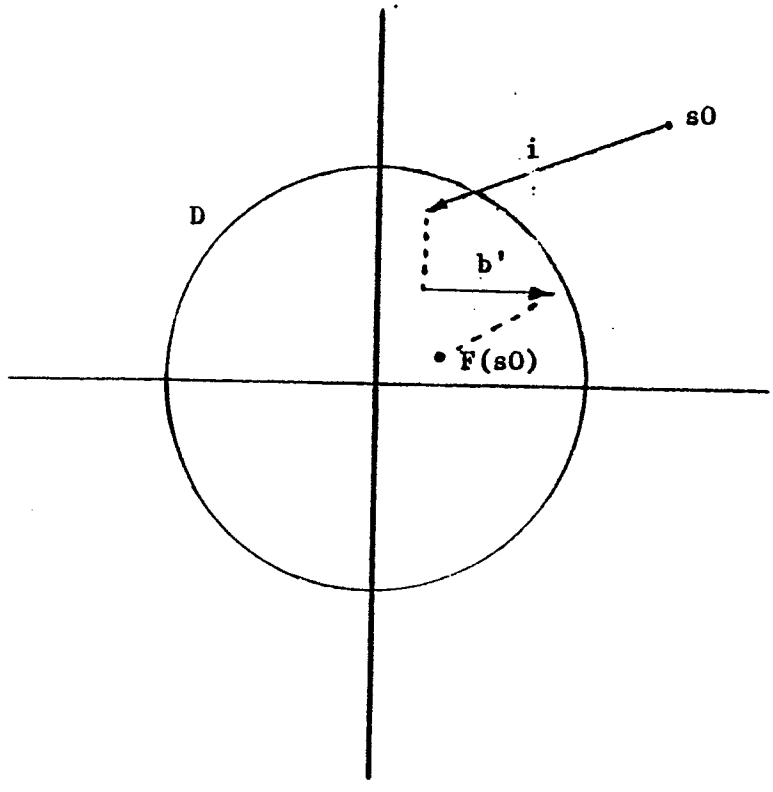
$$s \in D \ \& \ s \in C(s0).$$

Dans la figure (B1), les quadrants représentent les classes d'équivalence de S modulo EF et le cercle représente D. Pour un s0 donné, les états l-recouvrables sont représentés par le quadrant contenant s0 et les états l-cohérents sont représentés par l'intersection de ce quadrant avec le cercle D. Dans les figures (B2), (B3) et (B4), b', b" et b* représentent des versions erronées du segment b, le corps de boucle. Dans la figure (B2), l'état généré par b' n'est pas s-cohérent mais est l-cohérent: l'exécution peut continuer. Dans la figure (B3), l'état généré par b" n'est l-cohérent mais tout de même l-recouvrable: l'application de r à cet état génère un état l-correct. Dans la figure (B4) on montre deux versions erronées de b; l'une d'elles génère un état qui va converger vers une solution autre que F(s0); l'autre génère un état qui va peut être diverger.

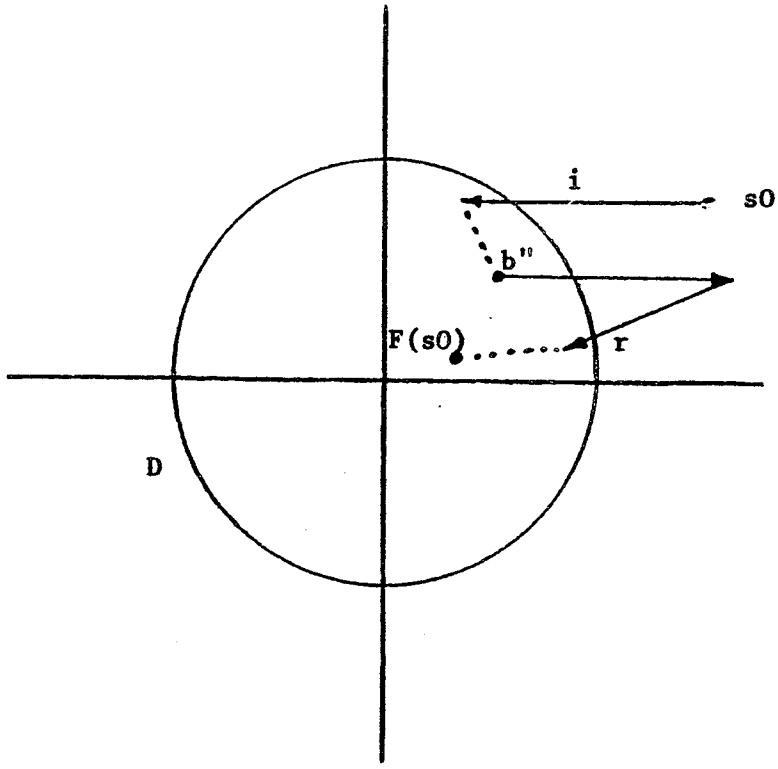
(B1)



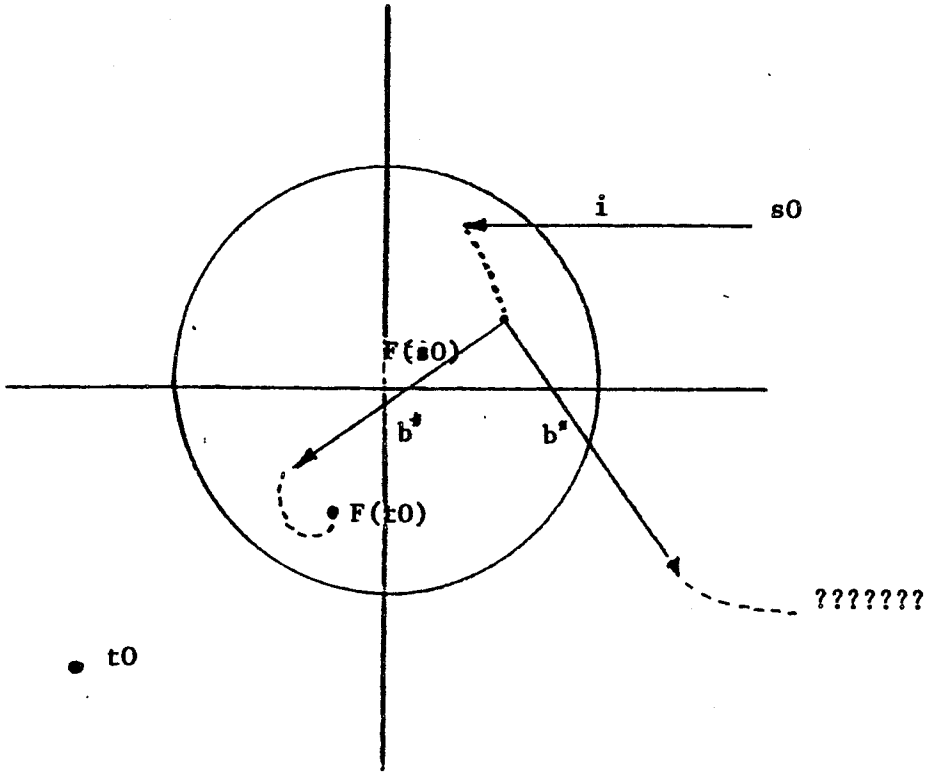

(B2)



(B3)




(B4)



## Appendice C.
## S-cohérence, L-cohérence et L-recouvrabilité: Séquences.

Le plan est partitionné par la relation

$$EF2 = \{(s,s') \mid F2(s)=F2(s')\}.$$

La condition de 1-cohérence peut être écrite

$$s \in D2 \ \& \ F2(s)=F(s0)$$

<=>

$$s \in D2 \ \& \ F2(s)=F2(F1(s0)).$$

Si on note s1=F1(s0), alors ceci devient

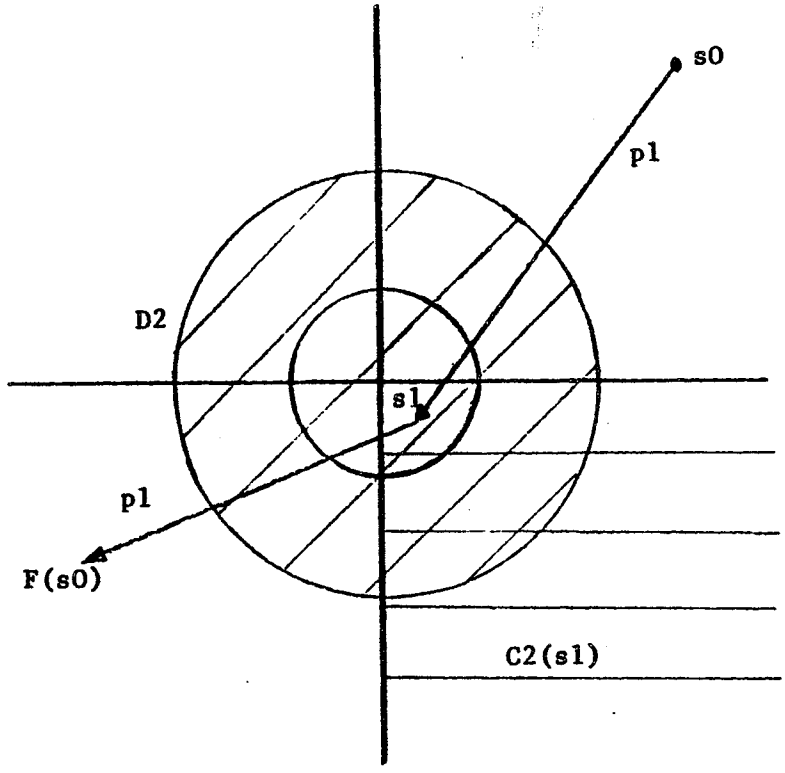
$$s \in D2 \ \& \ F2(s)=F2(s1).$$

Si on dénote par C2(s') la classe d'équivalence de s' modulo la relation EF2, la condition de 1-cohérence peut s'écrire

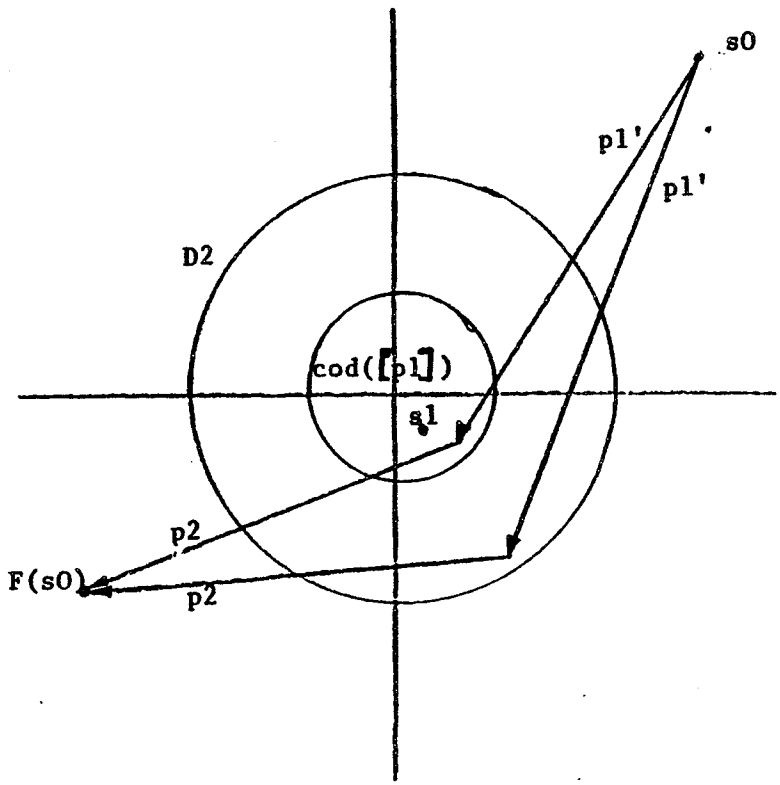
$$s \in D2 \ \& \ s \in C2(s1).$$

Dans la figure (C1), les quadrants représentent les classes d'équivalence modulo EF2, le cercle interne représente cod([p1]) et le cercle externe représente D2. Les figures (C2), (C3) et (C4) représentent des traces d'exécution du programme p dans des conditions où p1 génère (respectivement) un état 1-correct, un état 1-recouvrable et un état qui n'est pas 1-recouvrable.

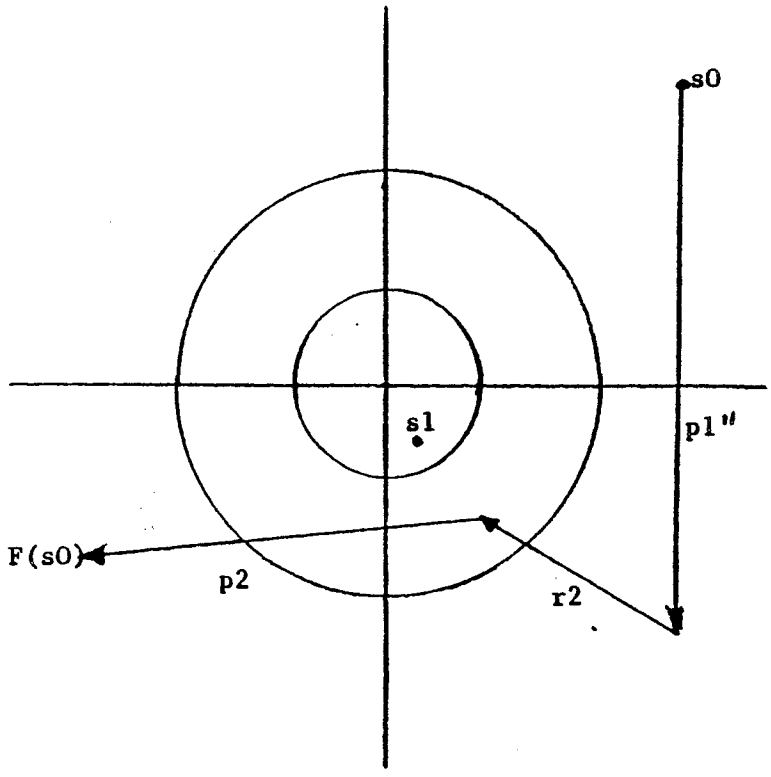
(C1)

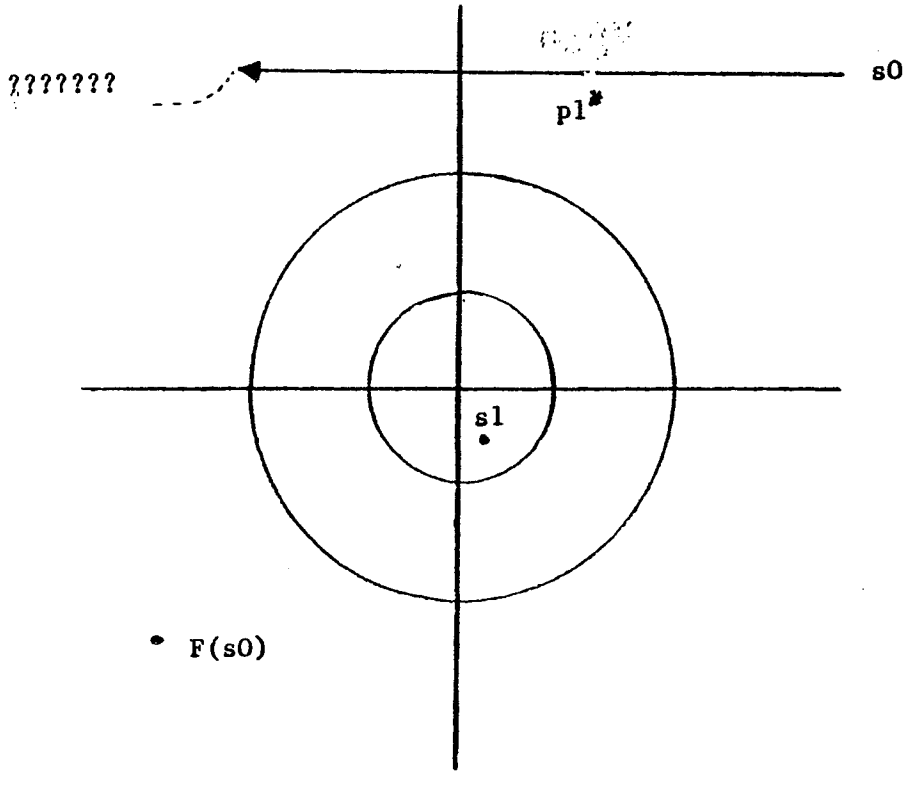



(C2)



(C3)




(C4)

# Conclusion

"Une Science a l'âge de ses instruments de mesure.
Attribué à L. Pasteur

Dans cette thèse nous avons pris une approche relationnelle à plusieurs aspects de l'activité de programmation. Cette approche a consisté à ce que l'on définisse certaines opérations algébriques sur l'ensemble des relations, ainsi que certaines propriétés des relations, ensuite que l'on utilise cet arsenal pour aborder certaines questions relatives à la programmation. Dans cette conclusion on passera brièvement en revue les principaux outils que l'on s'est donné ainsi que les principaux problèmes abordés; ensuite on évoquera quelques impressions générales et quelques perspectives de recherches futures.

Sur l'ensemble des relations binaires nous avons défini, outre les opérations ensemblistes, les opérations d'inversion, produit relatif, puissance relative, fermeture transitive, racine transitive et restriction. Parmi les propriétés de relations, nous avons introduit les propriétés usuelles d'équivalence et d'ordre. Enfin, nous avons introduit deux mesures qualitatives sur l'ensemble des relations, à savoir: une mesure qui reflète la quantité d'information portée par une relation; une mesure qui reflète le déterminisme d'une relation, soit son rapport d'arguments pour chaque image.

Cet arsenal relativement simple nous a permis d'aborder les aspects suivants de la programmation.

Nature d'une Spécification. Une spécification de programme étant une expression des exigences fonctionnelles d'entrée sortie du programme, il est fort naturel de la représenter par la relation contenant toutes les paires admissibles d'entrée-sortie.

Nature d'un Programme. Un programme n'est autre que la description d'un mécanisme par lequel un état initial (entrée) est transformé en un état final (sortie). Si l'on s'intéresse à la signification (sémantique) d'un programme plutôt qu'à sa forme, il est naturel de le représenter par la relation (déterministe, dans le cas de Pascal) contenant ses paires d'entrée/sortie. Dans le processus d'abstraire la signfication d'un programme à partir de sa forme, on vient à réaliser une identité intéressante qui existe entre les figures usuelles de la programmation et des opérations relationnelles; ainsi il y a identité entre une séquence et un produit relatif, une alternation et une union, une itératiuon et une fermeture transitive et enfin une concurrence et une intersection. Ces identités nous conduisent à percevoir un programme comme une expression relationnelle.

Processus de Spécification. Une analyse de la démarche naturelle de l'esprit lors du processus de conception nous conduit à penser que pour générer une spécification (relation) donnée, on génère des

relations qui sont moins définies qu'elles pour ensuite les combiner par l'union ou l'intersection.

Processus de Conception. Nous percevons le processus de conception de programmes comme étant une série de décisions de conception; chaque décision de conception résulte de l'application d'une règle, et revient essentiellement à résoudre une équation relationnelle. Dans le cas (fort courant) ou l'équation a plus d'une solution, nous avons trouvé qu'il est souvent avantageux de choisir la solution la moins définie possible.

Processus de Validation. Valider une spécification consiste à générer une propriété dont on pense que la spécification doit vérifier, ensuite confronter la spécification contre la propriété. Nous avons trouvé que non seulement les spécifications, mais aussi les propriétés sont naturellement représentées par des relations. Alors une spécification est valide par rapport à une propriété si elle est plus définie que cette propriété.

Processus de Vérification. Puisqu'une spécification est une relation et qu'un programme est une relation (déterministe), il est naturel de penser que la cohérence d'un programme par rapport à une spécification peut être vérifiée par une formule relationnelle. En effet, un programme est correct par rapport à une spécification s'il est plus défini que cette spécification. Il est intéressant -quoique pas surprenant- que le même lien lie une spécification valide à une propriété et un programme correct à une spécification.

Nature et Usage des Assertions. Une assertion exécutée dans un programme pour vérifier une certaine propriété entre l'état courant et un état précédent se prête fort naturellement à une représentation relationnelle. De plus, la façon dont plusieurs assertions locales se combinent pour produire une propriété globale se formule fort naturellement par composition de relations.

Nature et Usage de la Redondance. Le non-déterminisme potentiel des spécifications et la non-injectivité potentielle des programmes se combinent pour faire de sorte que l'état intermédiaire d'un programme peut fort bien être contaminé et demeurer quand même correct ou tout au moins recouvrable. Une approche relationnelle a été prise à l'étude de la redondance dans les programmes et de l'usage de cette redondance pour la tolérance aux pannes.

Bien qu'elle ne contribue pas de résultats fondamentaux nouveaux, l'approche relationnelle offre une perspective nouvelle de la programmation. En effet, elle permet d'exhiber certains mécanismes de la programmation que d'autres approches (approche prédicative, par exemple) n'exhibent pas. De plus, au même titre que l'approche fonctionnelle qui en est un prédecesseur, l'approche relationnelle fournit -par le biais de ses règles de conception- un argument en faveur de la programmation structurée; lequel argument est -nous pensons- bien plus convaincant que des polémiques concenrant l'instruction allera. Etant relativement jeune -au même titre que d'autres approches relationnelles de Parnas, Hatcher, Sanderson, Crespi-Reghizzi, ..- cette approche ne jouit pas d'une

expérimentation pratique significative. Il semblerait que le goulot d'étranglement majeur dans la mise en oeuvre de cette approche sur un projet à grande échelle soit le problème de représentation des relations.

Plusieurs extensions sont envisagées à ce travail. La première, liée au problème mentionné ci-dessus, concerne la représentation des relations. L'utilisation de la logique de prédicats pour la représentation de relations semble avoir des limitations sévères en pratique; nous envisageons d'étudier des représentations algébriques inspirées des travaux de R.C Lyndon. Parmi les extensions théoriques, on mentionnera des études plus approfondies du calcul relationnel afin de voir comment celui-ci peut nous permettre de résoudre les équations générées dans le chapitre 6. Ceci promets d'être un problème difficile au vu de tous les paramètres qui interviennet dans une décision de conception, mais toute perspective que l'on peut gagner sur la résolution de ces équations promets d'etre précieuse.

# Bibliographie

Les référence bibliographiques sont classées par ordre alphabétique puis par ordre chronologique (pour un même nom). On utilisera les abbréviation suivantes:

ACM TOPLAS: ACM Transactions on Programming Languages and Systems
AMS: American Mathematical Society
CACM: Communications of the ACM
HICSS: Hawaii International Conference on System Sciences
ICSE: International Conference on Software Emgineering
IEEE-TSE: IEEE Transactions on Software Engineering

[ANDE78]: Anderson T. and R.W. Witty. Safe Programming. BIT vol 18 (1978), pp 1-8.

[ANDE79]: Anderson R. Proving Programs Correct. John Wiley and Sons, 1979.

[ANDE81]: Anderson T. and P.A. Lee. Fault Tolerance: Principles and Practice. Englewood Cliffs (NJ): Prentice-Hall, 1981.

[ANDR79]: Andrews D.M. Using Executable Assertions for Testing and Fault Tolerance. Proceedings, Ninth International Symposium on Fault Tolerant Computing. Madison (WI), June 20-22 1979, pp 102-105.

[APT81]: Apt K. Ten Years of Hoare's Logic: A Survey -Part I. ACM TOPLAS, 3(4), (October 1981), pp 431-483.

[BALZ79]: Balzer R. and N. Goldman. Principles of Good Software Specification and their Implications for Specification Language. IEEE Symposium on Software Specification, 1979.

[BANA80]: Banatre J.P, B. Gamatié et F. Ployette. Outils de Structure de Logiciels Fiables. Proceedings, Second International Conference on Reliability and Maintainability. Perros-Guirec, September 1980.

[BASU75]: Basu S. and J. Misra. Proving Loop Programs. IEEE-TSE, SE-1(1), (1975), pp 76-86.

[BEST81]: Best E. and F. Cristian. Systematic Detection of Exception Occurrences. Science of Computer Programming, Vol 1 (1981), pp 115-141.

[BLIK83]: Blikle A. and A. Tarlecki. Naive Denotational Semantics. Invited Paper, 9th World Computer Congress (1983), Paris (France), September 19-23, pp 345-355.

[BRIN77]: Brinch Hansen P. The Architecture of Concurrent Programs. Englewood Cliffs (NJ): Prentice-Hall, 1977.

[BURS69]: Burstall R. Proving Properties of Programs by Structural Induction. Computer Journal, 12(1), (February 1969), pp 41-48.

[BURS77]: Burstall R. and J. Guoguen. Putting Theories Together to Make Specifications. Proceedings, Fifth International Joint Conference on Artificial Intelligence. Cambridge (MA), 1977, pp 1045-1058.

[CAPL75]: Caplain M. Finding Invariant Assertions for Proving Programs. Proceedings, International Conference on Reliable Software. Los Angeles, CA (April 1975), pp 165-171.

[CAPL78]: Caplain M. Langage de Spécifications. Thèse de Doctorat ès-Sciences, Institut National Polytechnique de Grenoble, 1978.

[CLAR76]: Clarke E. Programming Language Constructs for which it is Impossible to Obtain Good Hoare-like Axioms. Technical Report No 76-287, Computer Science Department, Cornell University, 1976.

[CLAR82]: Clarke L, J. Hassel and D. Richardson. A Close Look at Domain Testing. IEEE-TSE, SE-8(4), 1982, pp 380-390.

[COOK78]: Cook S. Soundness and Completeness of an Axiom System for Program Verification. SIAM Journal on Computing 7(1), (February 1978), pp 70-90.

[CERI83]: Ceri S. and S. Crespi Reghizzi. Relational Data Bases in the Design of Program Construction Systems. Rapporto interno 83-8. Laboratori di Calculatori. Politecnico di Milano. 1983.

[DAVI82]: Davis A. The Role of Requirements in the Automated Program Synthesis for Real-time Systems. Requirements Engineering Environments. Y Ohno (editor). North-Holland, 1982.

[DATE81]: Date C.J. An Introduction to Data Base. Addison Wesley, 1981.

[DEBA73]: DeBakker J. and Th. Meertens. On The Completeness of the Inductive Assertion Method. Mathematical Center, Amsterdam, December 1973.

[DEBA80]: DeBakker J. Mathematical Theory of Program Correctness. Englewood Cliffs (NJ): Prentice-Hall, 1980.

[DESH84]: Desharnais J. and A. Mili. Relations as the Basis for Program Specification, Analysis and Design. Proceedings, International Workshop on Models and Languages for Software Specification and Design. Orlando (FL) March 30th, 1984.

[DIJK75]: Dijkstra E. Guarded Commands, Nondeterminacy and the Formal Derivation of Programs. CACM 18(8), August 1975, pp 453-457.

[DIJK76]: Dijkstra E. A Discipline of Programming. Englewood Cliffs (NJ): Prentice-Hall, 1976.

[DUPR84]: Dupras M, F. Lemay and A. Mili. Some thoughts on Teaching First Year Programming. SIGCSE Bulletin, February 1984.

[FLOY67]: Floyd R. Assigning Meaning to Programs. Proceedings, AMS Symposium on Applied Mathematics. AMS, Providence (RI) 1967, pp 19-31.

[GERH82]: Gerhart S. Formal Validation of a Simple Data Base Application. Wang Institute Technical Report TR-82-02, December 1982.

[GOGU80]: Guoguen J. and R. Burstall. An Ordinary Design. Private Communication from SRI International to Honeywell Corporate Technology Center, 1980.

[GORD77]: Gordon M. The Denotational Description of Programming Languages: An Introduction. Springer-Verlag, 1977.

[GORE75]: Gorelick G. A Complete Axiomatic System for Proving Assertions about Recursive and Non-recursive Programs. Technical Report No 75, Computer Science Department, University of Toronto, January 1975.

[GRIE81]: Gries D. The Science of Programming. Springer-verlag, 1981.

[GROG78]: Grogono P. Programming in Pascal. Addison-Wesley, 1978.

[HATC74]: Hatcher W. A Semantic Basis for Program Verification. Journal of Cybernetics, 4(1), pp 61-69.

[HEHN83]: Hehner, E.C.R. Predicative Programming, Parts 1 and 2. CACM, Février 1984.

[HOAR69]: Hoare A. An Axiomatic Basis for Computer Programming. CACM 12(10) October 1969, pp 576-583.

[HOAR75]: Hoare A. Parallel Programming: An Axiomatic Approach. Computer Languages 1(2) (1975), pp 151-160.

[KAND80]: Kandzia P. and M. Manglemann. The Use of Transitively Irreducible Kernels of Full Families of Functional References in Logical Data Base Design. Lecture Notes in Computer Science, vol 100. Springer-Verlag, 1980.

[KROG80]: Kroger F. Infinite Proof Rules for Loops. Acta Informatica 14(4), 1980, pp 371-390.

[LAMP80]: Lamport L. The Hoare Logic of Concurrent Programs. Acta Informatica, 14(1), 1980 pp 21-37.

[LING79]: Linger R.C, H.D. Mills and B.I. Witt. Structured Programming: Theory and Practice. Addison-Wesley, 1979.

[LIU77]: Liu C.L. Elements of Discrete Mathematics. McGraw-Hill, 1977.

[LIVE78]: Livercy. Théorie des Programmes. Dunod -collection informatique. Paris, 1978.

[LYND50]: Lyndon R.C. The Representation of Relational Algebras. Annals of Mathematics, 51(3), may 1950, pp 707-729.

[LYND56]: Lyndon R.C. The Representation of Relational Algebras, II. Annals of Mathematics, 63(2), March 1956, pp 294-307.

[MANN74]: Manna Z. Mathematical Theory of Computation. McGraw-Hill, 1974.

[MANN78a]: Manna Z. and R. Waldinger. The Logic of Computer Programming. IEEE-TSE, SE-6(3), May 1978, pp 199-229.

[MANN78b]: Manna Z. and R. Waldinger. Is 'sometime' sometimes better than 'always'? Intermittent Assertions in Proving Program Correctness. CACM 21(2), February 1978, pp 159-172.

[METZ81]: Metze G. and A. Mili. Self-Checking Programs: An Axiomatization of Program Validation By Executable Assertions. Proceedings, Eleventh International Symposium on Fault Tolerant Computing. Portland (ME), June 24-26 1981.

[MILI81c]: Mili A. Error Recovery Without Backtracking: The Healing Power of Loops. Honeywell Corporate Technology Center, Minneapolis (MN). HR-81-260: 17-38 (March 1981).

[MILI81d]: Mili A. Self-Checking Programs: An Axiomatic Approach to the Validation of Programs by the Use of Assertions. PhD Dissertation, Department of Computer Science, University of Illinois (May 1981).

[MILI82c]: Mili A. Self-Stabilizing Programs: The Fault Tolerant Capability of Self-Checking Programs. Correspondence, IEEE Transactions on Computers, July 1982.

[MILI82k]: Mili A. System Requirements Specification: A Simplified Approach. Proceedings, International Conference on Requirements Engineering Environments. Kyoto, Japan (September 1982).

[MILI82t]: Mili A. A Closer Loop At Iteration: The Self-Stabilizing Capability of Loops. Proceedings, ICSE-6. Tokyo (Japan), September 1982.

[MILI83s]: Mili A. A Case For Teaching Program Verification. SIGCSE Bulletin, February 1983.

[MILI83j]: Mili A. and D. Reese. Representation and Manipulation of Information Systems: A Simplified Approach. The Journal of Systems and Software, 3(1), July 1983.

[MILI83p]: Mili A. The Bottom Up Analysis of While Statements: Strongest Invariant Functions. Proceedings, 9th World Computer

Congress. Paris (France), September 1983, pp 339-343.

[MILI83q]: Mili A. On The Use of Assertions in Structured Programs. Département d'Informatique, Université Laval. Quebec (Canada), Septembre 1983.

[MILI83i]: Mili A. Verifying Programs by Induction on Their Data Structure: General Format and Applications. Information Processing Letters, vol 17, October 1983, pp 155-160.

[MILI83a]: Mili A. A Relational Approach to the Design of Deterministic Programs. Acta Informatica, December 1983.

[MILI84h]: Mili A. On The Specification and Design of Programs: Set-theoretic Concepts. Proceedings, HICSS-17. January 1984, Honolulu (HI).

[MILI84s]: Mili A. and J. Desharnais. Toward the Automatic Symbolic Execution of While Statements. Proceedings, HICSS-17, January 1984, Honolulu (HI).

[MILI84o]: Mili A. and J. Desharnais. Assigning Meanings to Program Verification Methods. Proceedings, ICSE-7, Orlando (FL), March 1984.

[MILI85v]: Mili A. An Introduction to Formal Program Verification. New York: Van Nostrand-Reinhold, 1985.

[MILI85a]: Mili A., J. Desharnais and J.R. Gagné. Strongest Invariant Functions: Their Use in the Analysis of While Statements. Acta Informatica, Avril 1985.

[MILI85t]: Mili A. Toward a Theory of Forward Error Recovery. IEEE Transactions on Software Engineering, à paraître Août 1985.

[MILL72]: Mills H.D. Mathematical Foundations for Structured Programming. IBM Federal Systems Division, Gaithersburg (MD), February 1972. Report FSC-72-6012.

[MILL75]: Mills H.D. The New Math of Computer Programming. CACM 18(1), 1975, pp 43-48.

[MILL82]: Miller T. and B. Taylor. A Requirement Methodology for Complex Real-time Systems. Proceedings, International Conference on Requirements Engineering Environments. Kyoto, September 1982.

[MILL83]: Mills H.D. et al. The Calculus of Computer Programming. à paraître, par Allyn & Bacon (Boston, MA).

[MILN76]: Milne R. and C. Strachey. A Theory of Programming Language Semantics. John Wiley and Sons, 1976.

[MORR77]: Morris J.H. and B. Wegbreit. Program Verification by Subgoal Induction. Chapter 8 of Current Trends in Programming Methodology. R.T. Yeh, editor. Prentice-Hall, 1977.

[OWIC76]: Owicki S. and D. Gries. Verifying Properties of Parallel Programs: An Axiomatic Approach. CACM, 19(5), 1976, pp 279-285.

[PARN72]: Parnas D. A Tachnique for Software Module Specification with Examples. CACM, 15(5), 1972, pp 330-336.

[PARN83]: Parnas D. A Generalized Control structure and Its Formal Definition. CACM 16(8), August 1983, pp 572-581.

[PRAT76]: Pratt V. Semantical Considerations in Floyd-Hoare Logic. Proceedings, Symposium on The Mathematical Foundations of Computer Science. Houston (TX), 1976, pp 109-121.

[RAND75]: System Structure for Software Fault Tolerance. IEEE-TSE, SE-1(2), June 1975, pp 220-232.

[ROBI77]: Robinson L. and O. Roubine. SPECIAL: A SPECIfication and Assertion Language. SRI International, report No CSL 46, AD/A 038-225, January 1977.

[SAND80]: Sanderson G. A Relational Theory of Computing. Lecture Notes in Computer Science. Vol 100. Springer-Verlag.

[SCHL80]: Schlichting R. and F. Schneider. Verification of Fault Tolerant Software. Technical Report TR 80-446, Computer Science Department, Cornell University, November 1980.

[SCOT71]: Scott D. and C. Strachey. Toward a Mathematical Semantics of Computer languages. Technical Monograph PRG-6, Programming Research Group, University of Oxford, 1971.

[STOY77]: Stoy J. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press, 1977.

[SUNS82]: Sunshine C, D. Thompson, R. Erickson, S. Gerhart and D. Schwabe. Specification and verification of Communication Protocols in AFFIRM Using State Transition Models. IEEE-TSE, SE-8(5), Septembre 1982, pp 460-489.

[SUPP72]: Suppes P. Axiomatic Set Theory. New York: Dover Publications, 1972.

[SURF81]: Surfin B. Formal System Specification: Notations and Examples. Lecture Notes in Computer Science, Springer-verlag, 1981.

[TARS41]: Tarski A. On The Calculus of Relations. The Journal of Symbolic Logic 6(3), September 1941, pp 73-89.

[WANG76]: Wang A. An Axiomatic Basis for Proving Total Correctness of Goto Programs. BIT Vol 16, 1976, pp 88-102.

[WAUG80]: Waugh D. Ada as a Design Language. Software Engineering Exchange, Special Ada version, 3(1), 1980, pp 8-12.

[WIRT73]: Wirth N. Systematic Programming: An Introduction.

Prentice-Hall, 1973.

[WIRT76]: Wirth N. Algorithms + Data structures = Programs. Prentice-Hall, 1976.

# AUTORISATION de SOUTENANCE

VU les dispositions de l'article 5 de l'arrêté du 16 avril 1974

VU les rapports de présentation de

. Madame G. SAUCIER, Professeur
. Monsieur ABRIAL
. Monsieur W. HATCHER, Professeur

**Monsieur MILI Ali**

est autorisé à présenter une thèse en soutenance en vue de l'obtention du grade de DOCTEUR D'ETAT ES SCIENCES.

Fait à Grenoble, le 2 octobre 1985

Le Président de l'U.S.M.G

Le Président

D. TANCHE

Le Président de l'I.N.P.-G

D. BLOCH
Président
de l'Institut National Polytechnique
de Grenoble

Résumé

Cette thèse présente une perspective relationnelle à plusieurs aspects de la programmation. L'algèbre relationnelle de Tarski est utilisée pour formuler -et parfois résoudre- des problèmes pertinents à la programmation tels que : la spécification de programmes, l'analyse fonctionnelle de programmes, la vérification de programmes, la conception de programmes et le traitement d'erreurs dans les programmes.

La perspective que nous adoptons dans cette thèse est caractérisée par les prémisses suivantes : les programmes sont spécifiés à l'aide de relations ; l'analyse fonctionnelle de programmes se fait par composition de fonctions ; la conception de programmes se fait par décomposition de relations.

Mots clés

Calcul relationnel, Analyse fonctionnelle de programme, Spécification de programmes.