d'ordre

THESE

présentée à

L'INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE

pour obtenir

LE DIPLOME DE DOCTEUR DE 3ème CYCLE

par

OUERGHI Mohamed Saïd

Spécialité : **Mathématiques**
Mention : **Informatique**

Sujet de la thèse :

SEMANTIQUE ALGEBRIQUE D'UN LANGAGE DE PROGRAMMATION SUPPORTANT LE CONCEPT DE PROCESSUS COMMUNICANTS

mission composée de :

Président

Examinateurs

J.P. FINANCE
J.P. THOMESSE
M. WIRSING

ERRATUM
=======

PAGE	LIGNE	TEXTE A CORRIGER	TEXTE CORRECTE

22	8	Fin;	Fin;
			
			On sous entend par ceci la spécification dont les équations sont les suivantes :
			∀ x∈ Nat \ {indef}
			pred(succ(x))= x;
			pred(zéro)= indef;
			pred(indef)= indef;
			succ(indef)= indef;

39	8 9	xor(true,true)= false; xor(false,false)= false;	xor(b,b)=false;
40	-1	Fin;	le(zero) ^ lt(x,y) ==> div(x,y)= zéro; Fin;
48	-3	Fin;	eval(varexp(x)), init)= valvar(x); eval(varexp(x), substval(σ,valvar(y),v))= si egvar(x,y) alors v sinon eval(varexp(x), σ) fsi; eval(varexp(desvar(x,p)), substid(σ, idvarident(y), idvarident(z)))= si egid(x,y) alors valvar(desvar(z,p)) sinon eval(varexp(desvar(x,p)),σ) fsi; Fin; Remarque: varexp dans ces équations est un abus d'écriture de la compositon (tout en respectant les profils) des opérations suivantes : variexp, varbexp, iexpexp et bexpexp.
58	14	egmen	egmem
59	18 21,24	Medelem egmen	Modelem egmem
60	11	M2	m2
82	-13 -11 -5	T2 (T1, T2) precedegal(t1, t2)=	t2 (t1, t2) ... precede(t1,t2)= ...

N° d'ordre

THESE

présentée à

L'INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE

pour obtenir

LE DIPLOME DE DOCTEUR DE 3ème CYCLE

par

OUERGHI Mohamed Saïd

Spécialité : **Mathématiques**
Mention : **Informatique**

Sujet de la thèse :

SEMANTIQUE ALGEBRIQUE D'UN LANGAGE DE PROGRAMMATION SUPPORTANT LE CONCEPT DE PROCESSUS COMMUNICANTS

Soutenue le 03 Mars 1984 devant la Commission composée de :

MM.	J.C. DERNIAME	Président
	J.P. BANATRE J.P. FINANCE J.P. THOMESSE M. WIRSING	Examinateurs

Ce n'est jamais facile de remercier les différentes personnes qui m'ont apporté leur soutien, directement ou indirectement, pour mener à terme ce travail de recherche.

C'est à monsieur J.P. FINANCE que je dois une mention toute spéciale. Dirigeant mes recherches, il a su me stimuler sans cesse. Je lui suis redevable pour les conseils attentifs et les suggestions nombreuses et pertinentes qu'il a su me prodiguer.

J'ai pu apprécier durant cette expérience, à travers des exposés des articles mais aussi en discutant, le travail de messieurs M. WIRSING et M. BROY que je tiens à remercier. C'est le premier qui a su m'apporter d'autres éclairages sur les types abstraits. Je lui suis gré de participer dans ce jury.

C'est pour moi un grand honneur de voir aujourd'hui monsieur J.C. DERNIAME présider ce jury. Je le remercie vivement pour l'attention qu'il porte à mon travail et pour toutes les occasions de discussions critiques et sympatiques.

Dans le cadre des réunions de l'ATP parallélisme j'ai pu profiter de l'expérience de monsieur J.P. THOMESSE. Je tiens à le remercier de l'intérêt qu'il porte à mon travail et d'apporter ici son jugement.

Monsieur J.P. BANATRE m'a fait bénéficier de ses remarques et critiques. Il me fait le plaisir de siéger dans ce jury. Je le remercie.

Je remercie également toutes les personnes qui m'ont enseigné, encouragé et aidé mais que je ne puis citer car je ne voudrais oublier personne.

Je dois enfin beaucoup et plus encore aux personnes qui me sont le plus proche et qui m'ont supporté pendant cette période plein de doutes et d'espoirs. Qu'elles trouvent ici le témoignage de ma plus sincère gratitude.

RESUME

RESUME :

Le thème principal abordé dans cette thèse est la mise au point d'une méthode de description algébrique de la sémantique des langages de programmation supportant les concepts de processus communicants et de nondéterminisme.

L'idée de base de la définition algébrique d'une sémantique est d'associer aux différents composants d'un langage de programmation une hiérarchie de types abstraits rendant compte des expressions, des états de mémoire et des calculs.
L'extension de ce cadre algébrique pour la prise en compte des processus communicants se fait en "temporisant" les différentes opérations du type et en décrivant un mécanisme de composition d'opérations temporisées. Ces mécanismes sont décrits sur un langage jouet baptisé LAGALERE.

Le second thème abordé ici, est la complémentarité des sémantiques des langages de programmation supportant les processus communicants, en particulier les sémantiques algébriques et axiomatiques.

L'objectif est de valider les règles de preuves définies par la sémantique axiomatique à l'aide de la sémantique algébrique. On fournit ainsi un outil utilisable par le programmeur.

MOTS CLES :

Nondéterminisme, parallélisme, processus communicants, spécifications algébriques, types abstraits, sémantique des langages de programmation, systèmes de preuve.

SOMMAIRE

SEMANTIQUE D'UN LANGAGE DE PROGRAMMATION SUPPORTANT LES CONCEPTS DE PROCESSUS COMMUNICANTS.

PARALLELISME, NON DETERMINISME ET SEMANTIQUE

PARALLESLIME, NON DETERMINISME ET SEMANTIQUE

1. NOTIONS DE PARALLELISME ET DE NON DETERMINISME

Le parallélisme recouvre la notion d'activité simultanée de plusieurs programmes ou "processus". Il peut être soit virtuel (dans une machine mono-processeur), c'est-à-dire résulter d'une abstraction des détails du fonctionnement réel, soit véritable (dans les réseaux ou les machines multi-processeurs), c'est-à-dire correspondre à une réalité physique.
Malgré les problèmes complexes qu'il pose, tels que ceux de concurrence et de synchronisation, le parallélisme est de plus en plus présent en informatique. Ceci résulte en fait de plusieurs raisons :

i) les progrès technologiques du matériel informatique autorisent l'emploi croissant de machines multi-processeurs,

ii) les progrès méthodologiques du développement des logiciels facilitent la conception modulaire de systèmes complexes par des découpages en processus,

iii) les caractéristiques mêmes de certains problèmes (par exemple la description d'un atelier flexible) font surgir le parallélisme de situation de façon naturelle

iv) le désir d'obtenir des solutions performantes (notamment en temps de réponse) pour des problèmes séquentiels, mène aisément au parallélisme de résolution.

Le parallélisme fait l'objet de très nombreuses recherches, afin de mieux cerner et comprendre les problèmes qui lui sont reliés (partage de données, coordination de calculs, etc ...), et un grand nombre de modèles de calcul parallèle et d'outils de mise en oeuvre ont été proposés. En fait, la mise en évidence de ce qu'est le parallélisme

réside principalement en les deux situations suivantes [BAN 80] :

- algorithmes indépendants "la plupart du temps".
  De temps en temps, il doivent se coordonner,
- partage de ressources.

Et les différents types de solutions se ramènent alors, soit au niveau des structures de contrôle, soit au niveau de la transmission d'informations.

Ainsi, les premières méthodes utilisées dans les systèmes par variables partagées ont proposé des primitives de synchronisation pour accéder à ces variables sans risque de conflit :
sémaphores [DIJ 68] (Algol 68), events (PL1), moniteur et files d'attente [HOA 72] (concurrent-Pascal [BRI 76]), régions critiques [CAM 74], ...

Puis d'autres méthodes, plus adaptées à un environnement de multi-processeurs à mémoire propre, ont proposé des primitives de communication pour échanger des messages :
DP [BRI 78], CSP [HOA 78], ADA [ICH 79], CCS [MIL 89], CHILL [CCI 80], LTR-V3 [CIM 81] ...

Le non déterminisme est présent dans un système lorsque pour une même entrée, le système peut avoir des sorties différentes ; le non déterminisme peut résulter de vitesses variables d'exécution des processus composant le système ; il peut aussi résulter de l'emploi de primitives non déterministes au sein des programmes [DIJ 75]. Le non déterminisme est fortement relié au parallélisme : l'exécution de plusieurs programmes par un processeur unique, par exemple, laisse au processeur la possibilité de choisir arbitrairement les actions à effectuer et donc peut engender un non déterminisme sur le résultat de l'exécution parallèle. Cependant, ces deux notions ne doivent pas être confondues, puisque différentes : le parallélisme peut sous entendre des simultanéités d'exécutions, c'est-à-dire des notions liées au temps, alors que le non déterminisme n'exprime que des propriétés fonctionnelles.

2. NOTIONS DE SEMANTIQUES DE LANGAGE DE PROGRAMMATION

L'étude des langages de programmation se ramène à la description formelle de ses domaines : syntaxique (qui définit à l'aide d'une grammaire la structure et la forme des expressions du langage), sémantique (qui définit la signification des phrases du langage) et pragmatique (qui définit l'utilisation du langage).

Les difficultés rencontrées dans de telles études formelles viennent en grande partie de la diversité des buts poursuivis. Certains se rapportent à l'utilisateur, et encore peut-on distinguer entre celui qui se contente d'une vue globale de l'action obtenue, et celui qui entre dans les détails du calcul ; d'autres touchent ceux qui écrivent un compilateur pour le langage ou encore ceux qui sont chargés de le modifier, de le comparer avec d'autres, de le standardiser, etc ...
A ces buts purement orientés vers l'homme, qui nécessitent une définition claire et brève, mais aussi formelle, s'opposent les buts visant à obtenir des démonstrations (automatiques si possible) sur un programme, sur le langage, sur ses compilateurs, et même à produire automatiquement ces compilateurs. L'intérêt de ces derniers n'est pas affaibli par de nombreuses questions indécidables telles que l'équivalence de programmes, la terminaison des programmes pour certaines données, etc ... [PAI 71].

A l'heure actuelle, les grammaires formelles, par exemple exprimées sous la forme normale de Backus, sont le moyen largement connu et adopté pour décrire une grande partie de la syntaxe des langages.

Pour la formalisation de la sémantique, le moyen largement adopté est loin d'être trouvé, cependant, on peut regrouper les techniques actuelles de description formelle de la sémantique en deux grandes classes :

- par l'intermédiaire d'une machine censée exécuter les programmes du langage ; cette technique porte le nom de sémantique opérationnelle,
- en termes mathématiques par l'association d'un objet d'un domaine mathématique aux programmes du langage ; cette technique est appelée la sémantique mathématique.

3. VOC ERAT IN VOTIS

Dans ce travail, nous nous intéressons à la formalisation du domaine d'un langage de programmation intégrant des concepts du parallélisme et du non déterminisme (l'introduction de l'évolution dynamique dans ce langage n'est pas discutée) pour lequel on supposera données :

- une définition formelle de la majeure partie de sa syntaxe à l'aide d'une grammaire à la Backus,
- une définition moins formelle des conditions syntaxiques (représentant grossièrement sa sémantique statique) et,
- une définition informelle de sa sémantique (dynamique).

Sémantique algébrique

Notre but dans la définition d'une sémantique est principalement d'indiquer (fig. 1) comment associer un ensemble S de significations claires, précises et complètes à tout programme, syntaxiquement correct, décrit dans le langage L.

fig. 1 $\quad L \overset{sem}{\rightarrow} S$

Les significations étant souvent des objets assez complexes et peu manipulables, on préfère définir des ensembles intermédiaires plus maniables : un langage pivot D, décrit par abstraction syntaxique du langage L et un ensemble de "descriptions sémantiques" associé à D (fig. 2) :

fig. 2 : $L \overset{sa}{\rightarrow} D \overset{sem'}{\rightarrow} S$

Nous proposons alors pour définir sem' de prendre des termes de types abstraits algébriques T (fig. 3) choix déjà proposé et justifié (*) par plusieurs auteurs [PAI 82, GAU 80, MOS 80, BRO 80 a-b, ...].

fig. 3 : $L \overset{sa}{\rightarrow} D \subseteq T \overset{sem'}{\rightarrow} S$

L'étude conduit donc à définir la sémantique du langage considéré par un type abstrait algébrique et une fonction d'abstraction syntaxique. La signification d'un tel type abstrait peut être donnée par différents modèles (algébriques) tels que le modèle initial et le modèle terminal, ou encore par l'ensemble des congruences de l'algèbre des termes. On disposera ainsi, pour le même langage, d'un ensemble de modèles sémantiques selon le choix du modèle associé au type abstrait.

(*) En termes de types abstraits algébriques un programme est vu comme étant un objet qu'on peut construire par un certain nombre d'opérations et évaluer par un certain nombre d'autres.

Sujet de la thèse

L'objectif principal de la présente contribution à l'étude du parallélisme concerne la mise au point d'une méthode de description algébrique de la sémantique d'un langage de programmation fondé sur le concept de communication.

Il s'agit plus précisément d'étendre le formalisme introduit par [PAI 82, GAU 80, BRO 80 a] pour prendre en compte les notions de processus et de communication. L'idée de base de ce formalisme est d'utiliser les techniques de spécification algébrique de types abstraits : la définition d'une sémantique se ramène à associer aux différents composants du langage de programmation considéré une hiérarchie de types abstraits rendant compte des expressions, des états de mémoire et des calculs.

Notre travail consiste à étendre ce cadre algébrique en introduisant les concepts de processus et de communications. Ceci se fait en "temporisant" les différentes opérations du type et en décrivant un mécanisme de compositions d'opérations temporisées.

Les autres objectifs visés par le présent travail sont :

i) d'une part, l'extension du formalisme aux concepts de non déterminisme,

ii) d'autre part, la construction de modèles de types abstraits définissant la sémantique du langage considéré.

iii) et finalement, la définition de la complémentarité de ce type de définition algébrique avec une définition axiomatique.

Cette dernière préoccupation se justifie par le désir de faire supporter au langage un certain système de preuve.

Plan de la thèse

La description de ce travail est décomposée en quatre parties indépendantes :

- la première décrit et compare assez brièvement les différentes approches de la formalisation de la sémantique des langages de programmation,

- la seconde décrit le cadre de travail dans lequel nous nous plaçons pour mener à bien une description "algébrique" de la sémantique d'un langage de programmation. On y trouvera notamment l'essentiel des définitions et des résultats des travaux sur les types abstraits et leur spécification algébrique, ainsi qu'une description de la construction hiérarchique d'un type abstrait de formalisation d'un langage de programmation,

- la troisième formalise un langage de programmation, nommé LAGALERE, dans ses parties : séquentielle, non déterministe et parallèle (exprimé par des processus communicants). Pour chacune de ces formalisations, nous donnons une description syntaxique de la partie en considération et une sémantique formelle précédée d'une description informelle. Par ailleurs, nous décrivons la complémentarité de la formalisation algébrique d'une partie de LAGALERE avec sa formalisation axiomatique,

- la quatrième évalue notre apport dans le domaine de la sémantique des langages de programmation supportant le non déterminisme et le parallélisme et décrit les prolongements de ce travail.

SEMANTIQUE FORMELLE DES LANGAGES DE PROGRAMMATION : LES DIFFERENTES APPROCHES

SEMANTIQUE FORMELLE DES LANGAGES DE PROGRAMMATION : LES DIFFERENTES APPROCHES

1. INTRODUCTION

Un des premiers objectifs d'une définition formelle d'un langage de programmation est de fournir un sens concis et non ambigü dans une forme indépendante d'une implantation particulière.

D'autres objectifs peuvent être définis :

i) fournir au programmeur un sens clair et non ambigÜ de chaque construction du langage,

ii) fournir une base logique pour la vérification des programmes écrits dans ce langage.

Plusieurs facteurs influent le type de la définition : le but d'une définition claire et concise, le désir de supporter la vérification de programmes et la preuve de compilateurs, l'exigence de la définition totale du langage, le souhait d'une implantation automatique, etc... Pour atteindre ces exigences, de multiples modes de description sémantique peuvent être utilisés :

i) la sémantique axiomatique pour la définition primaire des instructions du langage. Ce mode de définition offre plusieurs avantages, incluant la concision, la compréhension et l'applicabilité des vérifications de programmes,

ii) la sémantique mathématique pour l'évaluation des expressions et des opérateurs, et pour la définition des conversions des types de données, de la visibilité et de la portée des identi-

ficateurs. Ce mode de définition offre plusieurs avantages incluant la concision et la formalité des définitions,

iii) la sémantique opérationnelle pour la prise en compte des problèmes de gestion de temps de réponse, de priorité, de synchronisation et coopération de processus. Ce mode de définition présente plusieurs avantages dont principalement la compréhension et son rapprochement d'une implantation réelle du langage.

Lorsque plusieurs modes de définition d'un langage sont utilisés [HEN 81], il est nécessaire d'assurer leur complémentarité. La technique de preuve, souvent utilisée, est de montrer qu'une de ces définitions fournit un modèle mathématique pour les autres.

2. DESCRIPTION DES APPROCHES SEMANTIQUES

2.1. Les approches interprétatives (ou opérationnelles) [NEU 71, PLO 81, VAN 69, ...] définissent la sémantique d'un langage de programmation en terme d'un interprète abstrait de ce langage. L'interprète est alors caractérisé essentiellement par un ensemble d'états et une fonction de transition d'un état à un autre, et la signification d'un programme est décrite par son exécution par l'interprète. La spécification de l'interprète réduit le problème de la définition sémantique d'un nombre infini de programmes à celui de la spécification de la sémantique d'un seul : l'interprète.

De telles méthodes, si elles permettent d'exprimer complètement la sémantique du langage et suggèrent ses implantations, semblent permettre difficilement la résolution des problèmes de preuves et d'équivalences de programmes.

2.2. Les méthodes axiomatiques [HOA 74, HOW 76, OWI 76, APT 79, ...] associent au langage un système formel dont les axiomes et les règles d'inférences caractérisent les objets et les constructions du langage : on associe par la même un sens à tout programme de ce langage. La primitive de ces méthodes est notée {P} S {Q} où P et Q sont des formules propositionnelles et S un morceau de programme. L'interprétation de cette primitive est la suivante : si P est vraie avant l'exécution de S et si S se termine alors Q est vraie après l'exécution de S.

Un des intérêts de ces méthodes est de laisser facilement certaines libertés à l'implantation (qui peut imposer à son tour certains axiomes et règles d'inférence spécifiques) et de permettre de prouver des théorèmes sur des programmes.

2.3. Les méthodes mathématiques

2.3.1. L'approche dénotationnelle [GOR 79, SCO 71, TEN 76, ...] cherche à associer directement à tout programme une fonction de ses entrées dans ses sorties. A chaque règle syntaxique (sous la forme normale de Backus) est associée une règle sémantique qui permet la construction d'une fonction à partir des fonctions associées aux facteurs droits. Au lieu de décrire la sémantique au moyen de calculs, on considère tout programme comme un système d'équations (règles sémantiques) dont on cherche la solution dans un espace de fonctions approprié : cette solution s'obtient par des procédés classiques d'approximation et utilise largement la théorie du point fixe.

Un certain nombre de travaux effectués dans cette direction, en définissant la sémantique des langages de programmation, étudient les concepts fondamentaux de la programmation et construisent des méthodes de preuves.

2.3.2. L'approche calculatoire [FIN 76, PAI 82, COU 78, BRO 80 a-b, ...] explicite la sémantique d'un langage en termes de calculs sur certaines structures d'informations (un système formel associé au langage), où les concepts d'informations et de modifications élémentaires y sont décrits. Le calcul est alors la suite des modifications élémentaires traduisant une élaboration d'un programme (sémantique opérationnelle), pouvant être interprété, s'il est fini, comme une modification composée (sémantique dénotationnelle).

Le travail présenté ici utilise essentiellement cette notion de structure d'information que nous spécifions algébriquement par un certain type abstrait, où le calcul s'exprime comme une composition d'opérations de type abstrait. Il est cependant éloigné des études sur les preuves puisque essentiellement descriptif.

Remarques :

Pour rendre ces différentes sémantiques plus manipulables quelques techniques peuvent être mises en oeuvre :

- définir des sémantiques intermédiaires qui permettraient le passage progressif d'une sémantique à l'autre,

- définir un cadre mathématique dans lequel ces sémantiques pourront toutes s'exprimer (ainsi ces différentes sémantiques ne peuvent représenter que différentes facettes de la même réalité mathématique),

- définir des sémantiques très abstraites de manière à retarder le plus longtemps possible tout choix d'implantation.

3. COMPARAISON DES APPROCHES [BER 82]

Les trois méthodes sont schématisées dans la figure 1 où P dénote un programme écrit dans un langage pivot et D dénote l'espace état du programme.

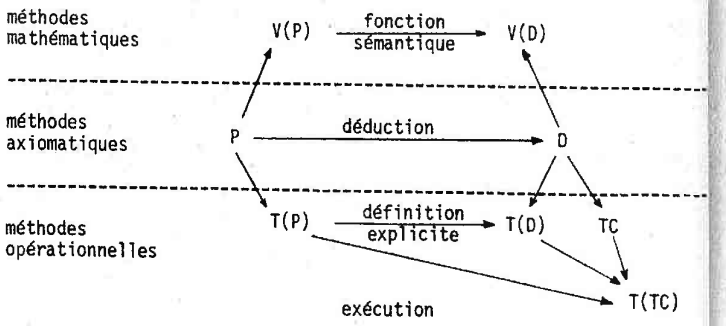


fig. 1 : comparaison des approches

Avec l'approche opérationnelle, le programme P est traduit en un programme T(P), écrit dans le langage de l'interprète dont T(D) est l'espace état, image de D. La sémantique du programme T(P) est définie implicitement par la fonction de transition de l'interprète. Pour définir explicitement la sémantique de P, un ensemble TC de tests par cas (données) est défini pour ce programme. Ces tests appliqués dans l'état T(D) donnent un nouvel ensemble T(TC) de tests par cas. La sémantique du programme est alors définie explicitement par l'exécution du programme T(P) avec les données T(TC).

Avec l'approche mathématique, un espace structure d'information V(D), est défini, dans lequel l'espace état D est appliqué. Le programme P est transformé en un programme V(P) dans le langage du système formel considéré. Finalement, les fonctions sémantiques sont définies de telle manière à associer les constructions de V(P) et V(P) lui-même, avec un sous espace de V(D). Ainsi la sémantique du programme V(P) est explicitement définie et par là même, celle du programme P.

L'approche axiomatique est plus directe : les constructions de P sont directement appliquées dans l'espace D par la sémantique explicitement définie pour P.

Dans la figure 2, nous résumons les développements associés à ces trois méthodes fondamentales pour la sémantique des langages de programmation, et montrons l'articulation de la présente étude par rapport à ceux-ci.

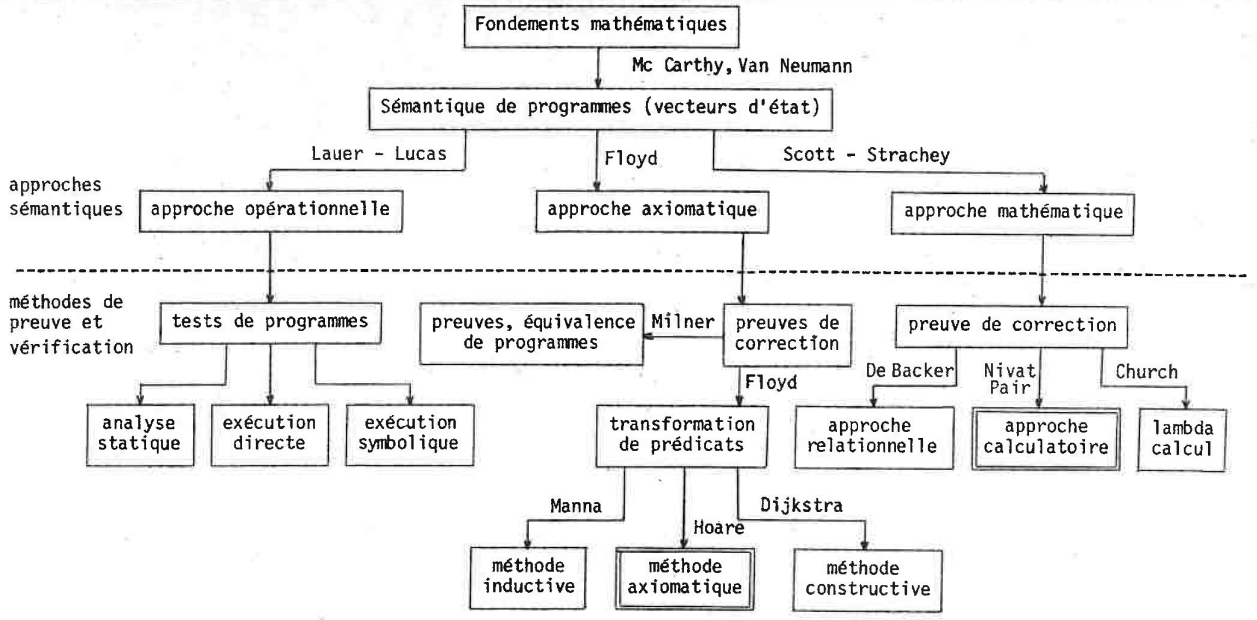


fig. 2 : approches sémantiques et les méthodes de preuve et de vérification induites

Sémantiques auxquelles on s'intéresse dans le but d'une définition formelle et d'une étude de complémentarité

CADRE ALGEBRIQUE DE LA FORMALISATION D'UNE SEMANTIQUE D'UN LANGAGE DE PROGRAMMATION

1. INTRODUCTION

En général, si l'on admet que le concept de types abstraits est étroitement lié à l'expression de propriétés indépendamment d'une représentation particulière, il est possible d'imaginer de nombreuses techniques de spécification.

De plus, nombreux sont les avantages des types abstraits de données : flexibilité incrémentale, pouvoir d'abstraction et de paramétrisation, conception descendante et modulaire, facilité de vérification, etc ... Lorsqu'on applique de telles techniques à la spécification d'un langage de programmation, on peut ajouter les points suivants à cette liste d'avantages :

i) la notion d'équivalence de programmes (une base du processus de développement des programmes) est directement obtenue de la sémantique lorsqu'elle est formalisée par des types abstraits algébriques,

ii) on n'a pas besoin d'avoir un métalangage que le programmeur doit connaître pour comprendre la signification des programmes,

iii) la définition sémantique du langage emploie la même technique que celle employée pour spécifier ces objets. Ceci mène à un seul cadre cohérent pour spécifier les structures de données et les structures de contrôle du langage.

C'est pour ces raisons que nous avons choisi de formaliser la sémantique d'un langage de programmation par le biais des spécifications de types abstraits. Différentes techniques de spécification ont été proposées.

Nous utilisons dans toute la suite l'approche algébrique qui bénéficie de nombreux travaux et se prête bien à notre démarche.

Ce chapitre a pour objet, d'une part, de rappeler les principales définitions habituellement utilisées, et d'autre part, de décrire la démarche de construction d'un type abstrait formalisant le langage considéré.

2. CONCEPTS ALGEBRIQUES ET TYPES ABSTRAITS

Le but de ce paragraphe est de fixer les notations et de préciser la terminologie employée dans cette thèse. Nous présenterons, sans aucune démonstration, l'essentiel des définitions et résultats que l'on rencontre classiquement dans les travaux sur "les types abstraits algébriques" que nous nous sommes autorisés à emprunter aux travaux de [GOG 78, WIR 83, LES 79, BID 81, PAI 80, REM 82, HUE 80, MUS 80 b, ZHA 83]. Pour une plus ample étude sur le sujet, nous recommandons au lecteur de se reporter à ces travaux.

2.1. Spécification

2.1.1. Syntaxe

Une spécification algébrique est un triplet <S,F,E> où :

. S est un ensemble de noms de domaines ou sortes et contenant une sorte d'intérêt notée s qui, intuitivement, est le nom du domaine que l'on est "en train" de définir,

. F est un ensemble de noms d'opérations munies de leur profil construit sur S^*. Ces profils précisent le type des arguments (les domaines) et du résultat (le codomaine) de chaque opération. La sorte d'intérêt doit apparaître dans chaque profil (appelé aussi signature).

. E est un ensemble d'équations conditionnelles entre des termes construits en utilisant les symboles de F et des symboles de variables universellement quantifiées.

Exemple :

Considérons la spécification du type NAT1 des entiers naturels où les différents constituants sont séparés par les mots clés Sorts, Opns et Eqns :

```
Sorts : Nat / Bool ;
Opns  : zero : → Nat
        succ : Nat → Nat ;
        nul  : Nat → Bool ;
Eqns  : x ∈ Nat ;
      nul (zero) = true ;
      nul (succ(x)) = false ;
Fin ;
```

Remarques :

i) On privilégie dans la définition précédente la sorte d'intérêt, les autres sont considérées comme déjà définies ailleurs et font partie de l'environnement (par exemple la sorte Bool des booléens dans l'exemple ci-dessus).

ii) Les équations servent à définir un ensemble de relations d'équivalence sur les objets du type abstrait. Intuitivement deux termes t et t' sont égaux par rapport à l'équation g = d, si t' dérive de t par remplacement de g (ou d) dans t par d (ou g). La fermeture réflexive, symétrique et transitive de cette relation de substitution est une relation d'équivalence notée $=_E$ dite la plus petite congruence de E (celle qui permet de valider les équations de E sans plus). Mais il existe une autre congruence, celle qui valide les équations pour n'importe quel modèle de la spécification.

2.1.2. Sémantique

Un modèle de la spécification <S,F,E> est une <S,F>- algèbre A telle que tous les axiomes de E soient valides dans A et A générée par les termes, i.e. tous les éléments des ensembles supports de A peuvent s'obtenir par l'interprétation t_A des termes clos de l'algèbre des termes $T_{<S,F,E>}$, et que l'on peut définir par :

$f \in F,\ t, t_1, \ldots, t_n \in T_{<S,F,E>}$

si $t = f(t_1, \ldots, t_n)$ alors $t_A = f_A(t_{1A}, \ldots, t_{nA})$

sous la condition que tous les t_{iA} soient définis,
sinon t_A est indéfini

Ceci revient à garantir que tous les axiomes du type soient valides dans le modèle.

Remarques : Soit T = <S,F,E> une spécification algébrique du type T

i) Soient A une F-algèbre de T et g = d une (F-) équation

- A est un modèle de l'équation g = d (ce que l'on note $A \models g = d$) ou encore A valide l'équation g = d si est seulement si pour toute interprétation i des variables g et d, on a i(g) = i(d), autrement dit g et d dénotent le même élément de support, quelque soit l'interprétation de leurs variables comme élément de A.
- A valide E ou encore A est un modèle de E (ce que l'on note $A \models E$) si et seulement si A valide chaque équation de E.
- la relation $=_A$ définie par $g =_A d$ si et seulement si $A \models g = d$ est une congruence sur l'algèbre libre engendrée par l'ensemble X des variables.

ii) Quand on choisit la relation $=_E$ (la plus petite congruence de E) pour valider les équations, on se place dans le cadre de l'algèbre initiale I :

$$I \models g = d \text{ ssi } \forall i : X \to I \quad i(g) =_E i(d)$$

iii) On peut au contraire considérer deux termes comme équivalents dès que l'on ne peut prouver qu'ils sont différents. C'est l'approche algèbre terminale T :

$$T \models g = d \text{ ssi } \forall i : X \to T \quad i(g) \neq_E i(d)$$

iv) En pratique, il semble raisonnalbe d'utiliser l'algèbre initiale lorsque l'on veut raisonner sur le type lui-même pour démontrer des équations, car c'est dans ce contexte que l'on fait le moins d'hypothèses.

v) Deux termes équivalents dans le modèle initial, le sont dans tout modèle du type, mais l'inverse n'est pas vrai. Par contre deux termes non équivalents dans l'algèbre terminale le sont dans tout le type.

Exemple :

Un modèle pour le type NAT1 donné en 2.1.1. est l'algèbre
$\mathcal{N} = (\mathbb{N}, \mathbb{B}; 0, s, V, F, N?)$ où :
. $\mathbb{N}$ est l'ensemble des entiers naturels,
. $\mathbb{B} = \{V,F\}$ est l'ensemble des valeurs de vérité,
. 0 est le nombre constant zéro,
. s est la fonction définie par $s = \lambda x . (x+1)$,
. N? est la fonction définie par $N? = \lambda x . (x=0)$,
pour lequel on peut interpréter les opérations comme suit :

$zéro_{\mathbb{N}} = 0$, $succ_{\mathbb{N}} = s$

$true_{\mathbb{N}} = V$, $false_{\mathbb{N}} = F$

et $null_{\mathbb{N}} = N?$

2.2. Construction

2.2.1. Spécification partielle de types abstraits

Cette technique de spécification présente l'originalité de s'écarter de celles qui considèrent la sémantique de la présentation d'un type

abstrait comme une algèbre hétérogène totale (i.e. une algèbre à plusieurs ensembles supports et où les fonctions sont définies sur tout leur domaine). Ici, parmi les modèles considérés on va avoir des algèbres partielles, c'est-à-dire des algèbres où les opérations ne sont pas définies sur tout leur domaine. Il n'y a donc pas de risque d'introduction des inconsistances par le biais de nouvelles valeurs : pour certaines opérandes, ces opérations ne retournent pas de résultats.

Une spécification est dite partielle si une de ses opérations est partiellement définie. La "totalisation" de telles spécifications revient à les décrire sous la forme $\langle S, F \cup F_{err}, E \cup E_{err} \rangle$ où E_{err} est l'ensemble des équations où figurent les opérations de F_{err}: un ensemble d'opérations avec "erreur".
Cette façon de procéder permet de définir des classes d'équivalence pour les termes "normaux" d'une part, et pour les termes "erronés" d'autre part ; elle ramène également l'étude des algèbres partielles à celle des algèbres totales.

Exemple :

Nous donnons ci-dessous une (nouvelle) présentation partielle du type NAT2 des entiers positifs :

```
Sorts : Nat ;
Opns  : zéro : → Nat ;
        succ, pred : Nat x Nat → Nat ;
Eqns  : x ∈ Nat ;
        pred(succ(x)) = x ;
        % pred(zéro) est non défini %
Fin ;
```

Pour "totaliser" cette spécification, nous allons définir le terme pred(zéro) comme une erreur.

La présentation avec erreur du type Nat2 est la suivante :


```
Sorts : Nat ;
OK-Opns : zéro : → Nat ;
          succ,pred : Nat x Nat → Nat ;
ER-Opns : indef : → Nat ;
OK-Eqns : x ∈ Nat ;
          pred(succ(x)) = x ;
ER-Eqns : pred(zéro) = indef ;
Fin ;
```

2.2.2. Spécification paramétrée de types abstraits

On veut souvent pouvoir utiliser une même spécification dans différents contextes, c'est pour cette raison que l'on souhaite construire plus des schémas de types abstraits que des types abstraits. La technique de spécification algébrique paramétrée apporte le remède à ce problème de construction de types abstraits génériques.

La spécification paramétrée se décrit par une paire PSPEC = (SPEC - 1, SPEC) où SPEC est une spécification des paramètres incluse dans la spécification SPEC-1. Les instances des paramètres formels de SPEC qui définissent la spécification effective SPEC - 2 sont des "morphismes de spécification" de SPEC vers SPEC - 2. Ainsi, une instance de la spécification paramétrée PSPEC correspond "en gros" à une spécification SPEC - 1 dans laquelle la partie SPEC a été renommée. Ce renommage est effectué à partir des spécifications utilisées comme paramètres effectifs. C'est une application de SPEC - 2 dans SPEC - 1 ∪ SPEC - 2 telle que l'image de chaque sorte paramètre formel T_i soit une sorte S_j dans SPEC - 2, celle de chaque symbole paramètre formel g_i soit un symbole f_j de SPEC - 2.

2.2.3. Spécification hiérarchique de types abstraits

Pour construire une spécification $\langle S,F,E \rangle$ on peut procéder par extensions ou enrichissements successifs. On part d'une spécification primitive $\langle S_0,F_0,E_0 \rangle$ dont l'axiomatisation est supposée être connue et inconditionnelle. L'enrichissement conserve (au sens de classes d'équivalence)

les objets du type existant (s'il est conservatif) et augmente sa définition par des opérations et des équations les définissant. On pourra ainsi enrichir le type NAT1 avec les opérations et l'axiomatisation suivantes :

Opns : add : Nat x Nat → Nat ;
le : Nat x Nat → Bool ;
Eqns : x,y ∈ Nat ;
add (x, zéro) = x
add (x,succ(y)) = succ (add(x,y))
le (zéro,x) = true
le (succ(x),zéro) = false
le (succ(x), succ(y)) = le (x,y)
Fin ;

L'extension, quant à elle, peut se définir sur un type donné $\langle S_0,F_0,E_0$ par de nouvelles sortes S, opérations F et équations E, tel que $S_0 \subseteq S$ $F_0 \subseteq F$ et $E_0 \subseteq E$. Ainsi, par exemple, on pourra définir les complexes par la spécification étendant celle présentée ci-dessous :

Sorts : Complex / Nat, Bool ;
Opns : make : Nat x Nat → Complex ;
re : Complex → Nat ;
Im : Complex → Nat ;
plus : Complex x Complex → Complex ;
Eqns : c ∈ Complex ; i,j,k,l ∈ Nat ;
re (make(i,j)) = i
im (make(i,j)) = j
plus (make(i,j), make (k,l)) = make (add(i,k), add(j,l))
Fin ;

Une spécification hiérarchique est ainsi une suite finie de triplets SPECH = $(\langle S_0,F_0,E_0\rangle, \ldots, \langle S_n,F_n,E_n\rangle)$, $n \in \mathbb{N}$, où pour tout i = 0, ...,

. $\langle S_i,F_i\rangle$ est une signature telle que :

$$\langle S_0, F_0\rangle \subseteq \ldots \subseteq \langle s_n,F_n\rangle$$

. E_i est un ensemble d'équations tel que E_0 est un ensemble d'équations inconditionnelles et $E_0 \subseteq \ldots \subseteq E_n$.

Remarques :

i) Pour i = 0, ..., n, $SPECH|_i = (\langle S_0,F_0,E_0\rangle, \ldots, \langle S_i,F_i,E_i\rangle)$ est aussi une spécification hiérarchique.

ii) Que l'on procède par enrichissements ou extensions d'un type donné pour construire la spécification d'un nouveau type, on doit s'assurer que les équations introduites ne modifient pas l'ensemble des classes d'équivalence, c'est-à-dire que la spécification obtenue est conforme à ce que l'on attend d'elle, et qu'elle n'ajoute pas de nouveaux objets aux types de la spécification initiale.

2.3. Qualités d'une spécification

Etant donnée une spécification <S,F,E> on peut se poser la question de savoir si elle est conforme à ce que l'on attend d'elle, si elle ne conduit pas à des contradictions, et si elle définit complètement l'ensemble d'objets et d'opérations du type traité. Ceci conduit à parler des propriétés de consistance, de correction, de complétude suffisante d'une spécification.

2.3.1. Consistance

Elle exprime le fait que l'axiomatisation de la spécification (autrement dit ses équations) ne conduit pas à des contradictions vis-à-vis des types déjà définis et utilisés dans la spécification. Autrement dit, la consistance équivaut à l'existence d'un modèle.

Prouver la consistance de la spécification <S,F,E> extension de la spécification $\langle S_0,F_0,E_0\rangle$ revient à démontrer que :

pour tous termes t_o, t'_o de $T_{\langle S_o,F_o,E_o\rangle}$

$$t_o =_E t'_o \Rightarrow t_o =_{E_o} t'_o$$

Considérons par exemple la spécification de NAT1 que l'on a déjà enrichie par les opérations "add" et "le" et les équations qui leur sont relatives. Cette spécification est consistante. Intuitivement, cela provient du fait que chaque équation caractérise des objets différents. Mais si on enrichit cette spécification par :

Opns : mult : Nat x Nat → Nat ;
Eqns : x,y ∈ Nat ;
mult (x,y) = mult (y,x)
mult (zéro,x) = zéro
mult (succ(x), y) = succ (mult (x,y))
Fin ;

on obtient une spécification inconsistante puisque d'une part,

le (mult(succ(succ(zéro)),zéro),succ(zéro)) =
le (mult(zéro,succ(succ(zéro))),succ(zéro)) =
le (zéro,succ(zéro)) =
true

et d'autre part,

le (mult(succ(succ(zéro)),zéro),succ(zéro)) =
le (succ(mult(succ(zéro),zéro))),succ(zéro)) =
le (succ(succ(mult(zéro,zéro))),succ(zéro)) =
le (succ(succ(zéro)),succ(zéro)) =
le (succ(zéro),zéro)) =
false

ce qui est contradictoire.

La preuve de consistance d'un système formel est en général un travail difficile à effectuer directement, cependant dans le cas des types abstraits algébriques, on peut la caractériser syntaxiquement.

En effet, on considère les équations comme des règles de réécriture et on cherche à prouver qu'elles possèdent la propriété de confluence : si on applique successivement une suite de règles de réécriture pour réduire une formule jusqu'à obtenir un terme irréductible, l'ordre d'application des règles de réécriture n'a aucune importance.

2.3.2. Complétude

Savoir qu'une spécification algébrique d'un type abstrait est consistante est en général insuffisant. En effet, lorsqu'on présente un type abstrait, on se contente de préciser un ensemble minimal de propriétés requises. Ainsi, bien que l'on puisse prouver la consistance de sa spécification, on ne sait pas comparer certains de ses termes. Plus précisément, on ne peut démontrer, en utilisant les équations, ni l'égalité, ni l'inégalité de certains termes ; ce qui est une définition de l'incomplétude. Cette incomplétude est d'ailleurs très intéressante dans la mesure où elle laisse beaucoup de possibilités à l'implantation.

La complétude d'une spécification <S,F,E> revient à prouver que :

étant donné u et v, deux termes de $T_{\langle S,F,E\rangle}$

soit $u =_E v$, soit non $(u =_E v)$

En termes de modèles, la complétude s'exprime de manière agréable : une spécification est complète si elle est consistante et que tous ses modèles sont isomorphes. En conséquence, une spécification incomplète admet des modèles non isomorphes.

2.3.4. Complétude suffisante

Si la complétude d'une spécification algébrique n'est, ni toujours réalisable, ni même souhaitable, il est cependant évident que n'importe quelle spécification n'est pas intéressante. La complétude suffisante est une propriété minimale caractérisant les "bonnes" spécifications.

Elle exprime intuitivement qu'une spécification <S,F,E> n'ajoute pas de nouveaux objets aux types déjà définis. Dans ce cas, pour chaque opération f de F dont le codomaine n'est pas le type d'intérêt s et pour chaque terme $f(t_1, \ldots, t_n)$ on doit trouver un terme u d'une sorte externe $s_i \neq s$ tel que $f(t_1, \ldots, t_n) = u$ est un théorème démontrable en utilisant l'axiomatisation E.

Par ailleurs, la complétude suffisante assure que toute formule de la forme u = v entre types externes est un théorème ou alors sa négation est un théorème. Intuitivement, ceci permet de décrire complètement "le comportement" du type vis-à-vis des opérations externes, i.e. de codomaine autre que s, sans se préoccuper de la description interne.

Remarque :

Il a été montré que la complétude d'un ensemble d'axiomes est en général indécidable. Cependant, si l'on impose quelques restrictions aux systèmes d'équations, alors il existe des conditions suffisantes pour vérifier cette propriété syntaxiquement.

Pour une spécification <S,F,E>

i) distinguer dans S, la sorte d'intérêt s et les autres sortes (dont celle des booléens) dites externes,

ii) partitionner l'ensemble F des opérations en trois ensembles disjoints :

- C : ensemble des opérations de construction des objets du type,
- R : ensemble des opérations de destruction (ou extension) des objets du type,
- O : ensemble des opérations d'observation des objets du type.

Ainsi, l'ensemble des opérations internes est formé par la réunion des constructions et des extensions, et l'ensemble des opérations externes par les observateurs.

iii) distinguer dans l'axiomatisation E, d'une part les équations caractérisant les extensions, et d'autre part, celles qui caractérisent l'effet des opérations d'observation sur les constructeurs, où chacun doit avoir une partie gauche de la forme o(c(x*),y*) et une partie droite vérifiant la décroissance d'imbrication des termes pour obtenir une spécification suffisamment complète.

2.4. Illustration

Nous présentons ci-dessous une spécification du type abstrait paramétré ensemble d'atomes

```
Type  : ENS [ATOM] ;
Sorts : Ens /                          % sorte d'intérêt %
        Atom, Bool ;                   % sortes externes %
Opns  : empty  : → Ens ;               % constante du type %
        insert : Ens x Atom → Ens ;    % constructeur binaire %
        delet  : Ens x Atom → Ens ;    % destructeur binaire %
        isempty : Ens → Bool ;         % observateur unaire %
        has    : Ens x Atom → Bool ;   % observateur binaire %
Eqns  : s ∈ Ens, i, j, ∈ Atom ;        % variables implicitement
                                         universellement quantifiées %

      % les équations relatives aux extensions %
        delet (empty,i) = empty
        delet (insert(s,i),j) = si eq (i,j)
                                  alors delet (s,j)
                                  sinon insert (delet(s,j),i)
                                fsi
```

```
% les équations relatives aux observateurs %
  isempty (empty) = true
  isempty (insert(s,i)) = false
  has (empty,i) = false
  has (insert(s,i),j) = si eq (i,j)
                        alors true
                        sinon has (s,j)
                       fsi
```

Fin ;

La spécification de ENS [ATOM] suppose l'existence d'une opération d'égalité entre les termes de la sorte des paramètres Atom dont le profil est eq : Atom x Atom → Bool.

Une équation de la forme r = si c alors u sinon v fsi n'est qu'un abus d'écriture du système d'équations

$$\begin{cases} c = \text{true} \Rightarrow & r = u \\ c = \text{false} \Rightarrow & r = v \end{cases}$$

Nous l'utiliserons largement dans la suite.

3. DESCRIPTION HIERARCHIQUE D'UN TYPE ABSTRAIT ALGEBRIQUE ASSOCIE A UN LANGAGE DE PROGRAMMATION

3.1. Les mises en oeuvre des spécifications algébriques de la sémantique des langages de programmation

Les types abstraits de données ont été suggérés par plusieurs auteurs [LIS 77, ZIL 81, GUT 78, DER 79, ...] comme étant un cadre d'abstraction flexible, uniforme et utile pour la spécification formelle des structures de données traitées dans les langages de programmation. Ces mêmes techniques de spécification algébrique pour la formalisation des langages - qu'ils soient applicatifs ou procéduraux - ont été appliquées avec succès. On a vu ainsi se définir la sémantique (algébrique) des fonctions récursives [BRO 80b], des variables et des affectations [PEP 79], des procédures [GAU 78], du non déterminisme [BRO 81] et du parallélisme [BRO 82, FIN 83].

Ces diverses applications diffèrent essentiellement dans la manière de considérer le concept d'état et de décrire la sémantique des instructions.
Dans [GOG 78, GAU 80, PAI 82 et WAN 75], où la construction du type est fondée sur un modèle fixe, les opérations qui évaluent les expressions (ou les identificateurs de variables) ont comme opérande un état et un environnement.
Dans [BRO 80 a et PEP 79], où la construction du type est fondée sur l'ensemble des modèles finiment engendrés, l'explicitation des états est évitée par la donnée d'un ensemble de règles de transformation des constructions du langage applicatif. La sémantique de ces derniers est définie en termes d'un type abstrait algébrique.
Dans [MOS 80], où la construction du type est fondée sur un modèle initial, les mentions explicites de l'état sont aussi évitées, mais les expressions et les instructions sont considérées comme des actions qui consomment et produisent des valeurs.

L'approche que nous avons adoptée associe aux différents composants du langage une hiérarchie de types abstraits rendant compte des expressions, des états de mémoires et des calculs. La construction de ces types abstraits sera basée sur l'ensemble des modèles finiment engendrés (en particulier, ceux qui présentent le caractère de minimalité [WIR 83]).

3.2. La construction hiérarchique du type abstrait associé à un langage de programmation

Dans l'approche algébrique, un langage de programmation peut être décrit par un type abstrait, tel qu'il est présenté dans [BRO 80 b] de la manière suivante :

- la syntaxe (abstraite) contexte libre correspond à l'algèbre des termes de la signature du type. Les sortes dénotent alors les entités syntaxiques,
- les conditions syntaxiques, que nous désignons par le terme de "sémantique statique", sont exprimées par des prédicats de définitions particulières restreignant ainsi l'algèbre des termes,
- la sémantique dynamique du langage est donnée par un nombre de nouvelles sortes (domaines sémantiques) et de formules équationnelles décrivant la signification des constructions du langage.

La signification d'un tel type abstrait (et par la même occasion, celle associée au langage de programmation) est expliquée par des modèles particuliers tels que les modèles initiaux ou terminaux mais aussi par l'ensemble des modèles possibles que l'on peut décrire par l'ensemble des congruences sur l'algèbre des termes. Il est ainsi possible d'avoir plusieurs modèles sémantiques pour les programmes du langage formalisé, selon le choix du modèle associé au type abstrait.

En pratique, pour décrire un langage de programmation, nous avons à considérer d'une part les valeurs qu'il traite (entiers, booléens, etc ...), et d'autre part les constructions et les phrases qu'il possède (expressions, déclarations, instructions, ...). Le type abstrait correspondant devra alors contenir des sortes pour les valeurs (Ent, Bool, ...), des sortes pour les expressions (Ident, Bexp, ...) et une sorte pour les déclarations et les instructions (Modif). Toute expression du langage doit pouvoir repérer des valeurs ; chaque valeur dépend d'un état d'une machine. Pour exprimer ce fait, une sorte des états (State) est introduite; ainsi qu'une opération eval qui, pour chaque état, évalue les expressions. Classiquement, la signification d'une instruction est un changement d'états (transformation d'un état en un autre). Ici on change de type abstrait en modifiant les équations et par conséquent les opérations. Pour rendre compte de cette modification, on définit des opérations qui font passer d'un type à un autre.

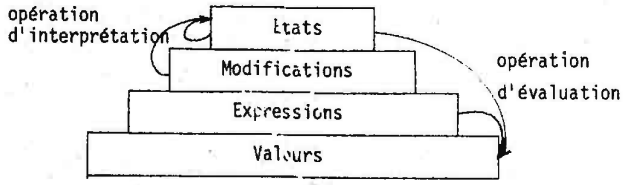
Pour définir la sémantique des langages de programmation, une méthode, souvent mise en oeuvre, consiste à étendre la sémantique, supposée connue, des objets de base (ou encore primitifs) aux autres objets qui sont les constructeurs du langage. Une construction progressive et structurée du type à associer est ainsi rendue assez systématique. On décrit :

- le niveau primitif par la définition des sortes de base,
- le niveau langage par la définition des sortes, des expressions et des modifications (déclarations de variables, des procédures,... et construction du langage).(*)
- le niveau structure de données par la définition de la sorte de l'état de l'interprête et de toutes sortes auxiliaires facilitant la description du langage.

(*) C'est à ce niveau qu'une fonction d'abstraction syntaxique peut être définie.

- le niveau évaluation par la donnée d'une opération reliant les sortes d'expression aux sortes valeurs correspondantes.
- le niveau interprétation par la donnée d'une opération reliant les modifications aux états.

La hiérarchie de la construction du type peut être schématisée par la pyramide suivante :



Construction hiérarchique d'un type formalisant un langage de programmation

SEMANTIQUE ALGEBRIQUE DE LAGALERE

SEMANTIQUE ALGEBRIQUE DE LAGALERE

le nom de ce langage est un acronyme pour : "LAngage de proGrAmmation paralLèle et non déteRministE".

Il est inspiré de PASCAL [WIR 72] et CSP [HOA 78] dans ses parties : séquentielle (nommée LPS) et parallèle (nommée LPP), et dont la syntaxe est précisée en annexe 1. Chacune de ces parties va être formalisée complètement ; et pour chaque formalisation, nous procédons en trois temps :

i) nous donnons tout d'abord une description de la syntaxe et de la sémantique informelle de la partie considérée,

ii) nous donnons ensuite les principales sortes, opérations et équations du type spécifiant la partie considérée,

iii) finalement, nous étudions le type spécifié en vue de décrire sa sémantique et par là même celle de la partie qu'il spécifie.

Pour simplifier l'exposé et supprimer de nombreux tests dans les équations sémantiques, nous ne considèrerons que les programmes qui répondent aux contraintes contextuelles; c'est-à-dire les programmes syntaxiquement corrects.

1. DESCRIPTION DE LPS

1.1. LPS Déterministe

LPS déterministe est apparenté à PASCAL en ce que tous les identificateurs de variables entières et booléennes doivent être déclarées.
Les instructions de LPS déterministe sont l'affectation habituelle, la composition séquentielle, la conditionnelle, la répétition TANT QUE et l'appel procédural.

1.1.1. Description syntaxique de LPS déterministe : un exemple

Nous donnons ici un simple aperçu de ce qu'est un programme décrit par LPS déterministe, dont la syntaxe formelle fait partie de celle présentée en annexe 1 pour le langage en entier.(*)

```
PROGRAM Nbrepremier ;
VAR p, n, prem : integer;
BEGIN read(n) ;
   IF n = 2 THEN prem := false
   ELSE  prem := (n - 2 * (n / 2)) < > 0 ;
         p := 1 ;
         WHILE (prem AND (p * p < n))
         DO   p := p + 2 ;  prem := (n - p * (n / p) < > 0
         OD
   FI ;
   IF prem THEN write (n, 'est premier')
   ELSE write (n, 'n'est pas premier')  FI
END.
```

(*) les instructions d'entrée/sortie sont ajoutées dans les exemples uniquement dans le but illustratif : elles ne font pas partie du langage.

1.1.2. Sémantique informelle de LPS déterministe

1.1.2.1. Les structures de données :

LPS déterministe permet de déclarer et manipuler des objets de types prédéfinis :

. INTEGER permettant de représenter un sous ensemble des entiers au sens mathématique du terme,

. BOOLEAN permettant de représenter des objets pouvant prendre deux valeurs TRUE ou FALSE, avec l'ordre FALSE < TRUE.

Ces objets sont, soit des constantes (représentées par des valeurs du type), soit des variables.

Pour toute variable les opérations suivantes sont toujours définies :

- l'affection d'une valeur à l'objet désigné,
- l'accès à la valeur de l'objet désigné,
- la comparaison des valeurs d'objets du même type.

La désignation d'un objet se fait par une déclaration d'identificateurs. Ainsi deux objets sont du même type si leurs déclarations mentionnent le même type ou s'ils sont déclarés en une seule déclaration.

1.1.2.2. Les expressions

Une expression décrit une valeur à partir des valeurs d'opérandes qui peuvent être variables ou des constantes. L'évaluation d'une expression en la valeur associée doit tenir compte d'une part du parenthésage et d'autre part de la priorité des opérateurs qui est donnée dans un ordre décroissant :

6) +, -, NOT (unaire)
5) *, /
4) +, -
3) =, <>, <, >, <=, >=
2) AND
1) OR

A priorité égale d'opérateurs, l'évaluation se fait de gauche à droite. Les expressions manipulées dans LPS déterministe sont :

i) les expressions arithmétiques :

Ce sont les expressions dont les opérandes sont du type entier. Les opérateurs prédéfinis pour ces opérandes sont + (la même valeur si unaire, l'addition sinon), - (la valeur opposée si unaire, la soustraction sinon), * (la multiplication) et / (la division entière).

Les opérations +, - et + provoquent l'exception "overflow" si le résultat n'appartient pas à la dynamique implantée par le type INTEGER. L'opération / provoque l'exception "zéro-divide" si le diviseur est nul.

ii) les expressions booléennes :

Ce sont les expressions dont les opérandes sont de type booléen ou des expressions de comparaison. Le résultat est de type BOOLEAN. Les opérateurs prédéfinis pour les opérandes booléens sont NOT (négation logique), AND (et logique) et OR (ou logique).
Les opérateurs de comparaison sont = (égalité), <> (inégalité), > (supérieur strictement), >= (supérieur ou égal), < (inférieur strictement) et <= (inférieur ou égal).

1.1.2.3. Les phrases

Outre les déclarations de variables et la composition séquentielle (notée ;), LPS déterministe fournit les primitives d'actions (instructions) suivantes :

i) l'affectation, notée "variable := expression", permettant d'affecter la valeur d'une expression à une variable ; l'expression et la variable doivent être du même type,

ii) la composition conditionnelle, notée IF expression booléenne THEN instructions ELSE instructions FI, spécifiant une expression booléenne conditionnant l'exécution de la phrase ; l'expression

booléenne est évaluée :

- si elle est vraie, les instructions suivantes du mot-clé THEN sont exécutées,
- sinon et si une partie ELSE est présente, alors les instructions suivantes de ce mot-clé sont exécutées, sinon la phrase est ineffective.

iii) la répétition TANTQUE, notée WHILE expression booléenne DO instructions OD, permettant de spécifier une expression booléenne conditionnant la poursuite de l'exécution itérée des instructions comprises entre les mots-clés DO et OD ; le test de l'expression booléenne est en début de boucle.

1.1.3. Sémantique algébrique de LPS déterministe

Nous allons nous intéresser successivement à la description des structures de données, des expressions, des phrases et de l'interprête abstrait de LPS déterministe ; ce qui nous permettra de spécifier complètement le type abstrait formalisant. Pour atteindre ces objectifs, nous utilisons partiellement des résultats dus à [GAU 80 et PAI 82].

1.1.3.1. Les structures de données

Occupons-nous tout d'abord du type des booléens :
trois opérations sont possibles correspondant aux opérations habituelles de la logique (et, ou, non) ; nous les spécifions en supposant les constructeurs constants (les valeurs de vérité) true et false.

```
Type  :  BOOL(EAN) ;
Sorts :  Bool ;
Opns  :  true, false : → Bool ;
         ¬ : Bool → Bool ;
         ∧, ∨, xor, eq : Bool x Bool → Bool ;
Eqns  :  b, b1, b2, b3 ∈ Bool ;
   xor (true, false) = true ;
   xor (true, true) = false ;
   xor (false, false) = false ;
   xor (b1, b2) = xor (b2, b1) ;
    ∧ (true, b) = b ;
    ∧ (false, b) = false ;
   ¬b = xor (true, b) ;
   eq (b1, b2) = xor (true, xor (b1, b2)) ;
   ∨ (b1, b2) = xor (b1, xor (b2, ∧ (b1, b2))) ;
   ∧(xor (b1, b2), b3) = xor ( ∧(b1, b3), ∧(b2, b3)) ;
   ∧(b1, xor (b2, b3)) = xor ( ∧(b1, b2), ∧(b1, b3);
Fin ;
```

Maintenant, décrivons le type des entiers : cinq opérations sont possibles, correspondant aux opérations classiques de l'arithmétique (l'addition, la soustraction, la multiplication, la division entière et le calcul de l'opposé) ; nous les spécifions comme suit :

```
Type  :  INT(EGER) ;
Sorts :  Int / Bool :
Opns  :  zéro : → Int ;
         succ, pred : Int → Int ;
         le, ge, eq, gt, lt, ne : Int x Int → Bool ;
         opp : Int → Int ;
         add, sub, mul, div : Int x Int → Int ;
Eqns  : x,y ∈ Int ;
        pred (succ(x)) = x ;
        succ (pred(x)) = x ;
```



```
le (zéro, zéro) = true ;
le (x,y) = true ⇒ le (x, succ(y)) = true ;
le (succ(zéro), zéro) = false ;
le (x,y) = false ⇒ le (succ(x),y) = false ;
le (x,y) = le (succ(x), succ(y)) ;
le (x,y) = le (pred(x), pred(y)) ;

ge (x,y) = le (y,x) ;
eq (x,y) = le (x,y) ∧ le (y,x) ;

ne (x,y) = ¬eq (x,y) ;
lt (x,y) = le (x,y) ∧ ne (x,y) ;
gt (x,y) = ge (x,y) ∧ ne (x,y) ;

opp (zéro) = zéro ;
opp (succ(y)) = pred (opp(y)) ;
opp (pred(y)) = succ (opp(y)) ;

add (x, zéro) = x ;
add (x, succ(y)) = succ (add(x,y)) ;
add (x, pred(y)) = pred (add(x,y)) ;

sub (x,y) = add (x, opp(y)) ;

mul (x, zéro) = zéro ;
mul (x, succ(y)) = add (mult(x,y), x) ;
mul (x, pred(y)) = sub (mult(x,y), x) ;

eq (x, zéro) = false ⇒ div (x,x) = succ (zéro) ;
div (x, succ (zéro)) = x ;
eq (y, zéro) = false ⇒ div (zéro,y) = zéro ;
eq (y, zéro) = false ⇒ div (opp(x), y) = opp (div(x,y)) ;
eq (y, zéro) = false ⇒ div (x, opp(y)) = opp (div(x,y)) ;
eq (y, zéro) = false ⇒ div (opp(x), opp(y)) = div(x,y) ;
Fin ;
```

Remarques :

i) Pour que cette spécification réalise la description informelle donnée plus haut, il nous faut la compléter par la définition de l'exception "zéro-divide". Nous ne le ferons pas : toutes les spécifications que nous présentons peuvent être partielles (i.e. certaines opérations ne seront pas décrites totalement).

ii) A partir de ce type, il est possible aussi de définir le type des "entiers bornés" par un minimum et un maximum et donc de prendre en charge l'exception "overflow" pouvant être déclenchée par l'addition, la soustraction ou la multiplication.

iii) Pour compléter les spécifications ci-dessus, nous regroupons sous le type VALUE les objets manipulés par le langage. Nous le spécifions par :

```
Type  : VALUE ;
Sorts : Value / Int, Bool ;
Opns  : intvalue : Int → Value ;
        boolvalue : Bool → Value ;
Fin ;
```

iv) Pour être tout à fait rigoureux, il serait peut être nécessaire à ce niveau d'associer à la syntaxe des constantes une sémantique faisant correspondre à chaque texte de constante un terme du type correspondant, cela pourrait être réalisé de la manière suivante :

```
V : constantes booléennes + constantes entières →
                        termes de Value

V [[ TRUE ]]  = true
V [[ FALSE ]] = false
V [[ 0 ]]     = zéro
V [[ 1 ]]     = succ (zéro)
V [[ 2 ]]     = add (V [[1 ]], V [[ 1]])
:
```



```
V [[9]]  = add (V [[8 ]], V [[1 ]])
V [[<nombre> <chiffre>]] = add (V [[<chiffre>]],
                                mul (V [[<nombre>]], add (V [[9 ]],
                                V [[1 ]])))
```

v) Pour des raisons de lisibilité on s'autorise à utiliser les opérations des types en mode infixé, à les abréger mnémoniquement et à les surcharger tant que la compréhension reste sans ambiguïté.

vi) Dans la suite nous confondons, tant que la compréhension reste sans ambiguïté, intvalue(x) et x de même que boolvalue(b) et b, etc ..

1.1.3.2. Les expressions

Les sortes de base étant introduites, nous allons spécifier successivement les sortes des identificateurs, des variables et des expressions.

Pour celà, nous allons considérer comme prédéfinies la sorte Text des "textes des terminaux" et l'opération d'égalité entre les textes : egtxt : Text x Text → Bool.

Les spécifications suivantes concernent la description de la sorte des identificateurs, où on distingue la sorte des identificateurs de variables booléennes et entières de celle des identificateurs des programmes.

```
Type  : IDVAR ;
Sorts : Idvar / Text, Bool ;
Opns  : id : Text x Text → Idvar ;
        typid : Idvar → Text ;
        desid : Idvar → Text ;
        egid : Idvar x Idvar → Bool ;
Eqns  : i, t ∈ Text ; x, y ∈ Idvar ;
        typid (id(i,t)) = t ;
        desid (id(i,t)) = i ;
        egid (x,y) = egtxt (desid(x), desid(y))
                   ∧ egtxt (typid(x), typid(y)) ;
Fin ;
```

```
Type  : IDPROG ;
Sorts : Idprog / Text ;
Opns  : prog : Text → Idprog ;
Fin ;

Type  : IDENT ;
Sorts : Ident / Idvar, Idprog ;
Opns  : idvarident: Idvar → Ident ;
        idprogident : Idprog → Ident ;
        père : Idvar → Idprog ;
Fin ;
```

Nous pouvons maintenant présenter le type des variables en se donnant les relations entre les identificateurs et les variables ainsi que les relations entre les valeurs et les variables.

```
Type  : VAR(IABLE) ;
Sorts : Var / Idvar, Idprog, Value, Bool ;
Opns  :
     desvar : Idvar x Idprog → Var ;
     valvar : Var → Value ;
     egvar  : Var x Var → Bool ;
Eqns  :
     egvar (v1,v2) = egvar (v2,v1) ;
     egvar (v1,v2) ∧ egvar (v2,v3) = true ⇒ egvar (v1,v3) = true ;
     egvar (v1,v2) = true ⇒ eq (valvar(v1), valvar(v2)) = true
Fin ;
```

A ce niveau tous les éléments pour décrire la sorte des expressions sont spécifiés ; nous la donnons par une définition d'un type union des expressions booléennes et des expressions entières.


```
Type  : BEXP ;
Sorts : Bexp / Var, Bool ;
Opns  :
        boolbexp : Bool → Bexp ;
        varbexp : Var → Bexp ;
        ¬ : Bexp → Bexp ;
        ∧, ∨ : Bexp x Bexp → Bexp ;
Fin ;

Type  : IEXP ;
Sorts : Iexp / Var, Int, Bexp ;
Opns  :
        intiexp : Int → Iexp ;
        variexp : Var → Iexp ;
        add, sub, mul, div : Iexp x Iexp → Iexp ;
        opp : Iexp → Iexp ;
        le, ge, ne, lt, gt, eq : Iexp x Iexp → Bexp ;
Fin ;

Type  : EXP(RESSIONS) ;
Sorts : Exp / Iexp, Bexp ;
Opns  : iexpexp : Iexp → Exp ;
        bexpexp : Bexp → Exp ;
Fin ;
```

La sémantique des expressions de LPS déterministe peut être alors aisément définie par les équations suivantes, sous l'hypothèse du renommage des opérations héritées des types composants (nous les avions différenciées par une écriture soulignée pour les opérations de Exp, et écriture en caractère normal pour les opérations de Value) :

$\mathcal{E}$: identificateur x (expressions + termes + facteurs + primaires + variables + constantes) → termes de Exp

$\mathcal{E}_p$ [[<termebool> OR <termebool>]] = $\underset{\sim}{\vee}$ ($\mathcal{E}_p$ [[<termebool>]], $\mathcal{E}_p$[[<termebool>]]

$\mathcal{E}_p$ [[<factbool> AND <factbool>]] = $\underset{\sim}{\wedge}$ ($\mathcal{E}_p$ [[<factbool>]], $\mathcal{E}_p$[[<factbool>]])

$\mathcal{E}_p$ [[<expsimple> = <expsimple>]] = $\underset{\sim}{eq}$ ($\mathcal{E}_p$ [[<expsimple>]], $\mathcal{E}_p$[[<expsimple>]]

⋮

$\mathcal{E}_p$ [[<expsimple> <= <expsimple>]] = $\underset{\sim}{le}$ ($\mathcal{E}_p$ [[<expsimple>]], $\mathcal{E}_p$[[<expsimple>]]

$\mathcal{E}_p$ [[- <terme>]] = $\underset{\sim}{opp}$ ($\mathcal{E}_p$ [[<terme>]])

$\mathcal{E}_p$ [[<terme> + <terme>]] = $\underset{\sim}{add}$ ($\mathcal{E}_p$ [[<terme>]], $\mathcal{E}_p$ [[<terme>]])

$\mathcal{E}_p$ [[<terme> - <terme>]] = $\underset{\sim}{sub}$ ($\mathcal{E}_p$ [[<terme>]], $\mathcal{E}_p$ [[<terme>]])

$\mathcal{E}_p$ [[<facteur> * <facteur>]] = $\underset{\sim}{mul}$ ($\mathcal{E}_p$ [[<facteur>]], $\mathcal{E}_p$ [[<facteur>]])

$\mathcal{E}_p$ [[<facteur> / <facteur>]] = $\underset{\sim}{div}$ ($\mathcal{E}_p$ [[<facteur>]], $\mathcal{E}_p$ [[<facteur>]])

$\mathcal{E}_p$ [[NOT <primaire>]] = $\underset{\sim}{\neg}$ ($\mathcal{E}_p$ [[<primaire>]])

$\mathcal{E}_p$ [[(<expression>)]] = $\mathcal{E}_p$ [[<expression>]]

$\mathcal{E}_p$ [[<constante>]] = $\mathcal{V}_p$ [[<constante>]]

$\mathcal{E}_p$ [[<variable>]] = desvar (<variable>, P)

1.1.3.3. Les phrases

Nous allons décrire la signification des phrases de LPS déterministe en définissant le type des "modifications" : la valeur sémantique de toute instruction ou déclaration (ou composition d'instructions ou composition de déclarations) est un terme de ce type.

Voyons tout d'abord la spécification des opérations relatives aux instructions :


```
Type  : INST(RUCTION) ;
Sorts : Inst / Var, Exp ;
Opns  : affect : Var x Exp → Inst ;  % affectation %
        cond : Bexp x Inst x Inst → Inst ;   % conditionnelle %
        iter : Bexp x Inst → Inst ;   % itération tant que %
        nop : → Inst ;   % instruction vide %
Eqns  : b ∈ Bexp ; m1, m2 ∈ Inst ;
        cond (b, m1, m2) = cond (¬(b), m2, m1)
Fin ;
```

La spécification de l'opération relative aux déclarations peut être décrite comme suit :

```
Type  : DECL(ARATION) ;
Sorts : Decl / Idvar ;
Opns  : declvar : Idvar x Idprog → Decl ;
Fin ;
```

La spécification des modifications se décrit alors aisément par :

```
Type  : MODIF(ICATION) ;
Sorts : Modif / Inst, Decl ;
Opns  : instmodif : Inst → Modif ;
        declmodif : Decl → Modif ;
        seq : Modif x Modif → Modif ;
Eqns  : m, m1, m2, m3 ∈ Modif ;
        seq (seq(m1, m2), m3) = seq (m1, seq(m2, m3)) ;
        seq (cond(b, m1, m2), m3) = cond (b, seq (m1, m3), seq (m2, m3)) ;
        iter (b, m) = cond (b, seq (m, iter (b, m)), nop) ;
Fin ;
```

A ce niveau, on peut décrire aisément l'association des phrases du langage aux modifications :

$\mathcal{S}$: programme + identificateur + instructions + déclarations → termes de Modif

$\mathcal{M}_p$: identificateur x (déclarations + instructions) → termes de Modif

$\mathcal{E}$: identificateur x (expressions + termes + facteurs + primaires + constantes + variables) → termes de Exp

$\mathcal{S}$ [[PROGRAM <identif> ; <déclarations> BEGIN <instructions> END.]] =

seq ($\mathcal{M}_{p}$ [[<déclarations>]], $\mathcal{M}_{p}$ [[<instructions>]])

$\mathcal{M}_p$ [[<declarat> ; <déclarations>]] =
seq ($\mathcal{M}_p$ [[<déclarat>]], $\mathcal{M}_p$ [[<déclarations>]])

$\mathcal{M}_p$ [[<identif> : <type>]] = declvar (varid(<identif>, <type>),p)

$\mathcal{M}_p$ [[NOP]] = nop

$\mathcal{M}_p$ [[<instruct> ; <instructions>]] = seq ($\mathcal{M}_p$ [[<instruct>]], $\mathcal{M}_p$ [[<instructions

$\mathcal{M}_p$ [[<variable> := <expression>]] = affect ($\mathcal{E}_p$ [[<variable>]], $\mathcal{E}_p$ [[<express

$\mathcal{M}_p$ [[IF <expression> THEN <instructions1> ELSE <instructions2> FI]] =

cond ($\mathcal{E}_p$ [[<expression>]], $\mathcal{M}_p$ [[<instructions1>]], $\mathcal{M}_p$ [[<instructions2>]])

$\mathcal{M}_p$ [[WHILE <expression> DO <instructions> OD]] =

iter ($\mathcal{E}_p$ [[<expression>]], $\mathcal{M}_p$ [[<instructions>]])

1.1.3.4. Sémantique

A ce niveau de description, la caractérisation des phrases de LPS déterministe n'est pas totalement faite. Pour l'exprimer, nous avons besoin d'introduire les états (d'un certain interprète abstrait) dans lequel nous pouvons évaluer les expressions et interpréter les phrases. Nous générons la sorte State des états de cet interprète par :

Type : STATE
Sorts : State / Ident, Value ;
Opns : init : → state ;
substid : State x Ident x Ident → State ;
substval : State x Value x Value → State ;
Fin ;

Pour décrire les évaluations d'expressions et les interprétations des modifications, nous sommes amenés à introduire :

- d'une part une opération eval : Exp x State → Value reliant sortes d'expressions et sortes valeurs ; les axiomes donnés pour cette opération sont les suivants :

Opns : eval : Exp x State → Value ;
Eqns : σ ∈ State ; i, j ∈ Exp ;
eval (opp(i), σ) = opp (eval(i, σ)) ;
eval (add(i,j), σ) = add (eval(i, σ), eval(j, σ)) ;
eval (sub(i,j), σ) = sub (eval(i, σ), eval(j, σ)) ;
eval (mul(i,j), σ) = mul (eval(i, σ), eval(j, σ)) ;
eval (div(i,j), σ) = div (eval(i, σ), eval(j, σ)) ;
eval (¬i, σ) = ¬(eval(i, σ)) ;
eval (∧ (i,j), σ) = eval (i, σ) ∧ eval(j, σ) ;
eval (∨ (i,j), σ) = eval (i, σ) ∨ eval(j, σ) ;
eval (eq(i,j), σ) = eq (eval(i, σ), eval(j, σ)) ;
eval (ne(i,j), σ) = ne (eval(i, σ), eval(j, σ)) ;
eval (lt(i,j), σ) = lt (eval(i, σ), eval(j, σ)) ;
eval (le(i,j), σ) = le (eval(i, σ), eval(j, σ)) ;
eval (gt(i,j), σ) = gt (eval(i, σ), eval(j, σ)) ;
eval (ge(i,j), σ) = ge (eval(i, σ), eval(j, σ)) ;
Fin ;

- d'autre part, une opération apply : Modif x State → State reliant modifications et états

Opns : apply : Modif x State → State ;
Eqns : σ ∈ State ; v ∈ Var ; vid ∈ Idvar ; p ∈ Idprog ;
e ∈ Exp ; b ∈ Bexp ; m1, m2 ∈ Modif ;

apply(declvar(vid,p),σ) = substid(σ, père(vid), p) ;
apply(affect(v,e), σ) = substval(σ, valvar(v), eval(e, σ)) ;
apply(nop, σ) = σ ;
apply(seq(m1, m2), σ) = apply(m2, apply(m1,σ)) ;
apply(cond(b, m1, m2), σ) =
si eval (b, σ)
alors apply (m1, σ)
sinon apply (m2, σ)
fsi ;
Fin ;

1.1.4. Etude du type associé à LPS déterministe

Nous allons étudier les propriétés du type formalisant LPS déterministe, notamment celles relatives à la correction, la complétude suffisante et la consistance.

Pour démontrer la consistance du type, l'ensemble de ses équations doit être tel qu'il n'y ait pas d'axiomes contradictoires. Autrement dit, on ne peut démontrer dans ce type une égalité entre termes primitifs qui n'est pas vérifiée dans l'algèbre de base. La preuve de la consistance peut se caractériser syntaxiquement en orientant les équations de gauche à droite et en vérifiant que le système de règle de réécriture ainsi obtenu possède la propriété de confluence : si on applique successivement une suite de règles de réécriture pour réduire une formule jusqu'à obtenir un terme irréductible, l'ordre d'application des règles de réécriture n'a aucune importance. Un outil s'articulant autour de l'algorithme de Knuth-Bendix peut, sous l'hypothèse de la terminaison finie, préciser si un ensemble d'équations possède cette propriété.

Nous supposerons que la présentation du type formalisant LPS déterministe est consistante, puisqu'apparamment chaque équation caractérise des objets différents, et qu'elle admet au moins un modèle.

Pour que le type soit suffisamment complet, l'ensemble de ses équations doit être tel qu'il ne manque pas d'axiomes. Autrement dit, on peut ramener par démonstration tout terme de base (i.e. de profil une sorte primitive) à un terme primitif. Ce qui s'exprime en termes de modèle : tous les modèles de la spécification sont isomorphes. Or vue la partialité de la définition de l'opération "eval", le type de LPS déterministe ne peut qu'être non suffisamment complet(*) :

il n'existe pas de valeur v de la sorte Value tel que par exemple :

eval (x, apply (seq (affect (x,'1'),
iter (gt (x, '0', affect (x, succ(x))))))),σ) = v

Remarquons toutefois que si nous oublions dans le type les descriptions relatives à l'opération d'interprétation "apply", pour tout terme e de Exp sans eval il existe un terme v de Value, ne contenant pas d'eval, tel que eval (e, σ) = v en supposant complète la définition du constructeur "init". Ainsi le type formalisant LPS déterministe n'est que partiellement complet : pour tout terme t défini de sorte primitive il existe un terme primitif p tel que par le biais des équations du type on puisse démontrer p = t.

Le type formalisant LPS déterministe possède entre autres les propriétés(**) suivantes :

i) Les modèles de ce type sont celles de ses algèbres qui vérifient les équations, parmi elles, nous avons les trois modèles suivants, où les images des termes formels dans les modèles sont confondues avec les termes eux-mêmes, et subst : State x Var x Value → State l'opération de substitution généralisée sur les états.

(*) Ici le problème de la non complétude suffisante est fortement lié à celui de la terminaison (cf. propriété v) relative à la caractérisation des états.

(**) Pour une étude approfondie des propriétés des types formalisant les langages de programmation, se conférer par exemple à [PAI 82], [BRO 80 a-b].

m1 : substitution sans effacement en respectant la chronologie
subst (subst (σ,v,x),w,y) ≠ subst (subst (σ,w,y),v,x)
subst (subst (σ,v,x), v, z) ≠ subst (σ,v,z)

m2 : substitution avec effacement
subst (subst (subst (σ,v,x),w,y),v,z) =
subst (subst (σ,v,z),w,y)

m3 : substitution sans effacement sans respect de la chronologie
subst (subst (subst (σ,v,x),w,y),v,z) =
subst (subst (subst (σ,v,z),w,y),v,x)

ii) Sous ces hypothèses, l'égalité entre objets de STATE n'est pas observable. En effet, l'indépendance de la chronologie et la possibilité d'effacer n'est pas prouvable :

∀ v, w ∈ Var ; ∀ x, y ∈ Value ;

l'égalité des termes subst (subst (init,v,x),w,y) et subst (subst (init,w,y),v,x) est non prouvable ainsi que celle entre les termes subst (subst (init,v,x),v,y) et subst (init,v,y).

Par contre, on peut la prouver, en se basant sur l'opération d'observation "eval". En effet, par induction structurelle, on a :

∀ x, y ∈ Value ; ∀ v, w ∈ Var ; ∀ e ∈ Exp ;

eval (e,subst (subst (init,v,x),w,y)) =
eval (e,subst (subst (init,w,y),v,x))
eval (e,subst (subst (init,v,x),v,y)) =
eval (e,subst (init,v,y))

iii) Parmi les modèles décrits en i)

. m1 représente le modèle initial du type : il est déterminé par l'équation de STATE : m1 ⊨ σ1 = σ2 ssi STATE ⊢ σ1 = σ2

. m2 représente un modèle terminal du type normal TN (sans l'opération "apply") : il est déterminé par l'opération visible :

m2 ⊨ σ1 = σ2 ssi (∀ σ1, σ2 ∈ State, ∀ e ∈ Exp, ∀ v ∈ Value
TN ⊢ eval (e, σ1) = v ssi TN ⊢ eval (e, σ2) = v)

vi) Le type formalisant LPS déterministe possède un modèle "faiblement" terminal, où deux états sont égaux si on peut démontrer qu'ils sont visiblement équivalents. Ce modèle se construit en étendant la définition de l'opération "apply" par :

appliquer (m,σ) = { σ' si apply(m,σ) = σ' ∈ State prouvable par les équations ; indéfini sinon }

et en interprétant l'équation relative à la composition séquentielle comme suit :

∀ σ, σ1 ∈ State ; ∀ m1, m2 ∈ Modif
apply(m1,σ) = σ1 ⇒ apply (seq (m1,m2), σ) = apply (m2, σ1)

v) Trois catégories de termes de la sorte State peuvent se présenter dans le type :

- ceux qui admettent une forme normale, correspondant à des programmes LPS déterministes qui se terminent,
- ceux qui conduisent à un calcul infini
  . provoquant une infinité de changement d'états et se produisant grâce par exemple à la modification :
  seq (affect (x, succ (zéro)),
  iter (gt (x, zéro), affect (x, succ(x))))
  . ne provoquant pas une infinité de changement d'états et se produisant grâce par exemple à la modification :
  iter (true, nop)

En conclusion, les modèles du type formalisant LPS déterministe sont ceux de ses algèbres qui vérifient les équations. Ce sont des modèles du type sous-jacent qui sont les algèbres du type basé. De plus, comme le type est non suffisamment complet, la restriction aux termes de base de la congruence associée à un modèle doit être strictement plus forte que celle de la congruence syntaxique : elle est obtenue en regroupant toute classe de congruence syntaxique sans termes primitifs avec une classe contenant un terme primitif et pour chacun de ces regroupements possibles on obtient une équivalence sous les termes de base. Cette équivalence est alors la restriction d'une congruence sur l'algèbre des termes si et seulement si elle est une congruence pour les observations de base [PAI 80].

1.2. LPS non déterministe

Le but ici est d'étudier le non déterminisme dans les programmes séquentiels. Dans la littérature (réduite) sur le sujet, l'introduction du non déterminisme qu'il soit envisagé d'un point de vue théorique ou conçu comme aide au développement des programmes, en particulier pour ceux qui énumèrent l'ensemble des solution d'un problème, par exemple en intelligence artificielle [DIJ 75, FLO 67, MAN 70, ...], se présente comme la possibilité fournie au programmeur de ne pas imposer la continuation de certains calculs partiels en indiquant un ensemble des continuations possibles. Ces choix sont syntaxiquement indiqués par une structure de contrôle particulière. Le caractère non déterministe de telles structures de contrôle ne signifie pas le déroulement "au hasard" de l'histoire (calcul) d'un processus.

1.2.1. Description syntaxique de LPS non déterministe

Le caratère non déterministe du langage est donné par une opération binaire de choix, décrite par la règle syntaxique suivante :

<instructions> ::= ALTOR <instruct> □ <instructions> ROTLA

La description syntaxique de LPS non déterministe fait partie de celle donnée en annexe 1. Pour concrétiser cette notion nous donnons ci-dessous un exemple de programme non déterministe :

```
PROGRAM estceraisonnable ;
VAR X,Y,a,b,u,v : integer ;

{Remarques :
 1- Les opérations d'entrée/sortie ne font pas partie du langage.
    Elles sont utilisées dans le but de l'"expression totale" du
    programme.
 2- Le programme fournit un résultat qui est soit le pgcd soit le
    ppcm des nombres en données.
}

BEGIN
   READ (X,Y) ;
   IF (X <= 0  OR  Y <= 0)
   THEN WRITE ('invariant global : X > 0 et Y > 0') ;
   ELSE
       a := X ; b := Y ; u := Y ; v := X ;
       WHILE a <> b  DO
         IF a < b  THEN  b := b - a ; u := u + v ;
                   ELSE  a := a - b ; v := v + u ;
         FI ;
       OD ;
       ALTOR
         WRITE ('pgcd(',X,',',Y,') = ', (a+b)/2) ;
       □
         WRITE ('ppcm(',X,',',Y,') = ', (u+v)/2) ;
       ROTLA ;
   FI ;
END.
```

1.2.2. Sémantique informelle de LPS non déterministe

Pour ne pas nous encombrer de trop de détails inutiles - déjà discutés lors de la formalisation de LPS déterministe - nous nous intéressons ici seulement à l'étude du concept du non déterminisme tel que nous l'avons introduit au paragraphe précédent.
L'opération de choix, notée □, est supposée ici être binaire, comme dans toutes les études théoriques sur le sujet [ARN 78, DEB 76, HEN 76, PLO 76, BRO 81, ...], ce qui n'est pas une restriction puisque cette opération est à la fois commutative et associative et dont la signification intuitive est la sélection indifféremment entre deux instructions ayant le même rôle.

Ce type de choix permet la construction des commandes gardées de [DIJ 75]. Rappelons que pour une commande gardée, si une garde est vraie, alors le groupe d'instructions correspondant est exécuté ; et au cas où aucune condition n'est vraie, on avorte. Le caractère non déterministe d'une telle commande réside dans le choix de la garde vraie dans le cas où plusieurs gardes sont vraies.

1.2.3. Sémantique algébrique de LPS non déterministe

Selon telles hypothèses ou telles autres, différentes formalisations sémantiques du concept de non déterminisme peuvent être définies, analysées et comparées [BRO 81].

Pour notre part,

i) nous avons délaissé la notion de non déterminisme qui exige un choix avant l'exécution d'une alternative ; choix ne pouvant se décrire que par une certaine stratégie conçue à l'avance telle que par exemple le mélange équitable moyennant des oracles décrits dans [BOU 81]. Ce type de non déterminisme convient agréablement aux stratégies de gestion dans les systèmes d'exploitation.

ii) nous avons délaissé également la notion du non déterminisme exigeant un choix après l'exécution des alternatives parmi l'ensemble des valeurs possibles, et qui convient bien à certains problèmes de recherche (algorithmes de "backtracking").

iii) nous avons choisi de décrire le non déterminisme, puisqu'il s'agit de calculer des ensembles ou plus généralement des relations, par l'union des valeurs sémantiques des alternatives. Nous présentons ci-dessous une formalisation basée à la fois sur les états et les modifications.

Pour mener à bien les descriptions formelles de LPS non déterministe nous oublierons tout ce qui est relatif aux déclarations de variables et nous utiliserons la majorité des spécifications algébriques du paragraphe précédent. En particulier nous supposerons définies et de base les types VALUE et EXP, dont les sortes d'intérêts sont :

Value : sorte des valeurs dont Int et Bool,
Exp : sorte des expressions dont Bexp et Iexp.

L'idée intuitive de la formalisation qui suit est d'associer à un programme un ensemble de modifications qui, appliqué à un ensemble d'états, fournit un ensemble d'états.

Partant de l'hypothèse de la définition des sortes de base : Value et Exp, nous allons décrire les sortes de modification et d'état ainsi que les opérations d'évaluation et d'interprétation.

Parmi les modifications, nous distinguons celles relatives aux phrases élémentaires, et celles correspondant aux phrases composées. Nous les spécifions successivement par :

```
Type  : MODELEM ;
Sorts : Modelem / Var, Exp, Bexp ;
Opns  : nop : → Modelem ;
        Bexpmodelem : Bexp → Modelem ;
        affect : Var x Exp → Modelem ;
Fin ;
```

```
Type  : MODCOMP ;
Sorts : Modcomp / Modelem, Bexp ;
Opns  : Modelemmodcomp : Modelem → Modcomp ;
        semi : Modcomp x Modcomp → Modcomp ;
        choix : Modcomp x Modcomp → Modcomp ;
        cond : Bexp x Modcomp x Modcomp → Modcomp ;
        iter : Bexp x Modcomp → Modcomp ;
Eqns  : m, m1, m2, m3 ∈ Modcomp ;
    iter (b,m) = cond (b, semi (m, iter(b,m)), nop) ;
    cond (b, m1, m2) = choix (semi(b, m1), semi (¬̃(b), m2)) ;(*)
    semi (m1, semi (m2, m3)) = semi (semi (m1, m2), m3) ;
    choix (m1, m2) = choix (m2, m1) ;
    choix (m1, choix (m2, m3)) = choix (choix (m1, m2), m3) ;
    semi (m1, choix (m2, m3)) = choix (semi (m1, m2), semi (m1, m3)) ;
    semi (choix (m1, m2), m3) = choix (semi (m1, m3), semi (m2, m3)) ;
Fin ;
```

La description sémantique de LPS non déterministe se définit à ce niveau comme une extension de celle donnée à LPS déterministe par :

$\mathcal{M}_p$ [[<instruct> □ <instructions>]] =
choix ($\mathcal{M}_p$ [[<instruct>] , $\mathcal{M}_p$ [<instructions>]])

(*) Pour une lisibilité des équations, nous oublions, intentionnellement, de manipuler des termes avec Bexpmodelem et Modelemmodcomp.

Nous avons besoin maintenant de décrire ce qu'est un état pour pouvoir donner une spécification aux constructions de LPS non déterministe. Nous l'entreprenons en considérant qu'un état est formé d'un ensemble d'espaces mémoire relatifs aux calculs des alternatives. Un espace mémoire est spécifié par :

```
Type  : MEMORY ;
Sorts : Memory / Var, Value, Bool ;
Opns  : build : → Memory ;
        dead : → Memory ;
        subst : Memory x Var x Value → Memory ;
        egmem : Memory x Memory → Bool ;
Eqns  :
        egmem (m, m) = true ;
        egmem (m1, m2) = egmem (m2, m1) ;
        egmem (m1, m2) ∧ egmen (m2, m3) = true
                                ⇒ egmem (m1, m3) = true
Fin ;
```

Un état peut dans ce cas être spécifié par

```
Type  : STATE ;
Sorts : State / Memory, Bool ;
Opns  : init : → State ;
        make : Memory → State ;
        union : State x State → State ;
        has : Memory x State → Bool ;
        egstate : State x State → Bool ;
Eqns  :  σ, σ1, σ2 ∈ State ; μ ∈ Memory ;
    union (σ, init) = σ ;
    union (σ1, σ2) = union (σ2, σ1) ;
    union (σ1, union(σ2, σ3)) = union (union(σ1, σ2), σ3) ;
    union (σ, σ) = σ ;
```

```
        has (μ, init) = false ;
        has (μ, make(μ)) = true ;
        has (μ, union(σ1, σ2)) = has (μ, σ1) ∨ has (μ, σ2) ;
        has (μ,σ) ∨ egmem (μ, dead) = true
                                ⇒ union (σ, make(μ)) = σ ;
        egstate (σ, σ) = true ;
        egstate (σ1, σ2) = egstate (σ2, σ1) ;
        egstate (σ1, σ2) ∧ egstate (σ2, σ3) = true
                                ⇒ egstate (σ1, σ3) = true ;
    Fin ;
```

Finalement, nous pouvons décrire l'évaluation des expressions et l'interprétation des phrases en se donnant :

```
    Opns : eval : Exp x Memory → Value ;
    Eqns :
          % celles déjà données lors de la description de LPS
            déterministe (voir §  1.1.3.4.) %
    Fin ;

    Opns : do : Medelem x Memory → Memory ;
    Eqns : b ∈ Bexp ; v ∈ Idvar ; e ∈ Exp ; μ ∈ Memory, m ∈ Modelem ;
        do (nop, μ) = μ ;
        egmen (μ, dead) = true ⇒ do (m, μ) = μ ;
        egmem (μ, dead) = false ⇒
             do (affect(v, e), μ) = subst (μ,v,eval(e, μ)) ;
        egmen (μ, dead) = false ⇒
             do (b, μ) = si eval (b, μ)
                         alors μ
                         sinon dead
                         fsi
       % Ce dernier axiome paraît un peu  surprenant : une expression
         booléenne est une modification. Ceci est un choix pour faci-
         liter la description du non déterminisme
       %
    Fin ;
```


L'interprétation des phrases composées se décrit intuitivement par l'ensemble des interprétations des phrases élémentaires composantes ; nous la spécifions par :

```
    Opns : apply : Modcomp x State → State
    Eqns : σ ∈ State ; m, m1, m2 ∈ Modcomp ; b ∈ Bexp ;
         egstate (σ, make(dead)) = true ⇒ apply (m, σ) = σ ;
         egstate (σ, make(dead)) = false ⇒
                  [ apply (b, σ) = run (b, σ) ;
                    apply (nop, σ) = run (nop, σ) ;
                    apply (affect (v,e),σ) = run (affect(v,e), σ) ;
                    apply (semi (m1,M2),σ) = apply (m2, apply(m1,σ)) ;
                    apply (choix (m1,m2),σ) = union (apply(m1, σ),
                                                     apply (m2, σ)) ;
                  ]
    Fin ;
```

avec comme spécification de l'opération "run" :

```
    Opns : run : Modelem x State → State
    Eqns : σ ∈ State ; μ ∈ Memory ; m ∈ Modelem ;
         run (m, make(μ)) = make (do (m, μ))
         run (m, union (σ1, σ2)) = union (run (m, σ1), run (m, σ2))
    Fin ;
```

Remarque :

La spécification de l'opération "apply" peut être raccourcie en se donnant, lors de la description de la sorte Modcomp, l'équation semi (m, nop) = m. Tout terme de la sorte Modcomp peut alors se ramener à la forme canonique :

choix (semi (m1, m2), m3)

où m1 est un terme de la sorte Modelem. La démonstration de cette forme canonique se fait par induction structurelle sur les termes de Modcomp.

1.2.4. Etude de la formalisation de LPS non déterministe

Nous allons donner ci-dessous les propriétés de cette formalisation sous l'hypothèse de la consistance de la spécification de son type :

i) Pour les même raisons explicitées lors de l'étude de LPS déterministe, la présentation du type associé à LPS non déterministe est faiblement suffisamment complète.

ii) Le type présenté possède un modèle initial I déterminé par les égalités de base sur les mémoires et les états :

$\forall \mu 1, \mu 2 \in$ Memory, $\forall \sigma 1, \sigma 2 \in$ State
$I \models \mu 1 = \mu 2$ ssi Memory $\vdash \mu 1 = \mu 2$
et $I \models \sigma 1 = \sigma 2$ ssi State $\vdash \sigma 1 = \sigma 2$

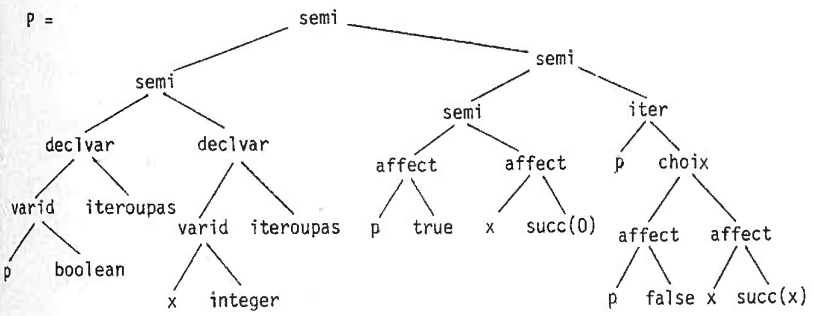
iii) il semble être difficile d'exhiber un modèle terminal T non trivial pour le type présenté si on n'étend pas l'opération d'observation 'eval' sur les états.

Exemple de sémantique de programmes non déterministes

Soit pour exemple le programme jouet suivant :

```
PROGRAM iteroupas ;
VAR p : boolean ; x : integer ;
BEGIN p := true ; x := 1 ;
      WHILE p DO p := false □ x := x + 1 OD ;
END.
```

A ce programme peut être associé l'arbre syntaxique suivant :



L'interprétation de p est la suivante :

apply (p, build) = apply (iter (p, w1), σ1)
où σ1 = subst (subst (subst (subst (build, père(p), iteroupas),
père (x), iteroupas
p, true),
x, succ(zéro))

w1 = choix (affect (p, false), affect (x, add(x, succ (zéro)))

soit m1 = iter (p, w1)
= cond (p, semi (w1, m1), nop)
= choix (semi (p, semi (w1, m1)), semi (¬̃(p), nop))

alors apply (p, build) = union (apply (semi (w1, m1), run (p, σ1)),
apply (semi (¬̃(p), nop), σ1)
= union (apply (semi (w1, m1), σ1),
apply (nop, apply (¬̃(p), σ1)))
= union (apply (semi (w1, m1), σ1), make (dead))
= apply (semi (w1, m1), σ1)
= apply (m1, apply (w1, σ1))
= apply (m1, σ2)

où σ2 = union (apply (affect (p, false), σ1),
apply (affect (x, add (x, succ (zéro)))))

= union (subst (σ 1, p, false),
subst (σ 1, x, succ(x)))

= union (σ 21, σ 22)

σ 21 = subst (subst (subst (subst (build, père (p), iteroupas),
père (x), iteroupas),
p, false),
x, succ (zéro))

σ22 = subst (subst (subst (subst (build, père(p), iteroupas),
père(x), iteroupas),
p, true,
x, succ(succ (zéro)))

⋮

Ainsi, par interprétation on arrive à la forme de récursion calculant $x \in \{1,2,3, \ldots\}$.

2. DESCRIPTION DE LPP

2.1. Introduction

Le terme "parallélisme" a une signification très générale et peut recouvrir bien des aspects de l'informatique : les systèmes d'exploitation, les bases de données réparties, la simulation, les réseaux téléinformatique, la conduite de procédés industriels... Dans chacun de ces aspects, c'est l'agencement des principaux composants et leurs relations qui est la première préoccupation.

Le parallélisme, qui, à notre avis, regroupe les notions de coopération, de concurrence, de communication, de synchronisation, de contrôle distribué, et de contrôle en temps réel est un des principes de base de l'organisation d'un système informatique pour améliorer sa qualité et optimiser ses performances.

La décomposition d'un système en sous-systèmes évoluant en parallèle et interagissant par transformation d'objets communs, peut alors paraître, non seulement comme une commodité, mais aussi comme une étape nécessaire à la démarche de la conception. L'interaction entre les différents sous-systèmes traduit alors leur coordination pour réaliser le comportement global du système dont ils font partie. Deux aspects fondamentaux sont alors à prendre en compte : la concurrence (pour l'acquisition des ressources nécessaires à leurs exécutions) et la coopération (l'utilisation par une action du résultat d'une autre).

Un système parallèle étant conçu comme un ensemble de sous-systèmes évoluant simultanément, il n'est pas réaliste, pour des raisons technologiques, logiques et méthodologiques de faire de quelques quelconques hypothèses sur l'existence de relations entre les durées des actions dans chacun de ces sous-systèmes.

Ceci amène à considérer l'asynchronisation et le non déterminisme comme caractéristiques fondamentales des systèmes parallèles. Pourtant, nous verrons plus loin, que, pour des raisons de formalisation, nous sommes conduits à introduire l'aspect temporel relatif à ces systèmes parallèles et à décrire un cadre de manipulation de différentes notions temporelles : dates de début et de fin de l'exécution d'une action, durée d'une action ...

L'évolution des domaines d'application, les progrès technologiques du matériel informatique ainsi que les développements "linguistiques", méthodologiques et théoriques, où intervient le parallélisme, font qu'une étude complète sur la matière devient assez longue et périlleuse à présenter [OUE 82 a, VER 80, ...].

Notre effort dans ce domaine s'est concentré dans les deux directions suivantes :

i) étude et proposition de mécanismes et de méthodes pour la spécification et la programmation du parallélisme [OUE 82, b,c]

ii) sémantique et propriétés formelles des langages de programmation parallèle [FIN 83].

L'intérêt porté à ces thèmes vient du fait qu'il apparaît difficile de décrire formellement le parallélisme et de vérifier ses propriétés invariantes et temporelles telles que le non interblocage, la non famine, la correction partielle et totale, la vivacité, l'atteignabilité, l'exclusion mutuelle, ...

En fait, les difficultés surgissent dès le moment où l'on veut décrire (formellement) la composition parallèle de plusieurs actions (soumises à des conditions de synchronisation qui imposent des conditions sur l'ordre dans lequel elles peuvent être réalisées). Ce qui n'est évidemment pas le cas de la composition séquentielle, puisque en termes mathématiques, elle s'exprime à l'aide de la composition habituelle des fonctions [APT 81] :

Pour $f : X \to \mathcal{P}(Y_\perp)$ et $g : Y \to \mathcal{P}(Z_\perp)$ où X,Y,Z sont des ensembles non vides, et $\mathcal{P}(Y_\perp)$ et $\mathcal{P}(Z_\perp)$ sont les ensembles des parties des ensembles $Y_\perp$ et $Z_\perp$ ($Y_\perp = Y \cup \{\perp\}$ et $Z_\perp = Z \cup \{\perp\}$)
on définit $f ; g = g^+ \circ f$ où $g^+ : \mathcal{P}(Y_\perp) \to \mathcal{P}(Z_\perp)$ est une fonction monotone dont la définition dépend étroitement de la définition de la fonction g et des domaines $\mathcal{P}(Y_\perp)$ et $\mathcal{P}(Z_\perp)$.

Pour décrire la composition parallèle de plusieurs actions, les solutions adoptées par plusieurs auteurs suggèrent de présenter le parallélisme par le non déterminisme dans le choix de l'action suivante. Ce type de description amène en général à :

i) calculer le non déterminisme indifféremment par "interprétation et choix" ou "choix et interprétation". Et toute la problématique réside dans la définition du choix qui doit être équitable de manière à ne pas privilégier indéfiniment une action au détriment des autres. Ceci mène alors, à fournir des opérateurs de mélange équitable [BOU 81].

ii) fournir une technique de preuve consistant à obtenir un programme non déterministe dont le comportement est, sous certaines hypothèses, "équivalent" au comportement du système parallèle considéré. Ainsi les propriétés de ce système se reflètent dans le programme non déterministe et peuvent être étudiées indirectement à l'aide de celui-ci. Par exemple, pour prouver qu'un système parallèle a un blocage total, il suffit de montrer que son équivalent séquentiel peut se terminer dans une situation ne correspondant pas à la terminaison dans aucun composant [GUE 81].

Cette démarche ainsi que d'autres [LEV 81, APT 80, ...] sont assez avantageuses car les techniques de preuve de programmes séquentiels sont assez classiques et leurs résultats fort bien établis ; néanmoins, elles ne s'attaquent pas directement aux problèmes du parallélisme.

C'est dans cet esprit que d'autres solutions ont vu le jour, suggérant de décrire et d'étudier la composition parallèle par des méthodes algébriques [MIL 80, DAR 80, BOU 83, ...].
Un processus p y est conçu abstraitement comme un objet qui accomplit certaines actions instantanées a et ce faisant, se reconfigure en un autre processus p', relation décrite par $p \xrightarrow{a} p'$. Les transitions initiales possibles d'un processus (ou plus exactement un agent) sont décrites par induction structurelle et la composition parallèle de deux agents peut se traduire par :

$(p \;//\; q) \xrightarrow{c} r$ ssi (i) $\exists p' : p \xrightarrow{c} p'$ et $r = (p' \;//\; q)$ ou bien
(ii) $\exists q' : q \xrightarrow{c} q'$ et $r = (p \;//\; q')$ ou bien
(iii) $\exists a, b \;;\; \exists p', q' : p \xrightarrow{a} p' \;,\; q \xrightarrow{b} q'$
$c = a.b$ et $r = (p' \;//\; q')$

où a.b est le produit de la co-occurence temporelle des actions a et b.

C'est-à-dire que les première actions de p // q sont celles de p ou q ou une action résultant de l'activité simultanée de p et q.

Cette démarche, ainsi que celles qui traitement les systèmes parallèles par le langage de comportement [ARN 81, ...] , présentent l'avantage de fournir un certain calcul qui permet de traiter commodément la composition parallèle et les problèmes qui lui sont reliés.

D'autres auteurs [LAM 83, PNU 79, MER 83, ...] se sont intéressés au problème en raisonnant sur les relations cause/effet et temporelle lors d'une composition parallèle de plusieurs actions. Ces travaux ont donné naissance à des systèmes de logique temporelle où les notions relatives au temps sont explicites et permettent de mener certains raisonnements à propos des programmes parallèles et leur correction (particulièrement en vérifiant les propriétés de propreté, safety en anglais, et de vivacité, liveness en anglais). Dans ce type de logique, les raisonnements

sont décrits par des assertions construites à partir de prédicats sur les états, des prédicats sur les actions, des opérateurs usuels de la logique et un ou plusieurs opérateurs temporels issus de la logique modale. Ce type de logique permet d'affirmer des choses du genre : si l'état présent possède telles propriétés et si on lui applique telles actions possédant telles propriétés, on atteindra alors (fatalement un jour) un état possédant telles autres propriétés.

Ce type de démarche présente l'avantage de fournir un cadre mathématique élégant (issu de la logique) pour décrire le parallélisme mais surtout de raisonner à propos des propriétés des programmes.

En survolant ce problème de composition parallèle, on s'aperçoit donc qu'il est possible de l'étudier de trois manières différentes :

i) en appliquant certaines extensions aux techniques issues des études sur la composition séquentielle,

ii) en décrivant l'ensemble des résultats possibles d'une composition parallèle dans un certain cadre comportemental formellement défini,

iii) en décrivant les aspcets temporels d'une composition parallèle. C'est à ce niveau que l'on peut distinguer différents modes de description (de ses aspects) selon notre compréhension de ce que l'on appelle communément "l'état suivant" (*) :

- l'"application simultanée" des actions où seules les dates minimales de la réalisation des actions est intéressante. Cette méthode, qui va faire l'objet de l'étude menée au § 2.4 se propose de construire l'état suivant à partir de l'état courant et des actions à lui imposer simultanément, ce qui permet d'affirmer par exemple que l'agent ayant atteint son action de communication se voit obligé, dans le cas d'une communication par rendez-vous,

(*) Lamport dit à ce propos : "When one talks about the next state, one really means the next state in which a significant change occurs - where significant means visible at the level of detail of the specification".

d'attendre son interlocuteur jusqu'au moment où il sera, lui aussi, prêt à communiquer.

- l'"application synchronisée" des actions où seules les dates les plus intéressantes sont celles qui correspondent à un évènement de synchronisation des activations futures du système. Cette méthode décrit les relations fonctionnelles et temporelles entre les évènements cruciaux du système.

La méthode préconisée, dans ce chapitre, pour décrire la composition parallèle de plusieurs actions est celle qui consiste à la retracer telle qu'elle est observable en termes de traces temporelles des actions de plus courte durée. Ce type de "spécification temporelle" nous amènera, dans le cadre de la formalisation des concepts de LPP à introduire les notions de dates et de durées (date courante, date d'arrivée d'une information, date de début du traitement d'une action, date de fin de traitement d'une action, ...) et les axiomatiser par un ensemble de spécifications formelles.

2.2. Spécifications temporelles

Plusieurs méthodes de spécifications formelles ont été proposées pour la programmation séquentielle et ont prouvé leur utilité et applicabilité. Un module séquentiel est en général spécifié en termes des valeurs retournées à ses ascendants. Ceci est inadéquat pour la programmation parallèle du fait que plusieurs parties du même programme peuvent s'exécuter en même temps et que le concept, largement connu, d'attente ne peut être exprimé en termes de valeurs retournées.

Dans un programme séquentiel et déterministe, il existe un seul chemin possible d'exécution qui peut atteindre un point de sortie, soit il avorte ou échoue pour un quelconque état intermédiaire. Il peut aussi

bien boucler infiniment. Par contre dans un programme parallèle il existe plus qu'un chemin d'exécution ; chacun d'eux pouvant faire l'une des options ci-dessus citées. Que signifie alors la notion de terminaison correcte ? Devra-t-on exiger qu'au moins un chemin d'exécution termine ? Ou bien que tous les chemins terminent avec une exécution correcte ?
C'est dans ce but que la logique temporelle a été suggérée comme le "moyen" approprié et efficace à la formalisation des programmes parallèles et l'étude de leurs comportements ; un module parallèle étant spécifié en terme de son comportement et non pas en terme des valeurs qu'il retourne.

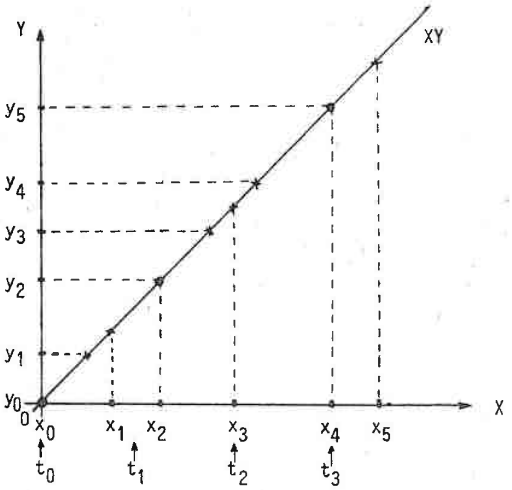
La méthode de spécification que nous allons décrire supporte dans un sens une certaine logique temporelle - nous la désignons par le terme "spécification temporelle" - et permet de spécifier un programme parallèle entier en considérant à la fois les changements d'états du programme et les actions qui causent ce changement.

Supposons que l'on est en présence d'un programme parallèle composé de n agents disjoints deux à deux (i.e. ne se partagent aucune mémoire) coopérant par des échanges d'informations (par le biais d'actions de communications spécifiques) ; et observons à différents instants (les changements de) l'état global de ce programme.
A la date t_o du lancement du programme, l'état global constitué des états relatifs aux agents composants est libre de toute modification ; nous dirons qu'il est parfait. A une date ultérieur t (précédant celle associée à la terminaison du programme), des séquences d'actions ont déjà été exécutées par les agents et il se peut que l'état global soit

i) "parfait" car il est libre de toute tâche ce qui signifie probablement qu'à cette date t, tous les agents viennent de finir l'exécution d'une ou plusieurs actions et qu'ils s'apprêtent à entamer l'exécution d'autres.

ii) "imparfait" car l'un des agents au moins se consacre à cette date à l'exécution de son action courante.

En prenant comme hypothèse que le nombre d'agents du programme considéré est de deux, nous pouvons représenter schématiquement ces observations par la figure suivante :



Composition simultanée d'actions sur un état global d'un système composé de deux agents.

légende :

. sur l'axe O-X sont représentées les actions x_1, x_2, ... de l'agent X celles de l'agent Y sont représentées sur l'axe O-Y ; la bissectrice supporte ainsi les actions communes aux agents composant le programme XY considéré.

. L'état d'un agent en début d'une action appropriée est toujours parfait et est noté par un tiret sur les axes.

. Les t_i représentent les dates d'observations de l'état global de XY : en t_0 et t_3 cet état est parfait (noté par • sur la bissectrice); par contre en t_1 et t_2 il est imparfait (même si l'état relatif à un agent est parfait : cas de t_2).

. Les croix sur la bissectice correspondent à des états imparfaits de l'état global où l'état d'un des deux agents est parfait.

D'après la description ci-dessus, on peut remarquer que :

i) chaque action (locale à un agent) possède une date de début (d'exécution), une date de fin (d'exécution) et une durée ;

ii) a priori n'importe laquelle de ces dates peut être inconnue (a fortiori lorsqu'on est en présence d'actions de communications impliquant l'attente des interlocuteurs) ;

iii) la notion de temps correspond à celle d'un observateur externe au système étudié. Cette notion peut être décrite par un temps linéaire discret dont le domaine de définition peut être représenté par l'ensemble des entiers naturels enrichi par l'infini ;

iv) des relations entre valeurs temporisées peuvent être définies par un ensemble d'axiomes (temporels) ; spécialement :

- un symbole de prédicat binaire "<" décrivant la précédence entre dates et définissant un ordre total sur le domaine du temps ;
- deux symboles de fonctions unaires décrivant la date de début et la date de fin d'un intervalle durée ;

v) à chaque objet "atomique" de l'état du système on peut associer sa valeur et sa date de changement de valeurs ;

vi) les notions temporelles qui précèdent se rapprochent de celles des systèmes de la logique temporelle avec le symbole de prédicat "before".

2.3. LPP : description informelle

2.3.1. Syntaxe de LPP

Le nom LPP est un acronyme pour Langage de Programmation Parallèle. LPP est apparenté à CSP en ce que tous les processus sont disjoints deux à deux (i.e. ne partagent aucune mémoire), tous les processus sont connus statiquement par leur identificateur, tous les processus sont lancés en même temps et en parallèle, tous les processus coopèrent en s'échangeant des informations par le biais d'actions d'envoi et de réception de messages typés (un message à la fois). Par contre, le type de communication diffère de celui de CSP : nous avons choisi dans le cadre de la formalisation de traiter le synchronisme (par rendez-vous) et l'asynchronisme.

La syntaxe de LPP est un sous ensemble de celle de LAGALERE (annexe 1) ; nous l'explicitons par l'exemple de programme suivant :

```
PROGRAM HOFGH1 ;
   PROCESS H0 ;
   VAR  x : integer ;
   BEGIN    x := 0 ;
         WHILE true DO    SEND x TO F ;
                          RECEIVE x FROM G ;
                   OD ;
   END ;
//
```



```
PROCESS H1 ;
VAR   x : integer ;
BEGIN     x := 1 ;
        WHILE true DO   SEND x TO F ;
                        RECEIVE x FROM G ;
                  OD ;
END ;
//
PROCESS F ;
VAR   x : integer  ; b : boolean ;
BEGIN     b := true ;
        WHILE true DO  if b THEN RECEIVE x FROM H0 ;
                            ELSE RECEIVE x FROM H1 ;
                       FI ;
                       SEND x TO G ; b := NOT(b) ;
                  OD ;
END ;
//
PROCESS G ;
VAR   x : integer ; b : boolean ;
BEGIN     b := true ;
        WHILE true DO  RECEIVE x FROM F1 ;
                       IF b THEN SEND x TO H0 ;
                            ELSE SEND x TO H1 ;
                       FI ;
                          b := NOT(b) ;
                  OD ;
END ;
END.
```

2.3.2. Sémantique informelle de LPP

Un programme non séquentiel décrit en LPP est formé d'une déclaration d'un ensemble de processus (statiquement connus, de noms différents et disjoints deux à deux) et d'un corps lançant, par le biais d'une commande parallèle (dénotée par l'opérateur //), l'exécution simultanée des processus qu'il nomme. L'achèvement d'un programme est obtenu lorsque tous les constituants ont terminé leur exécution.

Un processus est un morceau de programme séquentiel et non déterministe, constitué d'une partie déclarative et d'une partie corps où peut apparaître une composition séquentielle ou non déterministe d'actions de : communication, avortement, affectation, conditionnelle et répétition (boucle while). Le caractère séquentiel d'un corps d'un processus est noté par l'absence du non déterminisme explicite, exprimé par l'opérateur de choix □ décrit lors du traitement de LPS non déterministe.

La signification des phrases d'affectation, conditionnelles et répétitives a déjà été donnée lors du traitement de LPS, nous la conservons telle qu'elle ; celle des actions de communication est la suivante.

Les actions d'envoi et de réception (de messages) permettent aux agents d'une part de coopérer en s'échangeant, par désignation explicite (i.e. nommage de l'interlocuteur), des informations typées, et d'autre part de se synchroniser suivant le type de communication établi.

- SEND e TO X dans le corps de l'agent Y, est une "commande" de sortie (autrement dit d'envoi), signifiant que l'agent Y demande à envoyer une expression e à l'agent X,
- RECEIVE v FROM Y dans le corps de l'agent X, est une commande d'entrée (autrement dit de réception), signifiant que l'agent X demande à recevoir une valeur de la part de l'agent Y et à effecter cette valeur entrée à sa variable locale v.

La sémantique de ces actions d'envoi et de réception est complétée pour exprimer le type de communication : synchrone ou asynchrone. On distingue alors

i) un LPP synchrone où le mécanisme du rendez-vous est celui qui décrit les communications synchrones. Chaque fois qu'un agent X désigne dans son corps de programme un agent Y comme destinataire et que complémentairement Y désigne X comme expéditeur, une valeur en sortie est passée du premier agent au second si les désignations occurrent en même temps. Dans les autres cas de figure l'agent ayant atteint son action de communication en premier devra attendre que son interlocuteur atteigne l'action de communication complémentaire.

ii) et, un LPP asynchrone où la modélisation du type de communication est réalisée par des files de messages produits et consommés. Ces files sont gérées par FIFO (ainsi tout message produit peut être consommé une fois et une seule au bout d'un temps fini) et sont supposées être bornées par une limite (un nombre entier positif strictement)(*)(**).
Les agents agissent pratiquement indépendemment les uns des autres. Les consommateurs usent des messages dans leur file de réception des messages de leur interlocuteur tant qu'elle n'est pas vide, sinon attendent jusqu'à ce qu'il y ait un dépôt de messages dans cette file. Les producteurs déposent les messages dans la file de réception de leur interlocuteur si elle n'est pas pleine, sinon attendent jusqu'à ce qu'il y ait retrait de messages dans la file de l'interlocuteur.

(*) L'asynchronisme à une unité (i.e taille des files de messages = 1) est communément appelé synchronisme.

(**) L'asynchronisme total peut être décrit par les files de taille non bornée.

Remarques :

Si les files de réception sont bornées, le mécanisme de l'asynchronisme total n'exige normalement aucune attente de la part des interlocuteurs.

2.4. LPP : description formelle

Par souci de lisibilité et pour ne pas nous perdre dans trop de détails inutiles pour la compréhension des concepts fondamentaux que nous voulons formaliser, nous avons choisi d'alléger la présentation :

i) en oubliant les traitements relatifs aux déclarations d'objets à l'intérieur des processus. (La formalisation des déclarations d'objets est abordée au § 1.1.3.3. lors de la description de LPS déterministe).

ii) en décrivant la formalisation de LPP en deux étapes :

- la première met l'accent uniquement sur la communication entre les agents. Les deux types de communications choisis (par rendez et asynchronisme déterministe) seront décrits au paragraphe LPP déterministe. La formalisation sera décrite pour un système parallèle constitué uniquement de deux agents communiquants.
- la seconde prend en compte le nondéterminisme pouvant se trouver à l'intérieur des corps des agents. Nous le décrirons au paragraphe LPP non déterministe.

2.4.1. LPP déterministe

Dans les descriptions formelles relatives à LPP synchrone et asynchrone, nous utiliserons la majorité des spécifications algébriques présentées

pour LPS déterministe. En particulier, nous supposerons définies et primitives les sortes qui suivent et les opérations qui leurs sont associées.

Id : sorte des identificateurs d'un programme LPP (sa spécification est notamment augmentée de la sorte Idprocess des identificateurs de processus).

Value : sorte des objets sémantiques dont notamment les sortes Id, Int et Bool.

Exp : sorte des expressions d'un programme LPP, incluant en particulier la sorte Var des variables relatives à un agent particulier et la sorte Bexp des expressions booléennes.

Comme sortes non primitives, nous allons spécifier, tout en respectant le principe de construction hiérarchique du type formalisant LPP, les sortes relatives aux phrases du langage et de son interprète abstrait. Pour commencer, spécifions les sortes des modifications associées d'une part à un processus particulier et d'autre part à un programme parallèle.

Le type présenté ci-dessous concerne les modifications relatives à un processus (i.e celles qui correspondent à son corps de programme). Il peut être décrit comme étant une extension de celui présenté pour LPS séquentiel par les opérations et équations relatives à l'avortement et à la communication ; nous le décrivons en entier.

```
Type : MODIF(ICATION) ;
Sorts : Modif / Idprocess, Exp, Var, Bexp ;
Opns : nop, abort : → Modif ;
       affect : Var x Exp → Modif ;
       cond : Bexp x Modif x Modif → Modif ;
       iter : Bexp x Modif → Modif ;
       envoi : Idprocess x Idprocess x Exp → Modif ;
         % le producteur envoie au consommateur une expression %
       recept : Idprocess x Idprocess x Var → Modif ;
```

```
        % le consommateur reçoit du producteur un message
          qu'il affecte à un identificateur de variable locale %
      seq : Modif x Modif → Modif ;
Eqns : m, m1, m2, m3 ∈ Modif ; b ∈ Bexp ;
     seq (seq (m1,m2),m3) = seq (m1, seq (m2,m3)) ;
     iter (b,m) = cond (b, seq (m, iter (b,m)), nop) ;
Fin ;
```

Le type des modifications simultanées concernant l'activation en parallèle d'un exemple d'agents communicants peut être spécifié par :

```
Type : AGENT ;
Sorts : Agent / Idprocess, Modif ;
Opns : declagent : Idprocess x Modif → Agent ;
       parall : Agent x Agent → Agent ;
Eqns : p ∈ Idprocess ; a, a1, a2, a3 ∈ Agent ;
    parall (a1,a2) = parall (a2,a1) ;
    parall (parall (a1,a2),a3) = parall (a1, parall (a2,a3)) ;
    parall (declagent (P,abort),a) = declagent (P, abort) ;
    parall (declagent (P, nop), a) = a
Fin ;
```

La description qui précède est menée sous les hypothèses suivantes :

- l'opération "parall" est commutative et associative,
- la terminaison d'un agent n'implique pas la terminaison des agents qui évoluent en parallèle avec lui,
- l'avortement d'un agent entraîne l'avortement du programme dont il fait partie.

A ce niveau de construction nous avons à notre disposition tout le nécessaire pour présenter le type des programmes LPP :

```
Type : PROGRAM ;
Sorts : Program / Agent, Procid ;
Opns : defprog : Procid x Agent → Program ;
Fin ;
```


L'abstraction syntaxique d'un programme LPP peut ainsi s'écrire aisément par les équations :

$\mathcal{S}$ ⟦PROGRAM <identif> ; <declprocessus> END⟧ =
defprog ('<identif>' , $\mathcal{M}_{<identif>}$ ⟦<declprocessus>⟧) ;

$\mathcal{M}_p$ ⟦ <declprocess> // <declprocessus> ⟧ =
parall ($\mathcal{M}_p$ ⟦<declprocess>⟧ , $\mathcal{M}_p$ ⟦<declprocessus>⟧) ;

$\mathcal{M}_p$ ⟦PROCESS <identif> ; <déclarations> ; BEGIN <instructions> END⟧ =
declagent ('<identif>', seq ($\mathcal{M}_{<identif>}$ ⟦ <declarations>⟧ ,
$\mathcal{M}_{<identif>}$ ⟦ <instructions>⟧)) ;

Remarques :

i) certains $\mathcal{M}$, de la définition qui précède, correspondent à ceux définis dans le cas de LPS déterministe (§ 1.1.3.3).

ii) avec l'aide des équations des types présentés ci-dessus

- tout terme de la sorte Modif peut être transformé en un terme équivalent sous la forme
  seq (x1,x2)
  où x_1, x_2 sont deux modifications telles que x_1 est l'une des opérations nop, abort, cond, envoi et recept.
- tout terme de la sorte Agent est équivalent à un terme de la forme
  parall (declagent(X,x), parall(...,
  parall (declagent(Y,y), declagent (Z,z))...))
  où X,...,Y,Z dénotent des identificateurs de processus et x,...,y,z sont les modifications qui leur correspondent.

Ayant fini le traitement correspondant au niveau langage de la construction hiérarchique du type associer LPP nous allons passer à la description des niveaux supérieurs et ponctuellement au niveau structure de données (de l'interprète abstrait).

Rappelons tout d'abord que les agents ne partagent aucune mémoire et qu'ils ne coopèrent que par le biais d'actions de communication (du même type). Il semble alors être raisonnable de considérer l'état d'un programme LPP comme étant constitué de l'ensemble des états relatifs aux agents qui le composent.

Sur l'univers d'un agent (désormais nous désignerons par "univers", l'état relatif à un agent) ne peuvent opérer que des modifications séquentielles dont l'effet notoire est un changement des valeurs de ses composants locaux.
Ce changement affecte en fait :

i) d'une part, sa composante "mémoire" qui représente l'ensemble de ses objets locaux : substitution des valeurs des objets par d'autres, effet de l'application d'une modification séquentielle sur l'univers,

ii) d'autre part, sa composante "date" qui représente intuitivement son horloge (virtuelle) locale : incrémentation de la valeur de la date par la durée d'exécution de la modification séquentielle dans l'univers.

A ce niveau de description informelle de ce que représente intuitivement pour nous un état et un univers, se pose alors la question de savoir ce qu'est la durée d'exécution d'une action de communication ? Nous dirons, qu'a priori nous n'en savons rien, mais par hypothèse elle sera toujours finie.

Sur l'état d'un programme entier ne peuvent opérer que des modifications simultanées dont l'effet est la prise en compte des changements subis (simultanément) par les univers constituants. L'effet annexe

de ces modifications est la synchronisation, lorsqu'elle est nécessaire, des agents ; effet qui se traduit par la mise en oeuvre du mécanisme d'attente hérité du type de la communication établie.

En définitive, la définition de l'état (d'un programme LPP) requière une spécification des univers (des agents composants), qui elle même exige la description de ses composantes date et mémoire.
Commençons par décrire les dates. Soit Time la sorte des dates sur laquelle on peut opérer par des incrémentations successives à partir d'une date initiale constante. Sa spécification peut être vue comme étant une restriction des entiers à un renommage près de certaines opérations.

```
Type  : TIME ;
Sorts : Time / Bool ;
Opns  : tzéro : → Time ;
        succ : Time → Time ;
        incr : Time x Time → Time ;
        egal, precede, precedegal : Time x Time → Bool ;
Eqns  : t, t1, T2 ∈ Time ;
      incr (t, tzéro) = t
      incr (t1, succ (t2)) = succ (incr (T1,T2)) ;
      precedegal (tzéro, tzéro) = true ;
      precedegal (tzéro, succ(t)) = true ;
      precedegal (succ(t), tzéro) = false ;
      precedegal (succ(t1), succ(t2)) = precedegal (t1, t2) ;
      egal (t1,t2) = precedegal (t1, t2) ∧ precedegal (t2,t1) ;
      precedegal (t1,t2) = precedegal (t1,t2) ∧ ¬ egal (t1,t2)
Fin ;
```

La spécification que nous devons fournir pour décrire les mémoires des agents présente les mêmes caractérisitques opérationnelles (et observationnelles) que celle déjà donnée au § 1.1.3.4, pour décrire les

états (sorte State) d'un programme LPS déterministe. Nous la spécifions ici par la donnée de deux constructeurs : init qui génère la (constante) mémoire initiale et subst qui génère une nouvelle mémoire à partir de la mémoire courante par substitution (généralisée) de certains de ses objets par d'autres. En définissant la sorte Objet par une réunion disjointe des sortes Int, Bool, Value, ..., Modif, la présentation du type MEMORY peut être la suivante :

```
Type  : MEMORY ;
Sorts : Memory / Objet ;
        % Objet = union (Bool, Int, Value, Idvar, Idprog,
          Idprocess, Ident, Var, Bexp, Iexp, Exp, Modif)
        %
Opns  : init : → Memory ;
        subst : Memory x Objet x Objet → Memory ;
Fin ;
```

Occupons nous maintenant de la description des univers. Nous avons signalé plus tôt qu'un univers est composé d'une mémoire et d'une date (représentant une horloge virtuelle locale). En fait, nous avons besoin d'une troisième composante indiquant le fait qu'un univers est "parfait" ou "non parfait".
Nous avons choisi pour cette nouvelle composante une explicitation par la donnée de la modification en cours, avec comme hypothèses :

i) la modification en cours correspond à l'une des actions suivantes : substitution d'objets par d'autres (dans le cas d'une déclaration, appel procédural, affectation), avortement (dans le cas d'un abort) ou pas de changements (dans le cas d'un nop)(*).

ii) si la modification en cours est une substitution, l'univers est imparfait ; par contre si c'est un avortement, l'univers est dit avorté ; sinon il est parfait.

(*) ce sont les seules opérations correspondant à la notion d'action atomique.

Dans la spécification de Universe, nous utilisons, pour l'explicitation des modifications en cours, la sorte Modif. Signalons que, par les équations d'interprétation que nous allons donner plus loin, nous retrouvons tout à fait les besoins exprimés ci-dessus.

```
Type  : UNIVERSE ;
Sorts : Universe / Memory, Time, Modif ;
Opns  : <-,-,-> : Memory x Time x Modif → Universe ;
        mem : Universe → Memory ;
        date : Universe → Time ;
        mod : Universe → Modif ;
Eqns  : u ∈ Universe ; m ∈ Memory ; t ∈ Time ; s ∈ Modif ;
        mem  (<m, t, s>) = m ;
        date (<m, t, s>) = t ;
        mod  (<m, t, s>) = s ;
Fin ;
```

Finalement la spécification de la sorte State des états (des programmes LPP) peut être décrite. Elle est générée à partir des univers des agents composants le programme LPP considéré.

```
Type  : STATE ;
Sorts : State / Universe, Idprocess ;
Opns  : -_ : Universe x Idprocess → State ;
        <-,-> : State x State → State ;
Eqns  : u_x, u_y, u_z ∈ State ;
        <u_x, u_y> = <u_y, u_x> ;
        <<u_x, u_y>, u_z> = <u_x, <u_y, u_z>> ;
Fin ;
```

La description de l'état de l'interprête abstrait des programmes LPP correspond à la spécification qui précède. Nous la complétons ci-après par la donnée des opérations et équations relatives au niveau évaluation et interprétation de la construction hiérarchique du type formalisant LPP.

La fonction d'évaluation à décrire, doit associer à chaque expression, relativement à un état, une valeur donnée. Comme les expressions ne sont formulées qu'à l'intérieur du corps de programme d'un agent, et que ces derniers ne partagent pas de mémoires, il va de soi de décrire cette fonction d'évaluation simplement sur les univers des agents.
Notons par ailleurs, que cette association expressions à valeurs n'a de sens que lorsque l'univers correspondant présente le caractère d'être parfait. Ainsi donc, la fonction d'évaluation ne peut qu'être partiellement définie sur les univers ; sa spécification est la suivante :

Opns : evalue : Exp x Universe → Value ;
Eqns : e ∈ Exp ; u ∈ Universe ;
mod(u) = nop => evalue(e,u) = eval(e,mem(u))
Fin :

où l'opération eval : Exp x Memory → Value
reliant sortes d'expressions à la sorte des valeurs et dont les axiomes sont de la forme suivante :

$eval(f(x1,\ldots,xn), m) = \underset{\sim}{f}(eval(x1, m), \ldots, eval(xn, m))$
si f est une opération de base à n arguments dont $\underset{\sim}{f}$ est son correspondant construit sur la sorte Value.

Finalement nous donnons la description du niveau interprétation de la construction du type formalisant LPP, en définissant :

i) d'une part, la sémantique a priori des agents en les considérant indépendamment du système dont ils font partie. Cette formalisation concernera l'interprétation des modifications séquentielles, autres que les actions de communication, d'un agent dans son univers, et est achevée par le biais de l'opération :
apply : Modif x Universe → Universe

ii) d'autre part, la sémantique dite de liens des agents en les considérant liés deux à deux par l'opération "parall". Rappelons que cette opération est commutative et associative, ce qui nous permet de traiter deux agents à la fois. Cette formalisation concerne donc les modifications simultanées et est achevée par l'opération :
exec : Agent x State → State

2.4.1.1. Sémantique a priori

Dans cette section, nous nous intéressons à la définition des équations relatives à l'interprétation des modifications séquentielles d'un agent dans l'univers qui lui est associé, décrite par l'opération "apply".

Notons tout de suite que

i) cette définition est partielle car aucune interprétation ne sera donnée pour les opérations "envoi" et "recept" correspondant aux actions de communication. (Nous pouvions le faire à ce niveau en introduisant un quantificateur existentiel dans les spécifications algébriques et en se référant à l'interlocuteur désigné par ces opérations, mais nous préférons entreprendre sa définition lors de la description de la sémantique des liens).

ii) le "calcul" d'un agent séquentiel, commençant dans un univers initial parfait η_o = <init, tzéro, nop>, peut
- soit, se terminer dans un univers parfait $\eta_n = \langle \sigma_n, \Theta_n, nop \rangle$ où Θ_n est la date de fin du calcul et σ_n est la mémoire associée,
- Soit, se terminer dans un univers avorté $\eta_a = \langle \sigma_a, \Theta_a, abort \rangle$ où Θ_a est la date d'avortement due à la modification abort ou à un calcul erroné et σ_a la mémoire associée,
- soit, ne pas se terminer à cause d'une modification iter.

La spécification de l'opération d'interprétation apply et donnée ci-dessous par une étude de cas d'une part sur l'univers et d'autre part sur la forme de la modification subite par l'univers.

Opns : apply : Modif x Universe → Universe ;
% décrit partiellement l'interprétation des modifications séquentielles dans un univers associé à un agent %
Eqns : $\sigma \in$ Memory ; θ, δ_{nop}, δ_{eval}, $\delta_{\leftarrow} \in$ Time, $\eta \in$ Universe ;
μ, m, m1, m2 $\in$ Modif ; v $\in$ Var ; e $\in$ Exp ; b $\in$ Bexp ;
$\eta = \langle \sigma, \theta, abort \rangle \Rightarrow apply(m, \eta) = \eta$;
$\eta = \langle \sigma, \theta, nop \rangle \Rightarrow$
apply (m, η) = cas m de
nop alors $\langle \sigma, incr(\theta, \delta_{nop}), nop \rangle$;
abort alors $\langle \sigma, \theta, abort \rangle$;
affect (v,e) alors <subst (σ, valvar (v), evalue (e, η)),
incr (incr (θ, δ_{eval}), $\delta_{\leftarrow}$),
nop> ;
cond (b, m1, m2) alors si evalue (b, η)
alors apply (m1, $\langle \sigma, incr(\theta, \delta_{eval}), nop \rangle$)
sinon apply (m2, $\langle \sigma, incr(\theta, \delta_{eval}), nop \rangle$)
fsi
seq (m1, m2) alors apply (m2, apply (m1, η))
fcas ;
$\eta = \langle \sigma, \theta, \mu \rangle \wedge \neg (\mu = nop) \wedge \neg (\mu = abort) \Rightarrow$
apply (m, η) = apply (m, <memend(η), datend(η), nop>)
Fin ;

Dans la description de l'opération "apply", nous avons utilisé :

i) δ_{nop}, δ_{eval}, $\delta_{\leftarrow}$: trois termes constants de la sorte Time correspondant intuitivement aux durées d'exécution des opérations nop, eval et subst sur un processeur "logique" donné

ii) memend, datend : deux nouvelles opérations augmentant la spécification de Universe.

L'opération "memend" fournit la mémoire correspondant à la fin de l'exécution d'une opération dans un univers donné. Nous la décrivons partiellement par :

Opns : memend : Universe → Memory ;
Eqns : $\sigma \in$ Memory ; $\theta \in$ Time ;
memend ($\langle \sigma, \theta, nop \rangle$) = σ ;
memend ($\langle \sigma, \theta, abort \rangle$) = σ ;
memend ($\langle \sigma, \theta, affect(v,e) \rangle$) =
subst (σ, valvar(v), eval(e,σ))
Fin ;

L'opération "datend" fournit la date de fin d'exécution d'une opération dans un univers donné. Nous la décrivons partiellement par :

Opns : datend : Universe → Time ;
Eqns : $\sigma \in$ Memory ; θ, δ_{eval}, $\delta_{\leftarrow} \in$ Time ;
datend ($\langle \sigma, \theta, nop \rangle$) = θ ;
datend ($\langle \sigma, \theta, abort \rangle$) = θ ;
datend ($\langle \sigma, \theta, affect(v,e) \rangle$) =
incr (incr(θ, δ_{eval}), $\delta_{\leftarrow}$) ;
Fin ;

2.4.1.2. La sémantique des liens

Dans la précédente section, aucune équation n'a été donnée pour décrire l'interprétation des opérations de communication "envoi" et "recept". Ceci est dû aux faits :

i) dans l'approche adoptée, les équations sont uniquement universellement quantifiées,

ii) on est interessé a priori par la description, par le même outil algébrique (les spécifications temporelles), de tout type de communication exprimé dans la littérature par la combinaison des mécanismes synchrone, asynchrone et déterministe, non déterministe [PER 82, MON 81],

iii) certains protocoles de communication (et plus particulièrement le rendez-vous) sont liés fortement au contrôle et donc à l'aptitude des agents à communiquer et ne peuvent pas se décrire sans en faire référence.

Pour décrire une sémantique des liens pour un ensemble d'agents composants un programme LPP, notre choix est porté sur les deux types de communication suivants :

- la communication synchrone et déterministe telle qu'elle est décrite dans CSP : le mécanisme de rendez-vous imposé aux interlocuteurs oblige l'agent ayant atteint une action de communication (i.e apte à communiquer avec l'interlocuteur qu'il désigne explicitement dans son action de communication) d'attendre l'action de communication complémentaire correspondante chez son interlocuteur. Ce mécanisme va être décrit formellement dans le paragraphe LPP synchrone.
- la communication asynchrone et déterministe, où les interlocuteurs sont totalement indépendants mais évoluent en parallèle en respectant un certain mécanisme de synchronisation hérité du fait que les informations échangées transitent par des files de taille limitée par une borne donnée (entier positif strictement, pouvant être l'infini !). Ce type de communication est décrit dans le paragraphe LPP synchrone.

Notations :

Pour alléger l'écriture des équations et augmenter sensiblement leur lisibilité nous avons adopté le système de notation par opérateurs infixes des opérations nécessaires à l'axiomatisation :

incr : Time x Time → Time	θ1 + θ2
precede : Time x Time → Bool	θ1 < θ2
affect : Var x Exp → Modif	v := e
cond : Bexp x Modif x Modif → Modif	if b then m1 else M2 fi
envoi : Idprocess x Idprocess x Exp → Modif	X.Y ! e
recept : Idprocess x Idprocess x Var → Modif	X.Y ? v
seq : Modif x Modif → Modif	m1 ; m2
declagent : Idprocess x Modif → Agent	X :: m
parall : Agent x Agent → Agent	a1 // a2
defprog : Procid x Agent → Program	P = [a]

La description de la sémantique des liens vise le dernier niveau de la construction hiérarchique du type associé à LPP ; en effet elle vient compléter l'axiomatisation donnée pour l'opération "apply" de profil Modif x Universe → Universe, par celle de l'opération "exec" de profil Agent x State → State.

Un calcul d'un système décrit par p agents ($p \geqslant 2$) commence toujours dans un état parfait initial $\Sigma_0 = \langle \eta_{0_1}, \eta_{0_2}, \ldots, \eta_{0_p} \rangle$ où les univers composants sont de la forme

$$\eta_{0_i} = \langle \text{init}, \text{tzéro}, \text{nop} \rangle, \ 1 \leqslant i \leqslant p$$

et peut produire

i) soit, un état final parfait $\Sigma_n = \langle \eta_{n_1}, \eta_{n_2}, \ldots \eta_{n_p} \rangle$ où chaque univers composant est parfait et de la forme

$\eta_{n_i} = \langle \sigma_{n_i}, \theta_{n_i}, \text{nop} \rangle$, $1 \leqslant i \leqslant p$, avec $n = \text{maximum}\ (n_1, n_2, .., n_p)$

où les n_i sont les indices relatifs aux dates de fin (θ_{n_i}) d'exécution des agents i, $1 \leqslant i \leqslant p$.

ii) soit, un état final avorté $\Sigma_a = < \eta_{a_1}, \eta_{a_2}, \ldots, \eta_{a_p} >$ où au moins un des univers est avorté i.e $\exists\, j,\ 1 \leqslant j \leqslant p$ tq $\eta_{a_j} = <\sigma_{a_j}, \theta_{a_j}, \text{abort}>$ où $a = a_j$ et $a_i,\ 1 \leqslant i \leqslant p \wedge i \neq j$, sont les indices des dates θ_{a_j} correspondants à des univers parfaits de i agents.

iii) soit, une infinité d'états où un des univers possède la forme de bouclage se produisant grâce à la modification iter.

Le calcul est décrit par transformation de l'état du système ; transformation qui se réalise par un ensemble de transformations locales subies par les univers constituants.

Remarque :

L'opération "parall" étant commutative et associative la description du calcul correspondant à "exec" peut être donnée uniquement en raisonnant avec deux agents parallèles.

Axiomatisation

(1) $\eta_X = <\sigma_X, \theta_X, \text{abort}> \ \vee\ \eta_Y = <\sigma_Y, \theta_Y, \text{abort}> \Rightarrow$

$[\text{exec}\ (X :: x_1 ; x_2 \ //\ Y :: y_1 ; y_2, <\eta_X, \eta_Y>) = <\eta_X, \eta_Y> ;]$

(2) $\eta_X = <\sigma_X, \theta_X, \text{nop}> \wedge \eta_Y = <\sigma_Y, \theta_Y, \text{nop}> \Rightarrow$

$[x_1 = \text{if b then } x_{11} \text{ else } x_{12} \text{ fi} \Rightarrow$

$[\text{exec}\ (X::x_1 ; x_2 \ //\ Y::y_1 ; y_2, <\eta_X, \eta_Y>) =$

si evalue (b, η_X)

alors exec $(X::x_{11} ; x_2 \ //\ Y::y_1 ; y_2, <<\sigma_X, \theta_X + \delta_{X_{eval}}, \text{nop}>, \eta_Y>)$

sinon exec $(X::x_{12} ; x_2 \ //\ Y::y_1 ; y_2, <<\sigma_X, \theta_X + \delta_{X_{eval}}, \text{nop}>, \eta_Y>)$

fsi ;

]

$(x_1 = \text{abort} \vee x_1 = \text{nop} \vee x_1 = vx := ex) \wedge (y_1 = \text{abort} \vee y_1 = \text{nop} \vee y_1 = vy := ey) \Rightarrow$

$[\text{exec}\ (X::x_1 ; x_2 \ //\ Y::y_1 ; y_2, <\eta_X, \eta_Y>) =$

si datend (apply (x_1, η_X)) + θ_Y

$<$ datend (apply (y_1, η_Y)) + θ_X

alors exec $(X::x_2 \ //\ Y::y_2, <\text{apply}(x_1, \eta_X), <\sigma_Y, \theta_Y, y_1>>)$

sinsi datend (apply (y_1, η_Y)) + θ_X

$<$ datend (apply (x_1, η_X)) + θ_Y

alors exec $(X::x_2 \ //\ Y::y_2, <<\sigma_X, \theta_X, x_1), \text{apply}\ (y_1, \eta_Y)>)$

sinon exec $(X::x_2 \ //\ Y::y_2, <\text{apply}\ (x_1, \eta_X), \text{apply}\ (y_1, \eta_Y)>)$

fsi ;

]

]

(3) $\eta_X = \langle \sigma_X, \theta_X, nop \rangle \wedge \eta_Y = \langle \sigma_Y, \theta_Y, \mu \rangle$

$\wedge \neg(\mu = nop) \wedge \neg(\mu = abort) \Rightarrow$

$[x_1 = \text{if } b \text{ then } x_{11} \text{ else } x_{12} \text{ fi} \Rightarrow$

$[exec\ (X::x_1 ; x_2 \ //\ Y::y_1 ; y_2, \langle \eta_X, \eta_Y \rangle) =$

si evalue (b, η_X)

alors si $\delta_{X_{eval}} + \theta_Y < datend(\eta_Y)$

alors exec $(X::x_{11} ; x_2 \ //\ Y::y_1 ; y_2,$

$\langle \langle \sigma_X, \theta_X + \delta_{X_{eval}}, nop \rangle, \eta_Y \rangle)$

sinon exec $(X::x_{11} ; x_2 \ //\ Y::y_1 ; y_2,$

$\langle \langle \sigma_X, \theta_X + \delta_{X_{eval}}, nop \rangle,$

$\langle memend(\eta_Y), datend(\eta_Y), nop \rangle\rangle)$

fsi

sinon si $\delta_{X_{eval}} + \theta_Y < datend(\eta_Y)$

alors exec $(X::x_{12} ; x_2 \ //\ Y::y_1 ; y_2,$

$\langle \langle \sigma_X, \theta_X + \delta_{X_{eval}}, nop \rangle, \eta_Y \rangle)$

sinon exec $(X::x_{12} ; x_2 \ //\ Y::y_1 ; y_2,$

$\langle \langle \sigma_X, \theta_X + \delta_{X_{eval}}, nop \rangle,$

$\langle memend(\eta_Y), datend(\eta_Y), nop \rangle\rangle)$

fsi

fsi ;

]

$[x_1 = abort \vee x_1 = nop \vee x_1 = (vx := ex) \Rightarrow$

$[exec\ (X::x_1 ; x_2 \ //\ Y::y_1 ; y_2, \langle \eta_X, \eta_Y \rangle) =$

si $datend(apply(x_1, \eta_X)) + \theta_Y < datend(\eta_Y) + \theta_Y$

alors exec $(X::x_2 \ //\ Y::y_1 ; y_2, \langle apply(x_1, \eta_X), \eta_Y \rangle)$

sinsi $datend(\eta_Y) + \theta_Y < datend(apply(x_1, \eta_X)) + \theta_Y$

alors exec $(X::x_2 \ //\ Y::y_1 ; y_2, \langle\langle \sigma_X, \theta_X, x_1 \rangle,$

$\langle memend(\eta_Y), datend(\eta_Y), nop \rangle\rangle)$

sinon exec $(X::x_2 \ //\ Y::y_1 ; y_2, \langle apply(x_1, \eta_X),$

$\langle memend(\eta_Y), datend(\eta_Y), nop \rangle\rangle)$

fsi ;

]

]

2.4.1.2.1. LPP synchrone

Le point essentiel de l'axiomatisation qui suit est l'accomplissement de la communication par rendez-vous et où l'on peut noter

i) l'expression de l'attente dans le cas où un des agents concerné par une communication n'est pas apte à l'accomplir, i.e. il n'a pas atteint l'action complémentaire de celle décrite par son interlocuteur (1.1), (1.2), (2),

ii) l'expression du blocage total dans le cas où les agents concernés par une communication ont atteint des actions de communication non complémentaires (1.4),

iii) la réalisation de la communication par une affectation (à distance) de la variable de réception par la valeur de l'expression envoyée, dans le cas où les agents se désignent dans une communication par des actions complémentaires (1.3).

Axiomatisation

(1) $\eta_X = \langle \sigma_X, \theta_X, nop\rangle \wedge \eta_Y = \langle \sigma_Y, \theta_Y, nop\rangle \Rightarrow$

(1.1) $[(x_1 = X.Y!\,ex \vee x_1 = X.Y?\,vx) \wedge (y_1 = abort \vee y_1 = abort \vee y_1 = (vy := ey)) \Rightarrow$

$[exec\ (X::x_1 ; x_2\ //\ Y::y_1 ; y_2, \langle \eta_X, \eta_Y\rangle) =$

$exec\ (X::x_1 ; x_2\ //\ Y::y_2, \langle \eta_X, apply\ (y_1, \eta_Y)\rangle)$

$] ;$

(1.2) $(x_1 = X.Y!\,ex \vee x_1 = X.Y?\,vx) \wedge (y_1 = if\ b\ then\ y_{11}\ else\ y_{12}\ fi) \Rightarrow$

$[exec\ (X::x_1 ; x_2\ //\ Y::y_1 ; y_2, \langle \eta_X, \eta_Y\rangle) =$

$si\ evalue\ (b, \eta_Y)$

$alors\ exec\ (X::x_1 ; x_2\ //\ Y::y_{11} ; y_2, \langle \eta_X, \langle \sigma_Y, \theta_Y + \delta_{Y_{eval}}, nop\rangle\rangle)$

$sinon\ exec\ (X::x_1 ; x_2\ //\ Y::y_{12} ; y_2, \langle \eta_X, \langle \sigma_Y, \theta_Y + \delta_{Y_{eval}}, nop\rangle\rangle)$

$] ;$

$(x_1 = X.Y!\,ex \vee x_1 = X.Y?\,vx) \wedge (y_1 = Y.X!\,ey \vee y_1 = Y.X?\,vy) \Rightarrow$

$[exec\ (X::x_1 ; x_2\ //\ Y::y_1 ; y_2, \langle \eta_X, \eta_Y\rangle) =$

(1.3) $si\ x_1 = X.Y!\,ex \wedge y_1 = Y.X?\,vy$

$alors\ exec\ (X::x_2\ //\ Y::y_2, \langle \eta_X,$

$\langle subst\ (\sigma_Y, valvar(vy), evalue(ex, \eta_X)),$

$\theta_Y + \delta_{X_{eval}} + \delta_{Y\leftarrow}, nop\rangle\rangle)$

(1.4) $sinsi\ (x_1 = X.Y!\,ex \wedge y_1 = Y.X!\,ey)$

$\vee\ (x_1 = X.Y?\,vx \wedge y_1 = Y.X?\,vy)$

$alors\ exec\ (X::x_1 ; x_2\ //\ Y::y_1 ; y_2, \langle \langle \sigma_X, \theta_X, abort\rangle,$

$\langle \sigma_Y, \theta_Y, abort\rangle\rangle)$

fsi

$] ;$

$]$

(2) $\eta_X = \langle \sigma_X, \theta_X, nop\rangle \wedge \eta_Y = \langle \sigma_Y, \theta_Y, \mu\rangle$

$\wedge \neg(\mu = nop) \wedge \neg(\mu = abort) \Rightarrow$

$[x_1 = X.Y!\,ex \vee x_1 = X.Y?\,ux \Rightarrow$

$[exec\ (X::x_1 ; x_2\ //\ Y::y_1 ; y_2, \langle \eta_X, \eta_Y\rangle) =$

$exec\ (X::x_1 ; x_2\ //\ Y::y_1 ; y_2, \langle \eta_X,$

$\langle memend\ (\eta_Y), datend\ (\eta_Y), nop\rangle\rangle)$

$]$

$]$

2.4.1.2.2. LPP asynchrone

Le point essentiel de l'axiomatisation qui suit est la spécification des communications modélisées par des files de messages gérées en FIFO. Pour l'accomplir nous fixons quelques hypothèses :

i) chaque agent possède autant de files de messages à consommer que d'interlocuteurs possibles.

ii) les files de messages à consommer sont automatiquement systématiquement déclarées au sein de chaque agent comme étant des variables locales identifiées par le nom $filentrée_{idp,lim}$ où idp est l'identificateur de l'agent producteur et lim est la taille limite qui lui est associée.

iii) pour un même agent, les tailles de ses différentes files de réception ne sont pas obligatoirement les mêmes. Elles sont supposées être positives strictement et calculées aléatoirement lors de l'activation du programme.

iv) l'action de production, notée X.Y!e, du message e au sein de l'agent X pour l'agent Y se réalise, lorsque la file de réception des messages de X chez Y n'est pas pleine, par un depôt du message e dans celle-ci.

v) l'action de consommation, notée X.Y?v, d'un message produit par l'agent Y pour l'agent X, transite par la variable v locale à X lorsque le retrait du message en tête de la file de réception des messages en provenance de Y est possible ; i.e. la file en question n'est pas vide.

Avant d'entamer l'axiomatisation des communications asynchrones, nous allons spécifier le type des files de réception, que nous venons de décrire informellement. Ce sera un triplet formé de la file proprement dite, dont la présentation algébrique est celle donnée par le type QUEUE paramétré par les messages reçus, d'un identificateur de processus et de la taille limite (entier positif strictement) de la file.

```
Type  : QUEUE [VALUE] ;
Sorts : Queue / Value, Int, Bool ;
Opns  : qempty : → Queue ;
        inqueue : Queue x Value → Queue ;
        dequeue : Queue → Queue ;
        head : Queue → Value ;
        isempty : Queue → Bool ;
        length : Queue → Int ;
Eqns  : q ∈ Queue ; v ∈ Value ;
     isempty (qempty) = true ;
     isempty (inqueue(q,v)) = false ;
     dequeue (inqueue(q,v)) =
            si isempty(q)
            alors q
            sinon inqueue (dequeue (q), v)
            fsi ;
     head (inqueue(q,v)) =
            si isempty(q)
            alors v
            sinon head(q)
            fsi ;
```



```
     length (quempty) = 0 ;
     lenght (inqueue(q,v)) = succ (lenght(q)) ;
Fin ;
```

La spécification de la file de réception est alors la suivante :

```
Type  : FILENTREE [VALUE] ;
Sorts : Filentrée / Queue, Idprocess, Int, Bool ;
Opns  : make : Queue x Idprocess x Int → Filentrée ;
        limit : Filentrée → Int ;
        concern : Filentrée → Idprocess ;
        file : Filentrée → Queue ;
        vide, plein : Filentrée → Bool ;
        defiler : Filentrée → Filentrée ;
        enfiler : Filentrée x Value → Filentrée ;
        tête : Filentrée → Value ;
Eqns  : q ∈ Queue ; i ∈ Idprocess ; x ∈ Int ; f ∈ Filentrée ;
     limit (make(q,i,x)) = x ;
     concern (make(q,i,x)) = i ;
     file (make(q,i,x)) = q ;
     plein (f) = eq (lenght(file(f)), limit(f)) ;
     vide (f) = eq (lenght(file(f)), zéro) ;
     defiler (f) = make (dequeue(file(f)), concern(f), limit(f)) ;
     enfiler (f, v) =
        si plein(f)
        alors f
        sinon make (inqueue(file(f),v), concern(f), limit(f))
        fsi ;
     tête (f) = head (file(f)) ;
Fin ;
```

Remarques :

- dans la spécification du type QUEUE les opérations d'observations "head" et "dequeue" sont partiellement définies.
- dans l'axiomatisation qui suit, nous notons la file de réception f des messages en provenance de l'agent X et dont la longueur limite est l par $f_{X,l}$ au lieu de make(f,X,l).

Axiomatisation

En vue d'améliorer la lisibilité de l'axiomatisation de la communication asynchrone nous décrivons deux nouvelles opérations, prenant en compte les changements subis par les files d'entrée d'un agent lors d'une production ou d'une consommation d'un message.

Opns : send : Idprocess x Idprocess x Exp x Universe → Universe ;
receive : Idprocess x Idprocess x Var x Universe → Universe ;

Eqns :

send (P,Q,e, $\langle \sigma_Q, \Theta_Q, \text{nop} \rangle$) =

$\langle$ subst (σ_Q, $\text{filentrée}_{p,\gamma}$, enfiler($\text{filentrée}_{p,\gamma}$, eval(e, σ_p))),

$\Theta_Q + \delta_{p_{eval}} + \delta_{Q\leftarrow}$, nop $\rangle$

receive (P,Q,v, $\langle \sigma_p, \Theta_p, \text{nop} \rangle$) =

$\langle$ subst (subst(σ_p, v, tête($\text{filentrée}_{Q,\lambda}$)),

$\text{filentrée}_{Q,\lambda}$, défiler($\text{filentrée}_{Q,\lambda}$)),

$\Theta_p + \delta_{p_{eval}} + \delta_{p\leftarrow} + \delta_{p\leftarrow}$ + nop $\rangle$

Fin ;

(1) $\eta_X = \langle \sigma_X, \Theta_X, \text{nop} \rangle \wedge \eta_Y = \langle \sigma_Y, \Theta_Y, \text{nop} \rangle \Rightarrow$

exec (X:: X.Y! ex ; x // Y::y_1 ; y_2, $\langle \eta_X, \eta_Y \rangle$) =

cas y_1 de

nop, abort, vy := ey alors

si evalue (plein($\text{filentrée}_{X,Y}$), η_Y)

alors exec (X:: X.Y! ex ; x // Y::y_2, $\langle \eta_X$, apply (y_1, η_Y)$\rangle$)

sinon exec (X::x // Y::y_2, $\langle \eta_X$, apply (y_1, send(X,Y,ex,η_Y))$\rangle$)

fsi ;

if b then y_{11} else y_{12} fi alors

si evalue (plein($\text{filentrée}_{X,Y}$), η_Y)

alors si evalue (b, η_Y)

alors exec (X::X.Y! ex ; x // Y::y_{11} ; y_2, $\langle \eta_X$,

$\langle \sigma_Y, \Theta_Y + \delta_{Y_{eval}}$, nop$\rangle\rangle$)

sinon exec (X::X.Y! ex ; x // Y::y_{12} ; y_2, $\langle \eta_X$,

$\langle \sigma_Y, \Theta_Y + \delta_{Y_{eval}}$, nop$\rangle\rangle$)

fsi

sinon si evalue (b, η_Y)

alors exec (X::x // Y::y_{11} ; y_2, $\langle \eta_X$, send (X,Y,ex,

$\langle \sigma_Y, \Theta_Y + \delta_{Y_{eval}}$, nop$\rangle\rangle$)

sinon exex (X::x // Y::y_{12} ; y_2, $\langle \eta_X$, send (X,Y,ex,

$\langle \sigma_Y, \Theta_Y + \delta_{Y_{eval}}$, nop$\rangle\rangle$)

fsi

fsi ;

Y.X! ey alors
  si evalue (plein(filentrée$_{X,Y}$), η_Y)
  alors si evalue (plein(filentrée$_{Y,\lambda}$), η_X)
        alors < <σ_X, θ_X, abort>, <σ_Y, θ_Y, abort>>
        sinon exec (X::X.Y! ex ; x // Y::y_2, <send(Y,X,ey,η_X), η_Y>)
        fsi
  sinon si evalue (plein(filentrée$_{Y,\lambda}$), η_X)
        alors exec (X::x // Y::y_1 ; y_2, <η_X, send(X,Y,ex,η_Y)>)
        sinon exec (X::x // Y::y_2, <send(Y,X,ey,η_X), send(X,Y,ex,η_Y)>)
        fsi

Y.X? vy alors
  si evalue (plein(filentrée$_{X,Y}$), η_Y)
  alors exec (X::X.Y! ex ; x // Y::y_2, <η_X, receive (Y,X, vy, η_Y)>)
  sinon si evalue (vide(filentrée$_{X,Y}$), η_Y)
        alors exec (X::x // Y::y_1 ; y_2, <η_X, send (X,Y,ex, η_Y)>)
        sinon exec (X::x // Y::y_2, <η_X, send (X,Y,ex, receive (Y,X,vy,η_Y))>
        fsi
  fsi ;
fcas ;

(2) η_X = <σ_X, θ_X, nop> ∧ η_Y = <σ_Y, θ_Y, nop> ⇒
exec (X::X.Y? vx ; x // Y::y_1 ; y_2, <η_X, η_Y>) =

cas y_1 de
  nop, abort, vy := ey alors
    si evalue (vide(filentrée$_{Y,\lambda}$), η_X)
    alors exec (X::X.Y? vx ; x // Y::y_2, <η_X, apply (y_1, η_Y)>)
    sinon exec (X::x // Y::y_2, < receive (X,Y,vy,η_X), apply (y_1, η_Y)>)
    fsi ;
  if b then y_{11} else y_{12} fi alors
    si evalue (vide(filentrée$_{Y,\lambda}$), η_X)
    alors si evalue (b, η_X)
          alors exec (X::X.Y? vx ; x // Y::y_{11} ; y_2, <η_X,
                                  <σ_Y, θ_Y + $\delta_{Y_{eval}}$, nop>>)
          sinon exec (X::X.Y? vx ; x // Y::y_{12} ; y_2, <η_X,
                                  <σ_Y, θ_Y + $\delta_{Y_{eval}}$, nop>>)
          fsi
    sinon si evalue (b, η_X)
          alors exec (X::x // Y::y_{11} ; y_2, < receive (X,Y,vy,η_X),
                                  <σ_Y, θ_Y + $\delta_{Y_{eval}}$, nop>>)
          sinon exec (X::x // Y::y_{12} ; y_2, < receive (X,Y,vy,η_X),
                                  <σ_Y, θ_Y + $\delta_{Y_{eval}}$, nop>>)
          fsi
    fsi ;

Y.X! ey alors
  si evalue (vide(filentrée$_{Y,\lambda}$), η_X)
  alors exec (X::X.Y? vx ; x // Y::y_2, < send (Y,X,ey,η_X), η_Y>)
  sinon si evalue (plein(filentrée$_{Y,\lambda}$), η_X)
        alors exec (X::x // Y::y_1 ; y_2, < receive (X,Y,vx,η_X), η_Y>)
        sinon exec (X::x // Y::y_2, < send (Y,X,ey,receive (X,Y,vx, η_X)),η_Y>)
        fsi
  fsi ;

Y.X? vy alors
  si evalue (vide(filentrée$_{Y,\lambda}$), η_X)
  alors si evalue (vide(filentrée$_{X,Y}$), η_Y)
        alors < <σ_X, θ_X, abort>, <σ_Y, θ_Y, abort> >
        sinon exec (X::X.Y? vx ; x // Y::y_2, <η_X, receive (Y,X,vy, η_Y)>)
        fsi
  sinon si evalue (vide(filentrée$_{X,Y}$), η_Y)
        alors exec (X::x // Y::y_1 ; y_2, < receive (X,Y,vx,η_X), η_Y>)
        sinon exec (X::x // Y::y_2, < receive (X,Y,vx,η_X),
                                        receive (Y,X,vy,η_Y)>)
        fsi
  fsi ;
fcas ;

(3) η_X = <σ_X, θ_X, nop> $\wedge$ η_Y = <σ_Y, θ_Y, μ> $\wedge$ $\neg(\mu$ = nop) $\wedge$ $\neg(\mu$ = abort) $\Rightarrow$
  [x_1 = X.Y! ex $\vee$ x_1 = X.Y? vx $\Rightarrow$
   exec (X::x_1 ; x_2 // Y::y, <η_X, η_Y>) =
     exec (X::x_1 ; x_2 // Y::y, <η_X, <memend (η_Y), datend (η_y), nop>>)
  ] ;

2.4.1.2.3. Remarque

Nous avons formalisé dans les deux derniers paragraphes des systèmes parallèles développés uniquement par deux agents évoluant en parallèle. Une question s'impose à ce niveau est de savoir comment nous entendons étendre les résultats de cette formalisation pour prendre en compte des systèmes à plus de deux agents parallèles.

A première vue, les formalisations données par LPP synchrone et LPP asynchrone paraissent être à caractère explosif, puisque nous nous intéressons à chaque modification de chaque agent et nous essayons de les combiner, sachant que les unions sont de telle ou telle forme. Pour remédier à cette explosion des combinaisons possibles des modifications des différents agents, nous devons :

- utiliser, au mieux, les propriétés de commutativité et d'associativité de l'opérateur du parallélisme de manière à accorder plus d'attention aux agents les plus pertinents à savoir ceux concernés par des communications,
- "chronométrer" le déroulement du programme en calculant, après chaque modification intervenant sur l'état global du système considéré, la date de fin de la prochaine modification. Cette date est égale au minimum de l'ensemble des dates de fin des modifications courantes des différents agents constituants le système parallèle considéré, sachant que la durée de toutes les actions de communications complémentaires qui peuvent avoir lieu est une donnée constante.

Notons finalement que pour formaliser LPP asynchrone, il suffit de raisonner sur un agent uniquement, alors que pour formaliser LPP synchrone il faut entreprendre un raisonnement mettant en oeuvre les interlocuteurs à tout instant (d'ailleurs nous supposons que dans ce cas, la durée d'un envoi ou d'une réception est infinie si l'on considère les interlocuteurs indépendemment les uns des autres, alors que la durée de la communication lorsqu'elle a lieu est une constante connue).

Ces éclaircissements nous ont permis d'entreprendre une seconde formalisation de LPP synchrone (voir annexe 2), celle relative à LPP asynchrone étant plus facile à écrire nous ne la donnons pas.

2.4.2. LPP nondéterministe

Ce paragraphe est consacré à l'étude du non déterminisme, syntaxiquement exprimé par l'opérateur □ dans le corps des agents communicants. La formalisation de ce concept utilise la totalité des résultats présentés lors de la formalisation de LPP déterministe.

La spécification des agents non déterministes est basée sur celle relative aux modifications non déterministes et que nous présentons comme étant un enrichissement, par une opération de choix, de celle décrivant les modifications déterministes :

```
Type  : NMODIF ;
Sorts : Nmodif / Modif ;
Opns  : modifnmodif : Modif → Nmodif ;
        □ : Nmodif x Nmodif → Nmodif ;
Eqns  : m1,m2,m3 ∈ Nmodif ;
    m1 □ m2 = m2 □ m1 ;
    m1 □ (m2 □ m3) = (m1 □ m2) □ m3 ;
Fin ;
```


La présentation des agents non déterministes est alors équivalente à celle donnée aux agents déterministes au renommage près de la sorte d'intérêt et de la sorte des modifications :

```
Type  : NAGENT ;
Sorts : Nagent / Idprocess, Nmodif ;
Opns  :
        % celles données au paragraphe précédent au renommage
          près de la sorte d'intérêt et de la sorte des
          modifications %
Fin ;
```

La spécification des états (de l'interprète abstrait de LPP non déterministe), quant à elle, va être décrite comme étant l'ensemble de l'ensemble des univers des agents constituants un système parallèle :

```
Type  : NSTATE ;
Sorts : Nstate / State, Bool ;
Opns  :  ∅  : → Nstate ;
        {-} : State → Nstate ;
        -∪- : Nstate x Nstate → Nstate ;
        -∈- : State x Nstate → Bool ;
Eqns  : Π, Π1, Π2 ∈ Nstate ; Σ ∈ State ;
    Π ∪ ∅ = Π ;
    Π ∪ Π = Π ;
    Π1 ∪ Π2 = Π2 ∪ Π1 ;
    Π1 ∪ (Π2 ∪ Π3) = (Π1 ∪ Π2) ∪ Π3 ;
    Σ ∈ ∅ = false ;
    Σ ∈ {Σ} = true ;
    Σ ∈ (Π1 ∪ Π2) = Σ ∈ Π1  ∪  Σ ∈ Π2
Fin ;
```

Maintenant, il nous est possible de décrire les "calculs" relatifs à l'interprétation d'un agent non déterministe dans ses propres univers. Nous formalisons cette interprétation par la donnée de l'opération :

Opns : nexec : Nagent x Nstate → Nstate ;
Eqns :
nexec (X::x, {Σ} ∪ Π) =
si x = x_1 □ x_2
alors nexec (X::x_1, {Σ} ∪ Π) ∪ nexec (X::x_2, {Π} ∪ Π)
sinon {exec (X::x, Σ)} ∪ nexec (X::x, Π)
fsi ;

nexec (X::x_1 □ x_2 // A, Π) =
nexec (X::x_1 // A, Π) ∪ nexec (X::x_2 // A, Π)
Fin.

3. ETUDE DE LA FORMALISATION DE LPP

Nous nous intéressons au cours de cette étude, d'une part, à la description informelle des propriétés présentées par les différents types abstraits associés aux différentes parties de LPP, et d'autre part à la preuve de la complémentarité de la définition algébrique de LPP déterministe et synchrone par rapport à sa définition axiomatique.

3.1. Propriétés des types abstraits de la formalisation

Quelque soit la partie de langage considérée, il est à noter que le type abstrait la formalisant est non suffisamment complet à cause de la partialité des définitions de certaines de ses opérations. Ce qui n'est pas très génant dans la mesure où ceci permettra un grand nombre de modèles d'implantation ; bien entendu sous l'hypothèse de leur consistance par rapport aux présentations primitives.

Ce qui est notable, sous ces hypothèses, est que les modèles terminaux (non triviaux, lorsqu'ils existent) de LPP synchrone et LPP asynchrone présentent des "observations" communes sur la sorte Universe, puisque les termes de cette sorte admettent :

- soit, une forme avortée, due à la modification "abort" ou à un blocage global,
- ou bien, une forme infinie, due à un calcul (local) divergent produisant une infinité de changements dans l'univers,
- ou encore, une forme bloquée, causée par un calcul infini ne produisant pas de changements dans l'univers.

Ce qui les distingue par contre, sont les termes de Universe correspondant à des corps de programmes se terminant normalement :

- pour LPP synchrone, ces termes peuvent se mettre sous une forme normale construite à partir d'un univers initial de la forme <init, tzéro, nop> par des substitutions successives d'objets (bien définis) par d'autres.

- pour LPP asynchrone deux catégories de terminaisons sont possibles :

. la terminaison totale correspondant à des corps de programmes terminés normalement et où les files de messages à consommer sont vides.

. la terminaison partielle correspondant à des corps de programmes terminés sans pour autant que ses messages en entrée soient entièrement consommés (i.e. les files de réception ne sont pas toutes vides).

3.2. Complémentarité des formalisations algébrique et axiomatique de LPP synchrone et déterministe

3.2.1. Introduction

Pour un même langage de programmation, on constate que plusieurs définitions sémantiques peuvent être présentées [HOA 74, DON 75]. L'objectif commun à celles-ci est de donner un sens (global) à chaque programme écrit dans le langage considéré. Une question se pose alors est de savoir si ces différentes définitions donnent bien "le même sens" à chaque programme, ou si elles sont cohérentes entre elles, on dit qu'elles sont complémentaires. Pour répondre à une telle question, le concept de complémentarité des sémantiques doit être précisé. Dans la plupart des cas ceci revient à démontrer, par récurrence sur la structure du programme, que quelque soit la sémantique considérée, un programme réalise la même fonction de transformation des données en résultats. Un cas particulier de sémantique à regarder de près est la sémantique axiomatique (la plus abstraite puisque dégagée des considérations d'exécution). Nous consacrons le reste de ce chapitre à l'étude de la complémentarité de la formalisation algébrique et de la formalisation axiomatique de LPPSD : acronyme pour Langage de Programmation Parallèle Synchrone de Déterministe.

3.2.2. La sémantique algébrique de LPPSD

Le but ici est de rappeler brièvement le principal de ce qui a été entrepris au § 2.4.1. En effet, nous étions amené à décrire un certain nombre de domaines syntaxiques et sémantiques, ainsi qu'un certain nombre de fonctions d'évaluation et d'interprétation, dont les définitions mathématiques peuvent être les suivantes :

- les domaines syntaxiques :

```
Id, Bool, Int,
Var = Id + [Id x Exp],
Exp = Bool + Int + Var + [Uop x Exp] + [Bop x Exp x Exp],
Uop = {+, - , ¬},  Bop = {+ , - , * , / , ∧, ∨, < , ⩽ , = , > , ⩾, <>},
Modif = nop + abort +
        [Var x Exp] +              -- affectation
        [Exp x Modif x Modif]      -- conditionnelle
        [Exp x Modif]              -- itérative
        [Modif x Modif]            -- composition séquentielle
        [Id x Id x Exp]            -- envoi d'un message
        [Id x Id x Var],           -- réception d'un message
Agent = [Id x Modif] + [Id x Modif  x Agent],
Program = Id x Agent,
```

- les domaines sémantiques

```
Int, Bool, Time, Modif,
Value = Int + Bool,
Memory = Id → Value,
Universe = Memory x Time x Modif,
State = Universe + [Universe x State],
```

- Les fonctions d'interprétation :

```
eval : Exp → [Memory → Value]
evalue : Exp → [Universe → Value]
apply : Modif → [Universe → Universe]
exec : Agent → [State → State]
```

3.2.3. La sémantique axiomatique de LPPSD

Pour décrire la sémantique axiomatique d'un langage de programmation quelconque on a besoin de spécifier :

i) la syntaxe abstraite des constructions du langage,

ii) l'ensemble des expressions logiques ou assertions,

iii) un ensemble de formules, de la forme {P} S {Q} où P et Q sont des assertions et S un morceau de programme, définissant la signification des constructions simples du langage. L'interprétation intuitive de telles formules est "si P est vraie avant l'exécution de S et que S termine alors Q est vraie après l'exécution de S".

iv) un ensemble de règles d'inférence de la forme $\frac{H1, \ldots, Hn}{F}$, où les Hi ($1 \leqslant i \leqslant n$) sont les hypothèses et F est la conclusion, servant à décrire la signification des constructions composées du langage. L'interprétation intuitive de telles règles d'inférence est "si H1, ..., Hn sont vraies alors F est également vraie".

3.2.3.1. Syntaxe abstraite de LPPSD

Se rapporter à la description du § 2.3.

3.2.3.2. Les assertions

Les expressions logiques d'un système axiomatique obéissent à certaines règles syntaxiques et sémantiques. Celles que nous manipulons sont construites dans le langage d'assertions suivant, où <expression> est le non terminal correspondant aux expressions booléennes.

< assertion > ::= < assertion atomique > | ¬ < assertion > |
(< assertion > < connecteur > < assertion >) |
< assertion quantifiée > |

< assertion quantifiée > ::= < quantificateur > < quantifié >, < assertion >

< quantifié > ::= < identificateur > : < domaine > | < identificateur >

< domaine > ::= < expression > .. < expression >

< assertion atomique > ::= < expression > | < assertion > << substitution >>

< substitution > ::= < expression > / < expression >

< connecteur > ::= ∧ | ∨ | ⊃

< quantificateur > ::= ∀ | ∃

Le calcul des expressions logiques (on dit aussi calcul propositionnel) possède un ensemble de propriétés [MAN 74] dont les principales sont les suivantes : pour toutes assertions P, Q et R

. $P \supset (Q \supset P)$

. $(P \supset (Q \supset R)) \supset ((P \supset Q) \supset (Q \supset R))$

. $(\neg P \supset \neg Q) \supset (Q \supset P)$

. $\frac{P,\ P \supset Q}{Q}$ -- modus ponens

. $\forall x,\ P \supset P\langle x / t\rangle$ si t ne contient aucune variable libre de P -- substitution

. $\frac{P \supset Q}{P \supset \forall x,\ Q}$ si x liée dans P -- introduction ∀

et de manière tout à fait habituelle nous convenons que

$P \vee Q$ représente $\neg P \supset Q$

$P \wedge Q$ représente $\neg (\neg P \vee \neg Q)$

$\exists x\ P$ représente $\neg \forall x\ \neg P$

3.2.3.3. La sémantique des instructions de LPPSD

La définition axiomatique des constructions de LPPSD peut être donnée par le système de preuve (inspiré de [APT 80]) consistant en les axiomes et les règles d'inférence suivantes. En annexe 3, nous donnons, par le biais de ce système, la preuve de correction partielle d'un exemple de programme LPPSD.

A1. affectation : $\{P\langle v/e\rangle\}\ \ v := e\ \ \{P\}$
P $\langle v/e\rangle$ exprime le résultat de la substitution par e de toutes les occurrences libres de v dans P.

A2. instruction vide : {P} nop {P}

A3. instruction d'avortement : {False} abort {P}

A4. envoi de messages : {P} X.Y!e {P}
On suppose ici que le résultat d'une telle action est sans effet de bord, ce qui revient à dire que toute action d'envoi a un complémentaire chez le destinataire en question.

A5. réception de messages : {P} X.Y?v {Q}
Ici il est tout à fait normal que la postcondition soit différente de la précondition ; on exigera d'elle de vérifier la condition de coopération (CC) des preuves des interlocuteurs X et Y.

R1- instruction conditionnelle : $\dfrac{\{P \wedge b\}\ S1\ \{R\}\ ,\ \{P \wedge \neg b\}\ S2\ \{R\}}{\{P\}\ \underline{\text{if}}\ b\ \underline{\text{then}}\ S1\ \underline{\text{else}}\ S2\ \underline{\text{fi}}\ \{R\}}$

R2- instruction itérative : $\dfrac{\{P \wedge b\}\ \ S\ \ \{P\}}{\{P\}\ \underline{\text{while}}\ b\ \underline{\text{do}}\ S\ \underline{\text{od}}\ \{P \wedge \neg b\}}$

R3- composition séquentielle : $\dfrac{\{P\}\ S1\ \{Q\}\ ,\ \{Q\}\ S2\ \ \{R\}}{\{P\}\ \ S1;S2\ \ \{R\}}$

R4- composition parallèle

$$\frac{\{P1\}\ S1\ \{Q1\}\ ,\ \ldots\ ,\ \{Pn\}\ Sn\ \{Qn\}\ ,\ \text{satisfaction des communications}}{\{P1 \wedge \ldots \wedge Pn\}\quad A1::S1\ //\ \ldots\ //\ An::Sn\quad \{Q1 \wedge \ldots \wedge Qn\}}$$

Cette règle d'inférence décrit la composition parallèle d'un ensemble de morceaux de programmes. Les hypothèses d'une telle règle sont d'une part, les différentes esquisses de preuve {Pi} Si {Qi} ($1 \leqslant i \leqslant n$) et d'autre part l'assertion de la satisfaction des communications (CC) qui permet l'établissement de la coopération des différentes esquisses (et donc des postassertions des axiomes d'envoi et de réception de messages) et qu'on peut énoncer ainsi :

si [X::.... {P} Y?v {Q}.../.../ Y:: ... {R} X!e {R} ...]
alors (P $\wedge$ R) $\supset$ (Q $\wedge$ R) $\langle v/e\rangle$ assertion déduite du fait que la modélisation de la communication est entreprise par une affectation (à distance), que l'on peut établir par l'axiome

{true} X.Y?v // Y.X!e {v = e}

Noter que $\neg$ (P $\wedge$ R) empêche X et Y de se synchroniser et par la même occasion empêche la communication d'avoir lieu.
Par ailleurs, les agents d'un programme LPPSD n'interfèrent pas de nature puisqu'ils ne partagent aucune mémoire, il est alors tout à fait raisonnable de supposer que leur preuve n'interfèrent pas. Cependant, on a généralement besoin d'introduire des variables globales auxiliaires pour faciliter la preuve de certaines propriétés de programmes. Leur utilisation falsifie alors les preuves des esquisses et de satisfaction des communications, et une preuve de non-interférence devient nécessaire. Dans ce cas, on restreint les variables auxiliaires à n'apparaître que dans des assertions et comme paramètres des actions de communication et on prouve les axiomes

A6. préservation : $\{P \wedge pré(S)\}\ S\ \{P\}$
si aucune variable libre de P n'est sujette à un changement dans S.

L'assertion pré(S) correspond à la précondition de S.

A7. substitution : $(P \wedge pré(X.Y!e) \wedge pré(Y.X?v)) \supset P\langle v/e\rangle$
si P est une assertion contenue dans un processus Z s'exécutant en parallèle avec X et Y, et v est une variable libre dans P.

R5- conjonction : $\dfrac{\{P\}\ S\ \{Q\}\ ,\ \{P\}\ S\ \{R\}}{\{P\}\ S\ \{Q \wedge R\}}$

R6- conséquence : $\dfrac{P \supset R1,\ \{R1\}\ S\ \{R2\}\ ,\ R2 \supset Q}{\{P\}\ S\ \{Q\}}$

3.2.4. Complémentarité des formalisations

3.2.4.1. But et approche préconisée

Prouver que les définitions axiomatiques, i.e les axiomes et les règles d'inférence de la sémantique axiomatique, de LPPSD sont valides par rapport aux modèles fournis par la définition algébrique ; tel est le but de l'étude de la complémentarité (on dit aussi consistance) des définitions.
Nous procédons alors en trois étapes :

i) fournir une interprétation des assertions d'une définition axiomatique en terme des domaines de la définition algébrique,

ii) utiliser une technique similaire à celle de [DON 75] en interprétant les formules du système axiomatique comme des phrases du calcul des prédicats,

iii) prouver que les axiomes et les règles d'inférence sont valides par rapport à leur interprétation, i.e les axiomes sont vrais et les règles d'inférence préservent la vérité.

3.2.4.2. Interprétation des assertions

Nous entreprenons ici une description semblable à celle du § 3.2.2. En particulier, nous décrivons les domaines syntaxiques et sémantiques des assertions, ainsi que les fonctions sémantiques qui donnent leur signification.

i) Syntaxe abstraite :

Bool, Var, Id,
Var = Id + [Id x Exp]
Exp = Bool + Int + Var + [Uop x Exp] + [Bop x Exp x Exp]
Uop = {+, -, ¬} , Bop = {+, -, *, /, ∨, ∧, ⊃, <, ≤, =, ≠, >, ≥}
Ast = Exp + [Not x Ast] + [Binop x Ast x Ast]
+ [Qt x Id x Ast] + [Qt x Id x Exp x Exp x Ast]
+ [Ast x Exp x Exp],
Not = {¬} , Binop = {∧, ∨, ⊃} , Qt = {∀, ∃}

ii) sémantique :

- les domaines sémantiques :

Bool = {true, false} , Int, Time, Id, Modif, Value = Int + Bool,
Object = Bool + Int + Value + Id + Modif + Exp + Ast,
Memory = Id → Value,
Universe = Memory x Time x Modif,
State = Universe + [State x Universe]

- les fonctions sémantiques :

evalue : Exp → [Universe → Value]
evalue est la fonction déjà décrite dans le cadre de la définition algébrique de LPPSD.
substituer : Universe x Object x Object → Universe
substituer (η, e1, e2) = < subst(memend(η), e1, e2),
datend(η) + δ_4, nop >
avec subst la fonction de génération de mémoire décrite lors de la définition algébrique de LPPSD.

$I : Ast \to [Universe \to Bool]$

$\eta = \langle \sigma, \theta, abort \rangle \Rightarrow I[\![ast]\!]\eta = true$;

$\eta = \langle \sigma, \theta, \mu \rangle \wedge \neg(\mu = nop) \wedge \neg(\mu = abort) \Rightarrow$
$I[\![ast]\!]\eta = I[\![ast]\!](\langle memend(\eta), datend(\eta), nop \rangle)$;

$\eta = \langle \sigma, \theta, nop \rangle \Rightarrow$

[$I[\![exp]\!]\eta = evalue(exp, \eta)$;

$I[\![\neg ast]\!]\eta = \neg(I[\![ast]\!]\eta)$;

$I[\![ast1 \wedge ast2]\!]\eta = (I[\![ast1]\!]\eta) \wedge (I[\![ast2]\!]\eta)$;

$I[\![ast1 \vee ast2]\!]\eta = (I[\![ast1]\!]\eta) \vee (I[\![ast2]\!]\eta)$;

$I[\![ast1 \supset ast2]\!]\eta = (I[\![ast1]\!]\eta) \supset (I[\![ast2]\!]\eta)$;

$I[\![ast\langle exp1 / exp2 \rangle]\!]\eta = I[\![ast]\!](substituer(\eta, exp1, exp2))$;

$I[\![\exists id : exp1 .. exp2, ast]\!]\eta = loopfalse(id, exp1, exp2, ast, \eta)$;
avec loopfalse (id, x1, x2, ast, η)
si evalue (x2, η) < evalue (x1, η)
alors false
sinon ($I[\![ast]\!]$ (substituer, valvar(id) evalue (x1, η)))
$\vee$ (loopfalse(id, succ (x1), x2, ast, η)
fsi ;

$I[\![\forall id : exp1..exp2, ast]\!]\eta = looptrue(id, exp1, exp2, ast, \eta)$;
avec looptrue (id, x1, x2, ast, η)
si evalue (x2, η) < evalue (x1, η)
alors true
sinon ($I[\![ast]\!]$(substituer, valvar(id) evalue (x1, η)))
$\wedge$ booptrue (id, succ(x1), x2, ast, η)
fsi

] ;

$J : Ast \to [State \to Bool]$;

$J[\![ast]\!]\langle \eta_1, \ldots, \eta_n \rangle = \bigwedge_{i=1}^{n} I[\![ast]\!]\eta_i$

3.2.4.3. Interprétation des formules axiomatiques

Intuitivement une formule de la forme {P} S {Q} va être interprétée par : "si P est vraie avant l'exécution du morceau de programme S et que S termine alors Q est vraie après l'exécution de S". Formellement, nous définissons {P} S {Q}

i) dans le cas où S est un élément de Modif(*), par :

$$\forall \eta, (I[\![P]\!]\eta \supset I[\![Q]\!](apply(S, \eta)))$$

ii) dans le cas où S est un élément de Agent(*), par :

$$\forall \Sigma, (J[\![P]\!]\Sigma \supset J[\![Q]\!](exec(S, \Sigma)))$$

3.2.4.4. Preuves de consistance

Pour prouver que les définitions axiomatiques sont consistantes, on doit prouver que chaque axiome et règle d'inférence est valide par rapport aux définitions algébriques. Pour cela nous utilisons les techniques d'induction point fixe et d'induction structurelle [MAN 74, LIV 78] pour prouver respectivement les propriétés des fonctions et des domaines récursivement définis.

1) l'affectation :

théorème : l'axiome {P <v/e>} v:=e {P} est valide

démonstration :

Par l'interprétation des formules axiomatiques on a :

$$\forall \eta, (I[\![P\ \langle v/e \rangle]\!]\eta \supset I[\![P]\!](apply(v:=e, \eta)))$$

(*) Les programmes considérés ici sont déterministes

Supposons la préassertion P<v/e>. D'après les définitions algébriques on a :

apply(v := e, η) = substituer (η, valvar(v), evalue (e, η)).

Pour prouver ce théorème on va prouver un résultat plus riche :

$$I[\![P\langle v/e\rangle]\!]\,\eta = I[\![P]\!]\,\eta'$$

où η' = substituer (η, valvar(v), evalue (e, η)).

lemme 1 : ∀ P ∈ Ast, ∀ e ∈ Exp, ∀ v ∈ Var, ∀ η ∈ Universe,
$I[\![P\langle v/e\rangle]\!]\,\eta = I[\![P]\!]$ (substituer(η, valvar(v), evalue (e, η)))

Démonstration :
Le lemme se démontre par induction structurelle sur le domaine Ast.

- Par définition de Ast, l'étape de base correspond à une étude sur la structure du domaine Exp.

lemme 2 : ∀ E, e ∈ Exp, ∀ v ∈ Var, ∀ η ∈ Universe,
evalue (E<v/e>, η) = evalue (E, substituer(η, valvar (v), evalue (e, η)))

Démonstration :
De nouveau, on prouve le lemme par induction structurelle sur le domaine Exp des expressions.
+ l'étape de base est évidente
+ l'étape d'induction suppose le lemme vrai pour toute expression E et prouve le lemme pour tous les sous domaines de Exp.
On suppose la preuve des cas.

- Pour les autres cas, l'induction porte sur l'étude des prédicats :

si P = ¬Q
alors $I[\![P\langle v/e\rangle]\!]\eta = \neg(I[\![Q\langle v/e\rangle]\!]\,\eta) = \neg(I[\![Q]\!]\,\eta') = I[\![P]\!]\,\eta'$

si P = Q ∧ R
alors $I[\![P\langle v/e\rangle]\!]\eta = I[\![Q\langle v/e\rangle]\!]\,\eta \wedge I[\![R\langle v/e\rangle]\!]\,\eta$
$= I[\![Q]\!]\,\eta' \wedge I[\![R]\!]\,\eta'$
$= I[\![Q \wedge R]\!]\,\eta'$

etc ...

Finalement, à partir du lemme 1, il est claire que

$$I[\![P\langle v/e\rangle]\!]\,\eta \supset I[\![P]\!]\,(\text{substituer}(\eta, \text{valvar}(v), \text{evalue}(e,\eta)))$$

ce qui complète la preuve du théorème.

2) l'instruction vide :

théorème : l'axiome {P} nop {P} est valide.

démonstration :
Par l'interprétation des formules axiomatiques on a

$$\forall \eta,\ (I[\![P]\!]\,\eta \supset I[\![P]\!]\,(\text{apply}(\text{nop},\eta))$$

Sachant que dans le cas général on a

apply (nop, η) = < memend(η), datend (η) + δ_{nop}, nop >

Le résultat se déduit immédiatement du fait que $I[\![P]\!]$ se définit à partir de la fonction evalue et que cette dernière ne porte que sur des univers parfaits.

3) l'instruction d'avortement :

théorème : l'axiome {false} abort {P} est valide.

démonstration :
Par l'interprétation des formules axiomatiques on a

$$\forall \eta,\ (\text{false} \supset I[\![P]\!]\,(\text{apply}(\text{abort},\eta)))$$

Ce qui est valide.

4) la composition séquentielle :

théorème : la règle d'inférence $\dfrac{\{P\}\ S1\ \{Q\},\ \{Q\}\ S2\ \{R\}}{\{P\}\ S1\ ;\ S2\ \{R\}}$ préserve la validité des hypothèses.

Démonstration :
On suppose les hypothèses {P} S1 {Q} et {Q} S2 {R} et la premisse P de la règle de la composition séquentielle.
D'après l'interprétation des formules axiomatiques {P} S1 ; S2 {R} se définit par : $\forall \eta,\ (I[\![P]\!]\,\eta \supset I[\![Q]\!]\,(\text{apply}(S1; S2,\eta)))$

Or par définition même de l'opération apply on a

apply (S1 ; S2, η) = apply (S2, apply (S1,η))

Soit alors un univers η tels que $I[\![P]\!]\ \eta$, par l'hypothèse :

- si η_1 = apply (S1, η) alors $I[\![Q]\!]\ \eta_1$
- si $I[\![Q]\!]\ \eta_1$ et si η_2 = apply (S1;S2,η) = apply (S2, η_1) alors $I[\![R]\!]\ \eta_2$

et par transitivité de $\supset$ on tire le résultat.

5) l'instruction conditionnelle

théorème : la règle d'inférence $\dfrac{\{P \wedge b\}\ S1\ \{Q\},\ \{P \wedge \neg b\}\ S2\ \{Q\}}{\{P\}\ \underline{if}\ b\ \underline{then}\ S1\ \underline{else}\ S2\ \underline{fi}\ \{Q\}}$

préserve la validité des hypothèses

démonstration :

La validité de la règle est déduite directement à partir de l'étude de cas menée sur les valeurs possibles de l'évaluation de la condition b. Par l'interprétation des formules axiomatiques on définit la conclusion {P} if b then S1 else S2 fi {Q} par

$\forall \eta,\ (I[\![P]\!]\ \eta \supset I[\![Q]\!]$(apply (if b then S1 else S2 fi, η)))

- Cas 1 : evalue (b,η) = false

  de la définition algébrique de la conditionnelle, on a

  apply (if b then S1 else S2 fi, η) = apply (S2, η) = η'

  En supposant $I[\![P]\!]\ \eta$ on a par hypothèse :

  $I[\![P]\!]\ \eta \wedge \neg$evalue (b, η) $\supset I[\![P \wedge b]\!]\ \eta \supset I[\![P]\!]\ \eta \supset I[\![Q]\!]\ \eta'$

  Or d'après la définition de la conséquence la première implication se résoud, ainsi se termine la démonstration du théorème pour ce cas.

- Cas 2 : evalue (b, η) = true

  La démonstration est analogue à la précédente.

6) l'instruction itérative :

théorème : la règle d'inférence $\dfrac{\{P \wedge b\}\ S\ \{P\}}{\{P\}\ \underline{while}\ b\ \underline{do}\ S\ \underline{od}\ \{P \wedge \neg b\}}$

préserve la validité des hypothèses.

Démonstration :

Par induction point fixe, on peut former la suite d'approximation

$\underline{while}^0$ b do S od = ⊥

$\underline{while}^{i+1}$ b do S od = if b then S ; $\underline{while}^i$ b do S od else nop fi

Sachant que la définition algébrique de l'itération est :

while b do S od = if b then S ; while b do S od else nop fi

on utilise la séquence pour prouver

{P ∧ b} S {P} implique {P} $\underline{while}^i$ b do S od {P ∧ ¬b}

- l'étape de base de l'induction point fixe vient immédiatement du fait que par convention :

  $\forall \eta\ I[\![R]\!]$ (apply ($\underline{while}^0$ b do S od, η)) =

  $I[\![R]\!]$ (apply^0 (while b do S od, η)) =

  true

- l'étape d'induction suppose que la règle est valide pour $\underline{while}^i$ b do S od et démontre la validité de $\underline{while}^{i+1}$ b do S od étant donnée l'hypothèse {P ∧ b} S {b}.

  La démonstration est menée par une étude de cas selon l'évaluation de b.

  . Cas 1 : evalue (b,η) = false

  dans ce cas $\underline{while}^{i+1}$ b do S od = nop

  et la démonstration est alors écrite puisque

  $I[\![P]\!]\ \eta$ par hypothèse

  $I[\![P]\!]\ \eta \wedge$ evalue (b,η) = false $\supset I[\![P \wedge \neg b]\!]\ \eta$

  . Cas 2 : evalue (b,η) = false

  dans ce cas $\underline{while}^{i+1}$ b do S od = S ; $\underline{while}^i$ b do S od

  sachant que {P ∧ b} S {P}

  et {P} $\underline{while}^i$ do S od {P ∧ ¬b}

  on déduit, moyennant la règle de la composition séquentielle, la validité de la règle dans ce cas.

7) la règle de la conséquence

théorème : la règle d'inférence $\frac{P \supset R1,\ \{R1\}\ S\ \{R2\},\ R2 \supset Q}{\{P\}\ S\ \{Q\}}$

préserve la validité des hypothèses.

démonstration :

D'après l'interprétation des formules axiomatiques on définit l'hypothèse $\{R1\}\ S\ \{R2\}$ par :

$$\forall \eta,\ (I[\![R1]\!]\eta \supset I[\![R2]\!](\text{apply}(S,\eta)))$$

et d'après la définition de la sémantique des assertions on a

$$I[\![P \supset R1]\!]\eta = (I[\![P]\!]\eta) \supset (I[\![R1]\!]\eta)$$
$$I[\![R2 \supset Q]\!]\eta' = (I[\![R2]\!]\eta') \supset (I[\![Q]\!]\eta')$$

et comme par hypothèses $P \supset R1$ et $R2 \supset Q$, on en déduit que

$$\forall \eta,\ (I[\![P]\!]\eta \supset I[\![Q]\!](\text{apply}(S,\eta)))$$

ce qui prouve le théorème.

8) la règle de la conjonction

théorème : la règle d'inférence $\frac{\{P\}\ S\ \{Q\},\ \{P\}\ S\ \{R\}}{\{P\}\ S\ \{Q \wedge R\}}$ préserve la validité des hypothèses.

démonstration :

Par l'interprétation de $\{P\}\ S\ \{Q \wedge R\}$ on exige :

$$\forall \eta,\ (I[\![P]\!]\eta \supset I[\![Q]\!](\text{apply}(S,\eta)))$$

Comme par hypothèse $\{P\}\ S\ \{Q\}$ et $\{P\}\ S\ \{R\}$, il est clair qu'on à $I[\![P]\!]\eta$, $I[\![Q]\!]\eta'$ et $I[\![R]\!]\eta'$ avec $\eta' = \text{apply}(S,\eta)$, or d'après la définition des assertions :

$$I[\![Q \wedge R]\!]\eta' = (I[\![Q]\!]\eta') \wedge (I[\![R]\!]\eta')$$

ce qui résoud la conclusion puisqu'on a en fait besoin que de l'implication $(I[\![Q]\!]\eta') \wedge (I[\![R]\!]\eta') \supset I[\![Q \wedge R']\!]\eta'$.

9) composition parallèle

théorème : la règle d'inférence qui suit préserve la validité des hypothèses

$$\frac{\{P1\}\ S1\ \{Q1\},\ \ldots,\ \{Pn\}\ Sn\ \{Qn\},\ \text{satisfation des communications}}{\{P1 \wedge .. \wedge Pn\}\quad A1::S1 /\!/ \ldots /\!/ An::Sn\quad \{Q1 \wedge .. \wedge Qn\}}$$

démonstration :

Nous allons démontrer le théorème dans le cas d'un système composé de deux processus communicants X et Y.

Fait 1 : exec $(X::x /\!/ Y::y, \langle \eta_X, \eta_Y \rangle) =$

si $x = nop \wedge y = nop$

alors $\langle \eta''_X, \eta''_Y \rangle$

sinon exec $(X::x' /\!/ Y::y', \langle \eta'_X, \eta'_Y \rangle)$

fsi

où x, x', y, y' des termes de la sorte Modif,

X, Y des termes de la sorte Idprocess, et

$\eta_X, \eta'_X, \eta''_X, \eta_Y, \eta'_Y, \eta''_Y$ des termes de la sorte Universe

avec x', y', η'_X et η'_Y sont décrits par une étude de cas sur la forme de x, y, η_X et η_Y,

et η''_X, η''_Y sont des univers terminaux de la forme $\langle \text{memend}(\eta_Z), \text{datend}(\eta_Z), nop \rangle$ où Z est l'un des identificateurs de processus X, Y.

Fait 2 : on dispose des esquisses de preuve relatives aux composants du système parallèle, ainsi que des assertions relatives à la satisfaction des communications de ces composants.

Autrement dit, on sait que

- $I[\![P_X]\!]\eta_X \qquad I[\![Q_X]\!]\eta'_X$
- $I[\![P_Y]\!]\eta_Y \qquad I[\![Q_Y]\!]\eta'_Y$
- Pour toute paire de communications complémentaires dans X et Y

  si $\{\alpha_X\}\ X.Y!e\ \{\beta_X\}$, $\{\alpha_Y\}\ Y.X?v\ \{\beta_Y\}$,

  alors $(\alpha_X \wedge \alpha_Y) \supset (\beta_X \wedge \beta_Y)\ \langle v/e \rangle$

Remarque :
Pour toute assertion R et tout univers η, si les variables de R apparaissent libre dans η alors $I\ [\![R]\!]\ \eta$ = true.

Idée de la démonstration :
Etant donnée la définition récursive de la fonction "exec" (fait 1), utiliser un raisonnement par récurrence (induction point fixe) sur le nombre d'appel de cette fonction.

$$J\ [\![P_X \wedge P_Y]\!]\ \langle \eta_X, \eta_Y \rangle^\circ \supset$$
$$\text{exec}^i\ (X::x\ //\ Y::y, \langle \eta_X, \eta_Y \rangle^{i-1} = \langle \eta_X, \eta_Y \rangle^i \supset$$
$$J\ [\![Q_X \wedge Q_Y]\!]\ \langle \eta_X, \eta_Y \rangle^\eta$$

sous les hypothèses

H1 : $I\ [\![P_X]\!]\ \eta_X^\circ \supset I\ [\![Q_X]\!]\ \eta_X^\eta$

H2 : $I\ [\![P_Y]\!]\ \eta_Y^\circ \supset I\ [\![Q_Y]\!]\ \eta_Y^\eta$

H3 : satisfaction des communications à l'étape i, noté $\text{coopère}^i\ (\text{preuve}_X, \text{preuve}_Y)$

i) l'étape de base de l'induction vient immédiatement du fait de la convention établie lors de la définition de la fonction d'interprétation J.

ii) l'étape d'induction, suppose la démonstration du théorème à l'étape i, et le prouve à l'étape i + 1.

$$\langle \eta_X, \eta_Y \rangle^{i+1} = \text{exec}\ (X::x\ //\ Y::y, \langle \eta_X, \eta_Y \rangle^i)$$
$$= \text{exec}\ (X::x\ //\ Y::y, \text{exec}^i\ (X::x'\ //\ Y::y', \langle \eta_X, \eta_Y \rangle^{i-1}))$$

sous les hypothèses :
- validité de l'étape i
- $J\ [\![P_X \wedge P_Y]\!]\ \langle \eta_X, \eta_Y \rangle^\circ$
- $\exists\ R_X, R_Y \in \text{Ast}$ tq $J\ [\![R_X \wedge R_Y]\!]\ \langle \eta_X, \eta_Y \rangle^i$

- Cas 1 : $\langle \eta_X, \eta_Y \rangle^i$ est un état parfait (parce que les univers η_X et η_Y qui le composent sont tous les deux parfaits).

. Cas 1.1 : x et y sont des modifications autres que des communications.
En supposant $\{R_X\}\ x\ \{S_X\}$ et $\{R_Y\}\ y\ \{S_Y\}$, il est facile, moyennant la définition algébrique de exec et la définition de la fonction I sur les états non parfaits, d'établir

$$I\ [\![R_X]\!]\ \eta_X^i \supset I\ [\![S_X]\!]\ \eta_X^{i+1}$$
$$I\ [\![R_Y]\!]\ \eta_Y^i \supset I\ [\![R_Y]\!]\ \eta_Y^{i+1}$$

Et comme les modifications impliquées par ce cas sont différentes d'actions de communication, les preuves de X et de Y coopèrent à cette étape.
Finalement, moyennant la remarque ci-dessus et la définition de J

$$J\ [\![R_X \wedge R_Y]\!]\ \langle \eta_X, \eta_Y \rangle^i \supset J\ [\![S_X \wedge S_Y]\!]\ \langle \eta_X, \eta_Y \rangle^{i+1}$$

or par hypothèse de l'étape i

$$J\ [\![P_X \wedge P_Y]\!]\ \langle \eta_X, \eta_Y \rangle^\circ \supset J\ [\![R_X \wedge R_Y]\!]\ \langle \eta_X, \eta_Y \rangle^i$$

le théorème est alors démontré pour ce cas

$$J\ [\![P_X \wedge P_Y]\!]\ \langle \eta_X, \eta_Y \rangle^\circ \supset$$
$$J\ [\![R_X \wedge R_Y]\!]\ \langle \eta_X, \eta_Y \rangle^i \supset J\ [\![S_X \wedge S_Y]\!]\ \langle \eta_X, \eta_Y \rangle^{i+1} \supset$$
$$J\ [\![Q_X \wedge Q_Y]\!]\ \langle \eta_X, \eta_Y \rangle^\eta$$

. Cas 1.2 : x est une action de communication et y est une modification simple, autre qu'une communication.
D'après la définition algébrique de exec, on doit trouver

$$\langle \eta_X, \eta_Y \rangle^{i+1} = \langle \eta_X^i, \text{apply}\ (y, \eta_Y^i) \rangle$$

En supposant $\{R_Y\}\ y\ \{S_Y\}$, il est évident que

$$I\ [\![R_Y]\!]\ \eta_Y^i \supset I\ [\![S_Y]\!]\ \eta_Y^{i+1}.$$

La communication n'ayant pas lieu, l'hypothèse de la coopération des preuves de X et Y, héritée de l'étape précédente, reste valide à cette étape.

De plus, moyennant le fait que

$$\eta_X^i = \eta_X^{i+1} \quad , \quad I\,[\![R_X]\!]\,\eta_X^i \supset I\,[\![R_X]\!]\,\eta_X^{i+1} \ .$$

Ainsi $J\,[\![R_X \wedge R_Y]\!]\,\langle \eta_X, \eta_Y\rangle^i \supset J\,[\![R_X \wedge S_Y]\!]\,\langle \eta_X, \eta_Y\rangle^{i+1}$, et par hypothèse d'induction on démontre le théorème pour ce cas :

$$J\,[\![P_X \wedge P_Y]\!]\,\langle \eta_X, \eta_Y\rangle^{\circ}$$

$$J\,[\![R_X \wedge R_Y]\!]\,\langle \eta_X, \eta_Y\rangle^i \supset J\,[\![R_X \wedge S_Y]\!]\,\langle \eta_X, \eta_Y\rangle^{i+1}$$

$$J\,[\![Q_X \wedge Q_Y]\!]\,\langle \eta_X, \eta_Y\rangle^n$$

. Cas 1.3 : y est action de communication et x est une modification simple.

Démonstration analogue à celle du cas 1.2.

. Cas 1.4 : x est un envoi d'un message et y est une réception de message.

D'après la définition algébrique, en supposant x = X.Y!e et y = Y.X?v, on a

$$\langle \eta_X, \eta_Y\rangle^{i+1} = \langle \eta_X^i,\ \text{substituer}\,(\eta_Y^i,\ \text{valvar}(v),\ \text{evaluer}(e, \eta_X^i))\rangle$$

En supposant $\{R_X\}$ x $\{S_X\}$ et $\{R_Y\}$ y $\{S_Y\}$, et l'hypothèse de coopération des preuves de X et Y à l'étape i,

$$\text{coopère}^{i+1}\,(\text{preuve}_X, \text{preuve}_Y) = \text{coopère}^i\,(\text{preuve}_X, \text{preuve}_Y) \wedge ((R_X \wedge R_Y) \supset (S_X \wedge S_Y)\ \langle v/e\rangle)$$

Par hypothèse on a $J\,[\![R_X \wedge R_Y]\!]\,\langle \eta_X, \eta_Y\rangle^i$, $I\,[\![R_X]\!]\,\eta_X^i \supset I\,[\![S_X]\!]\,\eta_X^{i+1}$, $I\,[\![R_Y]\!]\,\eta_Y^i \supset I\,[\![S_Y]\!]\,\eta_Y^{i+1}$ et $(R_X \wedge R_Y) \supset (S_X \wedge S_Y)\ \langle v/e\rangle$.

Il parait alors évident que l'on ait :

$$J\,[\![(S_X \wedge S_Y)\ \langle v/e\rangle]\!]\,\langle \eta_X, \eta_Y\rangle^i .$$

Moyennant la définition de J et la remarque en début de ce paragraphe on va avoir $I\,[\![S_X\ \langle v/e\rangle]\!]\,\eta_X^i$ et $I\,[\![S_Y\ \langle v/e\rangle]\!]\,\eta_X^i$.

D'après la définition axiomatique de l'affectation on va avoir

$$I\,[\![S_X\ \langle v/e\rangle]\!]\,\eta_X^i \supset I\,[\![S_X]\!]\,\eta_X^{i+1}$$

et $I\,[\![S_Y\ \langle v/e\rangle]\!]\,\eta_Y^i \supset I\,[\![S_Y]\!]\,\eta_Y^{i+1}$

Finalement sachant que $\eta_X^{i+1} = \eta_X^i$ et moyennant la définition de J, on a

$$J\,[\![R_X \wedge R_Y]\!]\,\langle \eta_X, \eta_Y\rangle^i \supset J\,[\![S_X \wedge S_Y]\!]\,\langle \eta_X, \eta_Y\rangle^{i+1}$$

et ainsi on a démontré le théorème pour ce cas.

. Cas 1.5 : x est une réception d'un message et y est l'envoi de ce message.

Démonstration analogue au cas 1.4.

. Cas 1.6 : x et y sont des actions de communications non complémentaires.

η_X^{i+1} et η_Y^{i+1} sont alors des univers avortés.

Par hypothèse $J\,[\![R_X \wedge R_Y]\!]\,\langle \eta_X, \eta_Y\rangle^i$ et les preuves de X et Y coopèrent à l'étape i+1.

Ici puisque x et y ne sont pas complémentaires, il y a violation de la loi du miracle et on a :

$$\text{coopère}^{i+1}\,(\text{preuve}_X, \text{preuve}_Y) = \text{coopère}^i\,(\text{preuve}_X, \text{preuve}_Y) \wedge \text{false}$$

Donc nous pouvons écrire

$$J\,[\![R_X \wedge R_Y]\!]\,\langle \eta_X, \eta_Y\rangle^i \supset J\,[\![\text{false}]\!]\,\langle \eta_X, \eta_Y\rangle^{i+1} \supset J\,[\![Q_X \wedge Q_Y]\!]\,\langle \eta_X, \eta_Y\rangle^n$$

d'après la définition de J et l'axiome de l'avortement.

- Cas 2 : $\langle \eta_X, \eta_Y \rangle^i$ est imparfait (parce que l'un des univers composants est imparfait).
Supposons pour la démonstration, que η_X^i est parfait et η_Y^i est imparfait (la démonstration dans l'autre cas est analogue à celle qui suit).

. Cas 2.1 : x est une modification simple, autre qu'une action de communication.
En supposant $\{R_X\}$ x $\{S_X\}$, il est clair que

$$I [\![R_X]\!] \eta_X^i \supset I [\![S_X]\!] \eta_X^{i+1}$$

où η_X^{i+1} est calculé d'après la définition algébrique de exec.
Par hypothèse on a également $J [\![R_X \wedge R_Y]\!] \langle \eta_X, \eta_Y \rangle^i$ et les preuves de X et Y coopèrent à l'étape i.
Comme la modification s impliquée par ce cas n'est pas une action de communication, la coopération des preuves à l'étape i + 1 est la même que celle de l'étape précédente.
D'après la définition algébrique de exec, l'univers η_Y^{i+1} est soit le même que l'univers η_Y^i, soit l'univers parfait de η_Y^i. Dans tous les cas

$$I [\![R_Y]\!] \eta_Y^i \supset I [\![R_Y]\!] \eta_Y^{i+1}$$

Ainsi, moyennant la définition de J et les hypothèses on a

$$J [\![R_X \wedge R_Y]\!] \langle \eta_X, \eta_Y \rangle^i \supset J [\![S_X \wedge R_Y]\!] \langle \eta_X, \eta_Y \rangle^{i+1}$$

Et d'après l'hypothèse d'induction,

$$J [\![P_X \quad P_Y]\!] \langle \eta_X, \eta_Y \rangle^\circ$$
$$J [\![R_X \wedge R_Y]\!] \langle \eta_X, \eta_Y \rangle^i \supset J [\![S_X \wedge R_Y]\!] \langle \eta_X, \eta_Y \rangle^{i+1} \supset$$
$$J [\![Q_X \wedge Q_Y]\!] \langle \eta_X, \eta_Y \rangle^n$$

. Cas 2.2 : x est une action de communication
D'après la définition algébrique de exec on devra avoir
$\langle \eta_X, \eta_Y \rangle^i = \langle \eta_X, \langle \text{memend}(\eta_Y), \text{datend}(\eta_Y), \text{nop} \rangle \rangle^{i+1}$
Par hyptohèse, on a $J [\![R_X \wedge R_Y]\!] \langle \eta_X, \eta_Y \rangle^i$ et validité de la coopération des preuves de X et Y à l'étape i.
Il est clair d'après la définition de J, $I [\![R_X]\!] \eta_X^i$ et $I \|R_Y\| \eta_Y^i$.
Par définition même de la fonction I, on peut écrire

$$I [\![R_Y]\!] \eta_Y^i = I [\![R_Y]\!] \eta_Y^{i+1}$$

Ainsi, moyennant ces résultats

$$J [\![R_X \wedge R_Y]\!] \langle \eta_X, \eta_Y \rangle^i \supset J [\![R_X \wedge R_Y]\!] \langle \eta_X, \eta_Y \rangle^{i+1}$$

ce qui démontre le théorème dans ce cas :

$$J [\![P_X \wedge P_Y]\!] \langle \eta_X, \eta_Y \rangle^\circ$$
$$J [\![R_X \wedge R_Y]\!] \langle \eta_X, \eta_Y \rangle^i \supset J [\![R_X \wedge R_Y]\!] \langle \eta_X, \eta_Y \rangle^{i+1}$$
$$J [\![Q_X \wedge Q_Y]\!] \langle \eta_X, \eta_Y \rangle^n$$

3.2.4.5. En guise de conclusion

Nous venons d'exposer la sémantique axiomatique de LPPSD, ainsi que sa complémentarité avec la sémantique algébrique.
Si la formalisation axiomatique est supposée être la spécification définitive de cette partie du langage, alors le modèle algébrique est prouvé être l'implantation (abstraite et) correcte de cette partie ; ainsi les preuves établissent qu'il ne se produira jamais un résultat qui falsifie un théorème de la sémantique axiomatique.

Si par contre, la formalisation algébrique est supposée être la spécification définitive de cette partie de langage, la formalisation axiomatique est alors fournie pour donner une technique de preuve valide, qui ne permettra jamais de déduire un faux théorème. Cependant, la formalisation axiomatique n'est pas suffisamment riche pour établir toutes les propriétés des programmes écrits en LPPSD.

Bien sûr, il n'est pas du souci du spécifieur de choisir la formalisation définitive. L'existence de preuve de complémentarité montre que toutes les définitions du même morceau de langage sont dans un sens compatibles et que l'on peut utiliser à n'importe quel moment une d'elles selon le but choisi. Par exemple, la formalisation axiomatique pour prouver des propriétés de programmes, la formalisation algébrique pour guider l'implantation.

Finalement, il est à noter que le système axiomatique présenté est à critiquer comme tout système de logique à la Hoare ; logique qui, malgré tout, a le mérite de permettre une mise en oeuvre de méthodes de preuve pour des classes de programmes de plus en plus grands, dont certaines ont pu aboutir à des systèmes plus ou moins automatiques.
Donc, comme pour tous les systèmes formels, se posent pour celui que nous avons présenté les problèmes de consistance et de complétude [WAN 78, CLA 79, DON 82]. N'ayant nullement l'intention de nous étendre sur l'étude de tels problèmes dans le cadre de cette étude, nous nous contentons

i) d'affirmer que les axiomes A1 à A3 et les règles d'inférence R1 à R4 forment un système suffisamment puissant (d'après [COO 78]) pour prouver toute instruction correcte d'un programme itératif et séquentiel.

ii) de nous référer à [APT 80] pour ce qui concerne l'étude du système complet. Le seul reproche que nous faisons à cette méthode de preuve est le nombre de cas nécessaire pour tester la coopération qui, par exemple, pour deux processus se résoud en m1 * m2 tests où m1 et m2 sont proportionnels à leur longueur.

4. REMARQUES

Au terme de ce chapitre, nous souhaitons évaluer l'apport de l'outil de spécification algébrique pour la formalisation de la sémantique de LAGALERE. Il convient alors de noter que

i) le caractère formel de l'outil donne une définition rigoureuse de la sémantique du langage et favorise l'existance d'un cadre théorique sain où les preuves de programmes peuvent être menées.

ii) le caractère d'abstraction de l'outil, permet de ne faire intervenir aucune contrainte sur la classe des langages de programmation objets, le type de machines supportant le langage, etc. De plus il permet de n'avoir qu'un seul cadre cohérent où structures de données et structures de contrôle peuvent à la fois être spécifiées.

iii) le caractère structurant et évolutif de l'outil permet d'entreprendre une formalisation selon une méthode descendante et par niveau (construction hiérarchisée du type associé à la formalisation du langage). De plus il permet en général de faire correspondre un changement minime dans la spécification si l'on admet un changement minime dans le langage.

iv) l'outil permet d'estimer a priori la difficulté de réalisation d'un compilateur pour le langage en mettant en évidence les points cruciaux du non déterminisme et de communication.

v) l'outil permet aussi de donner une représentation concrète des objets sémantiques (des arbres décrits par des expressions composées à l'aide des constructeurs à partir d'un objet initial) sur lesquels est offerte la possibilité d'effectuer des calculs.

vi) l'outil permet de mieux comprendre les concepts à formaliser. Nous avons en effet remarqué que la sémantique de l'algèbre initiale ne garantit pas la correction totale de l'interprète du langage, car elle n'exprime rien sur la non terminaison. Par contre, si on considère des modèles prolongeant un modèle du type de base on y rend équivalents les termes sans forme normale. D'autre part, choisir un modèle terminal revient à rendre équivalentes les expressions ayant toujours la même valeur (par les opérations d'évaluation), les états où les expressions ont la même valeur, les modifications ayant le même effet (par les opérations d'interprétation). On peut ainsi se limiter à rendre équivalents parmi les états sans forme normale, ceux qui conduisent à un même calcul infini.

CONCLUSION

CONCLUSION

Il convient maintenant de faire le point sur cette contribution à l'étude de la sémantique du parallélisme, et d'esquisser les problèmes ouverts et les prolongements que nous envisageons pour nos futures recherches.

Notre préoccupation de départ fut l'étude du parallélisme et de ses problèmes tant par leurs mises en oeuvre dans les langages de programmation que par leurs spécifications dans des modèles d'expressions adéquats.

Un survol rapide de la littérature sur le parallélisme montre deux différentes approches : la première est fondée sur les primitives de synchronisation, la seconde est basée sur les opérations de communication. L'habilité de la seconde approche à fournir un support "méthodologique" adéquat pour le développement des programmes parallèles et de leurs preuves, nous a motivé pour entreprendre une telle expérience.

C'est alors que nous nous sommes penchés sur l'étude des fondements de la sémantique des langages de programmation. Il est intéressant de voir comment l'idée d'utiliser les techniques de spécification algébrique pour la formalisation de la sémantique a pu émerger. La théorie des types abstraits est extrêmement passionnante puisqu'elle permet de spécifier des structures de données assez riches par des techniques équationnelles et récursives. Le caractère très systématique de l'application d'une telle technique à la formalisation des langages de programmation séquentielle, nous a donné une direction de travail précise : étendre ces résultats aux langages supportant le nondéterminisme et certaines figures du parallélisme, notamment de la communication des processus.

Nous avons rencontré, au cours du développement de la formalisation du petit langage considéré (LAGALERE), la nécessité que les dates soient explicitement manipulées. Nous définissons celles-ci dans une spécification de base sur laquelle nous pouvions définir les opérations liées à l'exécution des modifications élémentaires relatives au processus.

Cependant, nous avons le sentiment d'avoir, au cours de ce développement, noté des résultats auxiliaires qui devront être détaillés, c'est le cas par exemple de l'applicabilité de l'outil (sémantique temporelle) aux systèmes temps réel, ou encore de la possibilité d'optimiser "les temps de réponse" dans le cas des systèmes de processus communicants par le mécanisme du rendez-vous (lorsque le résultat de l'action de communication n'est pas promptement utilisé par les modifications qui la succèdent), mais aussi de la complémentarité des formalisations axiomatiques et algébriques.

Néanmoins le travail le plus urgent à mener doit être l'étude d'une implantation (nous avons d'ailleurs tenu à montrer en annexe 4 les résultats d'une implantation en PASCAL) des idées principales de notre approche "sémantique temporisée". En même temps que ce travail nous désirons étudier :

i) l'applicabilité de la technique des spécifications algébriques pour la prise en compte des notions de définition de types de structures de données, de programmation "modulaire", et de processus dynamiques.

ii) l'apport de l'idée de manipulation explicite des notions temporelles liées à la simultanéité des "calculs" des processus pour "bien spécifier" des problèmes parallèle et fournir une aide méthodologique pour la mise en oeuvre de tels problèmes.

Nous imaginons que ce travail apporte une contribution, minime quelle soit, à l'étude du parallélisme et de sa formalisation même si de nombreux problèmes seront soulevés et resteront ouverts.

ANNEXES

ANNEXE I

SYNTAXE DE LAGALERE

1. Méta-langage et méta-symboles

La syntaxe est décrite dans un méta-langage de type BNF ; la signification des méta-symboles est la suivante :

méta-symbole	signification
::=	est défini comme
|	ou comme
{ }*	peut être répété 0,1 ou plusieurs fois
{ }+	peut être répété 1 ou plusieurs fois
{ }°	optionnel

2. Eléments lexicaux

Les programmes sont construits à partir du jeu suivant :
- les 26 lettres majuscules,
- les 26 lettres minuscules,
- les 10 chiffres décimaux,
- les caractères spéciaux suivants :

  : . , ; ( ) + - * / = > < □
- le caractère espace

Il y a quatre sortes de constituants lexicaux :
- les délimiteurs
- les mots clés (en majuscule dans la grammaire)
- les identificateurs
- les constantes littérales (true et false)

Il est possible d'insérer des commentaires dans un programme. Un commentaire commence par une accolade ouvrante et s'étend jusqu'à la fin de la ligne où figure une accolande fermante.

3. Syntaxe

programme ::= seqprog | parprog

seqprog ::= PROGRAM identif {; declvar}$^{+}$; BEGIN seqbody END.

declvar ::= VAR identif {, identif}* : type

type ::= boolean | integer

seqbody ::= instruct {; instruct}*

instruct ::= NOP | variable := expression |
ALTOR seqbody □ seqbody ROTLA |
IF expression THEN seqbody ELSE seqbody FI |
WHILE expression DO seqbody OD

parprog ::= PROGRAM identif ; declprocess {// declprocess}$^{+}$ END.

delprocess ::= PROCESS identif {; declvar}$^{+}$;
BEGIN parbody END ;

parbody ::= command {; command}*

command ::= NOP | ABORT | variable := expression |
SEND expression TO identif |
RECEIVE variable FROM identif |
IF expression THEN parbody ELSE parbody FI |
WHILE expression DO parbody OD |
ALTOR parbody □ parbody ROTLA

variable ::= identif

expression ::= boolterm {OR boolterm}$^{+}$

boolterm ::= boolfactor {AND boolfactor}$^{+}$

boolfactor ::= simpleexp {comparop simpleexp}$^{+}$

simpleexp ::= {addop}$^{\circ}$ term {addop term}$^{+}$

term ::= factor {multop factor}$^{+}$

factor ::= {NOT}$^{\circ}$ primar

primar ::= variable | constant | (expression)

addop ::= + | -

multop ::= * | /

comparop ::= = | <> | > | < | >= | <=

identif ::= letter {letter | digit}*

letter ::= A | B | C | ... | Z | a | b | c | ... | z

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

constant ::= number | litteral

number ::= digit | number digit

litteral ::= true | false

ANNEXE II

AXIOMATISATION DE LPP SYNCHRONE

L'axiomatisation qui suit généralise la définition de l'opération exec : Agent x State → State, décrite au § 2.4.1. ch. IV pour formaliser la sémantique des liens des programmes parallèles, pour la prise en compte des systèmes constitués de n agents communiquants ($n \geq 2$).

Nous avons basé cette axiomatisation sur les faits que

- d'une part, il faut profiter au maximum de la commutativité et de l'associativité de l'opérateur du parallélisme, en ce sens que dans la définition de l'opération "exec" les agents les plus intéressants (i.e. ceux qui communiquent) apparaissent en tête,
- d'autre part, le calcul de la date minimale de la fin d'exécution d'un agent est calculé en posant deux hypothèses à propos des communications.

i) la date de fin d'un envoi ou d'une réception d'un message est a priori inconnue (infinie) si l'on considère les agents indépendamment les uns des autres.

ii) la date de fin de la transmission d'un message est une constante connue (plutôt supposée l'être) dans le cas où deux agents sont aptes à se communiquer un message par l'intermédiaire d'actions de communication complémentaires.

La spécification de l'interprétation des agents de LPP synchrone est ainsi formulée :

Opns : exec : Agent x State → State ;
Eqns : A ∈ Agent ; Σ_A ∈ State ;
stopped(A) ∨ aborted(A) = true ⇒
exec(A, Σ_A) = Σ_A ;
stopped(A) ∨ aborted(A) = false ⇒
exec(A, Σ_A) = exec($proj_1$(activ(A, Σ_A)),
$proj_2$(activ(A, Σ_A))) ;
Fin ;

Pour compléter cette spécification nous allons décrire les opérations auxiliaires qui y interviennent.

Opns : stopped : Agent → Bool ;
Eqns : A,B ∈ Agent ; X ∈ Idprocess ; x ∈ Modif ;
stopped(A // B) = stopped(A) ∧ stopped(B) ;
x = nop ⇒ stopped (X :: x) = true ;
¬(x = nop) ⇒ stopped (X::x) = false ;
Fin ;

Opns : aborted : Agent → Bool ;
Eqns : A,B ∈ Agent ; X ∈ Idprocess ; x ∈ Modif ;
aborded (A // B) = aborted (A) ∨ aborted (B) ;
x = abort ⇒ aborted (X::x) = true ;
¬(x = abort) ⇒ aborted (X::x) = false ;
Fin ;

Opns : $proj_1$: Agent x State → Agent ;
$proj_2$: Agent x State → State ;
Eqns : A ∈ Agent ; Σ_A ∈ State ;
$proj_1$ (<A, Σ_A>) = A ;
$proj_2$ (<A, Σ_A>) = Σ_A ;
Fin ;

Opns : activ : Agent x State → Agent x State

% fonction de "transition" permettant de décrire l'évolution pas à pas d'un agent dans son propre état %

Eqns :

activ (X:: if b then x_1 else x_2 fi ; x // A, <n_X, Σ_A>) =
si eval (b, n_X)
alors activ (X::x_1 ; x // A, <n_X, Σ_A>)
sinon activ (X::x_2 ; x // A, <n_X, Σ_A>)
fsi ;

¬(datefin (X::x_1 ; x_2, n_X) = infini) ⇒
activ (X::x_1 ; x_2 // A, <n_X, Σ_A>) =
si datefin (X::x_1 ; x_2, n_X) ≤ datefin (A, Σ_A)
alors << X::x_2 // $proj_1$ (activ(A, Σ_A))>,
<apply (x_1, n_X), $proj_2$ (activ (A, Σ_A)) >>
sinon << X::x_2 //$proj_1$ (activ(A, Σ_A))>,
<< memend (n_X), datend (n_X), x_1>, $proj_2$ (activ(A, Σ_A))>>
fsi ;

```
datefin (X::x1 ; x2, nX) = infini ∧ x1 = X.Y! ex  ⇒
 activ (X::X.Y! ex ; x2 // Y::y1 ; y2 // A, <<nX, nY>, ΣA>) =
 si datefin (Y::y1 ; y2, nY) = infini
 alors si y1 = Y.X? vy
       alors si cstcomm ⩽ datefin (A, ΣA)
             alors <<X::x2 // Y::y2 // proj1 (activ (A, ΣA))>,
                    <<nX,
                      subst (nY, vy, eval(ex, nY)) >,
                      proj2 (activ (A, ΣA)) >>
             sinon <<X::x2 // Y::y2 // proj1 (activ (A, ΣA)) >,
                    <<nX,
                      ensubst (nY, vy, eval(ex, nX)) >,
                      proj2 (activ (A, ΣA)) >>
             fsi
       sinsi y1 = Y.X! ey
       alors <<X::abort ; x1 ; x2 // Y::abort ; y1 ; y2 // A>,
                                          <<nX, nY>, ΣA>>
       sinon <<X::x1 ; x2 // proj1 (activ (Y::y1 ; y2 // A, < Y, ΣA>))>,
              < X, proj2 (activ (Y::y1 ; y2 // A, <nY, ΣA>))>>
       fsi
 sinon <<X::x1 ; x2 // proj1 (activ (Y::y1 ; y2 // A, <nY, ΣA>))>,
              < X, proj2 (activ (Y::y1 ; y2 // A, <nY, ΣA>))>>
 fsi ;
```



```
datefin (X::xY ; x2, nX) = infini ∧ x1 = X.Y? vx  ⇒
 activ (X::X.Y? vx ; x2 // Y::y1 ; y2 // A, <<nX, nY>, ΣA>) =
 si datefin (Y::y1 ; y2, nY) = infini
 alors si y1 = Y.X! ey
       alors si cstcomm ⩽ datefin (A, ΣA)
             alors <<X::x2 // Y::y2 // Proj1 (activ (A, ΣA)) >,
                    <<subst (nX, vx, eval (ey, nY)),
                      nY >,
                      proj2 (activ (A, ΣA)) >>
             sinon <<X::x2 // Y::y2 // proj1 (activ (A, ΣA)) >,
                    <<ensubst (nX, vx, eval (ey, nY)),
                      nY >,
                      proj2 (activ (A, ΣA)) >>
             fsi
       sinsi y1 = Y.X? vy
       alors <<X::abort ; x1 ; x2 // Y::abort ; y1 ; y2 // A>,
                                          <<nX, nY>, ΣA>>
       sinon <<X::x1 ; x2 // proj1 (activ (Y::y1 ; y2 // A, < nY, ΣA>))>,
              <nX, proj2 (activ (Y::y1 ; y2 // A, <nY, ΣA>))>>
       fsi
 sinon <<X::x1 ; x2 // proj1 (activ (Y::y1 ; y2 // A, <nY, ΣA >))>,
              <nX, proj2 (activ (Y::y1 ; y2 // A, <nY, ΣA>))>>
 fsi ;

Fin ;
```

Pour décrire l'opération "activ" nous avons fait une étude par cas des modifications à entreprendre par un programme quelconque. Ainsi lorsque nous supposons qu'un agent est apte à communiquer (cas où l'opération "datefin" donne un résultat égal à l'infini), nous nous intéressons à l'agent avec qui il veut communiquer. Deux cas se présentent :

- soit, ce dernier veut entreprendre une action de communication complémentaire à celle spécifiée par le premier et dans ce cas nous établissons la communication par une affectation,
- soit, ce dernier ne possède pas d'action complémentaire à celle spécifiée par le premier et dans ce cas nous établissons l'attente du premier sur son action de communication.

Par contre, lorsque la modification est supposée être différente d'une action de communication, nous nous intéressons à sa date de fin d'exécution pour pouvoir la comparer avec celle des autres agents s'exécutant en parallèle et décrire ainsi le prochain état du système dans lequel au moins un agent possède un univers parfait.

La description de l'opération "activ" a été menée sous l'hypothèse de la spécification des opérations auxiliaires "datefin" et "ensubst" dont voici les opérations :

```
Opns : ensubst : Universe x Var x Value → Universe ;
Eqns :
    ensubst (η_X, vid, val) = <mem (η_X), date (η_X), vid := val>
Fin ;
```

```
Opns : min : Time x Time → Time
Eqns :
    min (t1, t2) = si t1 ⩽ t2 alors t1 sinon t2 fsi
Fin ;
```



```
Opns : datefin : Agent x Agent → Time ;
       % calcule la date de fin d'exécution de toute action
         d'un agent donné selon les hypothèses fixées au début
         de l'annexe %

Eqns :
    datefin (X::x, η_X) =
    si mod (η_X) = nop
    alors cas x de nop, abort, v := e, if b then x_1 else x_2 fi
                alors datend (apply(x, η_X))
                sinon infini
          fcas
    sinon datend (η_X)
    fsi ;
```

```
    datefin (X::x // A, <η_X, Σ_A>) =
    si ¬(datefin (X::x, η_X) = infini)
    alors min (datefin (X::x, η_X), datefin (A, Σ_A))
    sinon datefin (A, Σ_A)
    fsi ;
Fin ;
```

ANNEXE III

EXEMPLE DE PREUVE :
PARTITION DE LA REUNION DE DEUX ENSEMBLES

Enoncé

Soient S et T deux ensembles disjoints non vides. On veut partitionner la réunion de S et T en deux ensembles S' et T' de telle façon que :

i) #S = #S' et #T = #T' où #X est le cardinal de l'ensemble X,

ii) tout élément de S' soit plus petit que tout élément de T'.

Hypothèses

Nous supposerons que LAGALERE supporte la structure de données "ensemble" (SET OF type) et les opérations de leurs manipulations.

Solution séquentielle

Soient MAXI(X) et MINI(X) les fonctions donnant respectivement l'élément maximal et minimal de l'ensemble X.

```
PROGRAM partisionseq ;
VAR S,T : SET OF integer ;
    mn, mx : integer ;
BEGIN
        mx := MAXI(X) ; mn = MINI (T) ;
        WHILE mx > mn DO
           S := S \ [mx] ;  T := T U [mx] ;
           mn := MINI (T) ; T := T \ [mn] ; S := S U [mn] ;
           mx := MAXI (S) ;
        OD ;
END ;
```

Solution parallèle

On considère deux processus parallèles PART1 et PART2 qui s'échangeront le maximum courant de S et le minimum courant de T jusqu'à ce que ces valeurs courantes soient égales aux anciennes valeurs.

```
PROGRAM partitionpar ;
  PROCESS part1 ;
  VAR S : SET OF integer ;
      mx1, mn1 : integer ;
      b1 : boolean ;
  BEGIN
      b1 := false ;
      WHILE not(b1) DO
            mx1 := MAXI(S) ;
            SEND mx1 TO part2 ; S := S \ [mx1] ;
            RECEIVE mn1 FROM part2 ; S := S U [mn1] ;
            b1 := (mx1 = mn1) ;
      OD ;
  END ;

// PROCESS part2 ;
  VAR T : SET OF integer ;
      mx2, mn2 : integer ;
      b2 : boolean ;
  BEGIN
      b2 := false ;
      WHILE not(b2) DO
            RECEIVE mx2 FROM part1 ; T := T U [mx2] ;
            mn2 := MINI (T) ;
            SEND mn2 TO part1 ; T := T \ [mn2] ;
            b2 := (mx2 = mn2) ;
      OD ;
  END ;

END.
```


Preuve du programme séquentiel

On supposera que MAXI(0) = - ∞

{S ∩ T = ∅ & #S > 0 & #T > 0}
{S ∩ T = ∅ & #S = n > 0 & #T = m > 0 & MAXI(S) ∈ S}

mx := MAXI(S) ;

{S ∩ T = ∅ & #S = n > 0 & #T = m > 0 & mx = MAXI(S) & MINI(T) ∈ T}

mn := MINI(T) ;

{S ∩ T = ∅ & #S = n > 0 & #T = m > 0 & mx = MAXI(S) & mn = MINI(T)
& mn ∈ T & mx ∈ S}

WHILE mx > mn DO

{S ∩ T = ∅ & #S = n > 0 & #T = m > 0 & mx = MAXI(S) & MN = MINI(T)
& mx ∈ S & mn ∈ T & mx > mn}

S := S \ [mx] ;

{S ∩ T = ∅ & #S = n-1 >= 0 & #T = m > 0 & mx ∉ T & mx > MAXI(S)
& mn = MINI(T) & mx > mn}

T := T U [mx] ;

{S ∩ T = ∅ & #S = n-1 > 0 & #T = m+1 > 0 & mx > MAXI(S) & mx > mn
& mn = MINI(T) & MINI(T) ∈ T}

mn := MINI(T) ;

{S ∩ T = ∅ & #S = n-1 >= 0 & #T = m+1 > 0 & mx > MAXI(S) & mn ∈ T
mn = MINI(T)}

T := T \ [mn] ;

{S ∩ T = ∅ & #S = n-1 >= 0 & #T = m > 0 & mx > MAXI(S) & mn ∉ S
mn < MINI(T)}

S := S U [mn] ;

{S ∩ T = ∅ & #S = n > 0 & #T = m > 0 & mx > MAXI(S) & mn ∈ S
MAXI(S) ∈ S & mn < MINI(T)}

mx := MAXI(S) ;

{S ∩ T = ∅ & #S = n > 0 & #T = m > 0 & mx S & mx = MAXI(S)
& mn < MINI(T)}

OD ;

{mx <= mn & S ∩ T = ∅ & #S = n > 0 & mx = MAXI(S) & mn > MINI(T)}
{S ∩ T = ∅ & #S > 0 & #T > 0 & MAXI(S) < MINI(T)}

Preuve du programme parallèle

a) esquisse de preuve du processus PART1 :

On supposera que MAXI(∅) = - ∞

{S ≠ ∅}

b1 := false ;

{#S = n > 0 & b1 = false}

WHILE not(b1) DO

{#S = n > 0 & b1 = false & MAXI(S) ∈ S}

mx1 := MAXI(S) ;

{#S = n > 0 & b1 = false & mx1 ∈ S & MAXI(S)∈ S & mx1 = MAXI(S)}

SEND mx1 TO part2 ;

{#S = n > 0 & b1 = false & mx1 ∈ S & MAXI(S)∈ S & mx1 = MAXI(S)}

S := S \ [mx1] ;

{#S = n-1 >= 0 & b1 = false & mx1 ∈ S & mx1 > MAXI(S)}

RECEIVE mn1 FROM part2 ;

{#S = n-1 >= 0 & b1 = false & mx1 > MAXI(S) & mn ∉ S
& mn1 <= MAXI(S)}

S := S U [mn1] ;

{#S = n > 0 & b1 = false & mn1 ∈ S & MAXI(S) ∈ S & mx1 >= MAXI(S)}

b1 := (mx1 = mn1) ;

{#S = n > 0 & MAXI(S)∈ S & mx1 >= MAXI(S) & b1 = (mn1 = mx1)}

OD ;

{#S = n > 0 & b1 = true & mn1 = mx1 & mx1 >= MAXI(S))
{S ≠ ∅ & mn1 = mx1 & mx1 >= MAXI(S)}

b) esquisse de preuve de processus PART2

{T ≠ ∅}

b2 := false ;

{#T = m > 0 & b2 = false}

WHILE not(b2) DO

{#T = m > 0 & b2 = false}

receive mx2 FROM part1 ;

{#T = m > 0 & b2 = false & mx2 ∉ T}

T := T U [mx2] ;

{#T = m+1 > 0 & b2 = false & mx2 ∈ T & MINI(T) ∈ T}

mn2 := MINI(T) ;

{#T = m+1 > 0 & b2 = false & mn2 ∈ T & mn2 = MINI(T)}

SEND mn2 TO part1 ;

{#T = m+1 > 0 & b2 = false & mn2 ∈ T & mn2 = MINI(T)}

T := T \ [mn2] ;

{#T = m > 0 & b2 = false & mn2 ∉ T & MINI(T) T & mn2 < MINI(T)}

b2 := (mx2 = mn2)

{#T = m > 0 & b2 = (mn2 = mx2) & mn2 < MINI(T) & MINI(T) ∈ T}

OD ;

{#T = m > 0 & b2 = true & mn2 = mx2 & mn2 < MINI(T)}
{T ≠ ∅ & mn2 = mx2 & mn2 < MINI(T)}

c) regroupement des esquisses

fait1 : {S ≠ ∅} PART1{S ≠ ∅ & mn1 = mx1 & mx1 >= MAXI(S)}
fait2 : {T ≠ ∅} PART2{T ≠ ∅ & mn2 = mx2 & mn2 < MINI(T)}
invariant global = {S ∩ T = ∅}

satisfaction de la communication :

i) {pré (PART2 ! mx1) & pré (PART1 ? mx2)}
{#S = n > 0 & mx1 ∈ S & MAXI(S) = mx1 & b1 = false
& #T = m > 0 & b2 = false}

SEND mx1 TO part1 // RECEIVE mx2 FROM part2

{#S = n > 0 & mx1 ∈ S & mx1 = MAXI(S) & b1 = false
& #T = m > 0 & b2 = false
& mx1 = mx2}
{#S = n > 0 & mx1 ∈ S & mx1 = MAXI(S) & b1 = false
& #T = m > 0 & b2 = false}
{ post (PART2 ! mx1) & post (PART1 ? mx2)

ii) {pré (PART2 ? mn1) & pré (PART1 ! mn2)}
{#S = n-1 >= 0 & b1 = false & mx1 ∈ S & mx1 > MAXI(S)
& #T = m+1 > 0 & b2 = false & mn2 T
& mn2 = MINI(T)}

RECEIVE mn1 FROM part2 // SEND mn2 TO part1

{#S = n-1 > 0 & b1 = false & mx1 ∉ S & mx1 > MAXI(S)
& #T = m+1 > 0 & b2 = false & mn2 ∈ T
& mn2 = MINI(T) & mn1 = mn2}
{#S = n-1 >= 0 & b1 = false & mx1 ∉ S & mx1 > MAXI(S)
& #T = m+1 > 0 & b2 = false & mn2 ∈ T
& mn2 = MINI(T)}
{ post (PART2 ? mn1) & post (PART1 ! mn2)}

Règle de la composition parallèle :

{pré (PART1) & pré (PART2) & $S \cap T = \emptyset$}
{$S \neq \emptyset$ & $T \neq \emptyset$ & $S \cap T = \emptyset$}

PART1 // PART2

{$S \neq \emptyset$ & mn1 = mx1 & mx1 >= MAXI(S)
& $T \neq \emptyset$ & mn2 = mx2 & mn2 < MINI(T)
& mn1 = mn2 & mx1 = mx2
& $S \cap T = \emptyset$}
{$S \neq \emptyset$ & $T \neq \emptyset$ & MAXI(S) > MINI(T) & $S \cap T = \emptyset$}

ANNEXE IV

UN PROTOTYPE DE DESCRIPTION DE LA SEMANTIQUE DE LAGALERE

Dans un but demonstratif, nous avons programmé en PASCAL un prototype d'interprétation de programmes écrits en LAGALERE. L'interprète que nous allons décrire fonctionne sur une mémoire commune selon les spécificités de la formalisation du chapitre IV.

Le prototype décrit pas à pas, pour chaque programme, l'évolution en parallèle de ses processus. A chaque étape, cette description est formulée par deux résultats :

i) l'état du système (état d'avancement des processus) avant l'exécution de la (les) prochaine(s) instruction(s) de plus courte durée.

ii) l'état du système après l'exécution d'une (de) telle(s) instruction(s). Cet état peut être annoncé par l'une des alternatives suivantes :

- une fin normale notée par la terminaison de tous les processus composant le programme considéré,
- une fin anormale notée par un interblocage des processus ou par une attente infinie d'un rendez-vous dans le cas de LAGALERE synchrone,
- une activation notée pour un processus donné par soit une suspension en attente d'un rendez-vous dans le cas de LAGALERE synchrone, soit par terminaison, soit enfin par une exécution (le processus est alors soit en train d'exécuter une instruction, soit apte à entamer une nouvelle).

PROGRAMMATION LAGALERE SYNCHRONE :
==========

HYPOTHESES:
---

1) action = 0 ==> instruction de base
   action = -1 ==> action de réception
   action = 1 ==> action d'envoi

2) les dates de lancement peuvent être différentes

3) la durée d'une communication lorsqu'elle peut avoir lieu = constante
   la durée d'une instruction de base autre que "ABORT" = quelconque
   la durée du "ABORT" = 0

4) la description qui suit donne la forme d'un agent et son univers:

   agent:
   ---
   nom agent
   ---
   modification: (action, nom agent)
   ---

   univers :

   état de l'univers de l'agent
   ---
   date courante
   ---
   date de fin d'execution de la modification courante

EXEMPLES:

COMBIEN D'AGENTS: 3

entrée des donnees initiales
==========

```
--> agent 1
   action: 1
  datedeb: 1
id2 agent: 3
--> agent 2
   action: 0
  datedeb: 2
    durée: 3
--> agent 3
   action: 0
  datedeb: 1
    durée: 2
```

$[\ A1::A3!e_1^1\ ;\ \ldots\ //$

$A2::v_1^2 := e_1^2\ ;\ abort\ //$

$A3::v_1^3 := e_1^3\ ;\ abort\ ]$

sortie résultats
==========

résultats 1

```
-  0 agents en communication
-  1 agents en attente
-  0 agents en interblocage
-  0 agents en cours d'exécution
-  2 agents en état parfait
EN ATTENTE:   1  3  1 OK   1 999
AUTRE   PARFAIT:   2 99  0 OK   2   5
AUTRE   PARFAIT:   3 99  0 OK   1   3
```

résultats 2

```
PROCHAINE ETAPE:
TIMING MINIMAL: 2
   ===> EN COURS: 2
   ===> TERMINE: 3
   ===> EN ATTENTE: 1
```

entrée des données pour la poursuite de l'exécution
==========

```
 POUR L'AGENT   3
    + action suivante: 0
    + durée action: 0
```

sortie résultats
==========

résultats 1

```
-  0 agents en communication
-  1 agents en attente
-  0 agents en interblocage
-  1 agents en cours d'exécution
-  1 agents en état parfait
EN ATTENTE:   1  3  1 OK   3 999
AUTRE IMPARFAIT:   2 99  0 NO   4   5
AUTRE   PARFAIT:   3 99  0 OK   3   3
```

resultats 2

```
PROCHAINE ETAPE:
TIMING MINIMAL: 1
   ===> TERMINE: 2
   ===> TERMINE: 3
   ===> EN ATTENTE: 1
```

entrée des données pour la poursuite de l'execution
==========

```
 POUR L'AGENT   2
    + action suivante: 0
    + durée action: 0
```

```
sortie résultats
================
   résultats 1
   -----------
 -  0 agents en communication
 -  1 agents en attente
 -  0 agents en interblocage
 -  0 agents en cours d'exécution
 -  2 agents en état parfait
 EN ATTENTE:    1  3  1 OK    4 999
 AUTRE   PARFAIT:    2 99  0 OK    5   5
 AUTRE   PARFAIT:    3 99  0 OK    3   3

   résultats 2
   -----------
PROCHAINE ETAPE:
    ###  ATTENTE INFINI  ###
```

```
 COMBIEN D'AGENTS: 4


entrée des données initiales
============================
 --> agent 1
    action: 1
   datedeb: 2
 id2 agent: 2
 --> agent 2
    action: -1
   datedeb: 1
 id2 agent: 1
 --> agent 3
    action: -1
   datedeb: 1
 id2 agent: 4
 --> agent 4
    action: 1
   datedeb: 3
 id2 agent: 3
```

$$[\ A1 :: A2!e_1^1 ; v_1^1 := e_2^1 ; A2?v_2^1 ; ...//$$

$$A2 :: A1?v_1^2 ; A1?v_2^2 ; ... //$$

$$A3 :: A4?v_1^3 ; A4!v_2^3 ; ... //$$

$$A4 :: A3!e_1^4 ; v_1^4 := e_2^4 ; ...$$

$$]$$

```
sortie résultats
================
   résultats 1
   -----------
 -  4 agents en communication
 -  0 agents en attente
 -  0 agents en interblocage
 -  0 agents en cours d'exécution
 -  0 agents en état parfait
 COMMUNICANTS:    1  2  1 OK    2 999
 COMMUNICANTS:    2  1 -1 OK    1 999
 COMMUNICANTS:    3  4 -1 OK    1 999
 COMMUNICANTS:    4  3  1 OK    3 999
```



```
   résultats 2
   -----------
PROCHAINE ETAPE:
TIMING MINIMAL: 2
   ===> TERMINE: 1
   ===> TERMINE: 2
   ===> TERMINE: 3
   ===> TERMINE: 4


entrée des données pour la poursuite de l'exécution
===================================================
 POUR L'AGENT   1
    + action suivante: 0
    + duree action: 2
 POUR L'AGENT   2
    + action suivante: -1
    + id2 agent : 1
 POUR L'AGENT   3
    + action suivante: 1
    + id2 agent : 4
 POUR L'AGENT   4
    + action suivante: 0
    + durée action: 5


sortie résultats
================
   résultats 1
   -----------
 -  0 agents en communication
 -  2 agents en attente
 -  0 agents en interblocage
 -  0 agents en cours d'exécution
 -  2 agents en état parfait
 EN ATTENTE:    2  1 -1 OK    3 999
 EN ATTENTE:    3  4  1 OK    3 999
 AUTRE   PARFAIT:    1 99  0 OK    4   6
 AUTRE   PARFAIT:    4 99  0 OK    5   7

   résultats 2
   -----------
PROCHAINE ETAPE:
TIMING MINIMAL: 2
   ===> TERMINE: 1
   ===> EN COURS: 4
   ===> EN ATTENTE: 2
   ===> EN ATTENTE: 3


entrée des données pour la poursuite de l'exécution
===================================================
 POUR L'AGENT   1
    + action suivante: -1
    + id2 agent : 2
```

```
sortie résultats
================
   résultats 1
   -----------
 -  0 agents en communication
 -  1 agents en attente
 -  2 agents en interblocage
 -  1 agents en cours d'exécution
 -  0 agents en état parfait
 EN ATTENTE:    3  4  1 OK    5 999
 BLOQUE:    1  2 -1 OK    6 999
 BLOQUE:    2  1 -1 OK    5 999
 AUTRE IMPARFAIT:    4 99  0 NO    7  10

   résultats 2
   -----------
PROCHAINE ETAPE:
      ###   BLOCAGE GLOBAL   ###
```

```
 COMBIEN D'AGENTS: 3


entrée des données initiales
============================
 --> agent 1
    action: 1
   datedeb: 2
 id2 agent: 2
 --> agent 2
    action: 0
   datedeb: 2
     durée: 3
 --> agent 3
    action: 0
   datedeb: 1
     durée: 2
```

$[\ A1 :: A2!e_1^1\ ;\ v_1^1 := e_2^1\ ;$ abort //

$A2 :: v_1^2 := e_1^2\ ;\ A1?v_2^2\ ;\ v_3^2 := e_2^2\ ;$ abort //

$A3 :: v_1^3 := e_1^3\ ;\ v_2^3 := e_2^3\ ;$ abort

$]$

```
sortie résultats
================
   résultats 1
   -----------
 -  0 agents en communication
 -  1 agents en attente
 -  0 agents en interblocage
 -  0 agents en cours d'exécution
 -  2 agents en état parfait
 EN ATTENTE:    1  2  1 OK    2 999
 AUTRE   PARFAIT:    2 99  0 OK    2   5
 AUTRE   PARFAIT:    3 99  0 OK    1   3

   résultats 2
   -----------
PROCHAINE ETAPE:
TIMING MINIMAL: 2
   ===> EN COURS: 2
   ===> TERMINE: 3
   ===> EN ATTENTE: 1
```



```
entrée des données pour la poursuite de l'exécution
===================================================
 POUR L'AGENT   3
    + action suivante: 0
    + durée action: 1


sortie résultats
================
   résultats 1
   -----------
 -  0 agents en communication
 -  1 agents en attente
 -  0 agents en interblocage
 -  1 agents en cours d'exécution
 -  1 agents en état parfait
 EN ATTENTE:    1  2  1 OK    4 999
 AUTRE IMPARFAIT:    2 99  0 NO    4   5
 AUTRE   PARFAIT:    3 99  0 OK    3   4

   résultats 2
   -----------
PROCHAINE ETAPE:
TIMING MINIMAL: 1
   ===> TERMINE: 2
   ===> TERMINE: 3
   ===> EN ATTENTE: 1


entrée des données pour la poursuite de l'exécution
===================================================
 POUR L'AGENT   2
    + action suivante: -1
    + id2 agent : 1
 POUR L'AGENT   3
    + action suivante: 0
    + durée action: 0


sortie résultats
================
   résultats 1
   -----------
 -  2 agents en communication
 -  0 agents en attente
 -  0 agents en interblocage
 -  0 agents en cours d'exécution
 -  1 agents en état parfait
 COMMUNICANTS:    1  2  1 OK    5 999
 COMMUNICANTS:    2  1 -1 OK    5 999
 AUTRE   PARFAIT:    3 99  0 OK    4   4

   résultats 2
   -----------
PROCHAINE ETAPE:
TIMING MINIMAL: 2
   ===> TERMINE: 1
   ===> TERMINE: 2
   ===> TERMINE: 3
```

```
entrée des données pour la poursuite de l'exécution
===================================================
 POUR L'AGENT   1
    + action suivante: 0
    + durée action: 2
 POUR L'AGENT   2
    + action suivante: 0
    + durée action: 1


sortie résultats
================
   resultats 1
   -----------
 -  0 agents en communication
 -  0 agents en attente
 -  0 agents en interblocage
 -  0 agents en cours d'exécution
 -  3 agents en état parfait
 AUTRE    PARFAIT:    1 99  0 OK    7   9
 AUTRE    PARFAIT:    2 99  0 OK    7   8
 AUTRE    PARFAIT:    3 99  0 OK    4   4

   resultats 2
   -----------
PROCHAINE ETAPE:
TIMING MINIMAL: 1
   ===> EN COURS: 1
   ===> TERMINE: 2
   ===> TERMINE: 3


entrée des données pour la poursuite de l'exécution
===================================================
 POUR L'AGENT   2
    + action suivante: 0
    + durée action: 0


sortie résultats
================
   résultats 1
   -----------
 -  0 agents en communication
 -  0 agents en attente
 -  0 agents en interblocage
 -  1 agents en cours d exécution
 -  2 agents en état parfait
 AUTRE IMPARFAIT:    1 99  0 NO    8   9
 AUTRE    PARFAIT:    2 99  0 OK    8   8
 AUTRE    PARFAIT:    3 99  0 OK    4   4

   résultats 2
   -----------
PROCHAINE ETAPE:
TIMING MINIMAL: 1
   ===> TERMINE: 1
   ===> TERMINE: 2
   ===> TERMINE: 3
```



```
entrée des données pour la poursuite de l'exécution
===================================================
 POUR L'AGENT   1
    + action suivante: 0
    + durée action: 0


sortie résultats
================
   résultats 1
   -----------
 -  0 agents en communication
 -  0 agents en attente
 -  0 agents en interblocage
 -  0 agents en cours d'exécution
 -  3 agents en état parfait
 AUTRE    PARFAIT:    1 99  0 OK    9   9
 AUTRE    PARFAIT:    2 99  0 OK    8   8
 AUTRE    PARFAIT:    3 99  0 OK    4   4

   résultats 2
   -----------
PROCHAINE ETAPE:
    *** TERMINUS TOUT LE MONDE DESCEND  ***
```

PROGRAMMATION LAGALERE ASYNCHRONE DETERMINISTE :

HYPOTHESES:

1) action = 0 ==> instruction de base
   action = -1 ==> action de réception
   action = 1 ==> action d'envoi

2) les dates de lancement peuvent être différentes

3) la durée d'une communication lorsqu'elle peut avoir lieu = constante
   la durée d'une instruction de base autre que "ABORT" = quelconque
   la durée du "ABORT" = 0

4) la description qui suit donne la forme d'un agent et son univers:

agent:

nom agent
modification: (action, nom agent)

univers :

état de l'univers de l'agent
date courante
date de fin d'execution de la modification courante
file de reception des messages

EXEMPLES:

COMBIEN D'AGENTS: 2

entrée des données initiales

--> agent 1
action: 1
datedeb: 1
id2 agent: 1
--> agent 2
action: 0
datedeb: 3
durée: 5

[A1 :: A1!e_1^1 ; ... //

A2 :: $v_1^2 := e_1^2$; ...

]

sortie résultats

résultats 1

- 0 agents en communication
- 0 agents en attente
- 1 agents autobloques
- 0 agents en cours d'exécution
- 1 agents en état parfait

BLOQUE: 1 1 1 OK 1 999 0
AUTRE PARFAIT: 2 99 0 OK 3 8 0

résultats 2

PROCHAINE ETAPE:
BLOCAGE GLOBAL

COMBIEN D AGENTS: 2

entrée des données initiales

--> agent 1
action: 1
datedeb: 2
id2 agent: 2
--> agent 2
action: 0
datedeb: 1
durée: 3

[A1 :: A2!e_1^1 ; A2!e_2^1 ; $v_1^1 := e_3^1$; abort //

A2 :: $v_1^2 := e_1^2$; A1?v_2^2 ; A1?v_3^2 ; abort

]

sortie résultats

résultats 1

- 1 agents en communication
- 0 agents en attente
- 0 agents autobloques
- 0 agents en cours d'exécution
- 1 agents en état parfait

COMMUNICANTS: 1 2 1 OK 2 999 0
AUTRE PARFAIT: 2 99 0 OK 1 4 1

résultats 2

PROCHAINE ETAPE:
TIMING MINIMAL: 2
===> TERMINE: 1
===> EN COURS: 2

entrée des données pour la poursuite de l'exécution

POUR L'AGENT 1
+ action suivante: 1
+ id2 agent : 2

```
sortie résultats
================
   résultats 1
   -----------
 -  0 agents en communication
 -  1 agents en attente
 -  0 agents autobloques
 -  1 agents en cours d'exécution
 -  0 agents en état parfait
 EN ATTENTE:    1  2  1 OK    4 999    0
 AUTRE IMPARFAIT:    2 99  0 NO    3    4    1

   résultats 2
   -----------
PROCHAINE ETAPE:
TIMING MINIMAL: 1
   ===> TERMINE: 2
   ===> EN ATTENTE: 1


entrée des données pour la poursuite de l'exécution
===================================================
 POUR L'AGENT   2
    + action suivante: -1
    + id2 agent : 1


sortie résultats
================
   résultats 1
   -----------
 -  2 agents en communication
 -  0 agents en attente
 -  0 agents autobloques
 -  0 agents en cours d'exécution
 -  0 agents en état parfait
 COMMUNICANTS:    1  2  1 OK    5 999    0
 COMMUNICANTS:    2  1 -1 OK    4 999    2

   résultats 2
   -----------
PROCHAINE ETAPE:
TIMING MINIMAL: 2
   ===> TERMINE: 1
   ===> TERMINE: 2


entrée des données pour la poursuite de l'exécution
===================================================
 POUR L'AGENT   1
    + action suivante: 0
    + durée action: 3
 POUR L'AGENT   2
    + action suivante: -1
    + id2 agent : 1
```



```
sortie résultats
================
   résultats 1
   -----------
 -  1 agents en communication
 -  0 agents en attente
 -  0 agents autobloques
 -  0 agents en cours d'exécution
 -  1 agents en état parfait
 COMMUNICANTS:    2  1 -1 OK    6 999    1
 AUTRE   PARFAIT:    1 99  0 OK    7  10    0

   résultats 2
   -----------
PROCHAINE ETAPE:
TIMING MINIMAL: 2
   ===> TERMINE: 2
   ===> EN COURS: 1


entrée des données pour la poursuite de l'exécution
===================================================
 POUR L'AGENT   2
    + action suivante: 0
    + durée action: 0


sortie résultats
================
   résultats 1
   -----------
 -  0 agents en communication
 -  0 agents en attente
 -  0 agents autobloques
 -  1 agents en cours d'exécution
 -  1 agents en état parfait
 AUTRE IMPARFAIT:    1 99  0 NO    9  10    0
 AUTRE   PARFAIT:    2 99  0 OK    8   8    0

   résultats 2
   -----------
PROCHAINE ETAPE:
TIMING MINIMAL: 1
   ===> TERMINE: 1
   ===> TERMINE: 2


entrée des données pour la poursuite de l'exécution
===================================================
 POUR L'AGENT   1
    + action suivante: 0
    + durée action: 0
```

```
sortie résultats
================
   résultats 1
   -----------
 -  0 agents en communication
 -  0 agents en attente
 -  0 agents autobloques
 -  0 agents en cours d'exécution
 -  2 agents en état parfait
 AUTRE    PARFAIT:    1 99   0 OK   10   10    0
 AUTRE    PARFAIT:    2 99   0 OK    8    8    0

   résultats 2
   -----------
PROCHAINE ETAPE:
    *** TERMINUS TOUT LE MONDE DESCEND  ***
```

BIBLIOGRAPHIE

REFERENCES BIBLIOGRAPHIQUES.

_ A _

APT79 K.APT
Ten years of Hoare's logic : a survey
partI : 5th Scandinaviam logic symp. Jan 1979
partII: LITP-Paris VII n° 82-32, 1982

APT80 K.APT, N.FRANCEZ & W.DE ROVER
A proof system for communicating sequential processes
ACM-TOPLAS 2,8,1980 p359-385

APT81 K.APT & G.PLOTKIN
A Cook's tour of countable nondeterminism
LNCS 115,1981 p479-494

ARN78 A.ARNOLDT & M.NIVAT
Algebraic semantics of nondeterministic recursive program schemes
LITP-Paris VII n°78-4, 1978

ARN81 A.ARNOLDT
Sémantique des processus communicants
RAIRO informatique théorique 5,2,1981 p103-139

_ B _

BAN80 J.P.BANATRE
Contribution à l'étude de méthodes et d'outils de construction de programmes parallèles et fiables
Thèse d'état, univ Rennes 1980

BER82 H.BERG, W.BOEBERT, W.FRANTA & T.MOHER
Formal methods of program verification and specification
Prentice-Hall 1982

BID81 M.BIDOIT
Une méthode de présentation des types abstraits algébriques
Thèse 3ème cycle, univ Paris-sud, 1981

BOU81 F.BOUSSINOT
Réseaux de processus avec mélange équitable : une approche temps réel
Thèse d'état, univ Paris VII, 1981

BOU83 G.BOUDOL & D.AUSTRY
Algèbre de processus et synchronisation
INRIA n°187, 1983

BRI76 H.BRINCH
The programming language Concurrent-Pascal
LNCS 6, 1976 p83-110

BRI78 H.BRINCH
Distributed processes : a concurrent programming concepts
CACM 21,11,1978 p934-941

BRO80a M.BROY & M.WIRSING
Programming languages as abstract data types
5ème CAAP, Lille 1980

BRO80b M.BROY & M.WIRSING
Algebraic definition of a functional programming language
TUM-I8008, tech univ München 1980

BRO81 M.BROY & M.WIRSING
On the algebraic specification of nondeterministic programming languages
6ème CAAP, Genova 1981

BRO82 M.BROY & M.WIRSING
On the algebraic specification of finitary infinite communicating sequential processes
IFIP TC2 Garmish-Partenkirchen 1982 p135-162

_ C _

CAM74 R.H.CAMPBELL & A.N.HABERMANN
The specification of process synchronisation by path expressions
LNCS 46, 1974 p89-102

CCI80 CCITT
Définition du langage CHILL
Commission d'étude XI, CCITT août 1980

CIM81 CIMSA
LTR-V3 : spécification technique
Division ICEV, centre électronique de l'armement 1981

CLA79 E.M.CLARK
Programming languages constructs for which it's impossible to obtain good Hoare axioms systems
JACM 26,1,1979 p129-147

COO78 S.A.COOK
Soundness and completeness of an axiomatic system for program verification
SIAM Journal of Computing 7,1,1978 p70-90

COU78 B.COURCELLE & M.NIVAT
The algebraic semantics of recursive program schemes
IRIA-LABORIA 1978

_ D _

DAR80 P.DARANDEAU
Processus non séquentiel et leurs observations en univers non centralisé
IRISA n°134, Rennes 1980

DEB76 J.W.DE BAKKER
Semantics and termination of non deterministic recursive programs
Proc ICALP Edinburgh 1976 p435-477

DER79 J.C.DERNIAME & J.P.FINANCE
Types abstraits de données : spécification, utilisation et réalisation
CRIN 79-E-57, Nancy 1979

DIJ68 E.W.DIJKSTRA
The structure of THE multiprogramming systems
CACM 11,5,1968 p341-346

DIJ75 E.W.DIJKSTRA
Guarded commands, nondeterminacy and formal derivation of programs
CACM 18,1975 p453-457

DON75 J.E.DONAHUE
Complementary definitions of programming language semantics
LNCS 42,1975, 172p

DON82 M.J.O'DONNEL
A critique of the foundation of Hoare style programming logics
CACM 25,12,1982 p927-935

_ F _

FIN76 J.P FINANCE
Une formalisation de la sémantique des langages de programmation
RAIRO informatique théorique 10,8 et 10,12, 1976

FIN83 J.P FINANCE & M.S OUERGHI
On the algebraic specification of concurrency and communication
ICPP'83 proc IEEE, Bellaire-Michigan 1983, p281-288

FLO67 R.W FLOYD
Nondeterministic algorithms
JACM 14,1976 p636-644

_ G _

GAU78 M.C GAUDEL, P.DESCHAMP & M.MAZAUD
Semantics of procedures as an algebraic abstract data type
IRIA-LABORIA n°334, 1978, 23p

GAU80 M.C GAUDEL
Génération et preuve de compilateurs basées sur une sémantique formelle des langages de programmation
Thèse d'état, INPL Nancy 1980

GOG78 J.A GOGUEN, J.W THATCHER & E.G WAGNER
An initial algebra approach to the specification, correctness and implementation of abstract data types
Current trends in prog. methodology IV, Yeh ed, 1978

GOR79 M.J GORDON
The denotational description of programming languages an introduction
Springer-Verlag ed 1979, 160p

GUE81 P.GUERREIRO
Sémantique relationnelle des programmes non déterministes et des processus communicants
Thèse 3ème cycle, IMAG Grenoble 1981

GUT78 J.V GUTTAG, E.HOROWITZ & D.R MUSSER
The design of data type specification
Current trends in prog. methodology IV, Yeh ed, 1978

_ H _

HEN76 M. HENNESSY & E.A ASHCROFT
The semantics of nondeterminism
Proc ICALP Edinburgh 1976, p478-493

HEN81 J.L HENNESSY & R.B KIEBURTZ
The formal definition of a real time language
Standford univ Computer Science Lab, 1981, 36p

HOA74a C.A.R HOARE
Monitors : an operating system structuring concept
CACM 17,10,1974, p549-557

HOA74b C.A.R HOARE & P.E LAUER
Consistent and complementary formal theories of the semantics of programming languages
Acta Informatica 3,1974, p135-153

HOA79 C.A.R HOARE
Communicating sequential processes
CACM 21,8,1979, p666-677

HOW76 J.H HOWARD
Proving monitors
CACM 19,5,1976, p273-279

HUE80 G. HUET & D. OPPEN
Equations and rewrite rules : a survey
Tech report CSL-111, SRI international 1980, 52p

_ I _

ICH79 J.D ICHBIAH & al
Preliminary ADA reference manual
ACM Sigplan Notices 14,6,1979

_ L _

LAM83 L.LAMPORT
Specifying concurrent program modules
ACM-TOPLAS 5,2,1983, p190-222

LES79 P. LESCANNE
Etude algébrique et relationnelle des types abstraits et leur représentation
Thèse d'état, INPL Nancy 1979

LES83 P. LESCANNE
Computer experiments with the REVE term rewriting system generator
10ème symp POPL 1983, 26p

LEV81 G.R LEVIN & D. GRIES
A proof technique for communicating sequential processes
Acta Informatica 15,1981, p281-302

LIS77 B. LISKOV, A. SYNDER, A. ATKINSON & C. SCHAFFERT
Abstraction mechanisms in CLU
CACM 20,8,1977, p564-576

LIV78 C. LIVERCY
Théorie des programmes
Dunod informatique 1978

_ M _

MAN70 Z. MANNA
The correctness of nondeterministic programs
Artificial Intelligence 1,1970, p1-26

MAN74 Z. MANNA
Mathematical theory of computation
Mc Graw-Hill ed 1974

MER83 D. MERY
Une méthode axiomatique de preuves de propriétés de fatalité de programmes parallèles avec hypothèse d'exécution équitable
Thèse 3ème cycle, INPL Nancy 1983

MIL80 R. MILNER
A calculus of communicating systems
LNCS 92, 1980

MON82 C. MONGENET
Définition et implantation de la communication
Rapport de DEA, Nancy 1982

MOS80 P. MOSSES
A constructive approach to compiler correcteness
LNCS 94,1980, p189-210

MUS80a D.R MUSSER
Abstract data type specification in the AFFIRM system
IEEE trans. on soft. eng. vol SE-6,n°1, 1980, p24-32

MUS80b D.R MUSSER
On proving inductive properties of abstract data types
7ème symp POPL 1980, p154-162

_ N _

NEU71 E.J NEUHOLD
The formal description of programming languages
IBM System Journal 2,1971, p87-113

NIV79 M. NIVAT
La synchronisation des processus
Revue technique de Thomson-CSF 11,1979, p899-919

_ O _

OUE82a M.S OUERGHI
Contribution à l'étude du parallèlisme
CRIN 82-R-044, Nancy 1982

OUE82b M.S OUERGHI & A. ZAKARI
Un exemple de systèmes communicants
CRIN 82-R-032, Nancy 1982

OUE82c M.S OUERGHI & A. ZAKARI
Vers une bonne expression de la communication dans un milieu parallèle
CRIN 82-R-061, Nancy 1982

OWI76 S. OWICKI & D. GRIES
An axiomatic proof techniques for parallel programs
Acta Informatica 6,1976, p319-340

_ P _

PAI71 C. PAIR
La formalisation des langages de programmation
Mathématiques et Sciences Humaines n°34,1971, p71-86

PAI80 C.PAIR
Sur les modèles des types absraits algébriques
CRIN 80-P-052, Nancy 1980

PAI82 C. PAIR
Application of abstract data types to the definition of the semantics of programming languages
TCS 18,1,1982, p1-31

PEP79 P.PEPPER
A study on transformational semantics
LNCS 69,1979, p382-405

PER82 G.R PERRIN
Specification and verification of communication of processes
CRIN 82-R-020, Nancy 1982

PLO76 G. PLOTKIN
A power domain construction
SIAM Journal on Computing 5,1976, p452-487

PLO82 G. PLOTKIN
An operational semantics of CSP
IFIP TC2 Garmish-Partenkirchen 1982, p185-208

PNU79 A. PNUELLI
The temporal semantics of concurrent programs
LNCS 70,1979, p1-20

_ R _

REM82 J.L REMY
Etude de systèmes de réecriture conditionnels et applications aux types abstraits algébriques
Thèse d'état, INPL Nancy 1982

_ S _

SCO71 D. SCOTT & C. STRACHEY
Toward a mathematical semantics for computer languages
Proc Symp on Computers and Automata 1971, p19-46

_ T _

TEN76 R.D TENNENT
The denotational semantics of programming languages
CACM 19,8,1976, p437-453

_ V _

VAN69 A. VAN WINJNGARDEN & al
Report on the algorithmic language ALGOL68
Numer. Math. n°14,1969, p79-218

VER80 J.P VERJUS, D. HERMAN & F. ANDRE
Contrôle du parallèlisme et de la répartition
Cours d'été AFCET, Aix-en-Provence 1980

_ W _

WAN75 M. WAND
First-order identities as defining languages
Indiana Univ Tech report n°29, 1975

WAN78 M. WAND
A new incompleteness result for Hoare's system
JACM 25,1,1978

WIR71 N. WIRTH
The programming language PASCAL
Acta Informatica 1,1971, p35-63

WIR83 M. WIRSING, P. PEPPER, H. PARTSCH, W. DOSCH & M. BROY
On hierarchies of abstract data types
TUM-I8303, Tech univ München 1983

_ Z _

ZHA83 H. ZHANG
Validation des types abstraits par test
Rapport de DEA, Nancy 1983

ZIL81 S.N ZILLES
A look at algebraic specification
IBM San José research lab 1981, 22p

institut
national
polytechnique
de lorraine

idonat,

Scol.

AUTORISATION DE SOUTENANCE DE THESE DE DOCTORAT 3ème CYCLE

-:-:-:-:-:-:-:-:-:-:-:-:-

VU LE RAPPORT ETABLI PAR :

Monsieur le Professeur FINANCE Jean-Pierre

le Président de l'Institut National Polytechnique de Lorraine autorise :

Monsieur OUERGHI Mohamed Saïd

à soutenir, devant l'I.N.P.L., une thèse de Doctorat intitulée :

"SEMANTIQUE ALGEBRIQUE D'UN LANGAGE DE PROGRAMMATION SUPPORTANT LES CONCEPTS DE PROCESSUS COMMUNICANTS"

en vue de l'obtention du titre de DOCTEUR 3ème CYCLE

Spécialité "INFORMATIQUE"

Fait à NANCY, le 28 Février 1984

Le Président de l'I.N.P.L.
P/O Le Vice- Président du Conseil Scientifique
de l'I.N.P.L.

M. LUCIUS
A. MAILFERT

INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE NANCY

RESUME :

Le thème principal abordé dans cette thèse est la mise au point d'une méthode de description algébrique de la sémantique des langages de programmation supportant les concepts de processus communicants et de nondéterminisme.

L'idée de base de la définition algébrique d'une sémantique est d'associer aux différents composants d'un langage de programmation une hiérarchie de types abstraits rendant compte des expressions, des états de mémoire et des calculs.
L'extension de ce cadre algébrique pour la prise en compte des processus communicants se fait en "temporisant" les différentes opérations [illegible] et en décrivant un mécanisme de composition d'opérations temporisées. Ces mécanismes sont décrits sur un langage jouet baptisé LAGALERE.

Le second thème abordé ici, est la complémentarité des sémantiques des langages de programmation supportant les processus communicants, en particulier les sémantiques algébriques et axiomatiques.

L'objectif est de valider les règles de preuves définies par la sémantique axiomatique à l'aide de la sémantique algébrique. On fournit ainsi un outil utilisable par le programmeur.

MOTS CLES :

Nondéterminisme, parallélisme, processus communicants, spécifications algébriques, types abstraits, sémantique des langages de programmation, systèmes de preuve.