Université de Nancy I

UER de Mathématiques
CRIN

SPECIFICATION et VALIDATION de SYSTEMES de PROCESSUS COMMUNICANTS

Thèse présentée pour obtenir le diplôme
de Doctorat de l'Université de Nancy I
en Informatique

par

Marina MARMONIER

**Thèse soutenue le 17 février 1989 devant
la Commission d'Examen :**

Messieurs :	Jean-Luc REMY,	CNRS CRIN (Président) (Rapporteur)
	Philippe JORRAND,	Institut National Polytechnique de Grenoble (Rapporteur)
	Jean-Marc ADAMO,	Ecole Nationale Supérieure de LYON
	Jean-Pierre FINANCE,	Université de Nancy I
	Guy-René PERRIN,	Université de Franche-Comté

Université de Nancy I

UER de Mathématiques
CRIN

SPECIFICATION et VALIDATION de SYSTEMES de PROCESSUS COMMUNICANTS

Thèse présentée pour obtenir le diplôme
de Doctorat de l'Université de Nancy I
en Informatique

par

Marina MARMONIER

**Thèse soutenue le 17 février 1989 devant
la Commission d'Examen :**

Messieurs :	Jean-Luc REMY,	CNRS CRIN (Président) (Rapporteur)
	Philippe JORRAND,	Institut National Polytechnique de Grenoble (Rapporteur)
	Jean-Marc ADAMO,	Ecole Nationale Supérieure de LYON
	Jean-Pierre FINANCE,	Université de Nancy I
	Guy-René PERRIN,	Université de Franche-Comté

Je ne saurais trop témoigner de gratitude envers Jean-Pierre Finance d'abord pour m'avoir accueillie dans son équipe, ensuite pour le temps qu'il m'a accordé et l'orientation qu'il a donné à ce travail. Il m'a conduite vers l'utilisation d'outils formels et m'a inlassablement guidée vers la recherche de la rigueur : ce travail ne serait pas ce qu'il est sans lui.

Je tiens tout particulièrement à remercier Monsieur Philippe Jorrand d'avoir bien voulu s'intéresser à mon travail et accepter d'en être le rapporteur et pour ses encouragements.

Que Jean-Luc Rémy trouve ici le témoignage de ma profonde gratitude non seulement pour avoir accepté de juger ce travail mais aussi pour le soutien et les conseils qu'il m'a apportés à tout moment. Je lui dois d'avoir persévéré au delà des moments de doute. Il me fait l'honneur de présider ce jury et je l'en remercie.

J'adresse ici toute ma reconnaissance à Monsieur Jean-Marc Adamo pour l'attention qu'il a portée à mon manuscrit ; je le remercie pour ses remarques pertinentes qui m'ont aidée à faire le point sur mon travail.

Guy-René Perrin a défini avec Jean-Pierre Finance le sujet de ma thèse. Je tiens ici à lui exprimer toute ma reconnaissance pour le sérieux avec lequel il a suivi ce travail, la clarté de ses explications et l'éventail des thèmes de recherche vers lesquels il m'a guidée. Je ne le remercierai jamais suffisamment pour sa disponibilité et la motivation pour le domaine du parallélisme qu'il a sû m'insuffler.

Que tous les membres de l'équipe COMETE, avec lesquels il m'a été agréable de travailler, qu'ils soient de Besançon ou de Nancy, trouvent ici l'expression de mon amitié. En particulier je tiens à remercier Jacques Julliand qui a guidé mes premiers pas au sein du projet COMETE et à qui je dois beaucoup : je regrette que les "textes officiels" ne l'autorisent pas à faire partie de ce jury. Merci également à Dominique Mery pour les discussions fructueuses que nous avons pu avoir au sujet de la sémantique des programmes parallèles.

Je tiens à remercier Monsieur Jean-Marie Kauffmann, Directeur de l'IUT de Belfort, de m'avoir permis d'organiser mon service pour mener à bien cette thèse et m'avoir aidé à résoudre divers problèmes techniques.

Je n'oublie pas mes collègues de l'IUT de Belfort qui ont rendu possible les aménagements de mon service afin que je puisse mener à bien la rédaction de cette thèse et en particulier Liliane et Jacques.

J'adresse ma profonde gratitude à tous ceux qui ont facilité mes déplacements sur Nancy et Besançon en m'hébergeant chez eux notamment Marie-Christine, Nadia, Christine, Brigitte et Jacques.

Enfin je ne voudrais pas terminer sans remercier tous ceux qui de près ou de loin ont aidé à la réalisation technique de ce document : Corinne, Muriel, Jocelyne, Camel ainsi que Irène Souchier qui en a effectué la reprographie avec sa gentillesse habituelle.

SOMMAIRE

CHAPITRE II : LE LANGAGE D'EXPRESSION DE SYSTEMES PARALLELES

CHAPITRE III : L'INTERPRETATION (LES PRINCIPES)

CHAPITRE IV : REALISATION DE L'INTERPRETE

CHAPITRE V : VALIDATION DES PROGRAMMES

INTRODUCTION

INTRODUCTION

I. CONTEXTE

Actuellement le développement des systèmes multiprocesseurs et des architectures réparties motive l'intérêt pour la recherche dans le domaine de la programmation parallèle.

Ce domaine est vaste et pour le cerner on peut, par exemple, établir la classification suivante, selon quatre grands axes :

- l'architecture des systèmes parallèles et leurs applications spécifiques : architectures systoliques, processeurs vectoriels, multiprocesseurs, réseaux de sites distribués, etc,...

- l'algorithmique des applications réparties,

- les langages d'expression du parallélisme,

- la sémantique des langages parallèles et les systèmes de preuve associés.

C'est aux deux derniers centres d'intérêt que se rattache notre travail. Cette thèse a été réalisée dans le cadre du projet COMETE (COMmunication Et TEmps), développé au sein des laboratoires CRIN à Nancy et LIB à Besançon et soutenu par le projet CNRS C^3.

Le projet COMETE se préoccupe de méthodologie de construction de programmes parallèles ; les principaux thèmes de recherche développés dans ce projet sont les suivants :

(i) la définition d'un cadre formel complet pour la spécification des problèmes et l'expression de leurs solutions : la notion de "communication" définie dans la thèse de GR. Perrin [Per85] est l'outil central de la démarche proposée,

(ii) l'étude de la représentation algorithmique des solutions

obtenues en termes de systèmes de processus communicants, de leur programmation et de leur validité,

(iii) la validation de la notion de "communication" comme outil méthodologique : dérivation systématique de solutions, transformation des solutions, développement de solutions pour des problèmes réels, etc,... .

Le travail présenté dans ce document est une contribution à l'étude des deux premiers thèmes énoncés ci-dessus.

II. OBJECTIFS

L'objet de cette thèse est de proposer des outils pour la conception et la validation de programmes parallèles :

- outils méthodologiques pour la spécification des problèmes et l'expression algorithmique des solutions,

- outils logiciels pour la validation des programmes.

Nous développons successivement la présentation de ces deux objectifs.

Une démarche méthodique de conception de programmes parallèles a été définie dans [Per 85] et [JMP 86] ; notre apport, dans cette thèse, est de formaliser les notions introduites en termes de *types abstraits de données* et de prouver la validité de la représentation algotihmique des solutions, sous forme de types abstraits appelés *"types de communication".*

Ainsi le premier chapitre de cette thèse est consacré à la présentation d'un cadre formel pour la conception de programmes parallèles ; cette présentation d'outils formels est donnée de manière relativement indépendante de la démarche qui les utilise, dans un souci de simplification de la lecture. Pour guider le lecteur dans la compréhension et la justification de ces outils formels nous allons tout d'abord exposer de manière informelle , dans cette introduction, la

démarche proposée par le projet COMETE.

Pour illustrer cette démarche, nous joignons en [Annexe 0] la définition complète de la gestion d'une unité de disque (issue de [JMP 86]); nous présentons ici de manière intuitive les différents concepts introduits par cette démarche.

Qu'il s'agisse de concevoir un système *"transformationnel"* ou *"réactif"*, selon la classification de [HaP 85], la spécification d'un problème peut s'exprimer en termes d'une **relation** entre un ensemble de données et un ensemble de résultats, relation au sens le plus général pouvant faire intervenir le **temps.**

La définition de cette relation met en évidence des objets intermédiaires et des relations entre les suites des valeurs de ces objets. En particulier ces relations exhibent des suites résultats **extraites** de suites de données et dont chaque terme est défini à partir de l'état **courant** du système spécifié ; par exemple pour définir la gestion d'une unité de disque, selon la stratégie dite "de l'ascenseur" (qui minimise les mouvements du bras), une première spécification de la suite résultat s à partir de la suite d des requêtes d'accès au disque est la suivante :

<u>données</u> : d, suite de demandes d'accès, de type n° de piste :
$d = (d_0, d_1, ..., d_i, ...)$

<u>résultats</u> : s, suite des services, de type n° de piste :
$s = (s_0, s_1, ..., s_i, ...)$
<u>telle que</u> :
$\forall j \in S$ (domaine de s), $s_j = d_i$
<u>avec</u> $i \in D$ (domaine de d) défini par : $i = \phi(E_j)$
<u>où</u> E_j est l'état du système et ϕ une application telle que la contrainte de déplacements minimaux du bras soit vérifiée.

A partir d'une spécification de ce type, la construction d'une solution réside dans la définition de l'état "courant" et de l'application ϕ.

Pour modéliser l'aspect temporel de telles définitions nous introduisons le concept de **"suite temporelle"** : une suite temporelle définit la séquence datée des valeurs d'un objet ; nous appelons alors **"relations de communication"** les relations entre de tels objets.

La définition d'une solution au problème spécifié ci-dessus est alors informellement, la suivante :

Si d est la *suite temporelle* des requêtes d'accès, suite de couples de la forme < n°-piste, date-requête >, alors la *suite temporelle résultat* s, suite de couples de la forme < n°-piste, date-service > vérifie les contraintes suivantes, caractéristiques d'une relation de communication :

$$\forall\, j \in \text{domaine}(s),\ \exists\, i \in \text{domaine}(d) \text{ tel que :}$$
$$\begin{cases} s_j.\ \text{valeur} = d_i.\ \text{valeur} \\ s_j.\ \text{instant} \geq d_i\ \text{instant} \\ \tau(d,s,i,j) \end{cases}$$

où l'on note *valeur* et *instant* les sélecteurs des deux composantes d'un terme d'une suite temporelle et où τ est la relation caractéristique de la stratégie "de l'ascenseur" qui lie les termes s_j et d_i en correspondance dans les suites de résultats et de données.

Cette relation τ est définie à partir d'un certain nombre d'intermédiaires (cf Annexe 0) qui formalisent **"l'état courant"**, notamment :

- le numéro de la piste atteinte au $(j-1)^{è}$ service,
- la date du $(j-1)^{è}$ service,
- l'ensemble des requêtes émises et non encore servies à cette date,

- la direction courante du bras (vers l'intérieur ou vers l'extérieur).

A partir de telles définitions des solutions nous visons une expression algorithmique en termes de systèmes de processus communicants. Intuitivement une relation de communication lie deux processus : le premier émet (produit) une suite de valeurs dont certains termes sont utilisés (consommés) par le second selon les caractéristiques de la relation.

L'étape qui suit, dans la démarche, c'est à dire la description fonctionnelle des solutions est donc celle de représentation algorithmique des "relations de communication". Pour cela nous introduisons la notion de **"type de communication"** : un type de communication est un type abstrait algébrique paramétré. Ce type définit des objets qui représentent les suites de valeurs mises en évidence lors de l'étape de spécification ; la construction de ces objets est définie par des opérations spécifiques de la relations représentée, sans manipuler explicitement le temps.

Nous montrons au chapitre I, que la notion de "type de communication" est une représentation algorithmique valide des "relations de communications" décrites en termes de type abstrait algébrique . Pour cela nous définissons la sémantique des types de communication dans le domaine des "*suites temporelles*".

A partir de cette représentation algorithmique, les solutions sont décrites en termes de systèmes de processus communicants dans un langage qui intègre la notion de type de communication : les communications sont réalisées de manière **asynchrone** par l'intermédiaire de ports de communication **typés** par les "types de communication".

Ce langage a été défini dans [Per 85] ; le style d'expression est un style impératif dans une syntaxe proche du langage Pascal en ce qui concerne le noyau de programmation séquentielle classique. L'expression du parallèlisme repose sur les quatres entités de *processus*, *port de communication* , *donnée communiquée* et *type de communication* et sur l'introduction des deux primitives de communication *produire* et *consommer*.

Le langage ne prétend pas être un langage de plus pour la programmation de systèmes parallèles : il a été conçu dans [Per 85] pour permettre au concepteur la validation intuitive de ses solutions ; notons par ailleurs que le projet COMETE s'intéresse à l'implantation des solutions dans des langages "concrets" tels que ADA [JKP 86] et OCCAM [EJ 88].

Nous abordons ici le deuxième aspect de notre travail qui concerne la validation des programmes parallèles.

Pour permettre la validation des systèmes décrits nous avons réalisé un logiciel d'**interprétation** des programmes et de **visualisation** des communications.

Cette réalisation s'appuie sur la définition formelle de la sémantique du langage proposé ; la définition de cette sémantique constitue la seconde "brique" importante de notre apport, au plan formel.

L'objectif visé dans cette définition est d'obtenir une sémantique qui permette de déduire "naturellement" un modèle d'exécution des programmes dans le but d'en définir l'interprétation.

Trois approches, au moins, étaient possibles :

- celle proposée dans [Lam 85] qui définit une sémantique axiomatique, compositionnelle basée sur la logique temporelle,

- l'approche "opérationnelle", et compositionnelle ainsi que le définit [Plo 81], adoptée dans [Bou 87],

- l'approche "sémantique algébrique" [Gau 80], présentée dans [FiO 83].

Nous avons écarté la première possibilité qui nous semble mieux adaptée à la définition d'un compilateur qu'à celle d'un interpréteur et avons retenu les avantages des deux autres approches :

- le type de formalisme introduit dans [Bou 87] permet de déduire

"simplement" la sémantique des programmes parallèles à partir de celle de ses processus constituants par l'intermédiaire d'un système de règles d'inférence,

- la prise en compte de la durée des actions élémentaires, ainsi qu'il est proposé dans [FiO 83], permet d'exprimer la simultanéité d'évènements.

La définition du langage et de sa sémantique fait l'objet du second chapitre de ce document.

Nous abordons maintenant le second objectif de notre travail qui se rapporte à la production d'outils logiciels pour la validation de programmes parallèles.

Les chapitres III et IV sont consacrés respectivement à la définition et à la réalisation de l'interprète ; le chapitre V présente une ébauche de l'utilisation de ce logiciel pour vérifier certaines propriétés des programmes.

La définition de l'interprète repose sur la notion de **"séquentialisation"**. Une *séquentialisation* est un modèle d'exécution d'un programme déduit de la sémantique et qui décrit un interclassement des traces d'exécution des processus composant le programme à partir d'une relation d'ordre sur les communications.
Les traces d'exécution d'un processus sont définies à partir de la notion d' **unité d'exécution** et du schéma de contrôle de ce processus ; la notion d'*unité d'exécution* est une mise en œuvre de la possibilité d'"abstraction" offerte par la sémantique du langage et permet de simplifier l'expression des traces d'exécution des processus : intuitivement une *unité d'exécution* est une "région" dans le texte d'un processus qui ne communique avec les autres processus du programme que par au plus une opération de communication et qui commence par cette opération.
Ainsi la notion de *séquentialisation* définit explicitement les dates des communications et constitue un modèle d'exécution cohérent avec l'expression en termes de *suites temporelles* adoptée lors de l'étape de spécification.

Nous caractérisons l'ensemble des séquentialisations possibles d'un programme parallèle à partir de l'ensemble des unités d'exécution

des processus composants et de leurs durées respectives.

La définition de la notion de *séquentialisation* fait l'objet de la première partie du chapitre III. Dans la seconde partie de ce chapitre nous décrivons fonctionnellement l'interprétation d'un programme comme la **simulation** de l'exécution de l'une des séquentialisations possibles ; cette simulation repose sur la définition d'un échéancier des dates de fin de réalisation des unités d'exécutions activées à chaque cycle de l'interprète.

La réalisation de l'interprète est décrite dans le chapitre IV de cette thèse. Cette réalisation a été programmée en Le_Lisp-CEYX [Cha 85] [Hul 84] et s'intègre dans un environnement de programmation développé par le projet COMETE.

Nous décrivons cette réalisation en définissant l'architecture du logiciel puis en exposant les principaux choix de représentation dans le langage CEYX et en détaillant l'interprétation des primitives de communication. Enfin nous terminons ce chapitre en présentant la visualisation graphique des communications.

Le dernier chapitre de ce document est consacré à l'esquisse d'un système de validation des programmes : nous y définissons les principes d'une extension de l'interprète pour permettre la vérification de la propriété d'**abscence de blocage**. Cette validation expérimentale est possible pour une certaine catégorie de programmes : les programmes dont la réalisation des opérations de communication ne dépend pas des *valeurs communiquées*.

Pour de tels programmes l'ensemble des séquentialisations peut être construit de manière statique.

L'introduction d'une **relation d'équivalence** sur les séquentialisations permet d'envisager une utilisation de l'interprète à cette fin de vérification par la simulation des comportements "représentants" des classes d'équivalence.
La définition de cette équivalence fait l'objet de la première partie du chapitre V. La seconde partie de ce chapitre est consacrée à la présentation des principes de cette validation par simulation. Enfin dans une dernière partie nous exposons les différents travaux en matière de vérification de programmes parallèles que nous avons

retenus de notre étude bibliographique.

Cet exposé se termine par une conclusion où nous faisons le bilan de notre travail et en présentons les prolongements possibles.

CHAPITRE I

CADRE FORMEL POUR LA CONCEPTION DE PROCESSUS PARALLELES COMMUNICANTS

CADRE FORMEL POUR LA CONCEPTION DE PROCESSUS PARALLELES COMMUNICANTS

L'objet de ce chapitre est de présenter un formalisme et des outils pour la **spécification** de problèmes et la **conception** de systèmes parallèles, thèmes de recherche du projet COMETE. Nous présentons ici les notions fondamentales développées par ce projet (suite temporelle, relation de communication et type de communication) ; l'apport de ce chapitre est la définition intuitive puis formelle de ces notions, en termes de types abstraits de données ainsi que la définition de la sémantique des types de communication et de la preuve de représentation des relations par les types de communication.

Dans un univers séquentiel on peut spécifier un problème par la définition d'une relation **statique** qui décrit une(des) suite(s) de résultats en fonction d'une (ou plusieurs) suite(s) de données [Fin 79] : on qualifie ce type de relation de **relation de calcul**.

Dans le cadre de systèmes complexes il s'agit d'exprimer la prise en compte d'événements en provenance de l'extérieur, leurs simultanéités et d'éventuelles contraintes de temps réel au niveau des réponses de système : toutes ces caractéristiques sont spécifiques de la classe des problèmes dits de "parallélisme de **situation**" [Ray 81].

La spécification de ce type de problèmes, qui conduit à la définition de systèmes **réactifs** [HaP 85], nécessite la prise en compte de l'état "**courant**" du système pour la détermination de chaque terme de la ou des suites résultats, en fonction des termes de la, ou des, suites de données : l'aspect **temporel** de ce type de relation et la notion intuitive d'état "courant" doivent être formalisés.

Le projet COMETE propose comme formalisme d'expression un modèle asynchrone du temps fondé sur les notions de **suite**

temporelle, et de **relation de communication**, mises en évidence respectivement dans [FiJ 80] et [Per 85].

Ces deux concepts sont utilisés pour la **spécification** des problèmes et la description fonctionnelle de leurs solutions ; nous les formaliserons à l'aide de types abstraits de données [Liz 75] [ADJ 77] [GuH 78], définis à partir du type support "SUITE".

L'expression algorithmique des solutions est décrite en termes de systèmes de processus communicants, ainsi que nous le verrons au chapitre II. La description des communications est formalisée par le concept de **type de communication**, type abstrait de données qui représente les relations de communication introduites au niveau spécification ; ce type abstrait a pour support les types "SUITE" et "TABLE".

Ainsi le premier paragraphe de ce chapitre a-t-il pour objet la définition des types abstraits algébriques "SUITE" et "TABLE", supports des types introduits ci-dessus.

Dans le second paragraphe nous définissons formellement les notions de **suite temporelle** et de **relation de communication**, en termes de types abstraits spécifiés de manière axiomatique [DeF 79] à partir du type support SUITE.

Le troisième paragraphe traite de la représentation algorithmique des relations de communication et comprend trois parties :

- dans la première partie nous présentons une spécification du type abstrait **type de communication** ,

- dans la seconde partie nous proposons une sémantique interprétative des types de communication dans le domaine des suites temporelles ,

- dans la troisième partie nous montrons que le concept de **type de communication** est une représentation algorithmique valide des relations de communication définies lors de l'étape de spécification; nous présentons un schéma général de preuve fondé sur la sémantique des types de communication et le développons sur un exemple.

I. DEFINITION DES TYPES ABSTRAITS "SUITE" ET "TABLE"

I.1. Spécification du type SUITE

Dans ce paragraphe nous nous limitons à la présentation de la spécification de ces deux types indépendamment de toute utilisation.

Nous proposons de définir formellement la notion de suite par le type abstrait algébrique paramétré SUITE dont la spécification, inspirée de [Fin 79], est décrite par la figure 1 ci-dessous :

Description informelle	Enoncé formel
V et le type des termes de la suite **ENT** désigne le type des entiers naturels **s-vide** et * sont les constructeurs **s-vide** construit la suite vide * construit une nouvelle suite par adjonction en queue d'une valeur **début** rend la suite privée de son dernier terme **der** délivre le dernier élément de la suite **prem** rend le premier terme de la suite **reste** délivre la suite privée de son premier terme **lg** donne le nombre de termes de la suite	type SUITE (V : type) opérations s-vide : → SUITE * : SUITE x V → SUITE der : SUITE → V début : SUITE → SUITE prem : SUITE → V reste : SUITE → SUITE lg : SUITE → ENT s-extrait: SUITE x ENT x ENT→ SUITE conc : SUITE x SUITE → SUITE axiomes : soient s, s1, s2 : SUITE ; v : V ; i, j : ENT der(s*v) = v

s-extrait délivre la suite extraite composée des éléments d'indice compris entre les valeurs des 2 paramètres **conc** réalise la concaténation de deux suites les fonctions der et prem ne sont pas définies pour la suite vide	début (s-vide) = s-vide début(s*v) = s prem(s*v) = si s = s-vide alors v sinon prem(s) fsi reste(s-vide) = s-vide reste(s*v) = si s = s-vide alors s-vide sinon reste(s)*v fsi lg(s-vide) = 0 lg(s*v) = lg(s) + 1 s-extrait(s*vide,i,j) = s-vide s-extrait(s*v,i,j) = si i > j alors s-vide sinon si j = lg(s) + 1 alors s-extrait(s,i,j-1)*v sinon s-extrait(s,i,j) fsi fsi conc(s, s-vide) = s conc(s1,s2*v) = conc(s1,s2)*v restrictions pré der(s) = s ≠ s-vide pré prem(s) = s ≠ s-vide

Figure 1 : Spécification du type SUITE

En mathématiques la notion de suite est définie comme une application d'un domaine D, intervalle d'entiers positifs, dans un ensemble image V : la spécification ci-dessus admet pour modèle les suites dont le domaine est un segment [0,n], avec $n \geq -1$.

A partir des opérations ci-dessus nous définissons les extensions suivantes :

1/ dom(s) = intervalle(0,lg(s))
qui délivre le domaine de définition d'une suite s.

Le type **intervalle d'entiers** (cf. définition en Annexe I) associe à deux entiers positifs l'intervalle des entiers compris entre ces deux bornes.

2/ La fonction **terme** de profil : SUITE(V) x ENT → V définit le terme d'une suite dont le rang est donné en argument, s'il fait partie du domaine de la suite.

terme(s,i) = si i = 0 alors prem(s) sinon terme(reste(s),i-1) fsi
avec pré terme(s,i) = (s ≠ s-vide ∧ i ∈ dom(s))

3/ est-vide(s) est équivalent au prédicat (s = s-vide).

Par la suite nous adopterons les conventions d'écriture suivantes :

- $(s_0 s_1 ... s_n)$ désigne la suite s de domaine [0...n]
- s_i est une abréviation pour terme(s,i)
- s(i...j) désigne la suite s-extrait(s,i,j)
- le domaine d'une suite est désigné par le nom de la suite écrit en majuscules : ex. S = dom(s)

I.2. Spécification du type TABLE

Intuitivement une table est définie comme une fonction de domaine I, ensemble quelconque totalement ordonné par une relation notée < , et de codomaine V.

Les éléments de la table, de type V, sont obtenus par une opération d'accès direct ; le domaine D d'une table est l'ensemble des éléments de I pour lesquels l'accès direct est défini.

Le type TABLE est paramétré par deux types : I, le type des indices de la table, objets du domaine D, et V le type des valeurs accédées.

La spécification algébrique suivante [figure 2] admet ce type pour modèle :

Description informelle	Enoncé formel
I type des indicatifs des éléments de la table muni de la relation d'ordre <	type TABLE (I : type, V : type)
V type des valeurs	opérations
ENT type des entiers naturels	table-vide : → TABLE
Ens-ord(I) ensemble des parties de I défini par le type abstrait "ensemble ordonné" dans [Per 85] et joint en Annexe I	ajouter : TABLE x I x V → TABLE retirer : TABLE x I → TABLE
table-vide et **ajouter** sont les constructeurs du type	domaine : TABLE → Ens-ord(I)
ajouter rend la table complétée par l'élément d'indice et de valeur donnés en argument (si cet indice ne figurait pas déjà dans le domaine)	accès : TABLE x I → V
retirer ôte du domaine de définition de la table l'indicatif donné en paramètre (s'il fait partie du domaine)	cardinal : TABLE → ENT
domaine définit le domaine d'une table à partir des opérations "ens-vid" et "union" du type "ensemble ordonné"	+ : TABLE x TABLE → TABLE
accès rend la valeur de l'élément dont l'indicatif est passé en paramètre, si cet indice appartient au domaine de la table	tr : TABLE x I x I → TABLE
	axiomes :
	soient t,t1,t2:TABLE ; i,j,k : I ; v : V
	retirer(table-vide,i)=table-vide
	retirer(ajouter(t,i,v),j) = si i = j alors t sinon ajouter(retirer(t,j),i,v) fsi
	domaine(table-vide) = ens-vide
	domaine(ajouter(t,i,v) = union(domaine(t),i)
	accès(ajouter(t,i,v),j) = si i = j alors v sinon accès(t,j) fsi

cardinal définit le nombre d'éléments d'une table (c'est-à-dire le nombre d'indicatifs auxquels sont associés des valeurs).	cardinal(table-vide) = 0 cardinal(ajouter(t,i,v))= cardinal(t)+1
+ délivre la table "union" des deux deux tables données en argument (union disjointe)	t1 + table-vide = t1 t1 + (ajouter(t2,i,v)) = si i ∈ domaine(t1) ou i ∈ domaine(t2) alors t1 + t2 sinon ajouter(t1 + t2,i,v) fsi
t r définit la "tranche" de la table paramètre dont les éléments ont des indicatifs compris entre les deux valeurs paramètres i et j	tr(table-vide,i,j) = table-vide tr(ajouter(t,k,v),i,j) = si i > j alors table-vide sinon si i ≤ k ≤ j alors ajouter(tr(t,i,j),k,v) sinon tr(t,i,j) fsi fsi restrictions pré ajouter(t,i,v)=non(i∈ domaine(t)) pré accès(t,j)= (j ∈ domaine(t))

Figure 2 : Spécification du type TABLE

• A partir des opérations du type TABLE nous définissons les extensions suivantes :

1/ dans? : TABLE x I → BOOLEEN
dans?(t,i) = (i ∈ domaine(t))

2/ vide? : TABLE → BOOLEEN
vide?(t) = (t = table-vide)

3/ min : TABLE → I
max : TABLE → I
min(t) = minimum(domaine(t))
max(t)= maximum(domaine(t))

où les opérations minimum et maximum, définies par le type "Ens-ord", délivrent respectivement le plus petit et le plus grand élément d'un ensemble quelconque totalement ordonné.

4/ premier : TABLE → V
dernier : TABLE → V
premier(t) = accès(t,min(t)) et dernier(t) = accès(t,max(t))

5/ mod : TABLE x V x I → TABLE :
modifie le contenu d'un élément de la table
mod (t,v,i) = ajouter(retirer(t,i),i,v)

Par la suite nous utiliserons le type TABLE avec un domaine défini sur les entiers positifs.

On remarque que le type SUITE peut être représenté par le type TABLE : intuitivement une suite est une table dont le domaine de définition est un intervalle d'entiers. Ainsi, au paragraphe III.3 certaines propriétés de tables qui représentent des suites sont héritées des suites qu'elles représentent. Dans ce but nous donnerons au paragraphe III 3.3. la définition de la représentation du type SUITE par le type TABLE.

II. LES RELATIONS DE COMMUNICATION

II.1. Notion de suite temporelle

Intuitivement une suite temporelle est une suite de couples (élément, date), croissante sur la composante "date" des couples.

Soit CHRONO un ensemble infini, discret, totalement ordonné par une relation notée ≤ et possédant un minimum noté ⊥ .

L'ensemble CHRONO, en association avec un ensemble quelconque ELT permet de définir des objets **datés** de type ELT. On appelle ELEM-DATE le type de ces objets : le type ELEM-DATE(ELT) est égal au produit cartésien (ELT x CHRONO). Les composantes d'un objet de ce type sont accédées par les deux sélecteurs respectifs "valeur" et "instant".

• Une **suite temporelle** s, définie à partir du type ELT, est une suite de valeurs de type ELEM-DATE(ELT) et de domaine S qui vérifie :

$$\begin{cases} s_0.\text{valeur} = \omega \wedge s_0.\text{instant} = \perp \wedge \\ \forall\ i,j \in S\ ,\ i < j \Rightarrow s_i.\text{instant} \leq s_j.\text{instant} \end{cases}$$

où ω désigne une valeur indéfinie et fictive dans l'ensemble ELT(*) .

Afin de pouvoir utiliser formellement la notion de **suite temporelle**, nous définissons [figure 3] le type abstrait **SUIT-TEMP** qui l'explicite.

Ce type abstrait, dérivé du type SUITE, est défini, de manière constructive à l'aide des opérations du type SUITE, par une spécification en termes de PRE et POST conditions [DeF 79].

Un certain nombre des opérations définies ici ont également été introduites dans [JJa 88] ; toutefois le type suite-temporelle défini dans cette publication n'est pas identique à celui développé ici : contrairement à notre définition la contrainte sur les instants des termes introduite dans cet article est une croissance **stricte** des instants. Le choix d'une croissance non stricte des instants des termes adopté ici sera justifié au paragraphe III puis au chapitre II.

Ainsi, par exemple, l'opération **"at"** définie dans [JJa 88] et qui délivre le terme d'une suite temporelle défini à un instant donné n'aurait pas un résultat déterministe sur un objet de type SUIT-TEMP puisque deux ou plusieurs termes consécutifs peuvent avoir des dates identiques. D'où la définition ici de deux opérations :

"at-max" qui délivre le dernier des termes défini à un instant donné
et
"at-min" qui délivre le premier de ces termes
(même démarche pour les opérations "index-max" et "index-min").

Le schéma donné [figure 4] illustre les principales opérations du type SUIT-TEMP.

(*) Les termes de rang 0 des suites temporelles sont des termes fictifs dont l'utilisation facilitera l'expression des propriétés.

Description informelle	Enoncé formel
Le type **SUIT-TEMP** est paramétré par ELT, le type de la composante "valeur" de ses termes ; il utilise le type ELEM-DATE et le type S-CHR des chronologies (suites croissantes, non strictement, d'instants) défini en Annexe I.1. Parmi toutes les suites à valeurs dans (ELT x CHRONO) on se restreint aux suites croissantes sur la composante "instant"	type SUIT-TEMP (ELT : type) = SUITE(ELEM-DATE(ELT)) spécification soit st: SUIT-TEMP invariant $(\forall\ i,j \in dom(st)\ ,\ i < j \Rightarrow st_i.instant \leq st_j.instant) \wedge st_0 = (<\omega,\perp>)$
	opérations
st-ajout construit une suite temporelle par adjonction en queue du couple <valeur, instant> formé par les 2 paramètres de type ELT et CHRONO	st-ajout(st: SUIT-TEMP, v : ELT, t : CHRONO) st' : SUIT-TEMP pré $(t \geq der(st).instant)$ post $st' = st * <v,t>$
valeur rend la suite composée des sélections "valeur" des éléments de la suite temporelle paramètre	valeur(st:SUIT-TEMP) : sv : SUIT-TEMP pré vrai post $dom(sv) = dom(st) \wedge (\forall\ i \in dom(sv), sv_i = st_i.valeur)$
instant définit la suite composée des sélections "valeur" des éléments de la suite temporelle paramètre et appelée **chronologie** de la suite paramètre	instant(st: SUIT-TEMP)sc:S-CHR pré vrai post $dom(sc) = dom(st) \wedge (\forall\ i \in dom(sc), sc_i = st_i.instant)$
index-max délivre le rang, dans la suite paramètre, du dernier des termes définis à l'instant t donné en paramètre	index-max(st:SUIT-TEMP, t: CHRONO) i:ENT pré vrai post $i \in dom(s)$ tel que $\forall\ j \in dom(s)$ $(j \leq i \Rightarrow st_j.instant \leq t \wedge j > i \Rightarrow st_j.instant > t)$

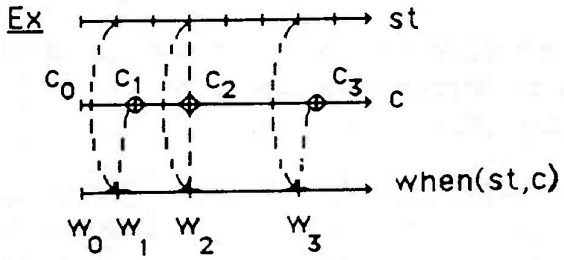
when construit une suite temporelle extraite de la suite st donnée en argument et "filtrée" par la chronologie, strictement croissante, c.  	when (st : SUIT-TEMP, c: S-CHR) st' : SUIT-TEMP pré ∀ i,j ∈ dom(c) i<j⇒ c_i<c_j post st' telle que dom(st')=dom(c)) ∧ (∀ i ∈ dom(c), st'_i = st_j avec j = index-max(st,c_i))
st-partie définit la suite composée des termes de st dont le rang est compris entre les valeurs des arguments i et j	st-partie(st:SUIT-TEMP,i:ENT, j:ENT) st':SUIT-TEMP pré vrai post st' = s-extrait(st,i,j)
s-temp construit la suite temporelle égale au produit cartésien des 2 suites arguments	s-temp(sv:SUITE(ELT),sc:SCHR) st:SUIT-TEMP(ELT) pré dom(sv) = dom(sc) post st telle que valeur(st)=sv instant(st) = sc
at-max délivre le terme de la suite argument d'indice défini par la fonction index-max appliquée à la suite et à l'instant paramètres	at-max(st:SUIT-TEMP,t:CHRONO) e:ELEM-DATE pré vrai post e = st_i avec i=index-max(st,t)
pred délivre le rang du terme de la suite argument dont l'instant est immédiatement inférieur, strictement, à celui du terme défini par index-max(st,t)	pred(st:SUIT-TEMP,t:CHRONO)i: ENT pré vrai post i = si index-max(st,t)= 0 alors 0 sinon max {j ∈ dom(st) / st_j.instant < at-max(st,t) .instant} fsi

Figure 3 : Spécification du type SUIT-TEMP

• A partir des opérations de base du type SUIT-TEMP nous définissons les extensions suivantes :

1/ st-tranche (st:SUIT-TEMP, i:ENT, t:CHRONO) : définit la suite des termes de st de rangs supérieurs ou égaux à i et d'instants inférieurs ou égaux à t.

st-tranche (st,i,t) = st-partie (st,i,j) avec j = index - max (st,t)

2/ st-jusqu'à (st:SUIT-TEMP,t:CHRONO) : délivre la suite des termes de st d'instants inférieurs ou égaux à la valeur de t :

st-jusqu'à (st,t) = st-tranche (st,0,t)

3/ st-entre (st:SUIT-TEMP,t:CHRONO,t':CHRONO) : définit la suite des termes de st d'instants supérieurs à t et inférieurs ou égaux à t'

st-entre (st,t,t') = st-partie (st,i+1,j) avec i = index-max(st,t)
j = index-max(st,t')

4/ st-après (st:SUIT-TEMP,t:CHRONO) : définit la suite des termes de st d'instants supérieurs au paramètre t

st-après (st,t) = st-partie (st,j+1,lg(st)) avec j = index-max(st,t)

5/ les projections sur la composante valeur des suite définies par les cinq opérations d'extraction de sous-suites ci-dessus sont :

partie :SUIT-TEMP x ENTxENT → SUITE (ELT)
partie (st,i,j) = valeur (st-partie(st,i,j)

tranche : SUIT-TEMP x ENT x CHRONO → SUITE(ELT)
tranche(st,i,t) = valeur(st-tranche(st,i,t))

jusqu'à : SUIT-TEMP x CHRONO → SUITE(ELT)
jusqu'à(st,t) = valeur(st-jusqu'à (st,t))

entre : SUIT-TEMP x CHRONO x CHRONO → SUITE(ELT)
entre(st,t,t') = valeur(st-entre(st,t,t'))

après : SUIT-TEMP x CHRONO → SUITE(ELT)
après(st,t) = valeur(st-après(st,t))

6/ La fonction index-min définit le rang du premier des termes d'instants égaux à l'instant donné en paramètre, s'il existe, et sinon rend l'indice défini par la fonction index-max : il s'agit dans tous les cas du successeur de l'indice défini par la fonction pred.

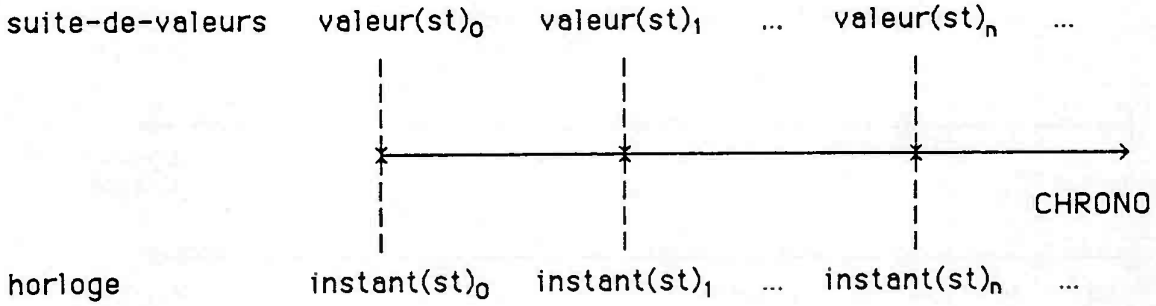
index-min : SUIT-TEMP x CHRONO → ENT

index-min(st, t) = si pred(st,t) = 0 alors 0 sinon pred(st,t) + 1 fsi

7/ Le terme défini par la fonction index-min est obtenu par la fonction at-min de profil SUIT-TEMP x CHRONO → ELEM-DATE :

at-min(st, t) = st_i avec i = index-min(st,t)

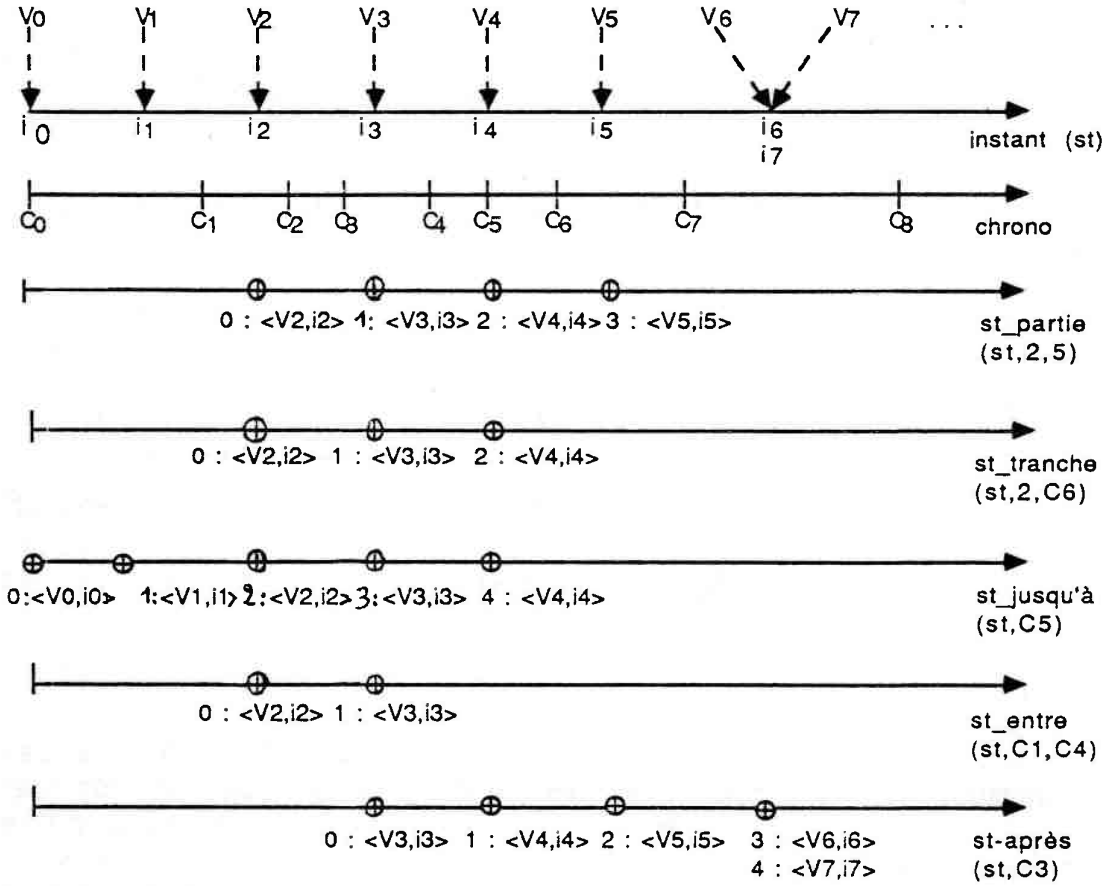
• Afin de schématiser certaines propriétés sur les suites temporelles, nous adopterons une présentation graphique de la forme :



où on appelle "suite-de-valeurs" la suite de type ELT définie par l'opération "valeur" pour une suite temporelle donnée, et "horloge" la suite résultat de l'opération "instant" (cette suite est appelée "chronologie" dans [JJa 88]).
Pour simplifier les schémas, on omettra de représenter les termes la suite horloge.

Remarque :

Dans la suite de ce chapitre, nous n'utiliserons pas l'ensemble des opérations sur les suites temporelles définies ci-dessus ; toutefois l'étude de relations de communication moins simples que celles que nous détaillons dans les exemples, notamment au paragraphe III.3, nécessite ces opérations diverses.



Légende :

- L'axe de référence est la droite chrono

- On note v_j la valeur du $j^{ème}$ terme de la suite de référence st, et i_j son instant.

- Pour les autres suites on matérialise chaque terme par ⊕ en indiquant par j:<valeur,instant> son rang et son contenu.

- Sur cet exemple

index-max(st,c4) = 3	at-max(st,c4) = <v3,i3>
index-max(st,c5) = 4	
index-min(st,c8) = 6	index-max(st,c8) = 7
pred(st,c8) = 5	pred(st,c4) = 2

Figure 4 : Les principales opérations du type SUIT-TEMP

Pour conclure ce paragraphe nous proposons la remarque suivante :

La notion de suite temporelle est similaire à la notion de variable dans le langage LUSTRE [CPH 87]. Une variable LUSTRE est représentée par une suite infinie de valeurs associée à une suite infinie de valeurs booléennes, appelée horloge, qui définit les instants d'apparition des valeurs de la variable, par rapport à une horloge de référence.

L'utilisation de ces deux notions dans leurs contextes respectifs est cependant différente : les horloges de LUSTRE servent à définir des opérations synchrones sur les flux de données tandis que la notion de suite temporelle permet de spécifier des relations temporelles asynchrones entre suites de données ainsi que nous allons le montrer ci-après.

II.2. Notion de relation de communication

II.2.1. Introduction

Cette notion introduite dans [Per 85] permet de **spécifier** des systèmes qui réalisent la solution de problèmes en définissant une ou des suites résultats à partir d'une ou plusieurs suites de données et en prenant en compte l'état courant du système pour définir chaque terme de la suite résultat.

Ainsi par exemple la relation de communication triviale que nous appelons "égalité" spécifie un système composé d'un processus "*producteur*" et d'un processus "*consommateur*" tel qu'à tout instant de communication :

- la valeur utilisée par le processus consommateur est la plus ancienne valeur émise par le producteur et non encore consommée,

- le processus producteur peut avoir plusieurs productions d'avance par rapport au processus consommateur.

Une stratégie totalement différente est par exemple celle qui sera spécifiée par la relation "à-retard-borné" :

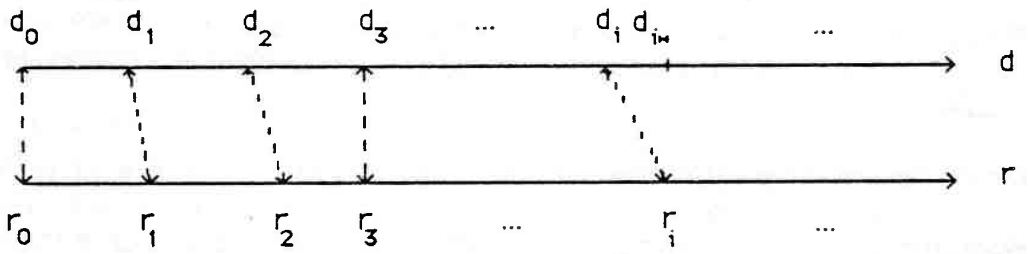
- toute valeur reçue par le processus consommateur est celle émise le plus récemment,

- le processus producteur peut être "en retard" vis à vis du consommateur et dans ce cas une valeur pourra être "consommée" plusieurs fois consécutivement ; cependant le retard pris par le processus producteur est borné : la différence entre le nombre d'occurrences de consommations et le nombre d'occurrences de productions reste bornée.

L'introduction explicite du temps au niveau des suites de données et de résultats permet de formaliser la notion "d'état courant" par des propriétés temporelles entre ces suites : la solution d'un problème est alors spécifiée par des relations dites de **communication** entre **suites temporelles** de données et de résultats.

Ainsi la relation de communication dite d'*égalité* lie une suite de données d et une suite de résultats r dont les termes sont identiques à un décalage dans le temps près, c'est-à-dire formellement :
$dom(r)=dom(d) \wedge (\forall\ i \in dom(r),\ r_i.valeur=d_i.valeur \wedge r_i.instant \geq d_i.instant)$

Cette relation peut être schématisée de la manière suivante :



Ainsi une relation de communication exprime un ensemble de contraintes que doivent vérifier les suites de valeurs échangées à l'intérieur d'un système, indépendamment de la manière dont ces suites seront construites. C'est la notion de type de communication, que nous développerons au paragraphe III, qui réalise la construction des relations.

Nous nous limitons ici à la définition de relations binaires de communication.

II.2.2. Présentation informelle et exemples

Une relation de communication $\mathcal{R}$ (de type ELT) est une relation binaire sur l'ensemble des suites temporelles de type ELT qui vérifie :

(P)

Soient d,r : SUIT-TEMP(ELT),
$\langle d,r\rangle \in \mathcal{R}$ ssi $\exists\, \varphi$, application : R → D telle que
$\forall\, j \in R, \forall\, i \in D, i = \varphi(j) \Rightarrow$
- rj.valeur = di.valeur
- rj.instant ≥ di.instant
- τ (d,r,i,j)

où τ est une relation sur $(SUIT\text{-}TEMP(ELT))^2$ x ENT x ENT qui **caractérise** $\mathcal{R}$, D et R désignent les domaines respectifs des suites d et r (de type [0,n]).

Remarques :

(1) On a vu en II.1. que les termes de rang 0 des suites temporelles, de valeur indéfinie ω et d'instant minimum $\perp$, sont des éléments fictifs. La définition d'une relation de communication est simplifiée par l'introduction de ces termes moyennant la convention suivante : toute relation τ, caractéristique d'une relation de communication $\mathcal{R}$, vérifie $\tau(d,r,0,0)$ pour tout couple $\langle d,r\rangle$ de suites temporelles.

(2) Comme nous ne nous intéressons ici qu'aux relations de communication binaires, les chronologies des suites temporelles mises en jeu sont **strictement** croissantes : intuitivement à un instant donné une seule valeur peut être émise (par un processus producteur) ou reçue (par un processus consommateur). Par conséquent l'utilisation des opérations index-max ou index-min sera indifférente dans les énoncés de propriétés de ce paragraphe.

(3) Nous ne nous intéressons qu'à des relations entre suites temporelles finies : intuitivement nous ne cherchons à spécifier des communications qu'entre des processus finis.

Exemples :

- Ex. 1 : relation "égalité"

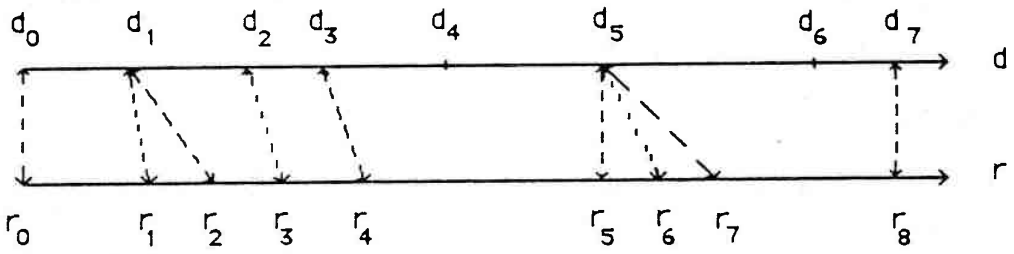
La relation $\tau(d,r,i,j) \equiv (i \in D \wedge j \in R \wedge i = j)$ définit une relation de communication triviale entre deux suites dont les valeurs des éléments sont identiques.

- Ex. 2 : relation "à-retard-borné" (par un entier p)

Cette relation de communication décrite intuitivement ci-dessus est caractérisée par une relation τ définie par :

$$\tau(d,r,i,j) \equiv (i \in D \wedge j \in R \wedge i = \text{at-max}(d, r_j.\text{instant}) \wedge i \geq j - p)$$

Le schéma ci-dessous illustre cette relation pour p = 2 :



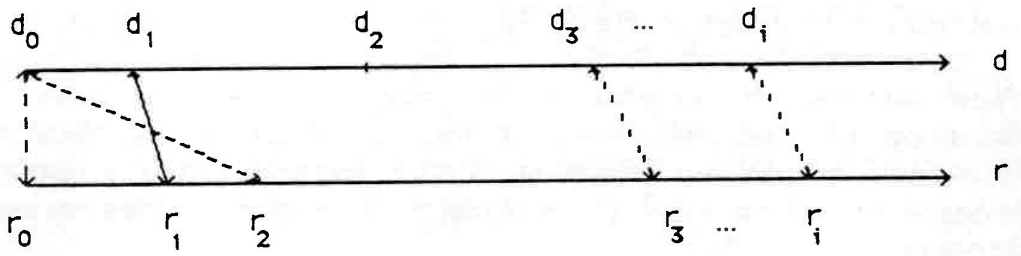
- Ex. 3 : relation "égalité-presque-partout"

Cette relation de communication est telle que les suites-de-valeurs associées aux suites temporelles d et r sont identiques à l'exception des éléments de rang i tels que l'instant de l'élément i dans la suite temporelle d est supérieur à celui du terme du même rang dans la suite temporelle r ; pour de tels éléments de la suite r, la composante "valeur" est égale à la valeur indéfinie "ω".

La relation τ associée est définie par :

$$\tau(d,r,i,j) \equiv (i \in D \wedge j \in R \wedge ((i = j \wedge d_i.\text{instant} \leq r_j.\text{instant}) \vee (i = 0 \wedge d_j.\text{instant} > r_j.\text{instant})))$$

Un exemple de suites temporelles liées par cette relation est figuré ci-dessous :



- Ex. 4 : relation "croissante" (définie pour un type ELT muni d'une relation d'ordre strict >, telle que ω soit la valeur minimum)

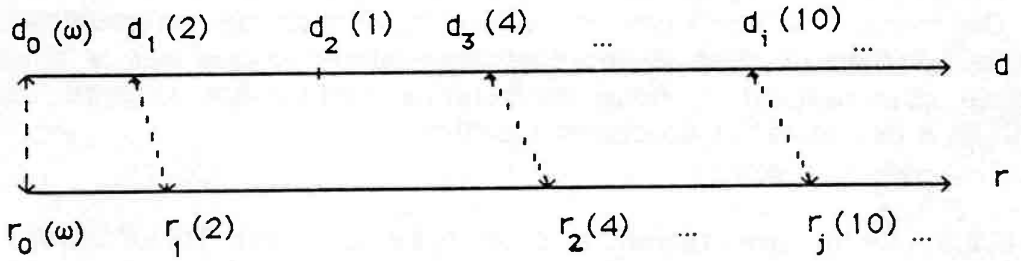
Cette relation est telle que la suite-de-valeurs associée à la suite temporelle r est une suite strictement croissante extraite de la suite valeur (d).

La relation caractéristique τ est définie par :

$\tau(d,r,i,j) = [i \in D \wedge j \in R \wedge ((i = 0 \wedge j = 0) \vee (j > 0 \wedge (d_i \bullet valeur > r_{j-1} \bullet valeur) \wedge$

$\exists i' \in D\ (\tau(i',j-1) \wedge i'<i \wedge \forall k \in D\ (i'<k<i) \Rightarrow d_k \bullet valeur \leq r_{j-1} \bullet valeur))]))]$

Le schéma ci-dessous illustre cette relation (les valeurs des termes des suites d et r sont indiquées entre parenthèses).



Remarques sur l'exemple 4 :

(1) La relation τ est "fonctionnelle sur j" c'est-à-dire que pour tout j de R il existe au plus un indice i de D tel que $\tau(d,r,i,j)$; de ce fait l'indice i' dans la définition de τ est désigné de manière unique.

(2) La relation τ est définie même lorsque la suite r n'est pas une sous-suite croissante de d (sur la projection valeur) : par exemple les suites d et r dont les suites de valeurs sont :

valeur(d) = (ω,1,0,1,0,1,...) et

valeur(r) = (ω,0,0,0,...)

vérifient $\tau(d,r,0,0)$ $\tau(d,r,1,1)$ $\tau(d,r,3,2)$ ...

Ainsi on met en évidence le fait que c'est l'ensemble des trois prédicats de la propriété $\mathcal{P}$ qui définit un couple de la relation de communication et pas seulement la relation caractéristique τ (On aurait pu énoncer le même type de remarque pour chacun des exemples précédents).

• Les exemples ci-dessus sont des exemples simples de stratégies de communication : d'autres exemples de relation de communication sont donnés en Annexe 2, en particulier la relation "ascenseur" qui permet de spécifier la gestion d'une unité de disque et illustre un système concret non trivial.

Les quatre exemples ci-dessus ont été choisis pour leur représentativité. Ce sont des relations qui conservent la chronologie des termes d'une suite à l'autre. La relation "égalité" est la plus simple des relations de communication et aussi l'une des moins asynchrones, comme il est montré dans [Per 85] ; la relation "à-retard-borné" est un exemple de relation de communication telle qu'une valeur peut être consommée plusieurs fois ; la relation "égalité-presque-partout" est un exemple de relation utilisant la valeur indéfinie "ω" : nous en verrons la signification aux chapitres suivants ; la relation "croissante" est un exemple de relation où les valeurs des termes sont prises en compte.

De manière à formaliser la notion de relation de communication et dans le but d'en prouver sa représentation algorithmique par la notion de type de communication, nous définissons maintenant le type abstrait RELCOMM des relations de communication.

II.2.3. Définition formelle : le type abstrait RELCOMM

Ce type abstrait est paramétré par τ la relation caractéristique d'une relation de communication et ELT le type des valeurs communiquées. Nous définissons le type RELCOMM (τ,ELT) comme l'ensemble des couples du graphe de la relation définie ; l'invariant de ce type est la propriété $\mathcal{P}$ énoncée en II.2.2.

La spécification proposée est de type axiomatique et utilise les opérations du type ENSEMBLE (supposé prédéfini) et du type SUIT-TEMP.

• **Spécification du type RELCOMM**

Description informelle	Enoncé formel
la relation τ est un élément de l'ensemble PSTI(ELT) des prédicats sur le domaine : (SUIT-TEMP(ELT) x SUIT-TEMP(ELT) x ENT x ENT) (PSTI : Prédicats sur les Suites Temporelles et leurs Indices) ELT est le type des valeurs communiquées ENS est le type "ensemble" D et R sont les domaines respectifs des suites temporelles d et r.	type RELCOMM(τ:PSTI(ELT), ELT:type)= ENS (<d:SUIT-TEMP(ELT), r:SUIT-TEMP(ELT)> spécification soit $\mathcal{R}$: RELCOMM(τ,ELT) invariant $\forall$ d,r:SUIT-TEMP(ELT),< d,r> $\in \mathcal{R}$ $\Leftrightarrow$ $\exists \varphi$:R $\rightarrow$ D telle que $\forall j \in R, \forall i \in D$ $i=\varphi(j) \Rightarrow \begin{cases} r_j.valeur = d_i.valeur \\ r_j.instant \geq d_i.instant \\ \tau(d,r,i,j) \end{cases}$

• On remarque,d'après l'invariant du type RELCOMM, que pour une suite temporelle d donnée il existe une infinité de suites temporelles r liées à d par la relation spécifiée : en effet la relation entre les horloges des suites d et r est une contrainte lâche sur les instants des termes liés par τ.

Or il s'agit de ne pas perdre de vue le domaine d'interprétation des relations de communication qui est celui des systèmes parallèles et les dates d'une suite résultat ne sont pas quelconques ; schématiquement une relation de communication lie deux processus : le premier émet une suite de valeurs que le second utilise (consomme) :

- selon une certaine stratégie, que l'on caractérise par τ,

- à des dates particulières définies d'une part par les dates des **requêtes** émises par le processus consommateur (intuitivement les

dates auxquelles il est prêt à recevoir une donnée communiquée), et d'autre part par l'**état courant** du système au moment des requêtes.

De plus la suite des dates des requêtes d'un processus n'est pas quelconque : un processus consommateur est bloqué dans l'attente d'une valeur et ne peut donc émettre de nouvelle requête tant que la précédente n'est pas satisfaite ; autrement dit, formellement, en notant req la suite des requêtes :

$\forall\ j \in dom(req)\ ,\ req_j \leq r_j.instant < req_{j+1}$

• Ainsi la prise en compte des dates des requêtes permet-elle de **caractériser** une suite résultat r pour une suite temporelle donnée d.

On appelle "communic" le prédicat qui lie une suite temporelle donnée, appelée **suite temporelle produite**, une suite de **requêtes de consommation**, suite d'instants croissants, et la suite **temporelle consommée** correspondante.

Cette opération du type RELCOMM a pour profil :

SUIT-TEMP(ELT) x S-CHR x SUIT-TEMP(ELT) → BOOL

et elle est définie par :

$$\begin{aligned} communic_{\mathcal{R}}(d,req,r) \equiv [&<d,r> \in \mathcal{R} \wedge dom(r) = dom(req) \wedge \\ &(\forall\ j \in dom(req),\ req_j \leq r_j.instant < req_{j+1}) \wedge \\ &(\forall\ j > 0 \in R,\ r_j.instant = \max(req_j,\ d_i.instant)) \\ &\quad \underline{avec}\ i \in D\ \underline{tel\ que}\ i = \varphi_{\mathcal{R}}(j)] \end{aligned}$$

Remarques :

1- Il n'est pas possible de donner une définition fonctionnelle d'une suite temporelle consommée à partir d'une suite temporelle produite donnée et d'une suite de requêtes : en effet la suite de requêtes de consommation est liée à la suite consommée par la contrainte $(\forall\ j \in dom(req),\ req_j \leq r_j.instant < req_{j+1})$.

2- L'équation qui définit l'instant de la $j^{ème}$ valeur consommée signifie : lorsque la $j^{ème}$ requête est émise si la valeur définie par le prédicat $\tau(d,r,i,j)$ est présente alors elle est immédiatement "consommée" sinon la $j^{ème}$ requête ne sera satisfaite que lorsque cette valeur aura été émise.

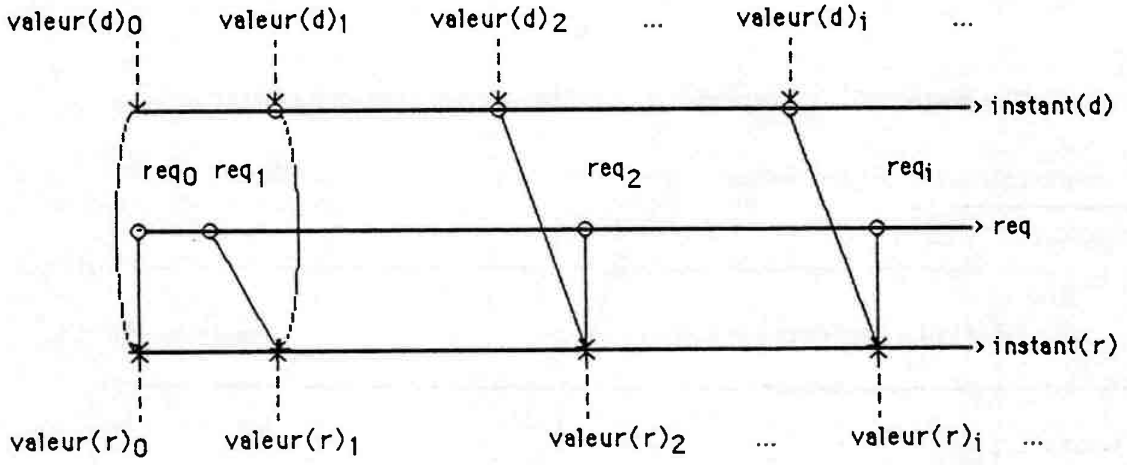
3- Toute relation τ, caractéristique d'une relation de communication, n'est pas forcément déterministe : le choix de l'indice i de D lié par τ à un indice j de R, lorsque plusieurs sont possibles n'est pas fixé par la spécification et repoussé à l'étape de représentation algorithmique.

4- Lorsque la relation τ est déterministe, la suite r est définie de manière unique pour un couple <d,req> donné.

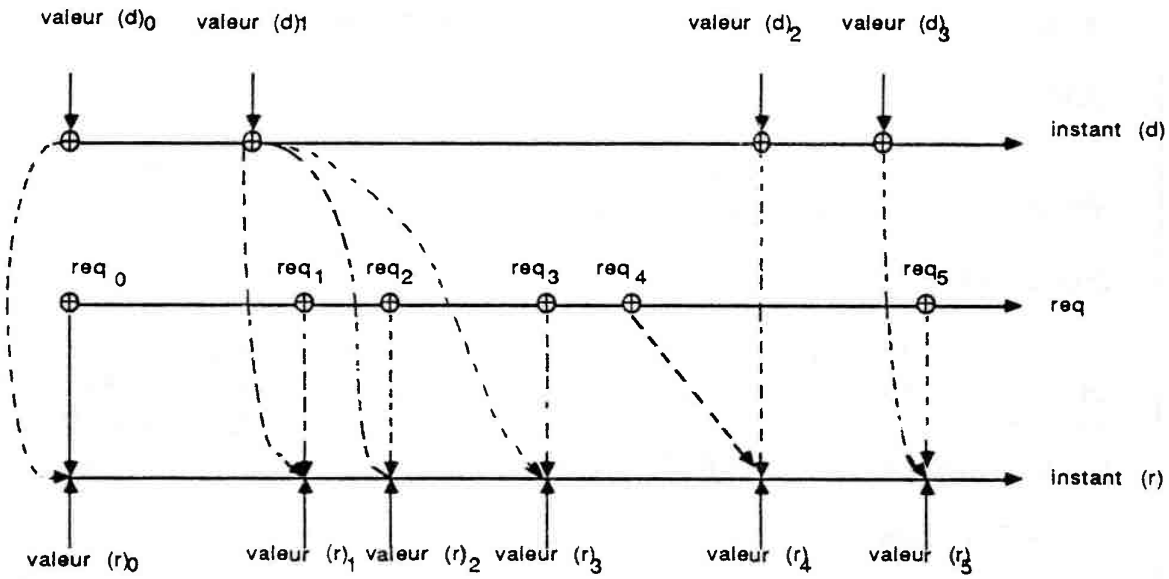
Exemples :

Les schémas ci-dessous illustrent les quatre relations de communication définies en II.2.2. pour une suite temporelle consommée en correspondance avec une suite temporelle produite et une suite de requêtes de consommation données.

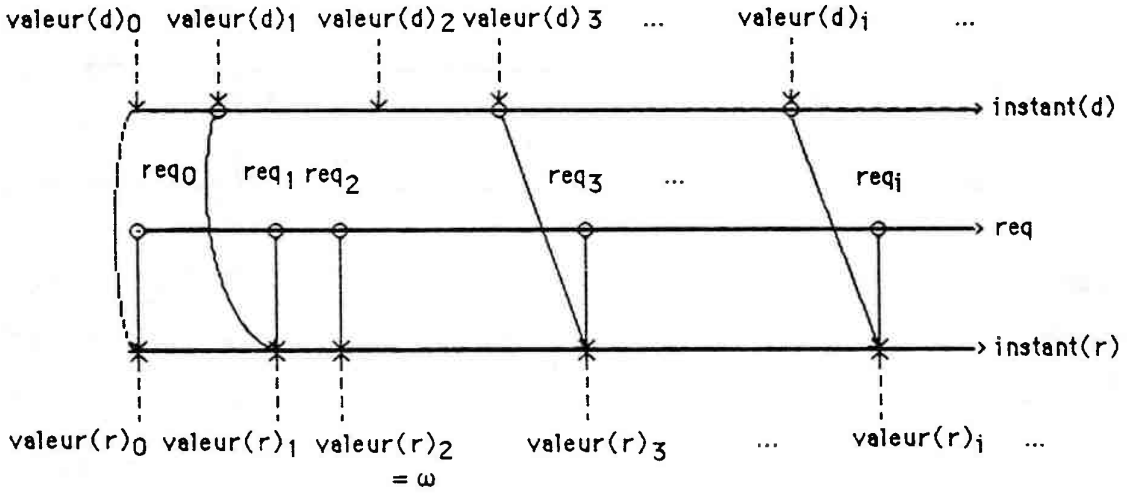
- Exemple 1 : relation ***égalité***



- Exemple 2 : relation ***à-retard-borné*** (par p = 2)



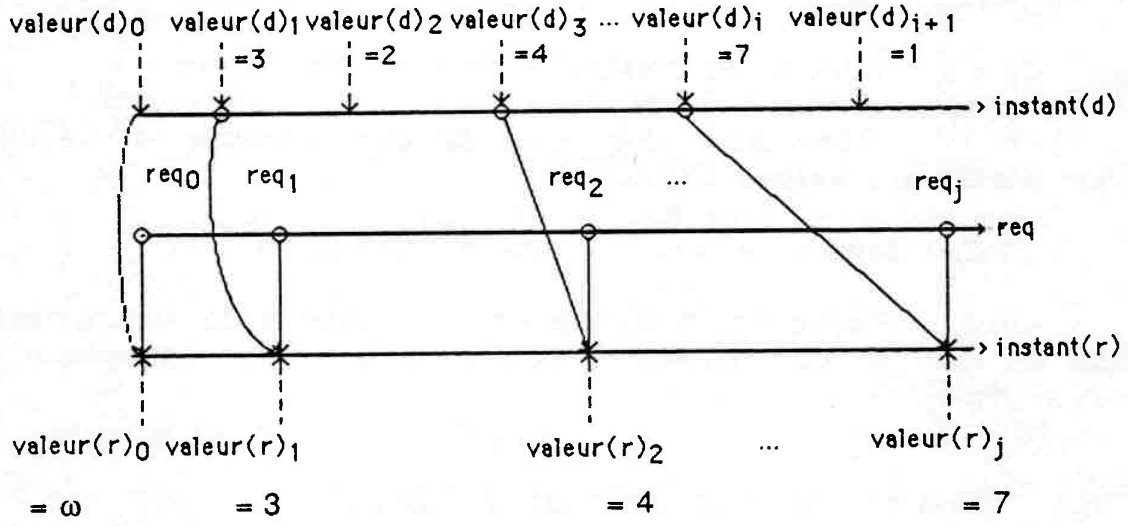
- Exemple 3 : relation ***égalité-presque-partout***



Remarque : La combinaison de l'inégalité $req_j \leq r_j.instant$ et de la définition du prédicat τ associé à cette relation entraîne que $r_j.instant = req_j$, pour toute requête de consommation : ainsi le processus consommateur n'est jamais bloqué dans l'attente d'une valeur,

la valeur 'ω' remplaçant la donnée communiquée lorsque celle-ci est produite "trop tard".
Ainsi, intuitivement, la relation "égalité-presque-partout" est plus asynchrone que la relation "égalité" : nous ne reviendrons pas ici davantage sur la classification des relations de communication donnée dans [Per 85] au chapitre V.

- **Exemple 4 :** relation ***croissante***



III. Notion de "type de communication"

La solution retenue pour l'expression algorithmique des relations de communication est une expression en termes de types abstraits de données, appelés **types de communication**. Cette représentation répond à l'objectif de modularité visé par l'expression algorithmique des systèmes parallèles, ainsi que nous le verrons au chapitre II, et permet de valider l'expression obtenue en regard de la spécification (cf. § III.3.).

Au-delà de la présentation des types de communication, décrite dans plusieurs articles, [JJu 81], [JuP 85], [JuM 86], ..., l'objectif de ce paragraphe III est double :

- d'une part prouver que pour toute relation de communication, le type de communication associé réalise correctement la construction d'un couple de suites temporelles liées par cette relation,

- d'autre part définir formellement les bases de l'interprétation des communications qui sera présentée au chapitre III.

Pour réaliser cet objectif nous proposons de définir la sémantique des types de communication par leur interprétation dans le domaine des suites temporelles.

Ce paragraphe est donc structuré de la manière suivante :

- III.1. : Présentation des types de communication et définition formelle du type abstrait TYPECOM

- III.2. : Sémantique des types de communication

- III.3. : Preuve de représentation des relations de communication par les types de communication : schéma de preuve et instanciation sur un exemple.

III.1. Définition du type abstrait TYPECOM

III.1.1. Présentation intuitive

Un type de communication est une **représentation algorithmique** d'une relation de communication entre des suites **temporelles** ; dans cette affirmation les trois mots clés sont : représentation, algorithmique et temporelle :

- **représenter** parce qu'il s'agit d'associer à tout couple de suites temporelles vérifiant une relation de communication $\mathcal{R}$ un objet du type de communication associé à $\mathcal{R}$,

- **algorithmique** signifie que les opérations d'un type de communication doivent permettre de construire de proche en proche l'objet représentant d'un couple de suites temporelles,

- **temporel** : ce qualificatif est important parce que les objets à représenter ne sont pas usuels dans la mesure où une suite temporelle est une suite de valeurs datées : ainsi la valeur, à un instant

d'observation donné, d'un objet de type "type de communication" doit-elle représenter l'état des communications à cet instant d'observation c'est-à-dire l'ensemble des termes de la suite temporelle produite définis à cet instant de même que ceux de la suite temporelle consommée.

• La structure de données retenue pour décrire les objets du type est composée :

- d'une suite de valeurs, appelée **"suite produite"** et notée **SP** : c'est une section commençante de la suite des valeurs des éléments de la suite temporelle produite.

- de la suite de valeurs construite par l'opération "CONSOMMER" du type "type de communication", pour une suite produite et une suite de requêtes de consommation données ; cette suite est appelée **"suite consommée"** et notée **SC** ; nous montrerons qu'à la $j^{ème}$ réalisation de l'opération CONSOMMER la suite SC est composée des valeurs des j premiers termes de la (*) suite temporelle consommée définie par la relation de communication,

- d'un sous-ensemble des valeurs de la suite produite appelée **"ensemble consommable"** et noté **A** : intuitivement il représente l'ensemble des valeurs dont on peut disposer pour définir chaque terme de la suite consommée **SC** ; la construction de cet ensemble est définie, spécifiquement à chaque type de communication, de manière à représenter efficacement les propriétés caractéristiques de la relation de communication.

• Ce triplet, noté <SP, A, SC>, est manipulé par deux opérations :

- PRODUIRE, qui exprime la construction de la suite temporelle produite par adjonction d'une valeur à la suite SP et à l'ensemble A,

- CONSOMMER, qui exprime la construction de la suite temporelle consommée en réalisant à chaque requête de consommation l'extraction d'une valeur de l'ensemble A, modification de cet ensemble et adjonction de cette valeur à la suite SC. Cette opération est soumise à

(*) Ou de l'une d'elles si plusieurs sont possibles dans le cas d'une relation τ non déterministe.

une pré-condition : intuitivement l'activation de cette opération n'est pas toujours possible au moment où la requête est émise.

Remarque :

Cette structure de données n'est pas la seule représentation possible des objets du type. En effet, s'il est nécessaire de représenter les suites temporelles produites et consommées (ici par les suites SP et SC), l'ensemble A n'est introduit que pour faciliter la définition de l'opération CONSOMMER, sans manipuler les instants des suites temporelles.

D'autres représentations ont été proposées dans divers travaux du groupe COMETE ; par exemple un marquage des éléments de la suite produite pour identifier les termes déjà consommés dans [Cha 86].

Dans [Car 86] les objets d'un type de communication sont composés:

- d'une "suite temporelle" produite,
- d'une "suite temporelle" consommée,
- d'une horloge qui date chaque objet du type pour permettre la définition de chaque terme de la suite temporelle consommée, pour une suite temporelle produite et une suite de requêtes de consommation données.

De plus le type "suite temporelle" retenu est un type plus complexe que celui que nous proposons : chaque terme d'une suite temporelle est muni d'un champ supplémentaire appelé "numéro" qui l'identifie et permet de retrouver, parmi les éléments de la "suite temporelle" produite, ceux qui ont déjà été consommés, à un instant donné de l'horloge du type.

L'avantage de la représentation de [Car 86] est d'être proche de la spécification ; son inconvénient est de manipuler explicitement les instants des suites temporelles. Avec la structure de données que nous retenons l'aspect **"temps"** des suites temporelles produite et consommée est représenté par l'évolution des suites de valeurs SP et SC à chaque activation des opérations PRODUIRE et CONSOMMER ; la définition d'un ensemble des valeurs "consommables" aux instants de requête de consommation est alors indispensable à la définition de l'opération CONSOMMER.

III.1.2. Spécification du type TYPECOM

Ce type abstrait est défini par une spécification en termes de PRE et POST conditions.

Les types support sont :

- le type SUITE : type des suites de valeurs SP et SC

- le type TABLE dans lequel est représenté l'ensemble A. En effet les propriétés caractéristiques de la relation de communication qu'il s'agit de représenter sont des propriétés quelconques sur les domaines, les valeurs et les instants des suites temporelles produite et consommée : la définition de l'opération CONSOMMER qui traduit ces propriétés sur l'ensemble A nécessite l'accès à tout élément de cet ensemble par son "indice", la définition du cardinal de cet ensemble, etc... ; ainsi la structure de table que nous avons définie au paragraphe I est bien adaptée à la représentation de l'ensemble A.

Lorsque l'expression de l'opération CONSOMMER est complexe, des définitions intermédiaires, portant généralement sur des parties de l'ensemble A, sont introduites (cf. le type "croissante" donné en exemple et le type "ascenseur" donné en Annexe 2).

• Le type TYPECOM est paramétré par :

- ELT le type des données communiquées
- les trois opérations caractéristiques du type de communication.

Ces trois opérations et leurs profils sont les suivants :

- **pré_cons**, de profil SUITE * TABLE * SUITE → BOOL, qui définit la précondition de l'opération de consommation

- **val_cons**, de profil SUITE * TABLE * SUITE → ELT, qui définit la valeur sélectionnée dans A lors de l'opération CONSOMMER

- **post_cons**, de profil SUITE * TABLE * SUITE → TABLE, qui définit le nouvel état de l'ensemble consommable suite à l'opération CONSOMMER.

Pour alléger l'écriture on notera T le triplet d'opérations caractéristiques d'un type de communication et on notera T.pré_cons, T.val_cons et T.post_cons les trois opérations respectives.

Par convention et pour être cohérent avec la définition des relations de communication on définit un objet particulier du type TYPECOM résultat de l'opération INIT et composé des suites SP et SC réduites à la valeur indéfinie "ω" et de la table A vide.

La spécification du type TYPECOM est donnée par la figure 5 ci-dessous :

```
type TYPECOM(ELT: type, T) = <SUITE(ELT),TABLE(ELT),SUITE(ELT)>

        spécification

        notation : tc = <SP,A,SC>

        opérations

         INIT () tc : TYPCOM
              pré : vrai
              post : <(ω), table-vide, (ω)>

         PRODUIRE(tc:TYPECOM, e:ELT) tc':TYPECOM
           pré  : vrai
           post : tc' = <SP',A',SC>
             avec    SP' = SP * e
                     A' = ajouter(A,lg(SP)+1,e)

         CONSOMMER(tc:TYPECOM) tc':TYPECOM
           pré  : T.pré_cons(<SP,A,SC>)
           post : tc' = <SP,A',SC'>)
             avec    A' = T.post_cons(<SP,A,SC>)
                     SC' = SC * T.val_cons(<SP,A,SC>)

fin TYPECOM
```

Figure 5 : Spécification du type TYPECOM

A titre d'exemple, nous donnons les quatre types de communication associés aux relations présentées au paragraphe II.

1- **le type de communication "*égalité*" :**

Les trois opérations caractéristiques de ce type sont les suivantes:

```
pré_cons(<SP,A,SC>) = non vide? (A)
val_cons(<SP,A,SC>) = premier(A)
post_cons(<SP,A,SC>) = retirer (A,min(A))
```

2- **le type de communication "*à-retard-borné*" :**

```
pré_cons(<SP,A,SC>) = non vide? (A)
val_cons(<SP,A,SC>) = dernier (A)
post_cons(<SP,A,SC>) =  si cardinal(SC) - max(A) < p
                        alors finir (A)  sinon  tuer (A)  fsi
```

où les opérations "finir" et "tuer" sont des extensions du type TABLE signifiant respectivement "ôter de la table toutes les entrées sauf celle d'indicatif maximum" et "rendre la table vide" ; elles sont définies formellement par :

```
finir (t) = ajouter(table-vide, max(t), accès(t,max(t)))
tuer (t) = table-vide.
```

3- **le type de communication "*égalité-presque-partout*" :**

```
pré_cons(<SP,A,SC>) = vrai
val_cons(<SP,A,SC>) = si dans? (A,lg(SC)+1)    alors
                      accès   (A,max(SC)+1)    sinon ω fsi

post_cons(<SP,A,SC>) = si dans? (A,lg(SC)+1)   alors
                       retirer   (A,max(SC)+1) sinon A fsi
```

4- **le type de communication "*croissante*"**

Les trois opérations caractéristiques de ce type sont définies à l'aide d'une suite intermédiaire : la suite des indicatifs des éléments de A supérieurs à la dernière valeur consommée, que nous notons ind-p-gds.

```
pré_cons(<SP,A,SC>) = non(est-vide (ind-p-gds(A,der(SC))))
val_cons(<SP,A,SC>) = accès(A,prem(ind-p-gds(A,der(SC))))
post_cons(<SP,A,SC>) = retirer(A,prem(ind-p-gds(A,der(SC))))
```

où la fonction intermédiaire "ind-p-gds" de profil

TABLE x ELT → SUITE(ENT)

délivre la suite des indicatifs, dans la table paramètre, des éléments de valeur supérieure à la valeur donnée en paramètre.

ind-p-gds(t,v) = si t = table-vide alors s-vide sinon
si dernier(t) > v alors ind-p-gds(retirer(t,max(t)),v)*max(t)
sinon ind-p-gds(retirer(t,max(t)),v) fsi fsi

• Par la suite nous conviendrons d'appeler "état de communication" l'état courant d'un objet du type TYPECOM.

III.2. Sémantique des types de communication (*)

III.2.1. Présentation intuitive

Rappelons l'objet de ce paragraphe III qui est de prouver que les types de communication sont une représentation algorithmique valide des relations de communication.

Classiquement, [Gau 80] [DeF 79] [Rem 82] , définir la représentation d'un type par un autre passe par les étapes suivantes :

- définir la représentation d'un objet du type à représenter (abstrait) par un objet du type représentant (concret)

- définir la représentation des opérations du type abstrait par des opérations du type concret

- montrer que l'invariant du type origine est conservé par la fonction de représentation, i.e. que l'invariant concret "implique" l'invariant abstrait.

Nous suivons cette approche en l'adaptant de manière à intégrer l'aspect "algorithmique" de la représentation et l'aspect "temporel" des objets à représenter que nous avons mis en évidence au paragraphe III 1.1.

Nous avons vu au paragraphe III 1.1., de manière intuitive, que pour une suite temporelle produite d et une suite de requêtes de

(*) Dans tout ce paragraphe on abrègera le nom d'un type de communication TYPECOM(V,T) par T.

consommation req données, les activations successives des opérations PRODUIRE et CONSOMMER réalisent la construction des suites de valeurs de d et d'une suite temporelle r liée à d età req par le prédicat communic de la relation considérée.Quant à la partie "instant" des suites temporelles d et r elle n'est pas explicitement représentée au niveau des types de communication : intuitivement les dates des termes de la suite d (resp. r) sont les dates d'activ ation de l'opération PRODUIRE (resp. CONSOMMER).

L'objet de la sémantique des types de communication est de formaliser cette description intuitive.

Comme lors de la définition constructive des objets du type RELCOMM, au paragraphe II 2.3., la suite de requêtes req n'est pas donnée mais liée aux dates d'activation de l'opération CONSOMMER : une requête ne peut pas être émise si la requête précédente n'a pas été satisfaite.

Nous proposons donc de définir la sémantique d'un type de communication TCOM par une relation $\mathcal{S}$ de profil : SUIT-TEMP (ELT) x SCHR x SUITE (TCOM) x SCHR → BOOL
telle que $\mathcal{S}$ (d, req, tc, Date) = vrai si et seulement si :

- tc est une suite d'objets de type TCOM,

- Date est une chronologie de même domaine que tc,

- tout terme tc_i est le résultat de l'application de l'opération PRODUIRE ou CONSOMMER au terme tc_{i-1}, réalisée à l'instant $Date_i$.

Une partie des termes de la suite tc est définie par l'hypothèse de la donnée de la suite temporelle produite d:
la $j^{ème}$ activation de l'opération PRODUIRE réalise la production de la $j^{ème}$ valeur de d en définissant un nouveau terme dans la suite tc à la date dj.instant.

L'autre partie de la suite tc, les termes définis par les activations de l'opération CONSOMMER, sera construite à partir de la suite req et de la suite d.

• De là, nous pourrons déduire la représentation des relations de communication par les types de communication ; la démarche que nous développerons au paragraphe III 3. et que nous illustrons [figure 6] est shématiquement la suivante :

- notons $\overline{tc} = < \overline{sp}, \overline{a}, \overline{sc} >$ le dernier terme d'une suite tc définie par la relation $\mathcal{S}$ pour un couple <d,req> donné;

- partitionnons la suite Date correspondante entre les suites Dp des dates de réalisation de l'opération PRODUIRE et Dc des dates de réalisation de l'opération CONSOMMER : chaque terme sp_i a été créé à la date Dp_i (égale à di par hypothèse) et chaque terme sc_j a été défini à la date Dc_j ;

- on peut alors construire les suites temporelles $\overline{d}$ et $\overline{r}$ suivantes :

$\overline{d}$ = s-temp ($\overline{sp}$,Dp)
$\overline{r}$ = s-temp ($\overline{sc}$,Dc)

où s-temp est l'opération du type SUIT-TEMP qui construit une suite temporelle

- comme par définition $\overline{d}$ = d est le type TCOM sera une représentation valide de la relation $\mathcal{R}$ considérée si et seulement si les suites d,req et $\overline{r}$ vérifient le précédent $communic_{\mathcal{R}}$.

Par analogie avec le schéma habituellement utilisé pour décrire le passage d'une spécification à une représentation (cf [Pai 78]) et pour résumer la démarche adoptéenous proposons le diagramme de la figure ci-dessous

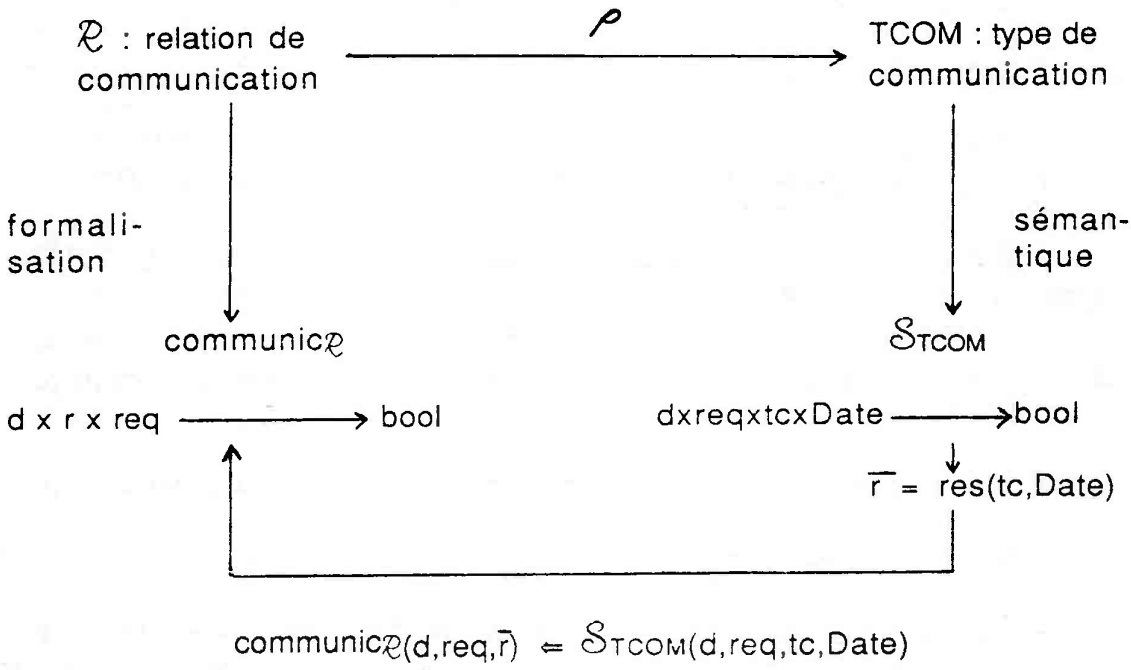


Figure 6: représentation d'une relation de communication $\mathcal{R}$ par le type de communication TCOM

Note : sur le diagramme [Figure 6] on note res(tc,Date) la fonction qui définit la suite $\overline{r}$, décrite informellement ci-dessus.

III.2.2. Définition formelle :

• Nous définissons par récurrence les suites tc et Date, liées par la relation $\mathcal{S}$ à un couple <d,req> donné.

Remarque : dom(tc) = dom(Date) et lg(tc) = lg(d) + lg(req) - 1

Le premier terme de la suite tc est l'objet défini par l'opération INIT du type TYPECOM (cf § III 1.), activée à l'instant $\perp$.

Chaque terme de la suite tc, de rang supérieur à 0, correspond soit à une production soit à l'activation de l'opération CONSOMMER suite à une requête. On peut donc associer à tout indice i du domaine de la suite d, l'indice k du terme de tc qui résulte de la production de la valeur d_i; de même à chaque requête req_j correspond un terme tc_k résultant de la $j^{ème}$ consommation.

Nous définissons donc, comme intermédiaire à la spécification des suites tc et Date et pour chaque pas de la récurrence, deux injections croissantes φ_p et φ_c respectivement du domaine de d et du domaine de req, dans le domaine de tc.

Avant de donner ces définitions nous illustrons par la figure 7 la construction des suites tc et Date et des injections φ_p et φ_c, pour le type de communication "égalité" dont nous rappelons les trois opérations caractéristiques

pré_cons (<sp,a,sc>) = non vide? (a)
val_cons (<sp,a,sc>) = premier (a)
post_cons (<sp,a,sc>) = retirer (a, min(a))

On note v_i la $i^{ème}$ valeur de la suite temporelle produite d. En regard de chaque terme tc_i on indique la valeur de la table a_i. Sur les flèches ⟶ qui symbolisent l'injection φ_c on indique la valeur consommée

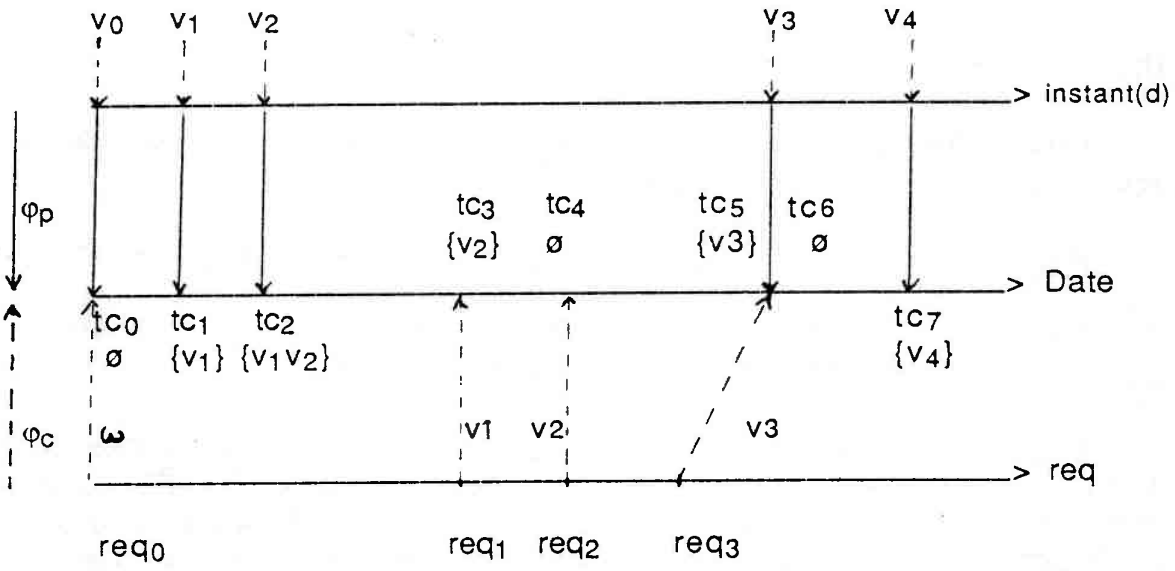


Figure 7 : Construction d'une suite d'états de communication.

Remarque :

Le schéma, [figure 7], met en évidence un cas de consommation (ici la 3ème) où, la pré-condition n'étant pas vérifiée à l'instant de requête, la réalisation de l'opération CONSOMMER a lieu à l'instant ultérieur de production qui rend la pré-condition vraie (ici d_3.instant).

Ainsi, deux termes consécutifs de la suite tc, et par conséquent de la suite Date, sont définis à la même date : le premier (ici tc_5) est le résultat de la production et le second (ici tc_6) celui de la réalisation de l'opération CONSOMMER.

Cette situation peut aussi se retrouver lorsqu'une requête de consommation et une production sont simultanées et que l'opération CONSOMMER est activable.

Ceci nous conduit à formuler une hypothèse d'interprétation de la simultanéité d'évènements, que nous notons H1.

Hypothèse H1
Lorsqu'une production et une consommation sont simultanées, priorité est donnée à la production : la consommation est activée sur l'objet qui résulte de la production, les 2 opérations sont consécutives tout en ayant la même date.

Nous revenons maintenant à la défintion par récurrence des suites tc et Date.

1/ Initialisation de la récurrence.

tc_0 = <(ω), table-vide, (ω)> : objet INIT du type TYPCOM

$Date_0 = \perp$: valeur minimum de l'ensemble CHRONO

$\varphi_p(0) = 0$: le terme d_0 est égal à $<\omega,\perp>$; il correspond donc à la 1ère "production" (fictive) qui a pour résultat tc_0.

$\varphi_c(0) = 0$: idem : la 1ère "consommation" définit le terme $<\omega,\perp>$ des suites temporelles consommées, représentée par la valeur initiale de la suite sc des états de communication.

2/ Etape k de la récurrence.

- On suppose que les suites tc et Date sont définies jusqu'au rang **k-1** et que l'origine de chaque indice j (≤k-1) est défini pour les injections φ_p et φ_c.

- Soit **(i-1)** le nombre de *productions* définies à la date $Date_{k-1}$: Comme nous posons comme hypothèse que les valeurs de la suite d sont produites dans la suite tc aux dates de la suite d, par définition :

$i-1$ = index-max $(d, Date_{k-1})$.

(rappelons que la fonction index-max du type SUIT-TEMP (§II.1) délivre le rang du terme d'une suite temporelle défini à l'instant donné en paramètre).

Donc, la prochaine production, après la date $Date_{k-1}$, sera celle de la $i^{ème}$ valeur de la suite d, avec i = index-max $(d, Date_{k-1})+1$.

- Soit **(j-1)** le nombre de consommations réalisées à la date $Date_{k-1}$:

le rang (j-1) est caractérisé par :

$$j-1 = \max \{ l \in dom\ (req),\ \varphi_c\ (l) \leq k-1\}$$

Deux cas sont possibles à la date Date $_{k-1}$:

(i) la $j^{ème}$ requête a déjà été émise, sans avoir pu être satisfaite : $req_j \leq Date_{k-1}$

(ii) la date req_j est supérieure à $Date_{k-1}$.

- Le prochain terme tc_k de la suite tc sera soit le résultat de l'opération PRODUIRE appliquée au terme tc_{k-1}, soit celui de l'opération CONSOMMER :

. dans le cas (i), la date de la prochaine production est supérieure à la date de la $j^{ème}$ requête : l'opération CONSOMMER sera donc appliquée si la précondition de consommation est vérifiée.

. dans le cas (ii), le terme tc_k résultera de l'opération PRODUIRE si la date du terme d_i est antérieure à la date de la $j^{ème}$ requête et sinon l'opération CONSOMMER sera appliquée au terme tc_{k-1} à condition que la précondition soit vérifiée.

On obtient la définition constructive suivante, qui met fin à la définition de la relation $\mathcal{S}$:

si ($req_j \leq Date_{k-1}$ ou ($req_j > Date_{k-1}$ et d_i.instant > req_j))
et pré_cons (tc_{k-1}) **alors**

-- la $j^{ème}$ requête doit être prise en compte avant la $i^{ème}$
-- production et l'opération CONSOMMER est possible

$Date_k = \max (req_j, Date_{k-1})$
tc_k = CONSOMMER (tc_{k-1})
$\varphi_c\ (j) = k$

sinon -- soit la $i^{ème}$ production précède la $j^{ème}$ requête
-- de consommation, soit l'opération CONSOMMER
-- n'est pas réalisable :

$Date_k = d_i$.instant
tc_k = PRODUIRE (tc_{k-1}, d_i.valeur)
$\varphi_p\ (i) = k$

fsi

III.3. Preuve de représentation des relations de communication par les types de communication

III.3.1. Définition de la représentation

Soient tc, la suite d'états de communication, et Date, la suite d'instants, définies par la relation sémantique $\mathcal{S}$, pour un type de communication TCOM et un couple <d, req> donné.

Nous en déduisons les résultats annexes suivants :

- **Définition 1:**

On appelle respectivement Dp et Dc les suites de dates de réalisations des opérations PRODUIRE et CONSOMMER qui définissent la suite tc :

$$\begin{array}{l} \text{dom (Dp)} = \text{dom (d)} \\ \text{dom (Dc)} = \text{dom (req)} \\ \forall\, i \in \text{dom (d)}, \quad Dp_i = Date_{\varphi p(i)} \\ \forall\, j \in \text{dom (req)}, \quad D_{cj} = Date_{\varphi c(j)} \end{array}$$

Remarque :

On déduit de manière triviale de la définition de la suite Date et de l'injection φ_p que D_p = instant (d)

- **Définition 2 :**

Soit $\overline{tc} = < \overline{sp}, \overline{a}, \overline{sc} >$ l'object du type TCOM défini par le dernier terme de la suite tc.

On appelle *suite temporelle de productions* et *suite temporelle de consommations* les suites $\overline{d}$ et $\overline{r}$ suivantes :

$$\begin{array}{l} \overline{d} = \text{s-temp (sp, Dp)} \\ \overline{r} = \text{s-temp (sc, Dc)} \end{array}$$

(L'opération s-temp du type SUIT-TEMP (§ II.1) définit une suite temporelle par le produit cartésien d'une suite de valeurs et d'une chronologie)

Remarque : On déduit rapidement de la définition de la suite tc et de la définition 1 que $\overline{d} = d$.

Nous pouvons maintenant énoncer la définition de la représentation d'une relation de communication par un type de communication.

- **Définition 3 :**

Un type de communication TCOM représente une relation $\mathcal{R}$ si et seulement si pour toute suite temporelle produite d et toute suite de requêtes de consommation req, la suite r définie ci-dessus est liée aux suites d et req par le prédicat communic$\mathcal{R}$. (cf Figure 6)

III.3.2. Schéma de la preuve

Rappelons l'énoncé du prédicat communic$\mathcal{R}$:

$$\begin{aligned}\text{communic}\mathcal{R}\,(d,req,r) = [&\text{dom}(r) = \text{dom}\,(req) \wedge \\ &(\forall j \in \text{dom}(req),\ req_j \le r_j.\text{instant} < req_{j+1}) \wedge \\ &(\forall j>0 \in R,\ r_j.\text{instant} = \max\,(req_j,\ d_i.\text{instant} \\ &\text{avec } i \in D \text{ tel que } i = \varphi\mathcal{R}\,(j)]\end{aligned}$$

où l'application $\varphi\mathcal{R}$ définie par l'invariant de la relation $\mathcal{R}$ est telle que :

$$\forall j \in R,\ \forall i \in D,\ i = \varphi\mathcal{R}\,(j) \Rightarrow \begin{cases} r_j.\text{valeur} = d_i.\text{valeur} \\ r_j.\text{instant} \ge d_i.\text{instant} \\ \tau\ (d,r,i,j) \end{cases}$$

Pour établir la preuve définie ci-dessus, chacune des trois propositions qui constituent le prédicat communic$\mathcal{R}$ $(d,req,\overline{r})$ doit être vérifiée : les deux premières sont établies directement et indépendamment de la relation de communication à représenter tandis que le schéma de preuve de la troisième proposition estélaboré par induction sur la longueur de la suite $\overline{r}$.

1/ $\text{dom}\,\overline{r} = \text{dom}\,(req)$

D'après la définition 2, $\text{dom}\,(\overline{r}) = \text{dom}\,(Dc)$
D'après la définition 1, $\text{dom}\,(Dc) = \text{dom}\,(req)$
donc $\text{dom}\,(\overline{r}) = \text{dom}\,(req)$.

2/ $\forall j \in dom(req),\ req_j \leq \overline{r}_j.instant < req_{j+1}$:

D'après les définitions 1 et 2, $\overline{r}_j.instant = Dc_j = Date_{\varphi c(j)}$
D'après la définition par récurrence du paragraphe III.2,
$Date_{\varphi c(j)} = \max (req_j, Date_{\varphi c(j)-1})$
donc $\overline{r}_j.instant \geq req_j$.

Comme d'autre part, la suite de requêtes est telle qu'aucune nouvelle requête ne peut être émise tant que la précédente n'est pas satisfaite, par définition :
Date $\varphi_{c(j)} < req_{j+1}$ et par conséquent :
$\overline{r}_j.instant < req_{j+1}$.

3/ Il reste à démontrer la troisième proposition que nous noterons (Q_α):

(Q_α) $\forall j \in$ dom $(\overline{r})$, $\exists\ i \in D$ **tel que** $\begin{cases} \overline{r}_j.valeur = d_i.valeur \\ \overline{r}_j.instant = \max (req_j, d_i.instant) \\ \tau\ (d,\overline{r},i,j) \end{cases}$

Nous allons donc procéder par induction sur la longueur de la suite r, en distinguant les deux cas suivants :

1°) la date du terme $\overline{r}_j$ est égale à req_j : dans ce cas la valeur de $\overline{r}_j$ est celle définie par l'opération val_cons appliquée au terme de tc défini à la date req_j.

2°) la date du terme $\overline{r}_j$ est supérieure à req_j : la valeur de $\overline{r}_j$ est celle définie par l'opération val_cons appliquée au premier terme tc_l, ultérieur à req_j, qui rend la précondition vraie.

Note : le rang 0 de la récurrence est toujours vérifié car par définition $SP_0 = SC_0 = (\omega)$, où ω est la valeur du terme de rang 0 de toute suite temporelle.

Supposant la proposition vérifiée jusqu'au rang (j-1) le shéma de démonstration de la proposition (Q_α), pour le $j^{ème}$ terme de $\overline{r}$, est alors le suivant :

1- étude du cas où la pré-condition est vérifiée à l'instant req_j :

a) définition de l'instant de $\bar{r}_j$ et comparaison avec l'instant donné par $\mathcal{Q}$,
b) définition de l'ensemble consommable A qui sera utilisé par l'opération val_cons ,
c) déduction du résultat de val_cons et comparaison avec la valeur di.valeur, donnée par $\mathcal{Q}$

2- étude du cas où la pré-condition n'est pas vérifiée à l'instant reqj :
même schéma qu'en 1.

L'instantiation de ce schéma de preuve sur des exemples utilise :
- un certain nombre de propriétés sur les suites définies par la relation sémantique $\mathcal{S}$,
- la représentation du type SUITE par le type TABLE.

A cet effet nous consacrons le paragraphe ci-dessous à la définitions de ces propriétés intermédiaires.

III.3.3 Propriétés complémentaires des objets de type TYPECOM

III.3.3.1. Propriétés des suites tc et $\bar{r}$

Note : Dans ce paragraphe, nous confondons la suite d'instants Date avec la suite temporelle isomorphe de même chronologie et de valeurs indéterminées de manière à pouvoir utiliser les opérations du type SUIT-TEMP.

• Propriétés P1 et P2 :(instant de la $j_{ème}$ consommation)

- Soit I le rang du terme de tc défini à la date req_j :
  I = index-min (Date,req_j).

- Si la précondition est vérifiée par l'objet tc_I alors la $j^{ème}$ opération CONSOMMER a lieu à l'instant req_j et $\bar{r}_j$.instant = req_j :

(P1) $\forall j \in dom(req), \forall I \in dom(tc)\ (I = index\text{-}min(Date, req_j) \wedge pré\_cons(tc_I)) \Rightarrow \bar{r}_j.instant = req_j$

Démonstration :

Il s'agit du premier cas de la définition par récurrence du § III.2 : la date de réalisation de l'opération CONSOMMER est alors déterminée par :

$Date_{l+1} = \max (req_j, Date_l)$.

Comme $l = \text{index-min}\ (Date, req_j)$, $Date_l \leq req_j$ alors

$Date_{l+1} = \bar{r}_j.instant = req_j$.

Lorsque la précondition de consommation n'est pas vérifiée par cet objet tc_l, elle ne l'est pas non plus pour chacun des termes de rang compris entre l et $\varphi_c(j)-1$ (d'après la définition par récurrence de tc). C'est ce qu'exprime la propriété P2 :

(P2)

$\forall j \in dom(req),\ \forall l \in dom(tc)\ (l = \text{index-min}\ (Date, req_j) \wedge$ non pré-cons $(tc_l))$

$\Rightarrow (\forall k \in dom(tc),\ l \leq k < \varphi_c(j)-1 \Rightarrow$ non pré-cons $tc_k)$

• Propriété P3 :

Soit i le nombre de productions définies à l'instant req_j, soit l le rang du terme de la suite tc défini à cet instant, alors la suite produite sp_l du terme tc_l contient les i premières valeurs de la suite temporelle d. C'est ce qu'exprime la propriété P3 :

(P3)

$\forall j \in dom(req)\ (i = \text{index-min}\ (d, req_j) \wedge l = \text{index-min}\ (Date, req_j))$

$\Rightarrow sp_l = \text{partie}\ (d,0,i)$

Démonstration évidente à partir de la définition de la suite tc (§III.2) et de l'opération "partie" du type SUIT-TEMP (§II.1) (qui extrait la suite des valeurs d'une suite temporelle comprise entre deux indices).

III.3.3.2. Représentation du type SUITE par le type TABLE

• Une suite s de type V définie sur un intervalle d'entiers est représentée par une table t de type TABLE(ENT,V).

La fonction de représentation ρ est définie par la représentation des opérations du type SUITE donnée ci-dessous :

ρ(s-vide) = table-vide

$\rho(s^*v)$ = ajouter($\rho(s)$, max($\rho(s)$+1, v)
ρ(der(s)) = accès($\rho(s)$, max($\rho(s)$)) = dernier($\rho(s)$)
ρ(reste(s)) = retirer($\rho(s)$, min($\rho(s)$))
ρ(lg(s) = cardinal($\rho(s)$)
ρ(début(s)) = retirer($\rho(s)$, max($\rho(s)$))
ρ(prem(s)) = accès($\rho(s)$, min($\rho(s)$)) = premier($\rho(s)$)
ρ(s-extrait(s,i,j)) = tr($\rho(s)$,i,j)
ρ(s1 conc s2) = $\rho(s1)$ + $\rho(s2)$

Cette représentation est valide puisque les axiomes du type SUITE sont "représentés" par des théorèmes :

Exemple :

?
ρ(lg(s*v) = cardinal($\rho(s)$) + 1
ρ(lg(s*v) = cardinal($\rho(s^*v)$) = cardinal(ajouter($\rho(s)$, max($\rho(s)$), v))
= cardinal($\rho(s)$) + 1 d'après la définition de cardinal

De cette définition de la fonction ρ on déduit les propriétés suivantes :

- **Propriété P4** :

Si la suite des valeurs de la suite temporelle d, comprises entre les indices j et k n'est pas vide, alors le premier élément de la table qui représente cette suite a pour valeur : d_j.valeur.
C'est ce qu'exprime la propriété P4 :

$$\forall\ d{:}\text{SUIT-TEMP(ELT)},$$
$$\text{partie}(d,j,k) \neq \text{s-vide} \Rightarrow \text{premier}(\rho(\text{partie}(d,j,k))) = d_j.\text{valeur}$$

- **Propriété P5** :

Pour tout type de communication TCOM tel que toute valeur consommée soit retirée de l'ensemble consommable (opérations val_cons et post_cons), la suite produite SP est égale à la "concaténation" de l'ensemble consommable A et de la suite consommée SC.
Un tel type de communication est caractérisé par :

$$\left\{\begin{array}{l} T.\text{val\_cons} \equiv \text{accès}\ (A,k) \\ T.\text{post\_cons} \equiv \text{retirer}\ (A,k) \end{array}\right.$$

où T désigne le triplet des opérations caractéristiques de TCOM.

La propriété énoncée intuitivement ci-dessus s'exprime alors par :

(P5) $\forall t : TCOM, \rho(t.SP) = t.a \oplus t.SC$

où l'opération $\oplus$, extension du type TABLE, est définie par :

$\forall t_1, t_2 :TABLE, s:SUITE\ (t_2 = t_1 \oplus s \Leftrightarrow t_2 = t_1 + \rho(s))$.

III.3.4 Réalisation de la preuve sur un exemple de type de communication

Nous nous limitons à l'exemple du type de communication "égalité", l'instantiation de la preuve pour d'autres types étant difficile à suivre.

Rappelons les trois opérations caractéristiques du type :

pré_cons : non vide?(A)
val_cons : premier(A)
post_cons : retirer(A, min(A))

L'instantiation de la proposition Q (§ III.3.2) à vérifier dans le cas de la relation "égalité" est la suivante:

$$\forall j \in dom(r), \exists i \in D \text{ tel que } \begin{cases} \overline{r}_j.valeur = d_i.valeur \\ \overline{r}_j.instant = max(req_j, d_i.instant) \\ i = j \end{cases}$$

Supposons la récurrence vérifiée jusqu'au rang (j-1).
Cela signifie qu' à la date de la $j^{ème}$ requête de consommation :

. les $(j-1)^{èmes}$ premières valeurs de la suite d ont été consommées et constituent la suite SC de l'objet tc_l (avec l = index-min(Date,req_j)),

. j' valeurs ont été produites avec j' ≥ j-1 (propriété P2),

. l'ensemble A de l'objet tc_l représente la suite des valeurs produites et non encore consommées : c'est à dire, d'après P5, la table constituée des valeurs des termes de d dont les indices sont compris entre j et j'.

Ainsi l'objet tc_l défini à la date req_j est caractérisé par:

$$\begin{cases} SP_l = partie(d,0,j') \\ A_l = \rho(\ partie(\ d,\ j,\ j')) \\ SC_l = partie(\ d,\ 0,\ j-1) \end{cases}$$

Montrons que la récurrence est alors vérifiée au rang j:

(1) étude du cas où pré_cons(tc_l) = vrai

. pré_cons (tc_l) = vrai $\Leftrightarrow$ $A_l \neq$ table-vide

. d'après P1, $\overline{r}_j$.instant = req_j

. $A_l \neq$ table-vide et $A_l = \rho$(partie(d, j, j')) $\Rightarrow$

(i) partie(d, j ,j') $\neq$ s-vide $\Rightarrow$ d_j.instant < req_j

(ii) premier (A_l) = d_j.valeur, d'après P4.

. Donc $\overline{r_j.valeur}$ = val-cons(tc_l) = $\underline{d_j.valeur}$

et $\underline{\overline{r}_j.instant = req_j}$, ce qui est défini par la relation dans ce cas puisque d_j.instant < req_j.

(2) Etude du cas où pré_cons(tc_l) = faux

. pré_cons (tc_l) = faux $\Leftrightarrow$ A_l = table-vide

. $A_l = \rho$(partie(d, j, j')) = table-vide $\Leftrightarrow$ partie(d, j, j') = s-vide

$\Rightarrow$ d_j.instant > req_j <u>et</u> j' = j-1

. Soit l' = $\varphi_c(j)$ - 1 : l' est le rang du terme de tc qui le premier vérifie la pré-condition :

$\Rightarrow$ $A_{l'} \neq$ table-vide <u>et</u> $\forall$ k $\in$ dom(tc), l $\leq$ k <l' $\Rightarrow$ A_k = table-vide (propriété P2)

$\Rightarrow$ l' = l+1 <u>et</u> $tc_{l'}$ = PRODUIRE(tc_l, d_j.valeur) <u>et</u> $Date_{l'}$ = d_j.instant, d'après la définition des suites tc et Date

$\Rightarrow$ (i) $A_{l'} = \rho((d_j.valeur)) \Rightarrow$ val_cons(tc_l) = d_j.valeur , d'après P4

(ii) Date $\varphi_{c(j)}$ = $Date_{l+1}$ = $Date_{l'}$, d'après la définition des suites tc et Date et de l'injection φ_c .

. Donc $\underline{\overline{r}_j.valeur = d_j.valeur}$ et $\underline{\overline{r}_j.instant = Dat\varphi_{c(j)} = d_j.instant}$, ce qui est défini par la relation dans ce cas puisque d_j.instant > req_j .

IV. CONCLUSION

Les objectifs de ce chapitre sont d'une part d' apporter un environnement formel à la démarche de conception de programmes définie au sein du projet COMETE et d'autre part de jeter les bases de la sémantique du langage qui sera définie au chapitre suivant. On peut aussi ajouter à ces objectifs celui de familiariser le lecteur avec les idées qui seront le fil conducteur de cette thèse et principalement le concept de "communication datée".
Le cadre formel que nous adoptons est bien sûr celui des types abstraits choisi dès les premiers travaux de J.Julliand et G.R.Perrin et qui constitue une des originalités du projet COMETE parmi les divers formalismes pour l'expression du parallélisme.

La première partie de ce chapitre, consacrée à la définition des types SUITE et TABLE, est en quelque sorte un exercice de style : elle pose simplement les bases de ce cadre formel.

Dans la seconde partie nous formalisons les notions de **suite temporelle** et de **relation de communication** jusqu' alors définies de manière "informelle". Bien sûr les exemples choisis pour illustrer ces notions sont des exemples simples : nous renvoyons le lecteur à la définition de l'unité de disque jointe en Annexe0 au chapitre d'introduction pour l'illustration de l'intérêt de ces notions dans un cas concret non trivial.
Dans la définition du type "SUIT-TEMP" nous nous sommes efforcé d'introduire les opérations adéquates non seulement pour nos propres besoins lors de la définition de la sémantique des types de communication mais surtout pour obtenir un outil qui permette de manipuler explicitement et simplement le temps dans les spécifications. Notons que si l'accès aux termes ou aux sous-suites d'une suite temporelle à partir des dates est direct (fonctions *at-min*, *at-max*, *index-min*, *entre*, etc ...) en revanche l'accès à partir des valeurs n'est pas immédiat (cf la relation *croissante* par exemple) : tel n'était pas l'objectif du type "SUIT-TEMP" et nous n'avons pas jugé bon d'introduire des opérations permettant des accès spécifiques par les valeurs qui relèvent plutôt du type ENSEMBLE ou TABLE que d'une structure de suite.
Dans la formalisation de la notion de **relation de communication** la principale difficulté concerne le rôle de la suite de requêtes de consommation. En effet dans les définitions de cette notion proposées jusqu' alors (cf [JMP 86] par exemple) la suite des requêtes était présentée comme une **donnée** du problème ce qui

permettait une définition fonctionnelle mais incorrecte de la notion de relation de communication.
Si le rôle de cette suite de requêtes est moindre dans le cas systèmes de type "transformationnel" il est primordial dans le cas de systèmes réactifs. La définition de la relation "communic" (§ II.2.3) correspond alors tout à fait à l'approche des systèmes réactifs établie dans [Pnu 85] : il s'agit de maintenir un certain équilibre du système en interaction avec son environnement.
Enfin notons que la formalisation des relations de communication en termes de types abstrait de données était indispensable à l'établissement de la preuve de représentation des relations par les types de communication.

La définition des types de communication et de leur sémantique a fait l'objet de la troisième partie de ce chapitre. Au delà de la formalisation de la notion de type de communication établie depuis longtemps, le résultat de notre travail est la définition de la sémantique des types de communication et l'établissement d'un schéma de preuve de la représentation des relations par les types de communication.
Remarquons que la preuve de représentation à établir n'était pas un "exercice de style" classique aux représentations de types abstraits et que si la littérature sur ce thème, [DeF 79] [Rem 82] entre autres, nous a été précieuse en revanche l'application au problème considéré n'était pas immédiate.
La définition de la sémantique que nous avons présentée procède par induction sur le nombre des opérations réalisées au niveau d'un objet de communication. Ce résultat est l'aboutissement de nombreux essais qui visaient une définition globale de la suite temporelle des états de communication : nous avons été conduit à une définition par récurrence faute d'obtenir une définition directe "simple" et lisible.
Notons que le schéma de preuve est illustré sur l'exemple simple de la relation *égalité* pour faciliter la compréhension : en effet l'instantiation de ce schéma pour des relations moins triviales s'avère vite illisible.

CHAPITRE II

LE LANGAGE D'EXPRESSION DE SYSTEMES PARALLELES

INTRODUCTION

Au niveau du projet COMETE ce langage a été défini, dans [Per 85], pour permettre la validation de la notion de communication comme outil de conception de systèmes parallèles.

Sa mise en oeuvre a donné lieu à la réalisation d'un environnement de programmation, appelé COMEDIE, à laquelle nous avons contribué par la réalisation d'un interprète dont la description fait l'objet du prochain chapitre.

Ainsi ce chapitre est consacré à la présentation du langage, en reprenant les propositions énoncées dans [Per 85], et à la définition formelle de sa sémantique qui constitue l'essentiel de notre apport personnel dans ce chapitre.

L'expression du parallélisme retenue dans [Per 85], est basée sur la coopération de processus non déterministes qui communiquent de manière asynchrone par des ports. L'originalité du langage, par rapport à d'autres expressions contemporaines du parallélisme en termes de processus communicants telles que CSP [Hoa 78], ADA [Ada 80], LCS [Ber 85] ou LC^3 [Lec 86] (pour ne citer qu'eux), réside principalement dans l'introduction du concept de type de communication ; nous avons montré dans [JuM 86] l'apport de ce concept en matière de conception des programmes, c'est à dire pour résumer :

- une définition des relations de communication entre processus indépendante de tout souci de synchronisation,
- une définition modulaire des programmes en terme d'objets [Mey 85] où les objets "processus" réalisent la fonction "calcul" des processus classiques et les objets "communication" définissent les relations de communication des processus.

Dans la première partie de ce chapitre (paragraphe I), nous présentons les caractéristiques du langage : concepts fondamentaux, syntaxe concrète, sémantique intuitive et schémas de composition des processus.

La définition de la sémantique formelle du langage fait l'objet de la seconde partie du chapitre (paragraphe II).

Les deux principaux objectifs poursuivis dans cette définition sont les suivants:

1- viser un modèle du parallélisme asynchrone "véritable" au sens où :

. les processus évoluent de manière indépendante selon leur propre rythme : la durée des actions élémentaires est explicitement

prise en compte, contrairement à la plupart des modèles actuels qui font l'hypothèse d'actions atomiques toutes de même durée,

. les communications sont réalisées de manière asynchrone par l'intermédiaire de ports,

. les seules synchronisations sont dues à la contrainte d'accès en exclusion mutuelle aux ports de communication et aux attentes qui se produisent lorsqu'un processus est prêt à réaliser une opération de consommation alors que la précondition n'est pas satisfaite (cf chapitre I § III),

2- décrire cette sémantique de manière à en déduire aisément la définition de l'interprète : ceci conduit naturellement à une sémantique de type "structurelle opérationnelle", tel que le définit Plotkin dans [Plo 81].

Pour réaliser ces objectifs nous avons étudié avec attention les propositions en matière de sémantique de processus communicants, ce que nous relaterons au dernier paragraphe de ce chapitre, et avons principalement retenu les idées de Boudol exposées dans [Bou 87]. L'idée maîtresse de cet article est de distinguer deux vues de la notion de programme :

- d'une part son **exécution** "symbolique", c'est à dire la séquence des actions réalisées par le programme,
- d'autre part, une **abstraction** de ce programme définie par une **opération** de modification de l'état de mémoire.

Cette distinction est bien sûr sans objet dans le cas de programmes séquentiels puisque la modification de l'état de mémoire est parfaitement définie par l'exécution d'un programme séquentiel, et déterministe pour un état initial donné. Par contre elle permet de définir avec élégance la sémantique de programmes parallèles, ce que nous argumenterons et utiliserons au paragraphe II.

I. PRESENTATION DU LANGAGE

La description d'un système parallèle dans le langage est une expression impérative dans une syntaxe d'un style proche de celui des langages Pascal ou Ada : la partie "séquentielle" du langage est un sous-ensemble de Pascal et nous ne la détaillerons pas ici, hormis l'introduction des constructions non-déterministes.

Commençons par présenter les concepts fondamentaux pour l'expression du **parallélisme** dans le langage.

I.1. Les entités caractéristiques du langage

Le langage est construit autour de quatre entités qui en constituent le noyau parallèle : le **processus**, le **port**, la **donnée communiquée** et le concept de **type de communication.**

Un programme parallèle, ou système, est décrit par un ensemble **hiérarchisé** de **processus communicants** : pour introduire de manière intuitive les concepts fondamentaux du langage nous commençons par présenter [figure 2] un programme "jouet" qui décrit un système "1 producteur - 1 consommateur" réalisant une communication de type "bipoint" [Ray 81].

Le schéma de la figure 1 ci-dessous représente la stucture de ce système :

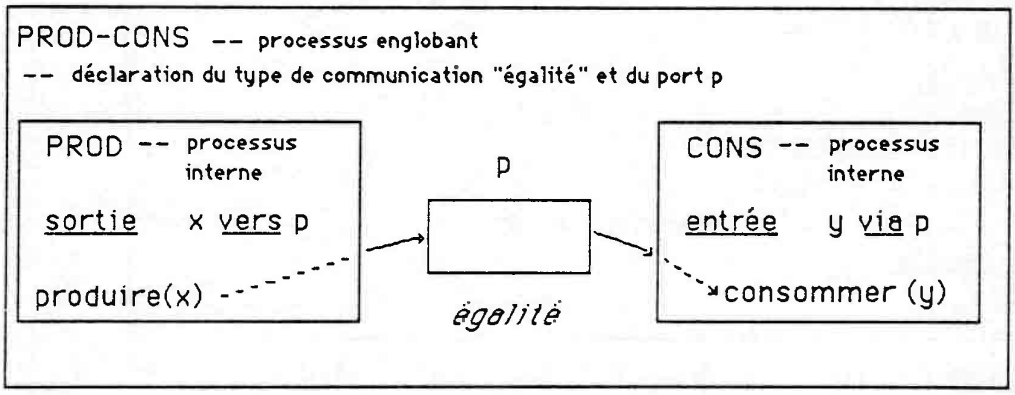


Figure 1 : Schéma d'un système à communication "bipoint"

```
programme PROD-CONS : :

    typecom        égalité (elt) : :    -- définition algébrique du type de
                                        -- communication égalité
    fin typecom ;

    port  p : égalité (entier) ;        --déclaration du port p, instance
                                        --du type égalité

    processus PROD : :                  -- ce processus calcule la suite des 100
      sortie x : entier vers p ;        -- premiers entiers et la communique au
      corps                             -- processus CONS
           x := 1;
           répéter  x < 101 →
                    produire (x) ;
                    x := x+1
           fin répéter
    fin PROD ;

    processus CONS : :                  -- ce processus reçoit, par le port p de
      entrée y : entier via p ;         -- type égalité, les valeurs successives
      fonction pair                     -- des 100 premiers entiers et affiche à
        (z : entier): booléen ;         -- l'écran les entiers pairs
      retourne ((z div 2)*2 = z)
      fin pair ;
    corps
           répéter  consommer (y);
                    z := y div 2 ;
                    si y = z * 2 alors écrire (y) fsi
           fin répéter
    fin CONS ;

    composition
           PROD // CONS             --  activation en parallèle des processus
fin PROD-CONS ;                     --  internes
```

Figure 2 : programme associé à un système "producteur-consommateur" simple

Deux types de processus entrent dans la formation d'un système parallèle :

- les **processus élémentaires** : feuilles de la hièrarchie (ex PROD et CONS [figure 2])
- les **processus composés** qui regroupent à leur tour des processus internes s'exécutant en parallèle et constituant un propre système ; différents schémas de composition de processus pourront être décrits comme nous le verrons au § I.4.

Les processus élémentaires sont des processus séquentiels non déterministes classiques, proches des processus CSP ou des tâches ADA pour les constructions : séquence d'énoncés, choix et itération non déterministes.

Tout processus communicant, élémentaire ou composé, a la structure générale suivante :

processus P : :

(1)
```
entrée : ...          -- définition des données communiquées :
sortie : ...             nom et type de la donnée, port utilisé
                         pour transmettre les valeurs de cette
                         donnée ; ce port aura été déclaré dans
                         un processus englobant
```

(2)
```
déclarations d'objets locaux au processus (variables, fonctions...) ou
des processus internes et des ports associés
```

(3)
```
corps                 -- définition des actions du processus
                      -- s'il est séquentiel
ou
composition           -- mise en composition // des processus
                      --internes, s'il s'agit d'un processus
                      -- composé.
```

fin P ;

La partie (1) constitue l'**interface** du processus ; dans le cas du système parallèle racine de la hiérarchie cette partie est vide.

La partie (2) déclare des éléments **internes** au processus c'est à dire non visibles depuis l'extérieur mais visibles depuis les processus internes.(*)

Pour un processus composé communicant, la partie (2) définit les processus internes et décrit leurs relations de communication par la déclaration :

- des **ports de communication** utilisés :
  - . nom du port
  - . type du port : nom du **type de communication** dont le port est une instance
- des **types de communication** associés et non prédéfinis (**), s'ils n'ont pas déjà été déclarés dans un processus englobant.

L'activation en parallèle des processus internes déclarés en partie (2) d'un processus composé est définie de manière **statique** : la partie (3) est alors réduite à cette mise en **composition parallèle.**

La partie interface des processus internes définit l'utilisation des ports déclarés au niveau du processus englobant ; cette définition est réalisée par les clauses "entrée" ou "sortie" qui déclarent les **données communiquées.** (**ex** : x et y [figure 1])

L'association d'un port et des noms de données communiquées liées par ce port définit un **canal de communication**, par exemple, le canal (p,x,y) dans le schéma figure 1. Un port peut être associé à plusieurs canaux (dans un système "1 producteur - n consommateurs" par exemple). L'ensemble des canaux de communication décrit le réseau de communications du système.

Les données communiquées déclarées dans la partie interface d'un processus sont utilisées, dans le corps du processus, s'il est élémentaire, ou dans celui d'un processus plus interne s'il est composé ; les activations des énoncés **produire** ou **consommer** réalisent les communications.

(*) Toute déclaration obéit à la règle de portée d'Algol 60.

(**) L'environnement de programmation "COMEDIE" à l'aide duquel sont développés les programmes permet la définition d'une bibliothèque de types de communication et de processus.

Ainsi les processus internes utilisent des ports déclarés: (cf [figure 3])

- soit au niveau du processus immédiatement englobant (**ex**: le processus C [figure 3]) et les processus internes (**ex:** C1 et C2) réalisent des communications **internes**.
- soit au niveau d'un processus englobant situé plus haut dans la hiérarchie (**ex**: le processus S figure 3) et les processus internes (**ex:** C1 et C2) réalisent les communications de leurs processus "père" (C [figure 3]) avec l'extérieur.

La signification des différents schémas de communication associés aux diverses organisations de hiérarchies de processus sera précisée au paragraphe I.4.

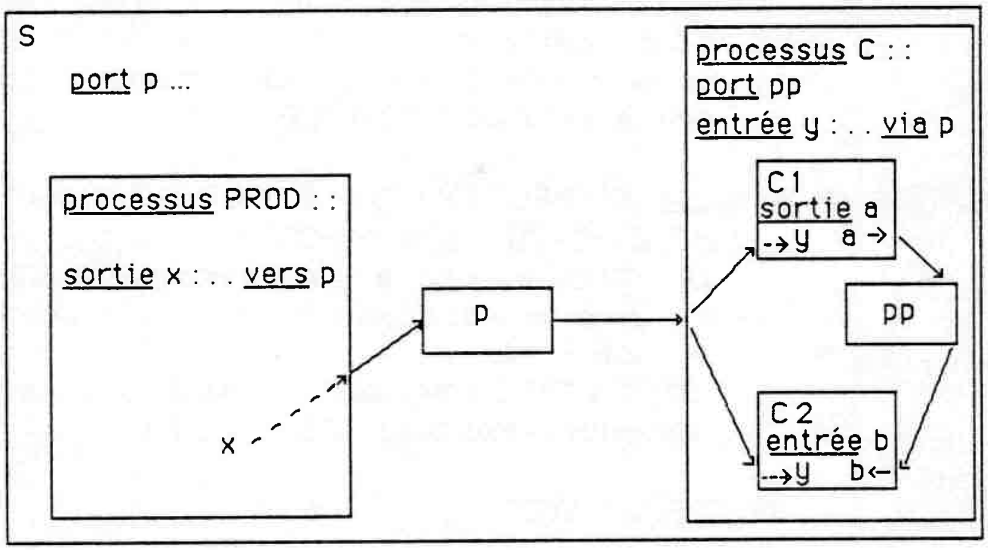


Figure 3 : exemple de communications interne à un sous-système et globale au système décrit.

I.2. Syntaxe concrète des principales constructions du langage

La description de cette syntaxe est donnée par une grammaire à contexte libre ; l'axiome initial est SYSTEME.

Les non-terminaux sont écrits en MAJUSCULES, les mots clés du langage sont soulignés.

La notation X!Y désigne le choix entre les clauses X et Y.

La notation {X}* indique la répétition 0 ou n fois de la clause X.

Tous les autres symboles de ponctuation, i.e. ; , : :: // ( ) , font partie

du langage.

SYSTEME → programme IDENT : : SYST
-- IDENT : nom de programme

SYST → {COMMUNICATION}* PROC-COMMUNIC
{PROC-COMMUNIC}* COMPOSITION

PROC-COMMUNIC → processus IDENT : : PROCESS !
modèle processus IDENT L-PARAM$^{0/1}$: : PROCESS
-- déclaration d'un processus ou d'un modèle
-- (éventuellement paramétré) de processus ;
-- toutes les instances de processus sont
-- activées simultanément de manière statique
-- dans la partie COMPOSITION

COMMUNICATION → {typecom ID-TYPECOM (elt) : : DESCR-TYPECOM}*
port LISTE-PORTS ; {LISTE-PORTS;}*
-- ID-TYPECOM : nom de type de communication
-- elt : paramètre désignant le type des données communiquées.
-- DESCR-TYPECOM : définition algébrique d'un type
-- de communication (cf chap I § III1)

PROCESS → PROC-SEQ ! SYST

PROC-SEQ → INTERFACE DECLARATIONS CORPS

COMPOSITION → composition (ID-PROC)$^{0/1}$ {//{ID-PROC ! VECT-ID-PROC}}* fin IDENT ;
-- création et activation // de processus ou d'un
-- vecteur d'instances d'un modèle de processus
-- (précédemment défini)
-- exemple : P1//P2// (i : 1..3, Ci :: CONS)
-- C// (i : 1..5, Pi :: PROD(pi))
-- pi : *nom de port précédemment déclaré*

L-PARAM → (PARAM {,PARAM}*)
PARAM → ID-PARAM : ID-TYPE
-- ID-PARAM : nom du paramètrre formel

-- ID-TYPE : type du paramètre

ID-PROC → IDENT ! IDENT1 :: IDENT2 L-ARGT$^{0/1}$
-- IDENT : nom d'un processus precédemment défini
-- IDENT1 : nom d'un processus instance du modèle
-- IDENT2 pour les paramètres effectifs contenus
-- dans L-ARGT

-- exemple PROD
C :: CONS (x,y) où x et y sont des noms de port

VECT-ID-PROC → indice : 1 . . nb, IDENTindice : : IDENT2 (L-ARGT)$^{0/1}$
-- vecteur de nb processus de modèle IDENT2
-- exemple (// i:1. .3, Pi : : PP)

L-ARGT → (ID {,ID}*)
-- ID : nom de variable ou de port ou constante

INTERFACE → ENTREE ! SORTIE ! ENTREE SORTIE

CORPS → corps ENONCES fin IDENT ;

ENTREE → entrée LISTE-ENTREES ;

SORTIE → sortie LISTE-SORTIES ;

LISTE-PORTS → LISTE-ID-PORTS : ID-TYPECOM (ID-TYPE) ;
-- ID-PORT : nom de port
-- ID-TYPE : nom du type des données communiquées par le port

LISTE-ID-PORTS → ID-PORT{,{ID-PORT)*!(ID-PORTindice, indice: 1. . nb}}*

LISTE-ENTREES → ID-DONNEE : TYPE via ID-PORT ;
{ID-DONNEE : TYPE via ID-PORT;}*
-- ID-DONNEE : nom de donnée communiquée

LISTE-SORTIES → ID-DONNEE : TYPE vers LISTE-ID-PORTS ;
{ID-DONNEE : TYPE vers LISTE-ID-PORTS ;}*

--une donnée peut être produite vers plusieurs ports
-- les listes d'entrées et de sorties sont disjointes
-- exemple : entrée x,y : entier via p1;
sortie z : entier vers (pi, i : 2..5);

DECLARATIONS → *-- déclarations classiques de types, variables sim-*
-- ples, structures, tableaux, fonctions

ENONCES → ENONCE ; {ENONCE;}*
-- mise en séquence d'énoncés par ;

ENONCE → VIDE ! AFFECTATION ! OP-COMMUNICAT !
CHOIX-INDETER ! ITER-INDETER !
conditionnelle et itérations classiques

VIDE →

AFFECTATION → ID := EXP *-- expressions classiques*

OP-COMMUNICAT → produire (ID-DONNEE) ! consommer (ID-DONNEE)

CHOIX-INDETER → select GARDE → ENONCES
{ou GARDE → ENONCES}* fin select

ITER-INDETER → répéter {GARDE → }$^{0/1}$ ENONCES
{ou {GARDE → }$^{0/1}$ ENONCES}* fin répéter

GARDE → EXP-BOOL
-- une garde est une expression booléenne, qui doit
-- être à "vrai" pour que la garde soit franchissable ;
-- la sémantique du choix et de l'itération indétermi-
-- nistes seront détaillées au § I.4

-- exemple :
select x < 2 → consommer (x) ; écrire (x)
*ou y ≥ 0 → x :=2*x ; produire (x)*
ou y > 10 → produire (y)
fin select

I.3. Sémantique "intuitive" des principales constructions du langage

I.3.1. Sémantique du "noyau parallèle" du langage

. Un système parallèle est décrit par une hiérarchie de processus communicants, élémentaires ou eux-mêmes composés ; l'ensemble des processus séquentiels nommés dans la partie "composition" des différents processus composés, par l'opérateur //, sont créés et activés simultanément. La création d'un processus consiste en la réalisation des déclarations d'objets, notamment celle des ports et des données communiquées, ce que nous décrivons ci-dessous.

. Ainsi les processus sont définis de manière **statique**, comme en CSP et contrairement à ADA : ce choix peut sembler à contre-courant d'autres propositions contemporaines de langage, telles que LCS [Ber 85] ou LC3 [Lec 86] mais la possibilité de créer dynamiquement des processus, bien que séduisante, a été écartée en raison du souci de description statique, conduisant à des solutions "sûres", c'est-à-dire que l'on peut prouver ainsi que nous le verrons au paragraphe II et l'utiliserons aux chapitres III et V.

Notons que l'on peut définir des instances d'un modèle de processus paramétré et les structurer en vecteurs si besoin. Cette construction est indispensable pour exprimer nombre de systèmes parallèles (architectures systoliques, exemples chapitre 3 dans [Per 85]).

. L'originalité du langage réside dans l'expression des communications qui met en oeuvre les trois concepts de **port**, **donnée communiquée** et **type de communication**.

. La déclaration d'un **type de communication** :

"typecom TC (elt)"

consiste en la définition algébrique des trois opérations caractèristiques du type nommé TC (*pré_cons, val_cons* et *post_cons*) ainsi que nous l'avons vu au chapitre I § III.

Les instances du type TC seront les ports de communication déclarés de ce type dans le même processus ou dans un processus interne.

. La déclaration de **port** :

"port p : TC (t) "

où t est un nom de type, a pour effet la création de l'objet de nom p et de modèle TYPECOM (TC), résultat de l'opération INIT définie dans le type abstrait TYPECOM(cf chap I § III.1).

Le port p est local au processus où il est déclaré, ce qui permet d'"encapsuler" les communications dans un sous-système à l'intérieur d'un système composé ainsi que nous l'avons vu au paragraphe I.1.

. La déclaration d'une **donnée communiquée** dans une liste d'**entrées** d'un processus consommateur Q :

"entrée y : t via p "

a pour effet :

- si le processus Q est un processus élémentaire, de créer une variable locale à ce processus de nom y et de type t,

- si le processus Q est un processus composé, de créer une variable locale à chacun de ses processus internes, de nom y et de type t,

- d'"associer" cette ou ces variables locales à l'objet port de nom p : cette "association" est nécessaire au moment de l'activation de l'énoncé "consommer (y)" comme nous le verrons ci-après.

. Une déclaration de **donnée communiquée** dans une liste de **sorties** d'un processus producteur Q :

"sortie x : t vers p1, p2, ..., pn "

a pour signification :

- créer une variable locale au processus Q, de nom x et de type t (ou une variable locale par processus interne si Q est un processus composé)

- "associer" cette, ou ces variable(s) locale(s) aux objets port p1, p2, ... pn.

. Dans le contexte de la déclaration

"sortie x : t vers p1, p2, ..., pn",

la sémantique de l'énoncé **"produire (x)"** est l'application simultanée de l'opération PRODUIRE, du type TYPECOM (cf chapitre I § III1), aux objets

désignés par p1, p2, ..., pn , la valeur produite étant celle de la variable x .

. Dans le contexte de la déclaration
"entrée y : t via p",

la sémantique de l'énoncé **"consommer (y)"** est l'activation de l'opération CONSOMMER définie par le type de communication associé au port p suivie de l'affectation à la variable y de la valeur définie par l'opération "*val_cons*".

Deux cas sont à envisager :
(Notons p l'état de communication désigné par p)

* la valeur *pré_cons* (p) est "vrai" au moment de la requête d'activation de l'opération CONSOMMER : cette opération est activée

* la valeur *pré_cons* (p) est "faux" et l'activation de l'opération CONSOMMER est différée jusqu'à ce que l'état de l'objet p soit tel que *pré_cons* (p) devienne "vrai" : le processus consommateur est "mis en attente" jusqu'au moment où *pré_cons* (p) devient "vrai".
  Notons qu'il s'agit de la seule exigence de synchronisation entre processus définie par le langage.
  Si plusieurs processus consommateurs liés à un même port p sont en attente, ils seront réactivés dans un ordre arbitraire, un seul pouvant à la fois accéder à un port.

Rappelons que lorsque plusieurs processus veulent accéder simultanément à un objet de communication, dans le cas de systèmes hiérarchisés notamment, les accès seront séquentialisés selon un ordre arbitraire sauf s'il s'agit d'une opération produire et d'une opération consommer simultanées, la production étant alors prioritaire.

I.3.2. Sémantique de la partie séquentielle du langage

. Nous n'expliciterons pas les éléments classiques de cette partie séquentielle : mise en séquence, conditionnelle et itération, opération vide, appel de fonction.

Notons simplement la possibilité de définir des itérations infinies en

utilisant la condition "vrai", qui peut être omise dans ce cas : nombreux sont les processus de systèmes parallèles qui utilisent cette construction.

. Les seules constructions nécessitant une définition précise sont les opérations non-déterministes .

- Le choix non-déterministe :

```
select
        g1 → énoncés-1
  ou        .
            .
  ou    gn → énoncés-n
fin select
```

où *gi* est une expression booléenne.

Cette opération a pour signification :
. évaluer simultanément toutes les gardes
. s'il existe une garde *gi* à "vrai", activer les opérations de *énoncés-i*,
. si plusieurs gardes sont à "vrai", choisir l'une d'elles au hasard,
. si aucune garde n'est à "vrai", l'état résultant est un état d'erreur.

- L'itération indéterministe :

```
répéter
        (g1 → )0/1  énoncés-1
  ou        .
            .
  ou    (gn → )0/1  énoncés-n
fin répéter
```

Les gardes *gi* sont des expressions booléennes qui peuvent être omises quand leur valeur est "vrai".

La signification de cette opération est la suivante :
. toutes les gardes sont évaluées simultanément,
. si une garde *gi* est à "vrai", activer les opérations de *énoncés-i*,
. si plusieurs gardes sont à "vrai", l'une d'elles est choisie au hasard,
. l'itération se termine lorsqu'aucune garde n'est à "vrai".

- Analogie et différence avec les opérations indéterministes de CSP ou ADA :

L'analogie est d'ordre syntaxique, les sémantiques attachées à ces constructions sont tout à fait différentes.

Contrairement à notre choix, où les gardes ne peuvent être que des expressions booléennes, le langage CSP permet l'utilisation de la commande de réception de valeur "?" dans une garde : la garde n'est franchissable que si le rendez-vous , avec le processus qui réalise la commande "!" correspondante, peut avoir lieu au moment de l'évaluation de la garde.

L'opération **select** de ADA permet aussi la prise en compte de rendez-vous dans le choix entre les alternatives.

Dans les deux cas, ces constructions lient le choix entre l'activation de différents énoncés, à la réalisation d'une communication entre processus : dans notre langage, c'est au niveau des types de communication que sont définies les différentes alternatives de communication (voir par exemple les types de communication "à-retard-borné", "très-aléatoire" joints en annexe du chapitre I).

Notons le gain en modularité apporté par le concept de type de communication :

- les processus élémentaires sont affectés à la "fonction calcul" et communiquent leurs résultats à leur environnement par le biais des ports de communication,

- l'expression des relations de communication entre processus est "encapsulée" dans les types de communication : le programmeur peut envisager différentes stratégies de communication sans remettre en cause le texte des processus.

I.4. Les schémas de composition de processus

Nous avons vu qu'un système parallèle peut être composé de processus communicants soit élémentaires, soit eux-mêmes systèmes de processus communicants.

Au-delà de la situation simple de communication **bipoint** [Ray 81], illustrée par le système "producteur-consommateur" donné précédemment [figure 2], différents schémas de composition ou situations de hiérarchie peuvent être envisagés. Nous donnerons pour chaque situation le mode d'expression obtenu dans le langage.

I.4.1. Situation de diffusion

On peut généraliser la communication **bipoint** pour un système "1 producteur - n consommateurs" où les valeurs de la donnée communiquée sont diffusées vers n ports, n défini statiquement : la situation de **diffusion** se présente alors comme un ensemble de n communications **bipoints**, utilisant soit toutes le même type de communication soit des types de communication distincts.

Un tel système peut être schématisé ainsi :

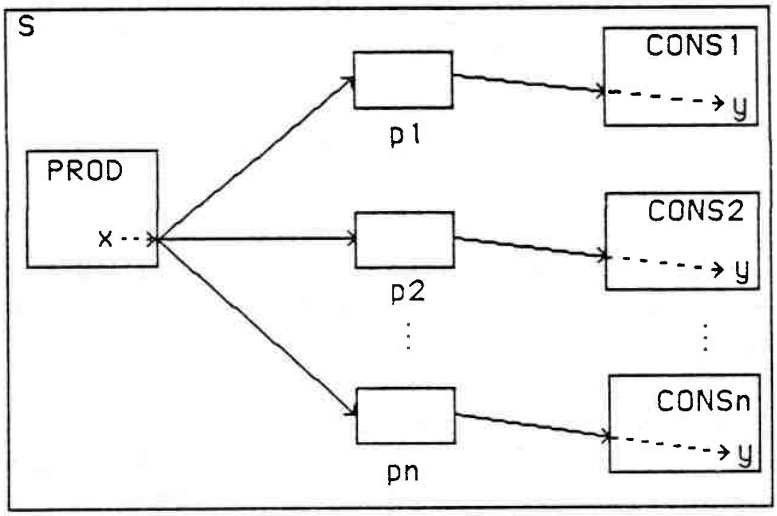


Figure 4 : Modèle de composition de processus de type **diffusion**

Les figures 5 et 6 présentent les deux descriptions de système possibles selon que les ports utilisés sont tous de même type ou non :

```
processus S : :                    -- les  n  ports  sont associés au même
                                   -- type  de  communication :  ils  sont
                                   -- alors déclarés  sous  la  forme d'un
                                   -- vecteur de ports

    port    (pi  : -- nom de type de communication, i:1..n);

    processus PROD : :
      sortie  x : . . .  vers  (pi,i:1..n);
      corps
              .
              .
              produire  (x) ;
              .
              .
    fin PROD ;

    modèle  processus CONS ( pp : port) : :

      entrée  y : . . .  via  pp ;-- dans le processus CONSi,  y sera asso-
      corps                       -- cié au port pi
              .
              .
              consommer  (y) ;
              .
              .
    fin CONS ;

    composition

            PROD (// i:1 . . n, CONSi: :  CONS(pi))
fin S;
```

Figure 5 : Système de processus de type **diffusion** où les n ports utilisés sont tous du même type

```
processus S ::                    -- les ports sont associés à des types de
                                  -- communications distincts
    port    a : ... ;
            b : ... ;
      .
      .

      processus PROD ::
                sortie x : ...  vers a,b, ... ;
                .
      fin PROD ;

      processus P ::
                entrée y : ...  via a ;
                .
                .
      fin P ;

      processus Q ::
                entrée z : ...  via b ;
                .
                .
      fin Q ;
        .
        .
      composition

            PROD // P // Q // ...
fin S ;
```

Figure 6 :Système de type **diffusion** avec des ports de types différents

De nombreux systèmes parallèles correspondent à cette situation de diffusion : voir les exemples tirés de [Per 85] et joints en Annexe (Décomposition en facteurs premiers)

I.4.2. Situation de divergence

Tout processus communicant - possédant une interface qui décrit des données en entrée et/ou en sortie du processus - peut lui-même composer un système parallèle.

Le rôle d'un tel processus se limite :

- à la mise en composition parallèle de ses processus internes, lesquels réalisent les opérations de communication avec l'environnement du processus englobant,

- à l'encapsulation des communications internes lorsque les processus internes communiquent entre eux.

Toute donnée déclarée en entrée du processus est susceptible d'apparaître dans une opération de communication "**consommer**" activée depuis l'un des processus internes (règle de visibilité).

Un système de ce type correspond à une "vectorisation" du processus consommateur et peut être schématisé ainsi :

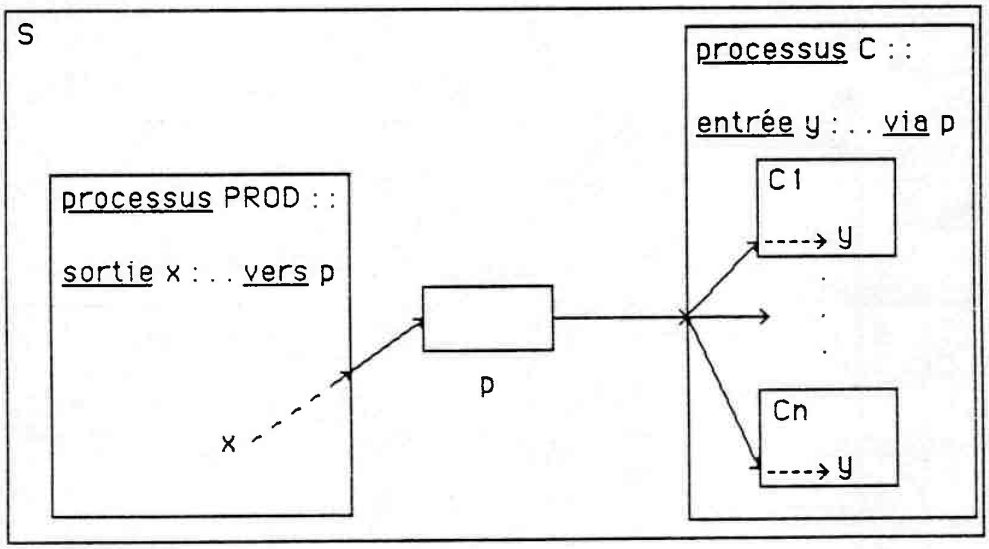


Figure 7 : Modèle de composition de processus de type **divergence**

Le système S réalise une relation de communication entre la suite des valeurs de x (calculées et produites par le processus PROD) et la suite des valeurs de y dont les éléments sont utilisés (consommés) par l'un quelconque des processus qui composent C, au hasard de leurs activations.
Le programme qui décrit ce système dans le langage est donné figure 8.

```
processus S ::

    port    p : ... ;

    processus PROD ::
      sortie  x : ...  vers  p ;
      corps
              .
              produire  (x) ;
              .
    fin PROD ;

    processus C ::
      entrée  y : ...  via  p ;
      modèle processus  CC ::
            .
            .
      corps
              .
              consommer  (y) ;
              .
      fin C ;

      composition
            (// i:1 . . n, Ci: :  CC)
    fin CC;

    composition

            PROD  //  C
fin  S ;
```

Figure 8 : Description d'un système de type **divergence**

Remarque :

. On peut qualifier ce type de divergence de "divergence avec mélange": en effet au niveau du port de communication la suite consommée SC contiendra les valeurs successives résultant des consommations réalisées par l'un quelconque des processus consommateurs.

Ce type de divergence ne permet pas de décrire des relations de communication où la stratégie de consommation dépend de l'"histoire locale" des consommations d'un processus interne.

(**Ex** : le type de communication "croissante" (cf chapitre I § III) qui exprime le fait que les valeurs successives consommées par un processus doivent former une suite croissante)

. Pour exprimer une situation de "divergence sans mélange" il faut étendre la définition de la structure de données d'un type de communication : ainsi le port p ci-dessus désignerait un objet composé :

- d'une suite produite SP
- d'un ensemble consommable A
- d'un n-uplet de suites consommées SCi, i ∈ [1,n] où chaque suite SCi est associée à un processus interne.

Ce second type de divergence pourrait être distingué du premier au point de vue syntaxique en déclarant une **entrée** yi par processus consommateur Ci, à la place de l'entrée y du processus C.

I.4.3. Situation de convergence

Cette situation est la symétrique de la précédente avec un processus **producteur** :

- en communication avec un processus consommateur par un canal de type (p, x, y)
- composé de processus internes réalisant les opérations de production de la donnée x au hasard de leurs activations.

Le schéma correspondant est donné [figure 9]:

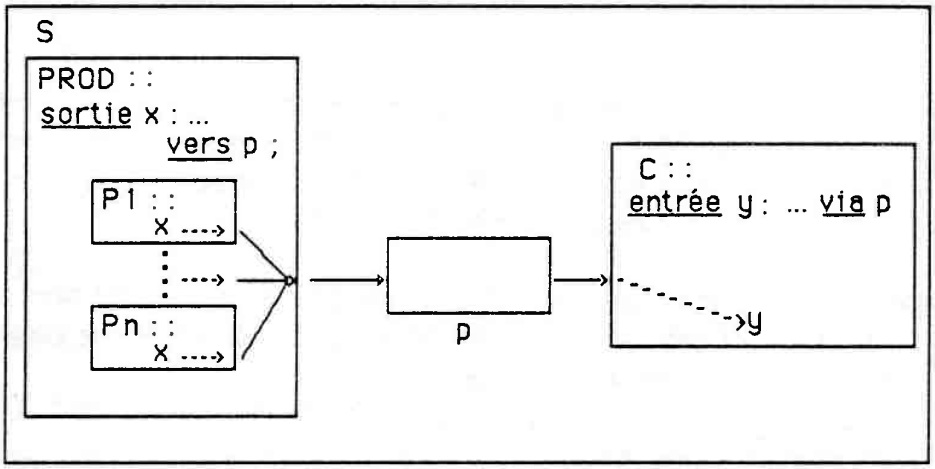


Figure 9 : Modèle de composition de processus de type **convergence**

Ce type de système permet une "vectorisation" du processus producteur (cf. exemple de l'algorithme d'Euclide joint en Annexe).

Dans ce cas aussi, deux situations sont possibles :

. le processus consommateur n'a pas besoin de connaître le processus origine de la valeur consommée : il s'agit alors d'une "convergence avec mélange" utilisant un port de communication classique.

. on veut discriminer les valeurs consommées selon leur origine : on qualifie cette situation de "convergence sans mélange" et la définition de la structure de données associée au port correspondant est :

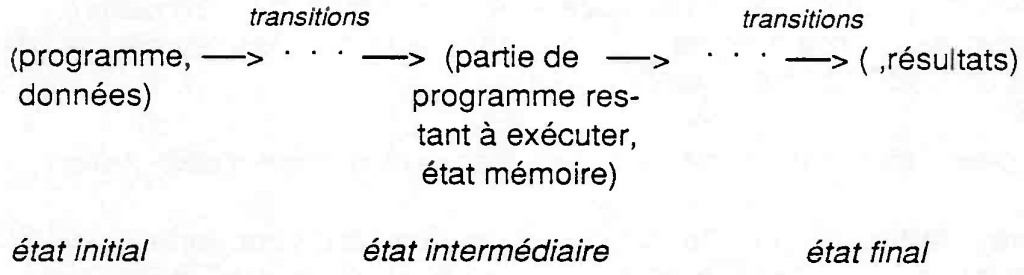
- un n-uplet de suites produites SPi, i $\in$ [1,n]
- un n-uplet d'ensembles consommables Ai
- une suite consommée SC.

Au point de vue syntaxique on peut utiliser une solution analogue à celle proposée pour la "divergence sans mélange" : déclarer une **sortie** x_i par processus producteur Pi.

II. SEMANTIQUE DU LANGAGE

Le langage d'expression de programmes parallèles proposé dans la première partie de ce chapitre, est un langage impératif au sens où il permet de définir des **calculs**, qui peuvent être interprétés comme une succession d'**actions** modifiant une **mémoire** : une telle interprétation conduit naturellement à la définition d'une sémantique qualifiée d'**opérationnelle** [Bak 69] [Plo 81].

Pour des langages séquentiels ce type de sémantique consiste en la définition d'une **machine abstraite**, ou automate, dont intuitivement le fonctionnement peut être schématisé ainsi [Liv 78] :



Plus précisément, cette machine abstraite est décrite par :

- un ensemble d'états : couples de la forme (texte de programme, état de mémoire)

- une fonction de transition sur les états définie par un ensemble de relations de transition de la forme :

(programme, mémoire) —action—> (programme', mémoire')

où les transitions sont étiquetées par les actions élémentaires de la machine qui correspondent aux constructions primitives du langage.

Dans le cas de programmes parallèles, il s'agit de définir le compor- tement d'une **machine parallèle** où plusieurs processus évoluent **simultanément,** agissent **concurremment** sur la mémoire, si le langage prévoit l'utilisation d'une mémoire partagée, et **communiquent**, de manière synchrone ou non.

On est alors naturellement conduit à définir cette machine parallèle comme une composition ou "produit" des automates qui

décrivent la sémantique des processus élémentaires. C'est cette approche qui a été adoptée dans [Per 85] pour définir la sémantique du langage.

Ainsi, selon que le modèle du parallélisme sous-jacent est de type synchrone, asynchrone ou "mixte", un tel automate produit réalise à chaque transition, soit une action composée à partir d'actions de chacun des processus, soit une des actions de l'un seulement des processus, soit une composition des actions de certains des processus.

De nombreux travaux reposent sur une telle notion d'automate, pour ne citer qu'eux par exemple [An 82] [Kel 76] [Sif 83], dont l'intérêt est de permettre non seulement de modéliser le comportement de systèmes de processus mais aussi de développer des systèmes de vérification de propriétés des programmes (citons par exemple les systèmes CESAR [Que 82], EMC [CES 86], MEC [AD 86]).

On peut cependant relever les inconvénients d'une telle approche :

- une définition lourde puisque toutes les combinaisons possibles d'actions simultanées ainsi que leurs effets sur la mémoire doivent être décrits,
- l'impossibilité de considérer comme atomique l'exécution d'un programme c'est à dire s'abstraire de la séquence d'actions réalisées pour définir séparément l'état mémoire atteint.
- l'impossibilité de définir des comportements véritablement asynchrones puisque toute action réalisée par une transition d'un automate produit a pour durée la plus longue des durées de ses actions composantes.

Dans [Bou 87], Boudol propose un formalisme pour remédier aux deux premiers de ces inconvénients : nous nous en inspirons largement et l'étendrons de manière à rendre compte explicitement du temps.

Ainsi dans cet article, le système de transition est décrit à l'aide de transitions étiquetées de deux types :

- celles qui définissent "l'exécution symbolique" des programmes, qualifées "**d'exec**",
- celles qui décrivent les modifications de l'état mémoire par l'activation d'actions ou de programmes, qualifées "**d'oper**".

L'explosion combinatoire inhérente à la définition des produits d'automates est évitée par l'introduction de règles d'inférence structurelle qui permettent de déduire l'activité de toute construction composée à partir de celle de ses composantes, ainsi que

le préconise Plotkin dans [Plo 81].

De plus la notion d'atome est introduite pour définir l'effet conjugué d'actions simultanées et concurrentes sur la mémoire. Par exemple, le rendez-vous CCS est défini par la composition d'opérations complémentaires α et α^{-1} : l'exécution de l'une des opérations ne peut avoir lieu sans le déclenchement simultané de l'autre et la synchronisation, définie par $\alpha \, . \, \alpha^{-1} = 1$, résulte de l'**entrelacement** (interleaving) des actions **élémentaires** qui composent les opérations α et α^{-1}. Ainsi il n'existe pas d'état de l'automate atteignable par l'exécution de l'opération α (ou α^{-1}), et seul un entrelacement des séquences d'actions de α et α^{-1} permet de définir l'état mémoire qui résulte d'un rendez-vous CCS.

Notons que nous ne retiendrons pas la notion d'atome : en effet nous ne cherchons pas à décrire de comportement explictement synchronisé tel que le rendez-vous.

Ainsi notre sémantique sera donnée par la définition d'une machine abstraite que nous décrirons en organisant ce paragraphe de la manière suivante :

- nous commençons au paragraphe II.1 par la définition de la syntaxe abstraite indispensable à l'approche structurelle que nous adoptons,
- les états et les actions atomiques activées par les transitions de la machine sont définis au paragraphe II.2,
- la relation de transition sur les états est définie par un système de déduction dont nous donnons les principes au paragraphe II.3 ; l'ensemble des axiomes et des règles d'inférence est présenté en trois étapes :
- le paragraphe II.4 définit les règles sémantiques des schémas élémentaires de processus séquentiels,
- le paragraphe II.5 est consacré à la définition de la sémantique des processus séquentiels, et
- le paragraphe II.6 définit les transitions qui décrivent le comportement global d'un programme.
- enfin nous concluons ce paragraphe II par l'étude d'un exemple (§II.7).

Afin de nous focaliser sur la sémantique du "noyau" parallèle du langage, dans cette présentation nous ne donnerons pas la partie "classique" de la sémantique associée aux déclarations, définitions de type, définitions et appels de fonctions et opérations

d'entrées/sorties.

II.1.Syntaxe abstraite des principales constructions du langage

Par souci de simplification nous nous limiterons ici à la syntaxe abstraite des instructions.

Nous appelons "abs" la fonction qui associe à une construction du langage concret, pour un environnement donné, une expression de la syntaxe abstraite.

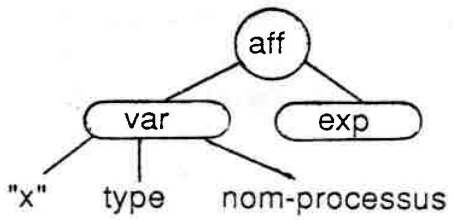
Commençons par les énoncés élémentaires du langage, que l'on interprètera par des actions de la machine abstraite au paragraphe suivant.

1 - l'instruction d'affectation :

abs(ε, < x := e >) = aff(abs(ε, < x >) , abs(ε, < e >))

où abs(ε, < x >) est l'arbre qui décrit la variable x, pour l'environnement ε, porteur notamment des attributs suivants : type de la variable, nom du processus où elle est déclarée, abs(ε, < e >) est l'arbre qui représente l'expression e.

On peut représenter schématiquement l'arbre de l'affectation ainsi :



où "var" est le type des nœuds représentant les variables et "exp" celui associé aux expressions

2 - L'instruction de production :

abs(ε, < produire (x) >) = prod (abs(ε, < x >), tc_1, ... , tc_k)

où les attributs tc_1, ... , tc_n sont les noms des ports de communication associés à la donnée communiquée x lors de sa déclaration.

3 - L'instruction de consommation :

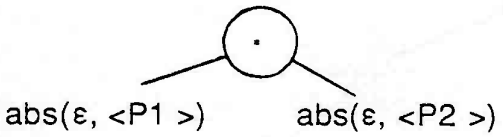
abs(ε, < consommer (y) >) = cons (abs(ε, < y >), tc)
où l'attribut tc est le nom du port de communication associé à la donnée communiquée y dans sa déclaration.

4 - Les compositions d'intructions :

Le corps des processus élémentaires est construit à l'aide des structures de contrôle suivantes :

4.1 - La séquence d'énoncés :

Si P1 et P2 sont deux textes de programmes ,
abs(ε,<P1; P2>) = abs(ε,<P1>). abs(ε,< P2 >)
où l'opérateur . réalise l'enracinnement des deux arbres qui représentent respectivement P1 et P2, ce que l'on peut représenter schématiquement:



Remarque : L'opérateur . est associatif (mais non commutatif !) ce qui permet de donner la même interprétation aux programmes abstraits (p.q.r), (p.q).r et p.(q.r).

4.2 - Les structures conditionnelles
Elles sont toutes définies à partir de l'opérateur "Cond", qui représente un programme soumis à une condition, et de l'opérateur "+" qui réalise un choix non déterministe entre plusieurs programmes conditionnés.

. Nous commençons par le **choix non déterministe** qui est la plus générale des conditionnelles :

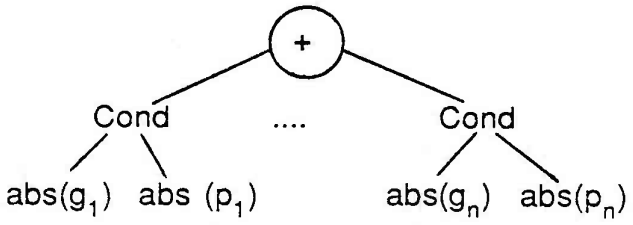
l'instruction select $g_1 \rightarrow p_1$
.
ou
.
. $g_n \rightarrow p_n$
fin select

est interprétée comme le choix arbitraire et exclusif entre les différents programmes p_i dont les gardes sont évaluées à "vrai" (rappelons que les gardes ne peuvent être que des expressions booléennes). Lorsqu'aucune garde n'est à "vrai" l'état de programme obtenu est l'état d'"erreur".

L'arbre correspondant est construit par l'application de l'opérateur "+" aux arbres qui représentent les n choix :

$$\begin{aligned} abs(\varepsilon, <\ &\underline{\text{select}}\ g_1 \rightarrow p_1 \\ &. \\ &\underline{\text{ou}} \\ &. \\ &\quad g_n \rightarrow p_n \\ &\underline{\text{fin select}}\ >) = + \underset{i=1,n}{\text{Cond}}\ (abs(\varepsilon,<g_i>), abs(\varepsilon,<p_i>)) \end{aligned}$$

Ceci est représenté schématiquement par :



. L'**alternative** < si C alors P sinon Q > est décrite par la "somme" des deux conditionnelles élémentaires exclusives : <si C alors P> et <si non C alors Q>. Ainsi :

abs(ε,<si C alors P sinon Q>) = Cond(abs(ε,<C>), abs(ε,<P>) +
Cond (abs(ε,<non C>), abs(ε,<Q>))

Remarque :
La conditionnelle élémentaire < si C alors P> est un cas particulier de l'alternative où dans le cas du sinon l'action nulle est réalisée.

4.3- Les structures itératives

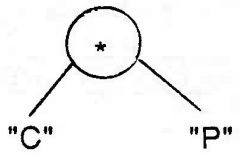
. L'itération "classique" <répéter C → P> est interprétée comme le programme << si C alors P > ;< répéter C → P >>

Telle est la signification de l'opérateur "*" utilisé pour

représenter la syntaxe abstraite de l'itération :

abs(ε, < répéter C → P >) = *((abs(ε,<C>), abs(ε,<P>))

Ce qu'on peut représenter graphiquement par :



. L'itération non déterministe :

< select $g_1 \rightarrow p_1$
ou
.
. $g_n \rightarrow p_n$
fin select >

est équivalente au programme

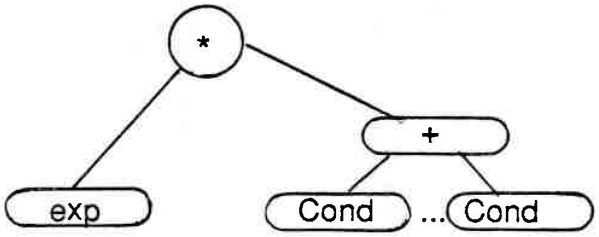
< si g_1 ou g_2 ou ... g_n alors < select $g_1 \rightarrow p_1$
ou
.
$g_n \rightarrow p_n$
fin select > ;
< répéter $g_1 \rightarrow p_1$
ou
.
. $g_n \rightarrow p_n$
fin répéter >>

L'itération non déterministe est donc un cas particulier de l'itération classique ce qui conduit à la syntaxe abstraite correspondante :

abs(ε, < répéter $g_1 \rightarrow p_1$
.
ou
.
.
$g_n \rightarrow p_n$
fin répéter >)

= * (abs(ε,<g_1 ou ... ou g_n>),
abs(ε,< select $g_1 \rightarrow p_1$
ou
.
.
$g_n \rightarrow p_n$
fin select >))

Cet arbre de syntaxe abstraite a la structure suivante :



g_1 ou g_1 ... ou g_1

5 - La composition parallèle de processus

Le corps d'un processus composé est réduit à l'énoncé <$Proc_1$// ... // $Proc_n$> qui est la directive de mise en composition parallèle des processus séquentiels $Proc_1$, ... , $Proc_n$; ces processus sont soit définis par leur déclaration soit déclarés par cette construction comme des instances d'un modèle de processus précédement défini, éventuellement paramètré par le nom d'un ou plusieurs ports de communication. L'interprétation de cet énoncé donnera lieu à la création simultanée des processus invoqués et à l'activation simultanée de leurs actions.

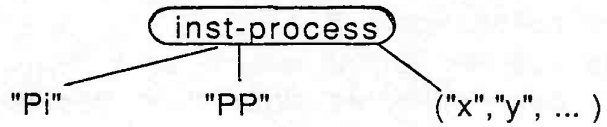
Ainsi l'énoncé $Proc_1$// ... // $Proc_n$ sera décrit par l'arbre
abs(ε, <$Proc_1$// ... // $Proc_n$>) = *comp-par* (abs(ε, <$Proc_1$>),...,
abs (ε, <$Proc_n$>))

où si <$Proc_i$> est le nom d'un processus précédemment déclaré l'arbre abs($Proc_i$) est un nœud de type "identificateur" porteur de la valeur "$Proc_i$" de manière à retrouver l'arbre de définition du processus "$Proc_i$ " :

ident

"$Proc_i$"

et si <$Proc_i$> est la déclaration d'un instance du processus PP de nom "Pi" pour les paramètres x,y, ... l'arbre correspondant est un nœud de type "instance-processus" muni de l'attribut "PP" et de la liste des paramètres effectifs "x", "y", ..; :



. Notons que dans les paragraphes qui suivent, dans un souci de simplification d'écriture, nous noterons par le symbole // les termes de la syntaxe abstraite définis par l'opérateur *comp-par.*

II.2. Les états et les actions de la machine abstraite

II.2.1 Définition des états

. Intuitivement, à un instant d'observation donné, l'état de la machine est défini par l'état courant de la mémoire ; or nous voulons rendre compte d'un parallélisme "véritable" et par conséquent à tout instant au cours de l'exécution d'un programme parallèle certains processus sont observables et d'autres pas : ainsi on est conduit à définir des états mémoire dits "parfaits" et d'autres dits "imparfaits" comme il est proposé dans [FiO 83].

L'introduction de ces deux types d'état n'est toutefois pas un obstacle à la définition de cette machine abstraite moyennant les principes, couramment admis, auxquels nous faisons référence ci-dessous.

. Tout d'abord la notion d'action **atomique** que nous utiliserons pour décrire les transitions est celle d'action **ininterruptible** et **indivisible** dans le temps, de durée nulle ou pas.

. Ensuite nous allons être amené à supposer l'existence d'un référentiel de temps global et à définir explicitement une horloge par processus interne : le temps est de manière dicrète et les horloges seront des entiers.

De nombreux modèles du parallèlisme reposent sur la définition d'une relation d'ordre sur l'ensemble des actions atomiques possibles, pour un programme et un état de mémoire initial donnés [Rei 85] [NPW 81] [CFM 83].

Ainsi dans [BC 88] le modèle retenu met en évidence trois relations qui partitionnent l'ensemble des événements (occurences d'actions) :

- une relation d'ordre partiel, que l'on note ≤ :

  a ≤ b signifie l'événement a est cause de l'événement b

- une relation en conflit, notée # :

a#b signifie que les actions a et b sont exclusives : cette relation lie les actions des différentes alternatives des choix.

- la relation complémentaire des deux précédentes, notée $\cup$:

$a \cup b$ définit l'indépendance des actions.

Il est montré dans [BC 88] que dans tout modèle d'**exécution** la relation de conflit est absente : intuitivement les choix réalisés au cours d'une exécution résolvent les conflits.

En supposant l'existence d'un référentiel de temps global, les relations $\leq$ et $\cup$ s'expriment alors sous la forme :

$a \leq b \Leftrightarrow$ date (fin(a)) $\leq$ date (début(b))

$a \cup b \Leftrightarrow$ date (début(a)) = date (début(b))

. Ainsi l'introduction explicite du temps permet-elle l'expression d'une relation d'ordre partiel sur les actions que nous utiliserons pour la définition des transitions. Par exemple les actions a et b de deux processus distincts n'utilisant pas un port de communication commun seront activées simultanément lors d'une transition si et seulement si leurs dates de début sont identiques, c'est à dire si les valeurs courantes des horloges des processus respectifs sont égales.

Par conséquent, un état contient l'ensemble des valeurs des horloges des processus et on dira qu'un état est parfait si l'ensemble des horloges ont la même valeur, et imparfait sinon : nous verrons au paragraphe II.5 comment l'utilisation de ces horloges permet de définir le comportement d'un programme parallèle.

. Définissons maintenant l'autre composante des états, plus classique : l'état mémoire.

La mémoire est un ensemble fini de liaisons variable-valeur, représenté par une table, défini par les déclarations du programme et modifié, classiquement , par l'exécution des affectations. Dans notre langage deux autres catégories d'opérations modifient l'état de la mémoire: les instructions de **communication** qui s'exécutent au travers des **ports** de communication.

Les ports qui désignent des instances de types de communication, constituent la mémoire partagée d'un programme.

L'utilisation de cette mémoire est régie par le principe d'**exclusion mutuelle** : les communications sont réalisées par

l'entrelacement des accès aux ports de communication. Ainsi nous ne définirons pas, par exemple, d'exécution simultanée pour la composition (prod(x,tc) // cons(y,tc)) puisque les données communiquées x et y sont associées au même port de communcation tc : ces deux opérations seront séquentialisées en donnant priorité à la production, ainsi que nous l'avons défini au chapitre I § III. Toute autre combinaison d'accès concurrents à un port de communication est réalisée par un entrelacement arbitraire. Rappelons que dans tous les cas, ces actions auront lieu à la même date puisque nous avons fait l'hypothèse, dès le chapitre I, que les communications sont instantanées.

. Ainsi la mémoire est constituée :

1- de la table des valeurs (*) des variables locales aux différents processus du programme.

On notera val(x, m) la valeur pour l'état mémoire m (ou simplement val(x) si le contexte est implicite), de la variable désignée par l'identificateur x , où x est le chemin complet du nom de la variable, de la forme : id (.id)* où id est le nom d'un processus et au niveau le plus interne le nom d'une variable dans un processus. Le chemin est élaboré grâce aux attributs associés aux arbres de syntaxe abstraite et propagé de l'arbre du processus le plus englobant vers celui du processus le plus interne.

Ex. : si x est le nom d'une variable locale au processus P1 interne au processus Q, lui-même composant du programme P, le nom complet de cette variable est P.Q.P1.x.

2 - de l'ensemble des valeurs des objets partagés : les **ports** de communication, désignés eux aussi par leurs noms complets : en effet dans le cas d'imbrications de processus, les ports ne sont pas visibles depuis n'importe quel processus (cf. § I.1.).

On désignera généralement par tc les ports de communication (instances de types de communication déclarés ou prédéfinis) et par val(tc,m) leurs valeurs courantes à l'état mémoire m (ou simplement val(tc) s'il n'y a pas d'ambiguïté possible).

. On suppose l'existence d'un mécanisme d'évalutaion des expressions que nous ne détaillerons pas ici.

(*) Nous omettons les piles nécessaires à la mise en oeuvre des fonctions récursives, prévues dans le langage.

On note cette évaluation des expressions $m \xrightarrow[eval]{e} v$ ce qu'il faut lire :

"dans le contexte m, état de la mémoire, le résultat de l'évaluation de l'expression e est la valeur v".

Ce mécanisme est purement applicatif : il ne correspond pas à une transition de la machine et l'évaluation d'une expression ne modifie pas l'état de la mémoire et est instantanée.

. Nous noterons généralement Σ un état de la machine défini par le produit cartésien :

$< m \times h_1 \times h_2 \times \ldots \times h_n >$ où m désigne l'état de mémoire

et h_i l'horloge du $i^{ième}$ procesus.

II.2.2. Les actions de la machine abstraite

Elles sont au nombre de sept. Les cinq premières correspondent aux schémas élémentaires de la syntaxe abstraite engendrés par les instructions élémentaires du langage : ce sont les actions *aff*, *prod*, *cons*, *nil* et *Cond(C,a)* , où *a* est une action atomique.
La sixième action déclenchable sur cette machine parallèle est l'activation parallèle d'actions atomiques indépendantes notée $Par(a_1,\ldots ,a_k)$ qui réalise l'exécution simultanée des actions a_1, ..., a_k lorsqu'elle est possible, c'est à dire :

- lorsque les horloges des processus associes aux actions a_i ($1 \leq i \leq n$, où n est le nombre de processus du sytème) ont même valeur.
- et que les actions a_i ne sont pas exclusives, c'est à dire n'opèrent pas sur le même port de communication.

(Nous verrons au § II.6 comment s'exprime formellement cette règle).

. La septième action atomique est l'action que nous appellerons *prod-cons* : elle réalise de manière atomique la séquence d'une production et d'une consommation simultanées sur le même port. En effet au chapitre I §III, nous avons posé l'hypothèse que des opérations d'accès simultanées à un objet de communication sont réalisées selon un ordre arbitraire sauf s'il s'agit d'une production et d'une consommation qui doivent être réalisées dans cet ordre.

. Toute action élémentaire est caractérisée par une durée. Les

seules hypothèses sur les durées des actions sont :

- les opérations de communication sont instantanées,
- l'opération *nil* est instantanée,
- les autres opérations ont une durée multiple de l'unité.

. Nous ne justifions pas davantage la nécessité évidente du caractère atomique de ces actions ; notons simplement que l'atomicité de l'action *Cond(C,a)* est garantie par le fait que l'évaluation des expressions ne déclenche pas de transition et qu'elle est instantanée .

L'introduction de l'action *Cond(C,a)* permet de décrire simplement les transitions des schémas de choix et des itérations, ainsi que nous le verrons aux paragraphes II.4 et II.5.

II.3. Formalisme et principes de la sémantique

La méthode pour spécifier les transitions est celle proposée dans [Plo 81] : la signification des constructions du langage est décrite par un ensemble de règles de déduction ; à chaque schéma de la syntaxe abstraite est associé un certain nombre de **règles structurelles** qui définissent ses transitions possibles à partir des transitions des sous-schémas qui le composent. Ainsi un programme est valide seulement si son comportement peut être **prouvé** par l'application d'un ensemble de règles d'inférence.

. Les règles d'inférence sont notées sous la forme :

$$\frac{A1, A2, \ldots, An}{B1, B2, \ldots, Bk}$$

où les Ai et Bj sont des relations de transition, ou des prédicats portant sur les actions à déclencher (cf. §II.3.3), et la signification d'une telle règle est : "de A1 et . . . et An on peut déduire B1 ou . . . ou Bk" ; la virgule a donc le sens de "ET" au-dessus de la barre d'inférence et de "OU" en dessous ; une règle introduit du non-déterminisme si plusieurs conclusions sont possibles.

Nous proposons trois types de transition :

- Les deux premiers types de transitions sont similaires de ceux définis dans [Bou 87] :

. les transitions que nous qualifions d'"*act*", pour "activation", et analogues aux transitions "exec" dans [Bou 87]; elles ont la forme suivante :

$$\varepsilon \text{ ; programme} \xrightarrow[act]{\text{action}} \text{programme, (*)}$$

ce qu'il faut lire : " dans le contexte ε un programme active une action élémentaire et se reconfigure alors en un nouveau programme"; elles spécifient les actions possibles d'un processus séquentiel ou d'un programme parallèle

. les transitions de la forme :

$$\varepsilon \text{ ; mémoire} \xrightarrow[modif]{\text{programme ou action}} \text{mémoire ;}$$

spécifient les modifications apportées à la mémoire par l'activation d'une action et dans le cas où la transition est étiquetée par un programme elles définissent l'**abstraction** de ce programme : on définit ainsi la fonctionnalité du programme sans faire référence à la séquence des actions réalisées.

- Le troisième type de transitions définit le **comportement global** d'un programme sous la forme , classique : (**)

$$\varepsilon \text{ ; (programme, état)} \xrightarrow{\text{programme ou action}} \text{(programme, état) ;}$$

Ces transitions seront toujours déduites à partir de transitions de type "modif" et "act" et de conditions de réalisation portant notamment sur les valeurs des horloges des processus présentes dans l'état de l'automate : nous verrons aux paragraphes suivants qu'un nombre relativement restreint de règles d'inférences et d'axiomes, lesquels portent sur les schémas élémentaires de programme, suffisent à définir cette sémantique.

(*) Nous ne définirons pas formellement l'environnement, noté ε constitué de l'ensemble des déclarations de variables, types ...;
(**) Nous ne qualifierons pas ce type de transitions : s'il fallait le faire ce serait par exemple par "exec" pour exécution.

. Il faut pouvoir exprimer la terminaison d'un programme et son arrêt sur un état d'erreur : nous noterons ces états de programme respectivement par ϕ (un programme terminé correspond à un texte vide) et "erreur" (ces symboles ne peuvent apparaître qu'en partie droite des transitions d'exécution)

. Enfin l'expression des règles est simplifiée par l'hypothèse des égalités suivantes :

$p//\phi = \phi//p = p$: cas de la terminaison d'un des deux processus activés en parallèle.

p//"erreur" = "erreur"//p ="erreur"
: terminaison anormale du programme composé.

$p//q = q//p$

$p//q//r = (q//p)//r = p//(q// r)$
: l'interprétation d'un système à plusieurs niveaux de hiérarchie est défini par "la mise à plat" de l'ensemble des processus élémentaires composants, la hiérarchisation servant uniquement à l'encapsulation des communications (cf §I.4).

. Chaque axiome ou règle du système sera désigné selon la terminologie suivante :

- les noms d'axiomes ou de règles se rapportant à une action, sont de la forme : xxx-act ou xxx-mod

où xxx est un mnémonique de l'action spécifiée et où le suffixe indique s'il s'agit d'une transition de type "act" ou de type "modif".

- les règles d'inférence générales sont désignées de manière similaire avec pour suffixe "act" si la conclusion de la règle est une transition de type "act", "mod" si c'est une transition de type "modif" et "exec" si c'est une transition qui porte sur le comportement global du programme.

II.4. Règles sémantiques des schémas élémentaires de programme

Nous commençons par définir les transitions d'exécution "symbolique" des 5 schémas élémentaires *aff, cons, prod, nil et Cond* qui permettent de construire les processus séquentiels, puis nous en décrirons la fonctionnalité.

II.4.1. Exécution "symbolique" des schémas élémentaires

Ce sont des axiomes qui spécifient l'exécution de ces schémas puisqu'ils correspondent à l'activation d'une action atomique : le résultat de ces activations est le programme vide.

(1) L'affectation

AFF-act $\quad \varepsilon \vdash aff(x,e) \xrightarrow[act]{aff(x,e)} \phi$

(2) L'opération de consommation

CONS-act $\quad \varepsilon \vdash cons(x,tc) \xrightarrow[act]{cons(x,tc)} \phi$

Notons qu'il n'est fait aucune référence à la pré-condition de l'opération : c'est la règle de type "modif" pour cette opération qui prendra en compte la valeur de la pré-condition et la réalisation de cette opération ne pourra être prouvée que par l'application des deux règles.

(3) L'opération de production

PROD-act $\quad \varepsilon \vdash prod(x,tc_1,...,tc_k) \xrightarrow[act]{prod(x,tc_1,...,tc_k)} \phi$

(4) L'opération vide

NIL-act $\quad \varepsilon \vdash \text{nil} \xrightarrow[act]{\text{nil}} \phi$

(5) Le schéma conditionnel élémentaire

Si le programme élémentaire p est interprété par l'action atomique a, alors le programme Cond(c,p) est réalisé soit par l'action Cond(c,a) soit par Cond(non c, nil) : telle est la signification des deux règles ci-dessous :

COND1-act

$$\frac{\varepsilon \vdash p \xrightarrow[act]{a} \phi}{\varepsilon \vdash \text{Cond}(c,p) \xrightarrow[act]{\text{Cond}(c,a)} \phi}$$

COND2-act $\quad \varepsilon \vdash \text{Cond}(c,p) \xrightarrow[act]{\text{Cond}(\text{non } c,\text{nil})} \phi$

II.4.2. Fonctionnalité des schémas élémentaires

Les règles "de base" pour décrire les modifications apportées à la mémoire par l'exécution des programmes sont celles des opérations élémentaires qui affectent la mémoire : l'affectation, les opérations de communication et l'action nil, qui est sans effet !

La fonctionnalité des programmes composés se déduira naturellement de la fonctionnalité des schémas élémentaires qui le composent, via les règles d'inférence que nous donnerons au paragraphe II.

Ces règles sont les suivantes :

(1) **NIL-mod** $\varepsilon \vdash m \xrightarrow[modif]{nil} m$

(2) **AFF-mod**

$$\frac{\varepsilon \vdash m \xrightarrow[eval]{e} v}{\varepsilon \vdash m \xrightarrow[modif]{aff(x,e)} m' = mod(m,x,v)}$$

où l'opération **mod**, définie par le type TABLE au chapitre I §I.2, modifie le contenu d'un élément d'une table.

(3) La signification de l'opération de consommation cons(x,tc) est la suivante : si la précondition de l'opération est vérifiée par l'état de communication désigné par tc (que l'on note tc) évaluer simultanément les expressions *val-cons* (tc) et *post-cons* (tc) et affecter simultanément le résultat de ces évaluations aux objets désignés par les variables x et tc.

Ce qui est décrit de manière formelle par le règle CONS-mod, où TC est le type de communication défini dans l'environnement e par la déclaration du port tc :

CONS-mod

$$\frac{\varepsilon \vdash m \xrightarrow[eval]{tc} tc,\ \varepsilon \vdash m \xrightarrow[eval]{TC.pré\text{-}cons(tc)} vrai,\ \varepsilon \vdash m \xrightarrow[eval]{TC.val\text{-}cons(tc)} v,\ \varepsilon \vdash m \xrightarrow[eval]{TC.post\text{-}cons(tc)} t}{\varepsilon \vdash m \xrightarrow[modif]{cons(x,tc)} m' = mod(mod(m,x,v),tc,t)}$$

On constate que cette règle, combinée avec la règle CONS-act, garantit que l'action cons(x,tc) ne pourra être activée que pour un état mémoie qui satisfait la précondition : comme il n'y a pas de règle

applicable lorsque la précondition est fausse, le processus qui invoque cette action ne peut pas évoluer tant que la valeur de la précondition est "faux". La transition sera franchie lorsque la précondition passera à la valeur "vrai" par l'opération d'autres processus sur la mémoire (par l'exécution d'opérations produire sur le port lié à la variable x).

(4) L'opération de production prod(x,tc_1, ... tc_k) est interprétée de la manière suivante :

-évaluer l'opération PRODUIRE, définie par le type TYPECOM (cf chapitre I §III.1), pour chacun des objets désignés par tc_1, ..., tc_k, avec comme valeur produite la valeur désignée par x,

- affecter de manière simultanée les résultats respectifs de ces évaluations aux ports tc_1, ...,tc_k.

La règle PROD-mod décrit cette interprétation :

PROD-mod

$$\forall\, i=1...k\ (\varepsilon \vdash m \xrightarrow[eval]{tci} \underline{tc_i}\ ,\ \varepsilon \vdash m \xrightarrow[eval]{PRODUIRE(\underline{tc_i},v)} t_i\ ),\ \varepsilon \vdash m \xrightarrow[eval]{x} v$$

$$\varepsilon \vdash m \xrightarrow[modif]{prod(x,tc_1,...,tc_k)} m' = mod(...(mod(m,tc_1,t_1)...)tc_k,t_k)$$

(5) La fonctionnalité de l'opération Cond(C,a) est celle de l'action a si la condition C a la valeur "vrai" dans le contexte d'activation :

COND-mod

$$\frac{\varepsilon \vdash m \xrightarrow[eval]{C} \text{'vrai'}\ ,\quad \varepsilon \vdash m \xrightarrow[modif]{a} m',}{\varepsilon \vdash m \xrightarrow[modif]{Cond(C,a)} m'}$$

Cette règle, combinée aux règles NIL-mod, COND1-act et

COND2-act, permet de déduire la fonctionnalité du programme élémentaire Cond(C,p) dans tous les contextes d'activation ainsi que celle des différentes alternatives et itératives ainsi que nous allons le voir dans le prochain paragraphe.

II.5. Sémantique des processus séquentiels

Le corps des processus séquentiels est engendré à partir des schémas élémentaires que nous venons de décrire par la composition en séquence de programmes et les structures de contrôle conditionnelles et itératives qu'offre le langage.

Commençons par donner les règles de la composition de programmes en séquence.

II.5.1. La composition séquentielle de programme : p.q

La séquence d'actions réalisée par le programme p.q est celle du programme p suivie de celle du programme q, ce que définit la règle ci-dessous où u est la première action engagée par le programme p :

SEQ-act

$$\frac{\varepsilon \vdash p \xrightarrow[act]{u} p'}{\varepsilon \vdash p.q \xrightarrow[act]{u} p'.q}$$

Le cas particulier de terminaison du premier processus est inclus dans cette règle moyennant la convention : $\varnothing .q = q$.

On peut établir l'opération d'un programme séquentiel sur un état mémoire initial donné à partir de la séquence d'actions qu'il réalise si la mémoire n'est pas simultanément modifiée par l'action d'un autre programme , c'est ce qu'exprime la règle SEQ1-mod ci-dessous :

SEQ1-mod

$$\frac{\varepsilon \vdash p \xrightarrow[act]{u} p',\ \varepsilon \vdash m \xrightarrow[modif]{u} m'',\ \varepsilon \vdash m'' \xrightarrow[modif]{p'} m'}{\varepsilon \vdash m \xrightarrow[modif]{p} m'}$$

La fonctionnalité d'un programme séquentiel isolé, c'est à dire son abstraction sur la mémoire , est le résultat de son calcul terminé, ce que l'on obtient grâce à la règle SEQ2-mod :

SEQ2-mod

$$\frac{\varepsilon \vdash p \xrightarrow[act]{u} \phi,\ \varepsilon \vdash m \xrightarrow[modif]{u} m'}{\varepsilon \vdash m \xrightarrow[modif]{p} m'}$$

A titre d'exemple définissons la fonctionnalité du programme :

$$(x := x + 1 ; y := y + 2 * x)$$

pour un état mémoire initial où x vaut 1 et y vaut 0.
Ce programme est représenté par le programme abstrait :

$$p.q = \text{aff}(x , x + 1) . \text{aff}(y , y + 2 * x)$$

. Le schéma de déduction est le suivant :

1) Calcul de la fonctionnalité du programme q pour un état m' où x vaut 2 et y 0 (*) :

$$(\rightarrow\text{AFF-mod})\ \frac{m' \xrightarrow[eval]{x} 2,\ m \xrightarrow[eval]{y} 0}{m' \xrightarrow[modif]{aff(y,y+2*x)} m''=m'[4/y]},\ (\rightarrow\text{AFF-act})\ q \xrightarrow[act]{aff(y,y+2*x)} \phi$$

$$(\rightarrow\text{SEQ2-mod})\ \frac{}{m' \xrightarrow[modif]{q} m''=m'[4/y]}$$

2) Calcul de la fonctionnalité de (p.q) pour l'état m initial:

$$\frac{p \xrightarrow[act]{aff(x,1)} \phi}{},\ (\rightarrow\text{AFF-mod})\ \frac{m \xrightarrow[eval]{x} 1}{},\ m' \xrightarrow[modif]{q} m''$$

$$(\rightarrow\text{SEQ-act})\ \frac{p.q \xrightarrow[act]{aff(x,1)} \phi \qquad m \xrightarrow[modif]{aff(x,1)} m' = m[2/x]}{m \xrightarrow[modif]{p.q} m'' = m[2/x,4/y]}$$

II.5.2 Les structures conditionnelles

Nous donnons les règles qui définissent la sémantique du schéma de programme $+_{i=1,n}\ Cond\ (g_i,p_i)$, où les g_i sont des expressions booléennes: en effet nous avons vu, au paragraphe §II.1, que ce schéma permet de décrire l'instruction *select*, aussi bien que l'alternative classique.

La règle qui définit les exécutions possibles lorsqu'au moins une condition aura la valeur 'vrai' est la suivante :

CHOIX1-act

$$\frac{\bigwedge_{i=1,n} \varepsilon \vdash p_i \xrightarrow[act]{u_i} p'_i}{\bigvee_{i=1,n} \varepsilon \vdash +_{i=1,n}\ Cond\ (g_i,p_i) \xrightarrow[act]{Cond(g_i,u_i)} p'_i}$$

où u_i est la première des actions qui composent le programme p_i et les symboles $\wedge$ et $\vee$ indiquent respectivement la conjonction et le choix exclusif et arbitraire.

. Dans le cas où aucune garde n'est à 'vrai' l'état atteint est l'état d'erreur ce qui est formalisé par l'axiome

CHOIX2-act

$$\varepsilon \vdash +_{i=1,n}\ Cond\ (g_i,p_i) \xrightarrow[act]{Cond(\bigwedge_{i=1,n}(non\ g_i),nil)} \text{"erreur"}$$

(*) la notation m' = m[e/x] signifie m' = m sauf pour x qui prend la valeur e.

Il est inutile de définir de règle particulière pour spécifier la fonctionnalité du programme $+_{i=1,n}$ Cond(gi,pi):

- la règle COND-mod va permettre le choix entre les deux règles d'exécution symbolique CHOIX1-act et CHOIX2-act
- ensuite la règle COND-mod combinée aux règles SEQ1-mod et SEQ2-mod permettant de déduire la fonctionnalité du programme selon le choix qui aura éré réalisé entre les différentes alternatives (ou l'ensemble des fonctionnalités lorsque plusieurs choix sont possibles).

. A titre d'exemple élaborons la fonctionnalité du programme

si x < 2 alors x := 0 ; y := 10 sinon x := 10 fsi

dans un contexte mémoire initial où x a la valeur 1.

Ce programme a pour forme abstraite :

Cond (x < 2, aff(x,0).aff(y,10)) + Cond(x≥2, aff(x,10))

que nous schématiserons par Q = Cond(c,p) + Cond(non c ,q).

$$(\rightarrow\text{COND-mod}) \quad \frac{m \xrightarrow[eval]{c} \text{'vrai'} \;,\; m \xrightarrow[modif]{aff(x,0)} m' = m[0/x]}{m \xrightarrow[modif]{Cond(c,aff(x,0))} m' = m[0/x]} \quad (1)$$

$$(\rightarrow\text{CHOIX1-act}) \quad \frac{p \xrightarrow[act]{aff(x,0)} aff(y,10)\;,}{Q \xrightarrow[act]{Cond(c,aff(x,0))} aff(y,10)} \quad (2)$$

$$(\rightarrow\text{AFF-mod}) \quad \frac{m' \xrightarrow[eval]{10} 10}{m' \xrightarrow[modif]{aff(y,0)} m'' = m'[10/y]} \quad (3)$$

$$(\rightarrow\text{SEQ1-mod}) \qquad \frac{(2),\ (1),\ (3)}{m \xrightarrow[modif]{Q} m'' = m\,[\,0/x, 10/y]}$$

Remarque :

La règle CHOIX1-act montre que :

- si le programme pi qui est activé ne termine pas alors le programme + $Cond(g_i,p_i)$ ne termine pas.

- la seule première action du programme choisi est gardée et l'exécution de cette première action forme une action indivisible avec l'évaluation des gardes.

II.5.3 les structures itératives

Rappelons l'interprétation de l'itération "classique" < répéter C →p>:

< répéter C →p> = < si C alors p ; répéter C →p>

Par conséquent l'exécution symbolique du programme *(C,p) qui représente une telle itération est définie par les deux règles ci-dessous :

ITER1-act

$$\frac{\varepsilon \vdash p \xrightarrow[act]{a} p'}{\varepsilon \vdash *(C,p) \xrightarrow[act]{Cond(c,a)} p'.\ *(C,p)}$$

ITER2-act

$$\varepsilon \vdash *(C,p) \xrightarrow[act]{Cond(\underline{non}\ c, nil)} \phi$$

. Comme l'itération indéterministe est décrite par le schéma du

type *(abs(g1 ou ... ou gn), + Cond (gi,pi)), son interprétation sera
$i=1,n$
déduite des règles de l'itération et du choix. Notons que l'état d'erreur défini par la règle CHOIX2-act ne peut pas être obtenu ici puisque l'itération non-déterministe a pour condition qu'une au moins des gardes soit vérifiée.

. Il est évident que la fonctionnalité des itérations pourra être inférée à partir de la règle COND-mod et des règles sur la séquence d'actions (SDEQ1-mod et SEQ2-mod)

. Par exemple élaborons l'exécution de l'itération :

répéter x < 2 ——> x := x + 1 fin répéter

dans le contexte initial de mémoire où x vaut 0.

Le schéma abstrait correspondant est: * Cond (x<2 , aff(x,x+1))

que nous noterons Q = * Cond(c,p)

(→COND-mod)
$$\frac{m \xrightarrow[eval]{c} \text{'vrai'},\ m \xrightarrow[modif]{aff(x,x+1)} m_1 = m[1/x]}{m \xrightarrow[modif]{Cond(c,aff(x,x+1))} m_1 = m\,[\,1/x]} \quad (1)$$

(→ITER1-act)
$$\frac{p \xrightarrow[act]{aff(x,x+1)} \phi}{Q \xrightarrow[act]{Cond(c,aff(x,x+1))} {}^{*}Cond(c,p) \equiv Q} \quad (2)$$

(→COND-mod)
$$\frac{m_1 \xrightarrow[eval]{c} \text{'vrai'},\ m_1 \xrightarrow[modif]{aff(x,x+1)} m_2 = m_1[2/x]}{m_1 \xrightarrow[modif]{Cond(c,aff(x,x+1))} m_1 = m\,[\,2/x]} \quad (3)$$

(→COND-mod)
$$\frac{m_2 \xrightarrow[eval]{\text{non } c} \text{'vrai'},\ m_2 \xrightarrow[modif]{nil} m_2}{m_2 \xrightarrow[modif]{Cond(\text{non } c,nil)} m_2 = m_1\,[\,2/x]} \quad (4)$$

$$(\rightarrow \text{ITER2-act}) \qquad Q \xrightarrow[act]{\text{Cond}(\underline{\text{non}}\ c\ ,\ nil)} \phi \qquad (5)$$

$$(\rightarrow \text{SEQ2-mod}) \qquad \frac{(5),\quad (4)}{m_2 \xrightarrow[modif]{Q} m_2 = m_1[2/x]} \qquad (6)$$

$$(\rightarrow \text{SEQ1-mod}) \qquad \frac{(2),\quad (3)\ ,(6)}{m_1 \xrightarrow[modif]{Q} m_2 = m_1[2/x]} \qquad (7)$$

$$(\rightarrow \text{SEQ1-mod}) \qquad \frac{(2),\quad (1)\ ,\ (7)}{m \xrightarrow[modif]{Q} m_2 = m_1[2/x]} \qquad (8)$$

Ce qui met fin au déroulement de la preuve.

Il reste maintenant à définir la sémantique de la composition parallèle de processus séquentiels.

II.6. Sémantique des programmes parallèles

Rappelons que le comportement d'un programme parallèle sera formalisé par une séquence de transitions, de la forme

$$(\text{programme, état}) \xrightarrow{\underline{\text{action}}} (\text{programme', état'}),$$

réalisées par la machine abstraite que nous avons définie au paragraphe II.2.

Ces transitions seront inférées à partir des règles qui définissent les actions possibles des processus composant le

programme, leurs fonctionnalités et à partir de conditions de déclenchement portant :

- sur les valeurs des horloges des processus : par exemple l'action a du processus 1 et l'action b du processus 2 ne peuvent être exécutées simultanément que si les horloges des processus sont identiques,

- sur la nature des actions que les processus peuvent engager à un instant donné : par exemple si l'action a du processus 1 et l'action b du processus 2 du programme agissent sur le même port de communication, elles devront opérer en entrelacement,

- sur l'état des objets de communication désignés par les ports à un instant donné.

Nous avons défini au §II.4 les actions atomiques réalisées par les processus séquentiels : rappelons que l'interprétation des programmes parallèles met en jeu deux autres actions (cf §II.2): l'action *Par* et l'action *prod-cons*.

Nous commençons donc par définir la sémantique de ces deux actions, dans un premier paragraphe , puis nous présentons les règles qui définissent l'exécution des programmes parallèles au paragraphe suivant.

III.6.1 Sémantique des actions Par et prod-cons

Commençons par définir la fonctionnalité de l'action Par : la composition parallèle d'actions n'est possible que si les actions peuvent s'exécuter simultanément c'est à dire si elles n'opèrent pas sur le même port de communication : nous notons "non exclusif $(a_1,...,a_n)$" le prédicat qui définit cette caractéristique d'un ensemble d'actions.
Pour spécifier de manière formelle ce prédicat, notons COM l'ensemble des actions de communication prod et cons, SYMBOLES(a_i) l'ensemble des noms des paramètres de l'action a_i et PORT l'ensemble des noms de port déclarés par le programme traité :

$$\text{exclusif}(a_1,...,a_n) \equiv \exists\, i,j \in [1,n]\ (a_i \in \text{COM} \wedge a_j \in \text{COM} \wedge \text{SYMBOLES}(a_i) \cap \text{SYMBOLES}(a_j) \cap \text{PORT} \neq \phi).$$

La règle qui définit la fonctionnalité de l'action Par est la

suivante :

PAR-mod

$$\frac{m \xrightarrow[modif]{a_1} \delta 1(m), \ldots, m \xrightarrow[modif]{a_n} \delta n(m), \text{ non exclusif}(a_1,\ldots,a_n)}{m \xrightarrow[modif]{Par(a_1,\ldots,a_n)} m' \quad = \bigcup_{i=1,n} \delta i(m)}$$

où n est le nombre de processus
$\delta i(m)$ désigne le nouvel état-mémoire atteint à partir de l'état m par l'exécution de l'action a_i : les modifications $\delta i(m)$ apportées par chacune des actions se juxtaposent au niveau de la mémoire car chacune d'elles est localisée (au niveau de la mémoire locale d'un processus ou d'un port de communication) et il n'y a pas d'"interférence" possible puisque les actions engagées ne sont pas "exclusives".

Toutefois il faut noter que lorsque les actions a_i, engagées simultanément par l'action Par n'ont pas la même durée, l'état m' défini par la règle *PAR-mod* n'est **parfait** qu'à la fin de l'action la plus longue ; cependant chaque état mémoire local à un processus (ou chaque objet de communication accédé par l'une des actions) est observable dès que l'action correspondante est terminée : nous allons utiliser cette possibilité pour activer séparément les processus qui sont dans un état local parfait même lorsque l'état global du programme ne l'est pas.
Notons que lorsque l'opération *Par* engage une opération de communication, l'objet de communication accédé est immédiatement de nouveau observable puisque les opérations de communication sont instantanées.

Définissons maintenant l'action **prod-cons** :
elle réalise l'activation consécutive mais atomique d'une action prod et d'une action cons sur le même objet de communication .
Sa fonctionnalité est décrite par la règle suivante :

PROD-CONS-mod

$$\frac{\varepsilon \vdash m \xrightarrow[modif]{prod(x,tc)} m', \quad \varepsilon \vdash m' \xrightarrow[modif]{cons(y,tc)} m''}{\varepsilon \vdash m \xrightarrow[modif]{prod\text{-}cons(x,y,tc)} m''}$$

Notons que cette action ne pourra être réalisée que si l'état m' qui résulte de la production est tel que la précondition de consommation soit vérifiée : comme une opération de production ne peut pas rendre fausse une précondition qui a la valeur 'vrai', l'opération *cons* activable simultanément à la production réalisée sera toujours activée. Par contre le résultat n'est pas forcément celui de l'opération *cons;prod*, en particulier pour les types de communication où la valeur consommée est la plus récente ("aléatoire", "rafraichie", ...)

III.6.2 Règles d'exécution des programmes parallèles (*)

La première règle pour décrire les transitions globales d'un programme est la règle R1-exec : elle définit l'une des transitions possibles d'un programme dans un état parfait, c'est à dire lorsque l'ensemble des processus possèdent la même valeur d'horloge.

Ce cas est celui où les actions possibles des processus composés en parallèle ne sont pas exclusives : l'action Par est alors activée.
On notera $\delta(a)$ la durée de l'action

R1-exec

$$\frac{\varepsilon \vdash p_1 \xrightarrow[act]{a} p'_1,\ \varepsilon \vdash p_2 \xrightarrow[act]{b} p'_2,\ \varepsilon \vdash m \xrightarrow[modif]{Par(a,b)} m'}{\varepsilon \vdash (p_1 // p_2, < m \times h \times h >) \xrightarrow{Par(a,b)} (p'_1 // p'_2, < m' \times h+\delta(a) \times h+\delta(b) >)}$$

Notons que le prédicat non exclusif(a,b) n'est pas nécessaire en prémisse de la règle puisque la transition de type "modif" sur l'action Par ne pourra être réalisée que si ce prédicat est vérifié.

Notons également que si les actions a et b sont des opérations de communication les horloges des processus gardent la même valeur puisque $\delta(a) = \delta(b) = 0$.

(*) Dans tout ce paragraphe nous décrirons, pour simplifer la compréhension, les règles pour 2 processus composés en parallèle : rappelons que le cas général s'en déduit moyennant la convention suivante :
p//q//r = p // (q//r) = (p//q)//r.

. Le second cas possible, toujours pour un état global parfait, est celui où les actions a et b sont exclusives : ici deux sous-cas sont à envisager selon que l'ordre d'activation de ces deux actions est arbitraire ou non. Rappelons que cet ordre est arbitraire s'il s'agit des combinaisons prod-prod ou cons-cons sur le même port et que sinon la production est prioritaire sur l'action consommer.

Dans ce dernier cas, il faut réaliser de manière consécutive l'action *prod* puis l'action *cons* mais de manière atomique car sinon la consommation risquerait de ne jamais être activée (situation de *famine*) (exemple du programme *(Prod(x,tc) // * (Cons(y,tc))).

C'est à cette fin qu' a été introduite au paragraphe II.2 l'action atomique prod-cons que nous avons définie ci-dessus .

La règle R2-exec définit le cas d'activation de cette action :

R2-exec

$$\varepsilon \vdash p_1 \xrightarrow[act]{cons(y,tc)} p'_1,\ \varepsilon \vdash p_2 \xrightarrow[act]{prod(x,tc)} p'_2,\ \varepsilon \vdash m \xrightarrow[modif]{prod(x,tc)} m',$$

$$\varepsilon \vdash m \xrightarrow[modif]{cons(y,tc)} m'',\quad \varepsilon \vdash m \xrightarrow[modif]{prod\text{-}cons\ (x,y,tc)} m'''$$

$$\varepsilon \vdash (p_1 // p_2, <m\ x\ h\ x\ h>) \xrightarrow{prod\text{-}cons(x,y,tc)} (p'_1 // p'_2, <m'''\ x\ h\ x\ h>)$$

. Le second cas d'actions exclusives, à réaliser selon un ordre arbitraire, est décrit par la règle R3-exec : soit l'action a soit l'action b est activée par la transition et cette règle a deux conclusions possibles.

On note arbitraire (a,b) le prédicat qui a la valeur 'vrai' si le couple (a,b) est de la forme (prod, prod) ou (cons,cons) .

R3-exec

$$\varepsilon \vdash p_1 \xrightarrow[act]{a} p'_1,\ \varepsilon \vdash p_2 \xrightarrow[act]{b} p'_2,\ \varepsilon \vdash m \xrightarrow[modif]{a} m',\ \varepsilon \vdash m \xrightarrow[modif]{b} m''$$

exclusif(a,b) , arbitraire(a,b)

$$\varepsilon \vdash (p_1 // p_2, < m \times h \times h >) \xrightarrow{a} (p'_1 // p_2, < m' \times h \times h >),$$
$$\varepsilon \vdash (p_1 // p_2, < m \times h \times h >) \xrightarrow{b} (p_1 // p'_2, < m'' \times h \times h >)$$

Notons que dans chacun des cas les horloges gardent les mêmes valeurs car il s'agit d'opérations de communication, donc instantanées.
Ces règles seraient plus complexes si on les exprimait pour n processus dont certains pourraient réaliser des actions de durée non nulle : les transitions obtenues réaliseraient une action de la forme Par(a1,...,ak) où les ai composés simultanément seraient soit des actions locales aux processus soit des opérations de communication deux à deux non exclusives.

. Enfin le dernier cas possible, pour un état global parfait, est celui où les actions potentielles ne sont pas exclusives mais ne sont pas toutes activables dans le contexte mémoire considéré : tel est le cas d'opérations de communication dont la précondition n'est pas satisfaite.

Les processus qui invoquent de telles opérations sont bloqués en **attente** d'opérations de production réalisées par d'autres processus de système : l'horloge de tels processus va progresser jusqu'à atteindre la date à laquelle est susceptible d'arriver une opération de production qui rende la précondition de consommation vraie ; dans le cas d'un système composé de deux processus, dont l'état est parfait, cette date au plus tôt est celle définie par l'activation du processus qui n'est pas bloqué.

Il faut noter que lorsque l'ensemble des processus du système sont dans une telle situation, le système est dans un état de blocage : aucune transition n'est plus activable.

La règle R4-exec décrit la situation où au moins un processus n'est pas bloqué :

R4-exec (*)

$$\frac{\varepsilon \vdash p_1 \xrightarrow[act]{a} p'_1,\ \text{est-cons?}(a,tc),\ \varepsilon \vdash m \xrightarrow[eval]{TC.\text{pré-cons}(\underline{tc})} \text{'faux'},\quad \varepsilon \vdash p_2 \xrightarrow[act]{b} p'_2,\ \varepsilon \vdash m \xrightarrow[modif]{b} m'}{\varepsilon \vdash (p_1 // p_2, < m \times h \times h >) \xrightarrow{b} (p_1 // p'_2, < m' \times h+\delta(b) \times h+\delta(b) >)}$$

Notons que l'état obtenu est parfait puisque les horloges prennent la même valeur.

Envisageons maintenant le cas des états imparfaits lorsque les horloges des processus n'ont pas toutes la même valeur. Comme nous l'avons dit précédemment, seuls sont activables les processus dont les horloges ont la valeur minimum de l'ensemble des horloges, à condition que l'état mémoire correspondant définisse une transition possible. Tel n'est pas le cas lorsque la précondition de la prochaine opération à activer pour un processus n'est pas vérifée.

On obtient ainsi les deux règles R5-exec et R6-exec ci-dessous.

. **R5-exec** : cas où le processus d'horloge minimum peut réaliser l'une de ses actions potentielles :

R5-exec

$$\frac{\varepsilon \vdash p_1 \xrightarrow[act]{a} p'_1,\quad \varepsilon \vdash m \xrightarrow[modif]{a} m',\ h_1 < h_2}{\varepsilon \vdash (p_1 // p_2, < m \times h_1 \times h_2 >) \xrightarrow{a} (p'_1 // p_2, < m' \times h_1+\delta(a) \times h_2 >)}$$

Notons que le nouvel état obtenu est parfait si $h_2 = h_1 + \delta(a)$.

(*) On note est-cons?(a,tc) le prédicat qui a la valeur 'vrai' si l'action a est une opération de la forme cons(x,tc). on note TC le type de communication associé au port tc et tc l'objet de communication désigné par tc.

Rappelons que cette transition est licite bien que le processus p_2 n'ait pas terminé l'action activée par la transition précédente :

- par hypothèse $h_1 < h_2$, donc p_2 ne peut être en train de réaliser une opération de communication car sinon $h_1 = h_2$,
- le processus p_1 peut donc réaliser soit une opération de communication soit une affectation locale.

. Le second cas est celui où le (ou les) processus d'horloge minimum est bloqué devant une opération de consommation dont la précondition n'est pas satisfaite, comme nous l'avons vu dans le cas d'un état parfait. De tels processus seront bloqués jusqu'à ce qu'un autre processus du système réalise une opération de production qui rende la précondition vérifiée.

Dans le cas d'un programme composé de deux processus, la prochaine opération de production aura lieu, au plus tôt, à la date de fin de l'action actuellement en cours à l'instant où est évaluée cette précondition : le processus ainsi bloqué est donc en attente jusqu'à au moins cette date.

La règle R6-exec décrit cette situation :

R6-exec

$$\frac{\varepsilon \vdash p_1 \xrightarrow[act]{a} p'_1,\ \text{est-cons?}(a,tc),\ \varepsilon \vdash m \xrightarrow[eval]{TC.\text{pré-cons}(tc)} \text{'faux'},\ h_1 < h_2}{\varepsilon \vdash (p_1 // p_2, < m \times h_1 \times h_2 >) \xrightarrow{nil} (p_1 // p_2, < m \times h'_1 = h_2 \times h_2 >)}$$

Notons que l'état obtenu est un état parfait puisque $h'_1 = h_2$.

Notons que pour cette règle aussi l'expression est simplifiée en se limitant à deux processus. Dans le cas d'un système quelconque, tous les processus d'horloge minimum ainsi bloqués devraient être "mis en attente" jusqu'à la date h immédiatement supérieure à l'horloge minimum : première date possible pour une éventuelle opération de production.

. Cette règle est la dernière de notre système de déduction. On

montrerait par une étude par cas de l'ensemble des configurations (programme, état) possibles que le système de règles est complet.

. En conclusion à ce paragraphe et avant de montrer sur un exemple l'exécution complète d'un programme parallèle, revenons sur un point que nous jugeons important. Il peut sembler à première vue artificiel que les opérations de communication soient instantanées alors que les autres actions, en l'occurence les affectations, prennent du temps. Cette hypothèse a été faite dès le premier chapitre lors de la définition des relations de communication (§ II.2) : cette hypothèse est réaliste dans la mesure où on souhaite qu'un événement , i.e. la production d'une valeur, soit immédiatement perçu. Toutefois l'hypothèse inverse aurait un sens dans le cadre de systèmes décentralisés. Mais cette hypothèse n'est en aucun cas simplificatrice : le système de règles serait un peu plus complexe (au niveau des règles R4-exec et R5-exec uniquement) mais resterait fondamentalement le même.

. Remarquons d'autre part que dans les six règles ci-dessus la transition sur l'exécution globale du programme aurait pu s'écrire systématiquement sous la forme

$$(p_1 // p_2, < m \times h_1 \times h_2 >) \xrightarrow{Par(a_1,a_2)} (p'_1 // p'_2, < m' \times h'_1 \times h'_2 >)$$

même lorsqu'une seule action est engagée : dans ce cas la seconde action, dans l'action composée Par, serait l'action nil, moyennant l'ajout au système de règles de l'axiome $\mathcal{A}$: $p \xrightarrow[act]{nil} p$.

Nous retiendrons ce choix de description au chapitre suivant, par commodité uniquement . Nous n'avons pas voulu ici faire a priori ce choix ni introduire l'axiome $\mathcal{A}$, comme le proposent les modèles de type CCS pour introduire l'asynchronisme sans manipuler le temps, car cette description masque la réalité; par exemple dans la transition définie par la règle R5-exec seule l'action a du processus p1 est engagée et simultanément le processus p2 termine une action activée précédemment: il n'est donc pas inactif comme le signifie l'action nil. Le seul cas réel d'inactivité est celui exprimé par la règle R6-exec lorsque les processus qui ont terminé une action sont bloqués en attente de consommation .

II.7. Exemple d'exécution

Nous allons décrire le déroulement d'un programme parallèle "jouet" dans un contexte d'activation tel que les deux processus composés en parallèle ont le même rythme.

Nous montrerons des situations d'attente, inhérentes au type de communication utilisé (l'"égalité") et non pas aux rythmes respectifs des processus.

Le programme est le suivant : R = P // Q avec

$P \equiv *(aff(x,x+1) \,.\, prod(x,tc))$, noté $*(u1.u2)$,

$Q \equiv *(cons(y,tc) \,.\, aff(z,y*2))$, noté $*(v1.v2)$,

où le port tc est de type "égalité".

Supposons que les affectations ont une durée égale à l'unité (les communications sont instantanées) : $\delta(u_1)=1$, $\delta(u_2)= 0$, $\delta(v_1) =0$, $\delta(v_2)=1$;

. Soit m le contexte mémoire initial où les variables x,y et z valent 0 et où l'ensemble consommable de l'objet désigné par tc est vide, ce que nous noterons

$$m = [x=0,\ y=0,\ z=0,\ tc=\{\ \}]$$

Soient h_1 et h_2 les horloges respectives des processus P et Q : à l'état initial elles sont toutes deux nulles.

. <u>1ère transition du système</u> :

l'état initial < m x h1 x h2 > est parfait ; la règle R1-exec ne peut pas s'appliquer bien que les premières actions respectives de P et Q , u1 et v1, ne soient pas exclusives car la préconditon de l'opération v1 n'est pas satisfaite: le processus Q est alors en **attente** ; c'est la règle R4-exec qui s'applique :

(R4-exec→)

$$\frac{\ldots\ldots\ ,\ \varepsilon \vdash m \xrightarrow[modif]{u1} m' = [x=1,\ y=0,\ z=0,\ tc=\{\ \}]}{(P//Q,\ < m x h_1=0 x h_2=0 >) \xrightarrow{u1} \underbrace{(u2.*(u1.u2))}_{P'} // Q\ ,<m' x\ h_1=1 x\ h_2=1>)}$$

. 2ème transition :

L'état <m' x h_1=1x h_2=1> est parfait : cette fois encore seul le premier processus est activable . L'opération u2 (production de la valeur de x) sera donc engagée par la transition, d'après la règle R4-exec, ce qui lèvera l'attente du processus Q à la transition suivante :

(R4-exec→)

$$\frac{\ldots\ldots\ ,\ \varepsilon \vdash m' \xrightarrow[modif]{u2} m''=\ [x=1,\ y=0,\ z=0,\ tc=\{1\}\ ]}{(P\ //\ Q,\ <\ m'\ xh_1=1xh_2=1>) \xrightarrow{u2} (\underbrace{*(u1.u2)}_{P}\ //\ Q,\ m''\ xh_1=1\ xh_2=1>)}$$

. 3ème transition :

L'état est parfait et les deux actions u1 et v1 peuvent avoir lieu simultanément et la règle R1-exec s'applique :

(R1-exec →)

$$\frac{P \xrightarrow[act]{u1} u2.P,\ Q \xrightarrow[act]{v1} v2.Q,\ m'' \xrightarrow[modif]{Par(u1,v1)} m_3 = [x=2,y=1,z=0,tc=\{\ \}]}{P//Q,<\ m''\ x\ h_1=1x\ h_2=1>) \xrightarrow{Par(u1,v1)} (\underbrace{u2.P}_{P'}//\underbrace{v2.Q}_{Q'},<m_3\ xh_1=2\ x\ h_2=1>)}$$

. 4ème transition :

L'état est imparfait : $h_1 > h_2$. Seul le processus Q' peut être activé et son action potentielle v2 est réalisable.

La règle R5-exec s'applique :

$$(R5\text{-}exec\rightarrow)\ \frac{Q' \xrightarrow[act]{v2} Q,\ m_3 \xrightarrow[modif]{v2} m_4 = [x=2,\ y=1,\ z=2,\ tc=\{\ \}],h_2<h_1}{P'//Q',\ <m_3\ x\ h_1=2xh2=1> \xrightarrow{v2} (P'//Q,\ <m_4\ xh_1=2\ xh_2=2>)}$$

. 5ème transition :

L'état obtenu est parfait mais seule l'action u2 est activable (la précondition de l'action potentielle v1 de Q n'est pas vérifiée) : on est dans un état similaire à celui de la 2ème transition. Ce sera la règle R4-exec qui s'appliquera et aura pour conclusion :

$(P'//Q, < m_4 \times h_1=2 \times h_2=2 >) \xrightarrow{u2}$
$(P//Q, <m_5=[x=2,y=1,z=2,tc=\{2\}] \times h_1=2 \times h_2=2 >)$

Ensuite le reste de l'exécution réalise le cycle

Par(u1,v1) v2 u2
[——————>, ——————>, ——————>] qui est décrit par la séquence des transitions n° 3, 4 et 5 détaillées ci-dessus :

- consommation sur le port tc par le processus Q sans attente simultanément à l'action u1 de P
- action v2 du processus Q : affectation de z
- production sur le port tc par p (action u2)
- l'état obtenu à la fin du cycle est toujours parfait et par conséquent chaque pas des itérations combinées des processus P et Q est réalisé par ce cycle.

On constate, sur cet exemple, que le cas d'attente pour le processus Q lors des transitions n°1 et 2 est lié au fait qu'au début de cette exécution la précondition n'était pas vérifiée (ensemble A du port tc vide) : l'attente a été levée à l'instant de la première production sur le port tc par le processus P (instant 1). Cette attente aurait été plus longue si le processus P avait été plus lent.
Ensuite le processus producteur réalise une production d'avance (transition n°5) et il n' y aura plus d'attente car à tout instant où sera évaluée la précondition de consommation, le nombre de productions sera égal au nombre de consommations plus un.

Notons que si le type de consommation avait été "*très aléatoire*", par exemple, (précondition = 'vrai') la première attente n'aurait pas eu lieu et le processus Q aurait consommé la valeur indéfinie 'ω'.

III. CONCLUSION

III.1. Travaux reliés

Il est difficile en peu de mots de résumer les caractéristiques des divers langages de programmation parallèle proposés à l'heure actuelle. Depuis l'introduction de la communication par envoi de message avec CSP, nombreuses sont les propositions de langages de processus communicants. On peut retenir les critères de classification suivants [Per 87]:

- le type des problèmes auquel ils sont dédiés : systèmes réactifs ou transformationnels,
- le style d'expression : impératif ou déclaratif,
- le mode de communication : synchrone ou asynchrone.

Pour la catégorie des systèmes transformationnels le style d'expression est généralement impératif, hormis le cas particulier des langages "data-flow" pour lesquels il n'y a pas d'expression explicite du contrôle. Pour les systèmes transformationnels la décomposition du programme en processus correspond à un partage du calcul des résultats avec coopération entre les processus pour la transmission des résultats intermédiaires. Les différences entre les divers langages impératifs dédiés à ces problèmes concerne l'expression des communications: synchrone ou asynchrone, destinataire désigné explicitement ou non, liaisons bipoints ou multipoint, ... Citons dans cette catégorie, outre les "pionniers" CSP et ADA, CESAR [Que 82], LC3 [Lec 86] , ARGUS [Lis 84] par exemple.

Pour les systèmes réactifs il s'agit de décrire comment le système maintient un certain équilibre en réponse aux sollicitations externes: les langages décrivent alors le comportement de tels systèmes. Le style déclaratif est particulièrement adapté à cette expression; citons, par exemple FP2 [Jor 84], LCS [Ber 85] et dans une approche "modélisation" CCS [Mil 80] et les calculs de processus dérivés tels que Meije [Bou 84] par exemple. Notons la présence des langages LUSTRE(applicatif)[CPH] et ESTEREL[BeC 85] (impératif) pour l'expression des problèmes temps réel.

En ce qui concerne la partie "sémantique" nous ne reviendrons pas ici sur les diverses expressions qui ont retenu notre attention et auxquelles nous avons fait référence dans ce chapitre.

III.2. Bilan

Dans la première partie de ce chapitre nous avons présenté le langage de description des programmes parallèles initialement défini dans [Per 85]. Ce langage n' a pas été conçu dans l'optique de définir un nouveau langage de programmation de systèmes parallèles mais comme un outil pour la représentation des solutions. Notons d' ailleurs que divers travaux de l'équipe COMETE portent sur des implantations des programmes en ADA [JKP 86] et en OCCAM [EJu 88]; mentionnons également qu'à l'heure actuelle une description fonctionnelle des programmes est proposée en lieu et place du langage impératif décrit ici [Jul 89].

Les principaux apports de ce langage, en matière d'expression du parallélisme, sont les suivants :

- la notion de type de communication constitue le noyau du langage dans la mesure où elle permet des communications asynchrones et décharge le programmeur de tout souci de synchronisation. Les contraintes de synchronisation exprimées dans la sémantique sont dues aux conflits d'accès aux objets partagés que sont les ports de communication et aux "attentes" des processus consommateurs lorsque les consommations ne sont pas immédiatement réalisables,

- les "stratégies" de communication sont "encapsulées" à l'intérieur des types de communication et le corps des processus est limité à la fonction de "calcul" : c'est un apport important en matière de modularité des programmes puisqu'une politique de communication peut être substituée à une autre sans modifier le corps du processus. Notons de plus l'avantage de cette modularité qui permet de ne pas lier les choix non-déterministes aux communications contrairement à la philosophie des "commandes gardées" de CSP; nous en mesurerons l'intérêt au chapitre V pour la validation des programmes.

La seconde partie du chapitre a été consacrée à la définition de la sémantique de ce langage. Nous avons envisagé plusieurs choix avant de retenir ceux que nous venons d' exposer et la définition que nous proposons ne prétend pas être "idéale" mais correspondre aux objectifs visés.

Nous avons cherché à obtenir une sémantique de type "opérationnelle", puisqu'elle doit servir de fondement à la définition de l'interprète, et qui n'assimile pas le caractère d'**atomicité** d'une action à la caractéristique de **durée élémentaire**. En effet, en général, les modèles de description du comportement des processus, hérités de la philosophie CCS, supposent que chaque instant de l'échelle du temps correspond à l'exécution d'une action atomique. Nous ne nous opposons pas à cette hypothèse et l'aurions probablement adoptée si nous avions eu à réaliser un compilateur du langage : dans ce cas chaque instruction du langage est implantée par une séquence d'actions élémentaires et il est raisonnable de supposer que ces actions élémentaires ont toutes même durée.

Une sémantique "axiomatique" telle que le propose Lamport dans [Lam 85] nous semble bien adaptée à ce type de description. Dans cette approche chaque primitive du langage est spécifiée par la donnée des prédicats "at" (devant l'action), "in" (pendant l'action) et "after" (après l'action).

Dans notre cas notre problème n'est pas d'implanter les primitives du langage mais d'interprèter les programmes et en particulier d' appréhender la simultanéité d' évènements et de mettre en évidence les communications. D'où le choix des actions élémentaires que nous avons fait et qui est d'ailleurs couramment adopté dans la sémantique des langages de programmation parallèle, [Lec 86], [Que 82] par exemple.

Dans un premier temps nous avions donné à l'opérateur de composition parallèle une signification analogue, par exemple, à celle de l'opérateur // du calcul Meije [Bou 85]; dans ce calcul la composition d'actions élémentaires a//b peut être interprétée par le déclenchement d'une des transitions suivantes :

$a//b \xrightarrow{a//b} \varnothing$, $a//b \xrightarrow{a} b$, $a//b \xrightarrow{b} a$, où l'action a//b a pour durée le maximum des durées de a et b.

Nous obtenions un système de transitions équivalent à celui proposé dans [Bou 87], sauf pour la partie "atomes": ce système était composé des règles de type *act* et *modif* présentées ici et complété d'une règle d'inférence exprimant la composition de ces deux types de règles pour déduire les transitions du programme de la forme

$(p,m) \xrightarrow{action} (p',m')$.

Etant donné notre choix d'actions atomiques il est apparu restrictif de s'interdire d'observer l'état du système lorsqu'une des deux actions engagées simultanément dans une composition parallèle termine avant l'autre.
D'où l'introduction explicite du temps qui permet de modéliser réellement la simultanéité d'évènements, sans alourdir considérablement le système de déduction qui définit cette sémantique : en effet seules les six dernières règles de type "*exec*" utilisent les horloges pour déduire la signification du programme à partir de celle des processus internes.

Enfin notons que nous obtenons une sémantique d'un "style" proche de celui de la "sémantique naturelle" exprimée à l'aide du formalisme "TYPOL" (cf [Kah 87] et [Des 86]); remarquons que si le système "CENTAUR" [BCD 87] avait été disponible au début de notre travail la réalisation de l'interprète aurait pu être développée sous ce système.

CHAPITRE III

L'INTERPRETATION
(LES PRINCIPES)

L'INTERPRETATION (LES PRINCIPES)

I. INTRODUCTION

L'objet de ce chapitre est de présenter les principes adoptés pour l'interprétation des programmes et de décrire fonctionnellement l'interprète séquentiel qui en découle.

Dans le cadre du projet COMETE la réalisation d'un interprète avait pour objectif, à l'origine, de permettre au concepteur de systèmes parallèles de valider "intuitivement" ses solutions par une simulation de l'exécution et notamment une visualisation graphique des communications ; nous verrons au chapitre V comment l'interprète réalisé constitue le noyau d'un système de preuves.

Un interprète d'un langage quelconque a pour rôle principal de réaliser, pour tout programme et tout jeu de données, un calcul conforme à la sémantique du langage.

Ainsi pour un langage séquentiel déterministe, le calcul réalisé par un interprète est décrit par une **séquence de transitions** de la machine abstraite associée au langage, ainsi que nous l'avons vu au chapitre précédent.

Pour un langage non déterministe, et a fortiori parallèle, à un couple (programme, données) ne correspond pas une suite linéaire

d'actions mais un ensemble de suites d'actions : on peut alors décrire le fonctionnement d'un interprète comme celui d'un automate non déterministe sur des arbres de suites d'actions [Bü 62] [Ra 70].

Ainsi nous proposons pour définir l'interprétation et la vérification de programmes de travailler à partir de l'arbre de comportements possibles d'un programme :

- l'**interprétation** d'un programme consiste à choisir l'un des comportements possibles pour un état initial donné, (ou éventuellement plusieurs de ces comportements en interaction avec l'utilisateur) et à le simuler sur une machine abstraite séquentielle,

- la **vérification** d'une propriété sera réalisée en montrant que cette propriété est vérifiée par tous les comportements possibles.

Pour définir l'ensemble des comportements possibles d'un programme parallèle nous procédons de la manière suivante :

1- A chaque processus élémentaire du programme est associé son **arbre d'exécution** qui représente l'ensemble des **traces d'exécution** ; une trace d'exécution est une suite d'actions définie par les règles d'**exécution** ("symbolique") de la sémantique pour un processus donné (cf chapitre II § II.4 et II.5)

2- Pour simplifier l'expression des traces d'exécution des processus séquentiels, et par conséquent celles des programmes composés, sans décrire l'ensemble des combinaisons possibles d'actions atomiques, nous utilisons la notion d'abstraction que nous avons définie au chapitre II §II.5 et introduisons la notion d'**unité d'exécution** : une unité d'exécution est une séquence d'actions atomiques dont seul le premier terme est une opération de communication. Intuitivement, l'exécution d'une telle action abstraite, une fois l'opération de communication réalisée, est sans effet sur les actions d'autres processus activées simultanément et le comportement d'un programme parallèle pourra ainsi être décrit en l'articulant autour des points stratégiques que sont les communications.

3. Au programme P=P1// ... //Pn nous associons un arbre d'exécution décrit par l'ensemble des séquences possibles de compositions parallèles d'unités d'exécution, que nous appellerons **séquentialisations** du programme.
Intuitivement toute trace (ou comportement) extraite de l'arbre d'exécution d'un programme est réalisée par une séquence de transitions datées telle que :

a- chaque transition active une composition parallèle d'au plus n unités d'exécution de processus distincts, (n étant le nombre de processus du programme),

b- les unités d'exécution apparaissent dans le même ordre que dans les traces d'exécution de leurs processus respectifs (notion de "satisfaction" de la trace d'un processus [Las 78]),

c- deux unités d'exécution (de processus distincts), ne peuvent pas être composées dans la même transition si elles utilisent des ports communs,

d- Il existe une relation d'ordre partiel sur les unités d'exécution : pour un port de communication donné tous les entrelacements d'opérations d'accès "syntaxiquement" possibles (i.e. qui "satisfont" les traces d'exécution de leurs processus respectifs) ne sont pas forcément "sémantiquement" réalisables (cf remarque 1). Intuitivement un processus "consommateur" ne pourra pas progresser dans son exécution si le processus "producteur" correspondant n'a pas effectué les opérations de production nécessaires pour qu'une opération de consommation puisse avoir lieu ; inversement une opération de production est toujours activable. Bien entendu cette relation d'ordre partiel sur l'ensemble des unités d'exécution dépend du rythme propre à chaque processus et la valeur des **horloges** des processus est **nécessaire** pour élaborer la prochaine transition, ainsi que nous l'avons vu dans la défintion de la sémantique au chapitre précédent.

Remarque 1 : nous utilisons ici les termes "entrelacements syntaxiquement possibles" et "sémantiquement possibles" pour marquer l'analogie avec la notion de "complémentarité" des opérations de communication de CSP définie dans [AFR 80], **syntaxique** (d'après les textes Pj ?x et Pi ?y des 2 processus communicants respectifs pi et pj) et **sémantique** si ces deux processus sont effectivement synchronisés lors de leur exécution par un rendez-vous au niveau de ces opérations complémentaires.

Ainsi nous proposons comme **modèle d'exécution** des programmes la notion de **séquentialisation** : pour un **rythme d'exécution** donné pour chaque processus, nous définirons par induction sur l'ensemble des unités d'exécution du programme, l'ensemble des séquentialisations de ce programme (cf remarque 2).

Par la suite la définition de l'interprète s'appuiera sur ce modèle d'exécution.

Remarque 2 : la relation d'ordre sur les activations des opérations de communication réalisées par les processus d'un système parallèle ne peut être établie sans connaissance de l'état des ports de communication que si les opérations des types de communication dont ces ports sont des instances ne portent pas sur les valeurs des données communiquées.

Ainsi la construction d'une séquentialisation ne pourra être réalisée de manière statique que pour ce type de programme.(ce qui est le cas le plus fréquent).

La suite de ce chapitre est organisée de la manière suivante :

- le paragraphe II est consacré à la définition de la notion de séquentialisation. Pour parvenir à cette définition on introduit les notions de trace d'exécution d'un processus et d'unité d'exécution (§II.1); on définit ensuite la sémantique de la composition parallèle d'unités d'exécution (§II.2) ; puis on introduit au paragraphe II.3 les assertions d'une unité d'exécution pour formaliser "l'activabilité" d'une unité d'exécution et décrire la relation d'ordre partiel sur les unités d'exécution. Enfin le paragraphe II.4 définit formellement la notion de séquentialisation et le paragraphe II.5 traite un exemple.

- Le paragraphe III présente la description fonctionnelle de l'interprète.

Deux modes de fonctionnement de l'interprète ont été envisagés :

- un mode automatique où une séquentialisation est construite, pour un rythme des processus fixé par l'utilisateur,
- un mode interactif où l'utiiisateur peut intervenir sur le choix des processus à activer afin d'exhiber plusieurs comportements du programme.

Nous ne nous intéressons ici qu'au premier mode de fonctionnement, le second n'ayant pas donné lieu à réalisation .

Le paragraphe III.1 est consacré à la spécification du processus d'interprétation et le paragraphe III.2 décrit fonctionnellement sa mise en œuvre.

II. PASSAGE DE LA SEMANTIQUE A L'INTERPRETATION : NOTION DE SEQUENTIALISATION D'UN PROGRAMME PARALLELE

L'exemple de programme donné [figure 1] servira à illustrer l'ensemble des définitions de ce paragraphe.

Nous commençons par définir les notions de trace d'exécution d'un processus et d'unité d'exécution (*) qui vont permettre de définir l'ensemble des comportements possibles d'un programme.

II.1 Définitions des notions de trace d'exécution, d'unité d'exécution et d'arbre d'exécution.

. Définition 1 :

On appelle **trace d'exécution** (ou calcul) d'un processus l'une des séquences d'actions possibles du processus définies par la sémantique.◊

. Exemple :

Rappelons que aff, cons et prod désignent les actions atomiques associées respectivement aux constructions :=, consommer et produire (cf chapitre II §II.2) ; on notera *s la répétition de la séquence d'actions.

La trace d'exécution du processus P1 donné figure1 est alors la suivante (elle est unique puisque ce processus ne comporte pas de choix indéterministe) :

TP1 = aff(f1,1);aff(n1,1);prod(n1,01);cons(n2,02) ;
* (aff(f1,f1*n1); aff(n1, n1 + 1) ; prod(n1,01) ; cons(n2,02))
n1<n2

. Les règles de transition de type "act" de la sémantique permettent de construire les traces d'exécution d'un processus (cf algorithme de construction en Annexe 1) mais sa fonctionnalité ne peut pas être définie indépendamment de l'activité des autres processus.

---

* La notion d'unité d'exécution avait été introduite dans [Per 85] dans le cadre d'un système de preuve : nous la reprenons ici en la formalisant pour définir un modèle d'exécution des programmes.

Cependant dans la séquence d'exécution d'un processus on peut isoler des sous-séquences dont l'opération sur la mémoire est indépendante de l'exécution des autres processus : il s'agit des séquences qui n'activent que des actions à effet local au processus : actions autres que les opérations de communication qui, elles , opèrent sur les objets partagés du programme que sont les ports de communication. D'où la notion d'**unité d'exécution** dont nous montrerons au paragraphe suivant que l'exécution peut être considérée comme atomique. Ainsi nous pourrons définir la notion de séquentialisation comme une séquence de transitions construite à partir de l'ensemble des unités d'exécution conformément aux règles de la sémantique définie au chapitre précédent.

. Définition 2 :

On appelle **unité d'exécution** un nœud, dans l'arbre de syntaxe abstraite d'un processus élémentaire, de type "séquence d'énoncés" , tel que :

- le premier fils de ce nœud soit de type "opération de communication" (prod ou cons)

- aucun autre descendant de ce nœud ne soit de type "opération de communication".

Un tel nœud situé dans un arbre de type "itération" définit n unités d'exécution potentielles que l'on désignera de manière indicée. ◊

Exemple :

Le programme P ci-dessous réalise une partie du calcul de factorielle(n)

```
processus P : :
     port     01, 02 : égalité (entier) ;

          processus P1 : :
               entrée   n2 : entier via 02 ;
               sortie   n1 : entier  vers 01 ;
               corps
                    var f1 : entier ;
                    début
                              f1:= 1 ; n1:= 1 ;
                              produire (n1) ; consommer (n2) ;
                              répéter n1 < n2   →   f1:= f1*n1 ;
                                                    n1:= n1 + 1 ;
                                                    produire (n1) ;
                                                    consommer (n2)
                              fin répéter

          fin  P1;

          processus P2 : :
               entrée   n1 : entier via 01 ;
               sortie   n2 : entier vers 02 ;
               corps
                    var f2 ,n : entier ;
                    début
                              f2:= 1 ; lire (n) ; n2:= n ;
                              produire (n2) ; consommer (n1) ;
                              répéter n1 < n2   →   f2:= f2*n2 ;
                                                    n2:= n2 - 1 ;
                                                    produire (n2) ;
                                                    consommer (n1)
                              fin répéter

          fin  P2;
          composition
                    P1//P2
fin P
```

Figure 1 : programme (partiel) du calcul de factorielle(n).

Note :

Dans certains cas on assimilera à une unité d'exécution un arbre qui ne contient aucune opération de communication : cas des actions situées au début du corps d'un processus ou d'une itération .
L'ensemble des définitions relatives aux unités d'exécution s'applique à ces cas particuliers et on obtient ainsi une expression plus simple des compositions de traces des processus.

Exemple du programme donné figure 1 :

- Le processus P1 commence par trois unités d'exécution :

```
e11  :  aff(n1,1).aff(f1,1)
e12  : prod(n1, 01)
e13  : cons(n2,02)
```

- Le $i^{ème}$ pas de l'itération du processus définit trois unités d'exécution :

```
e1i1: aff(f1,f1*n1).aff(n1,n1+1)
e1i2 : prod(n1, 01)
e1i3 : cons(n2,02)
```

- De même pour le processus P2 :

```
e21  :  aff(n,?).aff(n2,n-1).aff(f2,1)
e22  : prod(n2,02)
e23  : cons(n1,01)
```

-- le symbole ? désigne le résultat de l'opération lire

- Le $j^{ème}$ pas de l'itération définit les trois unités d'exécution :

```
e2j1 :  aff(f2,f2*n2).aff(n2,  n2-1)
e2j2 : prod(n2, 02)
e2j3 : cons(n1,01)
```

. On notera une unité d'exécution sous la forme uex = op.u où *op* désigne l'opération de communication et *u* l'abstraction de la séquence d'actions locales, qui est éventuellement vide.

Remarques :

1- L'opération sur la mémoire de l'action abstraite u est entièrement définie par l'application des règles SEQ1-mod et SEQ2-mod

(cf chapitre II §II.5) puisque cette séquence ne comporte que des actions à effet local sur la mémoire du processus et l'activation de cette séquence simultanément à d'autres actions ne change pas son opération sur la mémoire, ce que l'on utilisera au § II.2.

2- La notion d'unité d'exécution est similaire à celle de section parenthèsée d'un processus CSP introduite dans [AFR 80] pour définir la coopération de preuves des processus.

3- La démarche qui consiste à abstraire une séquence d'actions par son opération sur la mémoire correspond à la notion de "critère d'observation" introduite par [BOU 85]. Cette démarche est aussi utilisée pour définir la sémantique du comportement instantané d'un programme ESTEREL [Bof 87] .

. Définition 3 :

On appelle arbre d'exécution d'un processus, l'ensemble des traces possibles d'exécution du processus , construit à partir de l'ensembles des unités d'exécution du processus. ◊

- Cet ensemble est exprimé sous la forme d'une expression régulière sur l'alphabet des noms d'unités d'exécution , du type "expression de chemin" [CH 74] [LTS 79].
Les opérateurs utilisés sont les suivants :

";" réalise la trace formée de la séquence de deux unités d'exécution ou de deux traces.

"*(..)" dénote la répétition finie d'une trace d'exécution ; l'utilisation du symbole "+" à la place de "*" signifie que la trace est présente au moins une fois.

"," indique le choix non-déterministe mais exclusif d'une trace dans un ensemble fini de traces d'exécution.

"[...]" : dénote l'éventualité d'une trace d'exécution (cas du choix alternatif).
◊

- L'algorithme de construction de cet arbre est donné en Annexe 1.

Remarque :
Toute trace d'exécution d'un processus commence par une unité d'exécution particulière , appelée unité initiale et notée e0, et qui

consiste en l'initialisation des clauses "entrée" et "sortie" qui réalisent les associations donnée communiquée ↔ port.

Exemples :

. L'arbre d'exécution du processus "P1" (figure 1), ici réduit à une seule trace, est :

$$AP1 = e10 ; e11 ; e12 ; e13 ; +_i (e1i1 ; e1i2 ; e1i3)$$

. De même pour le processus "P2" :

$$AP2 = e20 ; e21 ; e22 ; e23 ; +_j (e1j1 ; e1j2 ; e1j3)$$

On peut représenter graphiquement le début de ces deux traces ainsi :

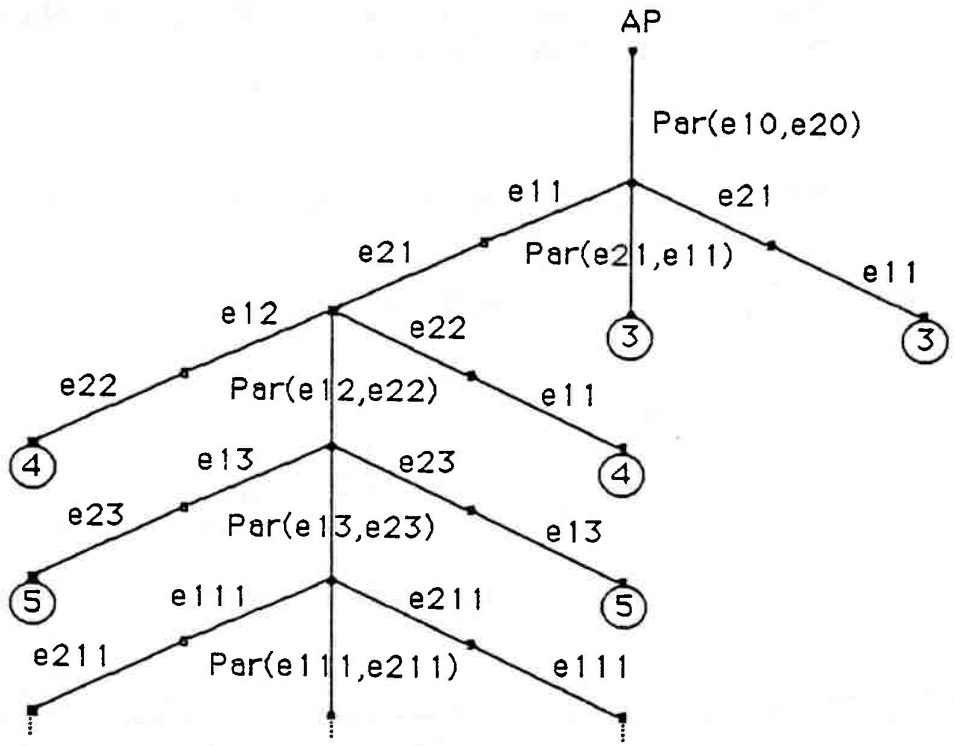
AP1 : . e10 . e11 . e12 . e13 . e111 . e112 . e113 . e121 .
AP2 : . e20 . e21 . e22 . e23 . e211 . e212 . e213 . e221 .

. On peut en déduire intuitivement le début de l'arbre d'exécution du programme P (figure 1) :

```
AP =   ( Par(e10, e20)) ;                                          (1)
       ((e11; e21) , (e21 ; e11) , Par(e11; e21));                 (2)
       ((e1; e2) , (e2 ; e1) , Par(e1; e2));                       (3)
       ((e13; e23) , (e23 ; e13) , Par(e13; e23));                 (4)
       ((e111; e2111) , (e211 ; e111) , Par(e111; e211)); ..       (5)
```

Rappelons que par principe les premières unités respectives des processus ,notées ei0 , sont activées simultanement.

Cet arbre d'exécution peut être schématisé ainsi :



où on symbolise par (i) l'arbre associé à la séquence d'exécution numérotée (i) dans la trace AP.

Remarques sur l'exemple :

1- Aucun autre début d'exécution n'est possible pour le programme P : en effet la 1ère opération de production sur le port 02 par le processus P2 (unité e22) précède obligatoirement la 1ère opération de consommation sur ce même port par le processus P1 (unité e13), quelles que soient les durées des actions des processus P1 et P2 (idem pour les opérations sur le port 01 contenues dans la trace n° 4).

Ainsi chaque trace dans l'arbre correspondra à une combinaison des rythmes possibles des processus.

Notons que cet exemple est particulier dans la mesure où les exécutions des processus P1 et P2 sont systématiquement entrelacées par les communications : cette situation est liée au type de communication choisi pour les ports 01 et 02 : "égalité". Le type de communication "aléatoire" produirait une exécution plus asynchrone (cf § II.5).

2- La séquence (u;v), si u et v sont des unités d'exécution de deux processus distincts est équivalente à la séquence (Par(u,ε);Par(ε,v)) où ε désigne l'unité d'exécution vide de durée nulle (équivalente à l'action nil).

Ainsi toute trace d'exécution d'un programme peut s'exprimer sous la forme d'une séquence d'activations parallèles d'unités d'exécution.(cf chap II §II.7)

Par exemple les traces

T1 = Par(e10,e20) ; Par(ε,e21) ; Par(e11,ε) ; Par(ε,e22) ; Par(e12,ε) et

T2 = Par(e10,e20) ; Par(e11,e21) ; Par(ε, e22) ; Par(e12,ε) ;

sont des débuts d'exécution du programme P ci-dessus. ◊

Nous avons montré un exemple de construction de l'arbre d'exécution d'un programme, sans faire intervenir explicitement le rythme des processus mais en exhibant la relation d'ordre sur les unités d'exécution.

Avant de définir formellement la notion de séquentialisation, commençons par déterminer les cas possibles de composition parallèle d'unités d'exécution et leur signification.

II.2. La composition parallèle d' unités d'exécution.

Rappelons qu'une unité d'exécution est un schéma abstrait de la forme : uex = op.a_1. .... .a_k, où les a_i sont des actions atomiques différentes de "prod" ou "cons" et l'action op, si elle existe, est une action "prod" ou "cons".

Le fait que les unités d'exécution commencent par une opération de communication est fondamental pour définir la composition parallèle d'unités d'exécution , comme nous allons le voir.

En assimilant les unités d'exécution à des actions atomiques, on peut utiliser les règles de la sémantique pour définir la signification de leur composition parallèle, c'est à dire leur activation simultanée.

D'après la règle "Par-mod" (chap II §II.6), l'action Par(uex_1,...,uex_n) n'a de sens que si les opérations de communication respectives des

unités $uex_1,...,uex_n$, obligatoirement simultanées dans ce cas, n'opèrent pas sur le même port : on étend donc naturellement le prédicat "exclusif" défini sur les actions atomiques aux unités d'exécution.

Ainsi lorsqu'à un instant donné de l'exécution d'un programme (*), deux unités d'exécution uex_1 et uex_2 sont activables , trois transitions sont possibles : (cf chap II §II.6)

(1) ces unités ne sont pas exclusives : elles sont activées simultanément; l'action $Par(uex_1, uex_2)$ est engagée (règle R1-exec)

(2) elles sont exclusives et réalisent toutes deux le même type d'opération de communication : elles sont séquentialisées selon un ordre arbitraire (règle R3-exec) ; les deux actions possibles sont $Par(uex_1,\varepsilon)$ et $Par(\varepsilon,uex_2)$

(3) l'une , uex_1, contient une opération prod et la seconde, uex_2 , une opération cons sur le même port : elles doivent être activées consécutivement mais de manière atomique (règle R2-exec) .
C'est à cet effet que nous avons introduit l'action atomique "prod-cons" au chapitre précédent. Par conséquent, si $uex_1 = prod_1.u_1$ et $uex_2 = cons_2.u_2$, la transition à réaliser est celle qui met en œuvre l'action $prod_1\text{-}cons_2$ suivie des actions u_1 et u_2 .
Pour que ceci soit réalisé , au couple (uex_1, uex_2) nous substituons les unités $uex_1' = prod_1\text{-}cons_2.u_1$ et $uex_2' = u_2$ et l'action activée sera $Par(uex_1',\varepsilon)$. L'unité uex_2' sera obligatoirement activée par la transition suivante puisqu'elle ne pourra être en conflit avec aucune autre unité d'exécution. Notons que les durées des unités uex_1 et uex_2 ne sont pas affectées par cette modification puisque les opérations de communication sont instantanées.

Pour être rigoureux nous devrions dans chacun des cas prouver les comportements décrits par l'application des règles de la sémantique : nous ne le ferons pas pour ne pas alourdir cette présentation.

Nous nous attachons maintenant à décrire formellement l'activabilité d'une unité d'exécution , ce qui va permettre de définir le modèle d'exécution des programmes que nous appellons séquentialisation.

(*) composé de deux processus pour simplifier

II.3- Activabilité d'une unité d'exécution : les assertions d'une unité d'exécution.

Une unité d'exécution est caractérisée par une pré-assertion et une post-assertion qui définissent sa situation vis-à-vis de l'ensemble des données communiquées du programme : la pré-assertion indique si l'unité d'exécution est **activable**, la post-assertion reflète l'effet sur les ports de communication de l'activation de l'unité d'exécution.

. Soient tc1, ... , tcr les *r* ports de communication définis par le système parallèle.

. Définition 4 :

On appelle pré-assertion d'une unité d'exécution *ei*, i ≠ 0, le r-uplet de variables à valeurs boolénnes :

pi = (préi1, préi2, ..., préir)

où pour tout j de i à r, préij est défini par :

(i) si j est tel qu'il existe un nœud de type cons(x,tcj) dans ei, où le port désigné par tcj est de type TCj, alors

préij = TCj.pré_cons(tcj)

(ii) pour toutes les autres valeurs de j

préij = vrai ◊

Exemples :

Les pré-assertions respectives des trois premières unités d'exécution du processus P1 (figure 1) sont : (cf note)

p11 = (vrai,vrai)
p12 = (vrai,vrai)
p13 = (vrai, non vide?(A2))

Les pré-assertions des unités de l'itération du processus P1 sont :

p1i1 = (vrai,vrai)
p1i2 = (vrai,vrai)
p1i3 = (vrai, non vide?(A2))

Les pré-assertions des trois premières unités du processus P2 sont :

p21 = (vrai,vrai)

p22 = (vrai,vrai)
p23 = (non vide?(A1), vrai)

Les pré-assertions des unités de l'itération du procesus P2 sont:
p1j1 = (vrai,vrai)
p1j2 = (vrai,vrai)
p1j3 = (non vide?(A1),vrai)

Note :
On note A1 et A2 les ensembles consommables associés aux ports 01 et 02 du programme ci-dessus.

. Définition 5 :

On appelle **post-assertion** d'une unité d'exécution ei, i ≠ 0, le r-uplet de variables à valeurs booléennes :
qi = (posti1, posti2, ... , postir)

où pour tout j de 1 à r , postij est défini par :

(i) si j est tel qu'il existe un nœud de type cons(x,tcj) dans ei, avec x donnée communiquée par le port tcj, de type TCj, alors
postij = TCj.*pré-cons*(TCj.*post_cons*(tcj))

(ii) si j est tel qu'il existe un nœud de type prod(x,tcj) dans ei, avec x donnée communiquée via le port tcj, alors
postij = TC.*pré-cons*(PRODUIRE(tcj,val(x)))

(iii) pour toutes les autres valeurs de j, la valeur, arbitraire, de postij est faux.

Exemples :

Pour certaines des unités ci-dessus, les post-assertions sont :

q1i2 = (non vide?(A1), faux) -- production sur 01
q1i3 = (faux, vrai) -- consommation sur 02
q2j2 = (faux, non vide?(A2)) -- production sur 02
q2j3 = (vrai, faux) -- consommation sur 01

. Définition 6 :

Les assertions de l'unité initiale *e0* d'un processus sont :

p0 = (vrai , ... , vrai)
q0 = (post01, ... , post0r)

où pour tout j de 1 à r, post0j = TCj.*pré-cons* (INIT)

où INIT est l'objet "initial" d'un type de communication égal au triplet <(ω),table-vide,(ω)> (cf chapitre I §III) ◊

. Dans l'exemple ci-dessus, q0 = (non vide?(A1), non vide?(A2)) où A1 et A2 ont la valeur "table-vide" donc q0 = (faux,faux).

. **Définition 7 :**

Une unité d'exécution *uex* est **activable**, à un point donné dans une trace d'exécution du processus qui la définit, si et seulement si :

- le r-uplet qui définit sa pré-assertion a la valeur (vrai, vrai, ...,vrai)

- l'unité "satisfait" la trace d'exécution du processus c'est à dire:
  soit *s* la séquence d'unités extraite de l'arbre d'exécution du processus et déjà activée au moment de l'évaluation de la pré-assertion de *uex* ,
  alors la séquence *s ; uex* est le préfixe de l'une des séquences d'exécution possibles du processus . ◊

Il reste maintenant à définir les compositions possibles des différentes unités d'exécution d'un système parallèle, que nous appellerons **séquentialisations** du système.

II.4 Définition de la notion de séquentialisation d'un système parallèle.

Considérons l'ensemble E = {e10,e11,...,e1k,...,en0,en1,...,enm} composé des unités d'exécution des n processus d'un système parallèle.

Intuitivement l'exécution de ce programme est modélisée par la suite datée des activations de l'ensemble des unités de E.

A un instant donné de l'exécution d'un programme parallèle un certain nombre de processus sont dans un état parfait : ce sont les processus qui ont terminé l'action précédemment engagée; leur horloge

a la valeur minimum de l'ensemble des horloges du système (cf chap II §II.6).
Leurs prochaines unités à activer, si elles sont activables , pourront être activées simutanément ainsi que nous l'avons vu au paragraphe II.2 à la condition que ces unités n'opèrent pas concurrement sur les mêmes ports de communication .

Notons que pour un rythme donné de chacun des processus, la durée des unités d'exécution est connue et par conséquent leurs dates d'activation dans une séquentialisation est directement élaborable, comme nous allons le définir ci-dessous.

Ainsi une séquentialisation est une suite de couples (U_i,H_i) où l'on note:

U_i une action composée de type $Par(uex_1,...,uex_n)$ telle que si le processus P_k est dans un état parfait et activable uex_k désigne la prochaine unité activable de ce processus ,et si le processus P_k n'a pas terminé l'action précédente ou si sa prochaine unité n'est pas activable uex_k désigne l'action nil , ce que l'on note $uex_k = \varepsilon$

.

H_i le produit cartésien des horloges des n processus défini par l'activation de l'action U_i : chaque horloge indique la date de fin de l'action engagée par le processus ou de l'action précédente si elle n'était pas terminée. L'activation de l'action U_i a lieu à la date égale à l'horloge minimum de H_{i-1} sauf si les processus d'horloge minimum sont bloqués en attente d'une consommation.
On notera $< h_1^i, h_2^i,..., h_n^i>$ le n-uplet H_i et $min(H_i)$ la valeur minimum de cet ensemble.

Nous définissons par induction une séquentialisation d'un système parallèle ; cette définition fait appel à la notion de "projection" et à la relation "inférieure" dont les définitions seront données immédiatement après.

. Définition 8 :

On appelle **séquentialisation** et on note $(<U_0,H_0>;<U_1,H_1>;...;<U_i,H_i>)$ la séquence d'exécution datée des actions composées $U_0,U_1,...,U_i$ telle que :

-($<U_0,H_0>$) est la séquentialisation définie par l'activation parallèle des unités de l'ensemble {e10,e20,...,en0} à la date initiale H_0= $< 0,...,0>$

- $\forall j$, $1 \leq j \leq i$, U_j définit un ensemble d'au plus n unités de E, activables (cf définition 7) et non exclusives tel que :
($<U_0,H_0>;<U_1,H_1>;...;<U_{j-1},H_{j-1}>$) est une séquentialisation ,
($<U_0,H_0>;<U_1,H_1>;...;<U_{j-1},H_{j-1}>$) est "inférieure" à $<U_j,H_j>$

- le choix des unités d'exécution de l'ensemble U_j lorsque plusieurs sont activables et exclusives à la date min(H_i) est celui défini au paragraphe II.2

- pour tout processus Pk du système parallèle la "projection" de la suite d'actions ($U_0;U_1;...;U_j$) sur Pk "satisfait" l'une des traces d'exécution de ce processus. ◊

. Définition 8-a :

La projection d'une séquence ($U_0;U_1;...;U_j$) sur le processus Pk est la sous-séquence d'exécution extraite, uniquement composée des unités d'exécution du processus Pk, définie par $\Pi_k(U_0;U_1;...;U_j)$.

La fonction Π_k de profil $(A_1 \cup \{\varepsilon\} \times ... \times A_n \cup \{\varepsilon\})^* \longrightarrow A_k$
(où A_k désigne l'ensemble des unités du processus Pk)

est définie par :

$\Pi_k(Par(uex_1,...,uex_n))$ = uex_k si $uex_k \neq \varepsilon$
= ^ ,le mot vide ,sinon

$\Pi_k(U_0;U_1;...;U_i) = \Pi_k(U_0;U_1;...;U_{i-1})\ \Pi_k(U_i)$ ◊

. Définition 8-b :

Pour j>0, une séquentialisation ($<U_0,H_0>;<U_1,H_1>;...;<U_{j-1},H_{j-1}>$) est **inférieure** au couple $<U_j,H_j>$) si et seulement si :
$qq_{j-1} \Rightarrow \tilde{p}_j$ et $H_j = < h_1^j, h_2^j, ..., h_n^j>$

où qq_{j-1} est la **post-condition** de la séquentialisation ($U_0;U_1;...;U_{j-1}$) et $\tilde{p}_j$ la **pré-assertion** de l'action U_j ,

H_j est le n-uplet des horloges des processus résultant de l'action U_j :

si U_j est l'action Par(uex_1,...,uex_n) , où uex_k désigne soit une unité du processus Pk(cas $uex_k \neq \varepsilon$) soit l'action nil (cas $uex_k = \varepsilon$), la valeur de l'horloge h_k^j est définie par :

(1): cas $uex_k \neq \varepsilon$ et $h_k^{j-1} = \min(H_{j-1})$,

-- l'unité uex_k est activée à la date h_k^{j-1} par cette
-- transition: elle se terminera à la date $h_k^{j-1} + \delta(uex_k)$, où
-- $\delta(u)$ désigne la durée de l'unité u.
$\Rightarrow h_k^j = h_k^{j-1} + \delta(uex_k)$

(2): cas $uex_k = \varepsilon$ et $h_k^{j-1} > \min(H_{j-1})$,

- - l'horloge du processus Pk à la date d'activation de U_j est
-- en avance par rapport aux horloges minimum : ce
- - processus n'a pas terminé l'action précédente et celle-
- - ci continue pendant la durée de l'action Uj
$\Rightarrow h_k^j = h_k^{j-1}$

(3): cas $uex_k = \varepsilon$ et $h_k^{j-1} = \min(H_{j-1})$,

-- soit il n'y a pas d'unité activable pour le processus
- - Pk à la date h_k^{j-1} , soit il existe une unité activable qui
- - a été écartée parce qu'en conflit avec une autre unité :
-- l'horloge du processus Pk doit progresser jusqu'à la date
-- du processus dont l'unité est activée.
$\Rightarrow h_k^j = h_m^{j-1}$ avec $m = \min \{ h_r^{j-1} / r = 1..n$ et $uex_r \neq \varepsilon \}$

(4): cas $uex_k \neq \varepsilon$ et $h_k^{j-1} > \min(H_{j-1})$,

- - cas où l'ensemble des processus d'horloge minimum sont
- - bloqués dans l'attente d'une consommation : les
- - processus d'horloge immédiatement supérieure dont une
- - unité est activable sont activés
$\Rightarrow h_k^j = h_k^{j-1} + \delta(uex_k)$

Ces quatre cas correpondent à l'application respective des règles R1-exec pour le cas (1), R5-exec dans le cas (2), R2 ou R3-exec dans le cas (3) et R6-exec dans le cas(4) (cf chapitre II §II.6).

. Définition 8-c :

(b1,b2,...,br) $\Rightarrow$ (b'1,b'2,...,b'r) si et seulement si $\forall j$, $1 \leq j \leq r$, bj$\Rightarrow$ b'j.

. Définition 9 :

9-a) La **pré-assertion** $\tilde{p}j$ d'un ensemble de n unités non exclusives Uj est la conjonction des pré-assertions de ses unités composantes:

$$\tilde{p}j = \bigvee_{k=1,n} pk$$

où pk, pour k de 1 à n, désigne la pré-assertion de la $k^{ième}$ unité d'exécution de l'ensemble U_j : un ensemble d'unités est activable si chacun de ses composantes l'est . ◊

Exemple :

Supposons que les unités e1i2 et e2j2 du programme ci-dessus soient activables simultanément :
e1i2 = cons(n2,02) et e2j2 = cons(n1,01)

Leurs pré-assertions repectives sont :

p1i2 = (vrai, non vide?(A2)) et p2j2 = (non vide?(A1), vrai).

L'action U = Par(e1i2,e2j2) a pour pré-assertion :
$\tilde{p}$ = p1i2 ∧ p2j2 = (non vide?(A1), non vide?(A2))

9-b) La **post-assertion** $\tilde{q}j$ d'un ensemble d'unités non exclusives Uj est l'"union" des post-assertions de ses unités composantes :

$$\tilde{q}j = \bigvee_{k=1,n} qk$$

où qk est la post-assertion de la $k^{ième}$ unité d'exécution de l'ensemble U_j . ◊

L'effet sur les ports de communication de l'exécution d'un ensemble d'unités non exclusives est la séquence, dans un ordre quelconque, de l'exécution de ses unités composantes, chacune opérant sur un port différent ; d'où la disjonction entre les booléens, attachés à un même port, des post-assertions de chacune des unités de l'ensemble.

Exemple :

La post-assertion de l'action U = Par(e1i2,e2j2) ci-dessus est :
$\tilde{q}$ = q1i2 ∨ q2j2
= (faux,vrai) ∨ (vrai,faux) = (vrai,vrai)

. **Définition 10 :**

La **post-condition** qqj d'une séquentialisation ($<U_0,H_0>;...;<U_j,H_j>$) est définie par :

- qq0 = q0

- - pour tout j>0, qqj est obtenue à partir de la post-assertion qj de U_j en substituant à toutes ses composantes égales à la valeur, arbitraire, faux leur valeur dans qqj-1. ◊

-- qqj représente le résultat de la séquence d'exécution ($U_0;U_1;...;U_j$) sur l'ensemble des ports de communication du système. On a vu que les "booléens d'état" des ports de communication non concernés par une unité d'exécution prenaient la valeur arbitraire faux dans la post-assertion de l'unité : il en est de même dans la post-assertion d'un ensemble d'unités; dans la post-condition d'une séquentialisation la valeur de ces booléens est alors celle résultant de la séquentialisation immédiatement inférieure.

. **Définition 11:**

On appelle **séquentialisation** d'un programme parallèle P, une séquence d'exécution notée ($<U_0,H_0>;<U_1,H_1>;...;<u_j,H_j>$) telle que chacune des unités d'exécution du programme apparaisse une fois, et une seule, dans l'un des ensembles U_j, pour j de 1 à i.

Remarques :

Toutes les séquentialisations d'un programme n'ont pas forcément la même opération sur la mémoire. En effet le principe d'exclusion mutuelle d'accès aux ports de communication, impose un choix entre les unités activables à un point donné de la construction d'une séquentialisation, lorsque certaines sont exclusives.
Par exemple l'opération sur un port de communication de type "égalité" de la valeur 1 suivie de la production de la valeur 2 n'aura pas la même conséquence que les productions dans l'ordre inverse (cas de *convergence* des processus producteurs). Tandis que pour un port de type "plus petite valeur" , par exemple, l'effet sur les consommations ultérieures serait le même.

Autre conséquence du principe d'exclusion mutuelle : lorsqu'une unité activable n'est pas retenue pour être activée elle peut devenir non activable suite à l'exécution de l'unité, concurrente pour le même port, choisie .

Par exemple ,dans le cas d'un schéma de type *divergence,* deux processus peuvent être prêts simultanément à consommer sue le même port : si ce port est de type "égalité" et ne contient qu'une valeur consommable, l'activation du consommateur choisi rend non activable le second candidat qui sera alors placé en position d' *attente* de consommation(cf §III.2).

II.5 Exemple de définition d'une séquentialisation

Définissons par induction les séquentialisations du programme P donné figure 1, dans l'hypothèse où les 2 processus P1 et P2 évoluent au même rythme et où chaque opération (affectation, lire) nécessite une unité de temps alors que l'initialisation des ports de communication, les opérations produire et consommer et l'évaluation des expressions sont instantanées.

Afin de faciliter la lecture du processus de définition des séquentialisations de ce programme, nous proposons [figure 2] une schématisation de l'exécution réalisée, sous les hypothèses énoncées.

unité activée	P1	hor-loge	unité activée	P2
e11	f1:=1, n1:=1	0	e21	f2:=1, lire(n),n2:=n
e12	produire(n1=1) attente de consommation	2		/* valeur lue : 4 */
		3	e22	produire(n2=4);
e13	consommer(n2=4);	3	e23	consommer(n1=1);
e111	f1:=1;n2:=2	3	e211	f2:=4 ;n2:=3;
e112	produire(n1=1)	5	e212	produire(n2=3);
e113	consommer(n2=3)	5	e213	consommer(n1=2);
e121	f1:=2;n1:=3;	5	e221	f2:=12;n2:=2;
e122	produire(n1=3)	7	e222	produire(n2=2);
e123	consommer(n2=2);	7	e223	consommer(n1=3);
	arrêt sur n1 > n2	7		arrêt sur n1 > n2

Figure 2 : Schéma d'exécution du programme (partiel) de calcul de factorielle(n) pour n = 4.

1) dans le système P ci-dessus, les "premières" unités à activer sont e10 et e20, puis respectivement pour le processus P1 et le processus P2:

e11 : aff(n1,1).aff(f1,1)
e12 : prod(n1,01)
e13 : cons(n2,02)

e21 : aff(f2,1) . aff(n,?) . aff(n2,n)
e22 : prod(n2,02)
e23 : cons(n1,01)

Montrons que $(\langle U_0,H_0\rangle;\langle U_1,H_1\rangle)$ avec U_1 = Par(e11,e12) et $H_1=\langle 2,3\rangle$ est une séquentialisation :

. qq0 = q0 = (faux,faux) : résultat de l'initialisation des 2 ports 01 et 02

. $\tilde{p}1 = p11 \wedge p21$ = (vrai,vrai) donc qq0 $\Rightarrow \tilde{p}1$ et comme $H = H_0 + \delta(e11), \delta(e21))$ la relation ($<U_0,H_0>$) "inférieure" ($<U_1,H_1>$) est vérifiée.

. Comme les séquences extraites (e10;e11) et (e20 ; e21) satisfont les traces d'exécution des processus P1 et P2, ($<U_0,H_0>$; $<U_1,H_1>$) est bien une séquentialisation

Remarque :

Cette séquentialisation est la seule possible pour le début de l'exécution du système, car aucune autre unité ne répond aux critères :

- pré-assertions des unités à "vrai"
- "respect" de la trace d'exécution des processus.

Cette séquentialisation définit comme date potentielle d'activation de l'action suivante, la date 2.

2) Montrons que ($<U_0,H_0>$; $<U_1,H_1>$; $<U_2,H_2>$) avec U_2 = Par(e12,ε) et H_2 = <2,3> est une séquentialisation

. qq1 = (faux,faux) est la post-condition la séquentialisation ($<U_0,H_0>$; $<U_1,H_1>$) telle que H_1=<2,3>

. $\tilde{p}2$, pré-assertion de U_2, a pour valeur (vrai,vrai) donc qq1 $\Rightarrow \tilde{p}2$.

. le processus P2 est "en avance" sur le processus P1 donc seule la prochaine action de P1 est activée. Les cas respectifs 1 et 2 de la définition 8-b définissent comme prochaines valeurs des horloges H_2=<2,3>. La relation "inférieure" est donc vérifiée.

. comme les séquences extraites (e10 e11 e12) et (e20 e21) respectent les traces d'exécution des processus P1 et P2 ,($<U_0,H_0>$; $<U_1,H_1>$; $<U_2,H_2>$) est une séquentialisation, que nous notons S_2.

. la post-condition de cette séquentialisation est
qq2 = $\tilde{q}2$ = (non vide?(A1), vrai))

3) Montrons que S_3=(S_2;$<U_3,H_3>$) avec U_3=Par(ε,e22) et H_3=<3,3> est une séquentialisation.

. $\tilde{p}3$, pré-assertion de U_3 , a pour valeur :(vrai,vrai) donc qq2 $\Rightarrow \tilde{p}3$.

. on est dans le cas où le processus d'horloge minimum P1, ne peut pas progresser car bloqué devant une opération de consommation: son horloge progresse jusqu'à la date du processus activable (date 3) et l'unité e22 du processus P2 qui est activable est activée ; on obtient H_3=<3,3> d'après les cas (3) et (4) prévus dans la définition 8-b et la relation inférieure est vérifiée.

. comme les projections respectives (e10 e11 e12) et (e20 e21 e22) respectent les traces d'exécution des processus P1 et P2, S_3 est bien une séquentialisation de post-condition.
qq3 = $\tilde{q}$3 = (non vide?(A1), non vide?(A2))

4) Montrons que S_4=(S_3;<U_4,H_4>) avec U_4=Par(e13,e23) et H_4=<3,3> est une séquentialisation.

. $\tilde{p}$4= (non vide?(A1), non vide?(A2)) donc qq3 $\Rightarrow$ $\tilde{p}$4.

. chacune des unités e13 et e23 sont donc activables et comme elles ne sont pas exclusives, elles sont activées simultanément ; le cas 1 correpondant de la définition 8-b définit H_4=H_3+<0,0>=<3,3> donc la relation "inférieure" est vérifiée.

. Comme ces unités "satisfont" la trace des processus, S_4 est bien une séquentialisation de post-condition qq4 = $\tilde{q}$4 = (vrai, vrai)

5) Ce point de l'exécution atteint, les unités restant à activer sont celles des itérations respectives de P1 et P2 : e1i1, puis e1i2 et e1i3 pour P1 et e1j1, puis e2j2 et e2j3 pour P2, répétées tant que (n1 < n2).

Les séquences d'actions de ces unités d'exécution sont :

e1i1 : aff(f1,f1*n1).aff(n1,n1+1)
e1i2 : prod(n1,01)
e1i3 : cons(n2,02)

e2j1 : aff(f2,f2*n2).aff(n2,n2-1)
e2j2 : prod(n2,02)
e2j3 : cons(n1,01)

Comme les ports de communication 01 et 02, utilisés pour communiquer les valeurs des données n1 et n2, sont de type "egalité", les pré-assertions de ces unités sont :

p1i1 = (vrai, vrai) p1i2 = (vrai, vrai) p1i3 = (vrai, non vide?(A2))

p2j1 = (vrai, vrai) p2j2 = (vrai, vrai) p2j3 = (non vide?(A1),vrai)

a) les prochaines unités qui satisfont les traces respectives des processus P1 et P2 sont les unités e111 et e211 : ces unités sont activables et forment l'action U_5 = Par(e111,e211) de pré-assertion $\tilde{p}5$=(vrai,vrai).
Comme qq4 $\Rightarrow \tilde{p}5$, la séquence $S_5 = S_4 ; <U_5,H_5>$ avec $H_5=<5,5>$ est une séquentialisation de post-condition qq5 = (vrai,vrai).

b) La post-assertion de la séquentialisation S_5 est qq5=(vrai,vrai) donc les seules unités activables sont e112 et e212, qui forment l'action U6 de pré-assertion $\tilde{p}6$=(vrai,vrai).

Comme la séquence $S_6 = (S_5 ;<U_6,H_6>)$ avec $H_6=<5,5>$ satisfait les traces d'exécution des processus P1 et P2 et vérifie la relation "inférieure" définie en 8-b, S_6 est bien une séquentialisation de post-condition qq6 = (non vide?(A1), non vide?(A2))

c) Les unités e113 et e213 forment l'action U_7 = Par(e113,e213) de pré-assertion $\tilde{p}7$= (non vide? (A1), non vide? (A2)) sont activables puisque qq6 $\Rightarrow \tilde{p}7$.
Ainsi la séquence $S_7 = (S_6;<U_7,H_7>)$ avec $H_7=H_6=<5,5>$ est une séquentialisation de post-condition qq7=(vrai,vrai) qui définit l'exécution du programme P jusqu'à la fin de la première itération des processus P1 et P2.

6) On constate que la post-condition <vrai,vrai> est invariante de l'itération combinée de P1 et P2 : on montrerait par induction sur i que la séquence S égale à

$$S_4 ; \overset{*}{_{i}} (<U_{i1},H_{i1}> ; <U_{i2},H_{i2}> ; <U_{i3},H_{i3}>)$$

est une séquentialisation du programme P ci-dessus, dans l'hypothèse d'actions toutes de durée égale à l'unité,

avec U_{i1} = Par(e1i1,e2i1)
U_{i2} = Par(e1i2,e2i2)
U_{i3} = Par(e1i3,e2i3)

et $H_{i1} = H_{i-1\ 1} + <2,2>$,
$H_{i2} = H_{i3} = H_{i1}$ si i >1
et = $H_{11} = <3,3>$

. On peut montrer, rapidement à l'aide d'un contre-exemple que c'est la seule séquentialisation possible, pour ce rythme des processus.

7) On montrerait de même que, quels que soient les rythmes des processus P1 et P2 (*), toute séquentialisation du programme P est de la forme :

$$S_0 ; \ *_i (<U_{i1},H_{i1}> ; S_i; <U_{i3},H_{i3}>)$$

où . S0 est une séquentialisation qui réalise l'activation des 3 premières unités respectives des processus P1 et P2 selon un entrelacement qui dépend des durées des unités e11 et e12 (notée $\delta 1$ et $\delta 2$) et tel que :

- les unités e12 et e22 (prod(n1,01) et prod(n2,02)) sont activées aux dates respectives $\delta 1$ et $\delta 2$

- la date de l'activation de e13 et e23, forcément simultanée, est égale à $\max(\delta 1,\delta 2)$ puisque chacune de ces consommations ne peut avoir lieu qu'après la production la plus tardive.
Notons t la date $\max(\delta 1,\delta 2)$

. U_{i1} est l'action définie ci-dessus : Par(e1i1,e2i1).
Soit di la date d'activation de cette action. En particulier d1,date de la première itération, est égale à t. L'action U_{i1} définit le n-uplet d'horloges $H_{i1} = < di + \delta'1, di + \delta'2>$
où $\delta'1$ et $\delta'2$ sont les durées des unités e1i1 et e2i1.

. S_i est une séquentialisation qui commence à la date $\min(di+\delta'1, di + \delta'2)$ et qui réalise les opérations **produire**(n1) (unité e1i2) et **produire**(n2) (unité e2i2) du $i^{ème}$ pas de l'itération des processus respectifs P1 et P2, soit en séquence soit simultanément selon la valeur des durées $\delta'1$ et $\delta'2$.

. U_{i3} est l'action Par(e1i3,s2i3) qui réalise les opérations **consommer**(n1) et **consommer**(n2) du $i^{ème}$ pas de l'itération à la date $d'i = \max(di+\delta'1, di+\delta'2)$. Ainsi la date de début de l'itération suivante est $di+1 = d'i$ et pour tout $i \geq 1$,

$$di=t + (i-1) \max(\delta'1, \delta'2)$$

avec $t = \max(\delta 1,\delta 2)$.

Ceci met fin à l'exemple et à ce paragraphe.

---

* Toujours sous l'hypothèse d'opérations de consommation instantanées.

III. DESCRIPTION FONCTIONNELLE DE L'INTERPRETE

Nous avons défini l'interprétation des programmes par la simulation, sous certaines hypothèses de temps, de leur éxécution parallèle et dans ce but avons introduit la notion de séquentialisation d'un programme parallèle.

Nous décomposons la fonction d'interprétation en trois phases, ainsi que l'illustre le schéma donné figure 3 :

- à partir d'un programme, décrit dans le langage, et des rythmes donnés des processus une des séquentialisations possibles est construite; cette étape est appelée **COMPIL** par analogie avec une phase de compilation produisant du code intermédiaire; cependant la construction d'une séquentialisation ne peut être réalisée de manière statique que pour des programmes où les opérations des types de communication utilisés ne portent pas sur les valeurs échangées

- la seconde phase réalise l'interprétation de la séquentialisation construite pour un certain jeu de données sur une machine virtuelle parallèle (MVP).
Un cycle de cette "machine" réalise l'interprétation d'un ensemble Ui d'unités d'exécution:

. à chaque unité d'exécution ue_{ij} correspond une action, notée A_{ij}, de la machine MVP,

. la fonction INTERP réalise la composition parallèle des actions A_{ij} en produisant une séquence équivalente (cf §II.2)

- enfin au niveau le plus interne, chaque action A_{ij} est interprétée pour les données du programme sur la machine séquentielle abstraite MS par une séquence des actions élémentaires qui composent l'unité d'éxécution ue_{ij}.

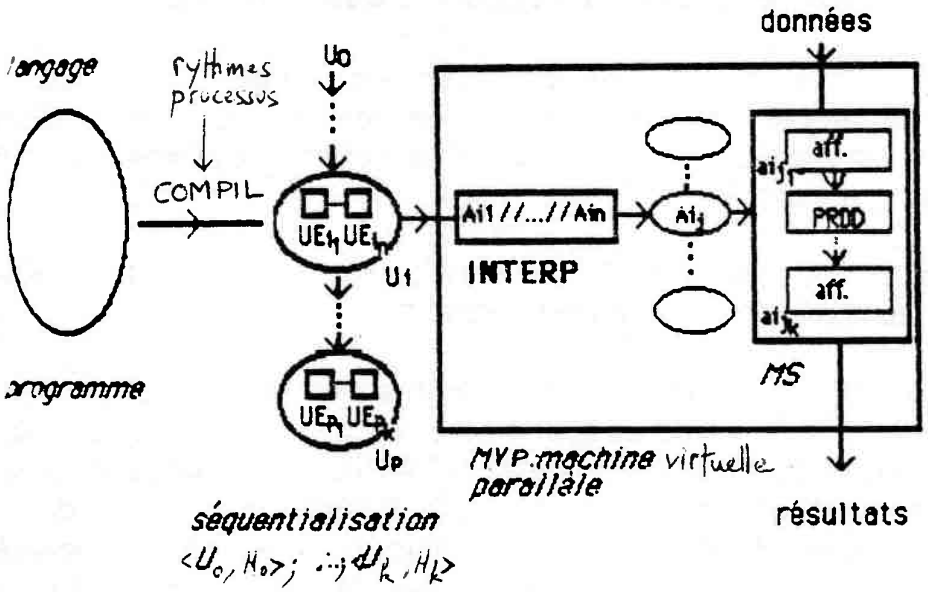


Figure 3 : Le processus d'interprétation

La présentation des spécifications de l'interprète fait l'objet du premier paragraphe et les principes de mise en œuvre sont donnés au second paragraphe.

III.1. Spécification de l'interprète

Précisons les trois phases du processus d'interprétation introduites ci-dessus.

III.1.1 Définition de la fonction COMPIL

Cette fonction a pour résultat une séquentialisation du programme à interpréter, ou toutes dans le cadre de son utilisation par le système de preuves (cf chap V).

Les séquentialisations d'un programme sont définies (cf § I.4) à partir de:

- l'ensemble E des unités d'exécution de tous les processus, de leurs pré et post-assertions et de leurs durées respectives,
- l'ensemble, noté AEX, des arbres d'exécution de chacun des n processus du programme .

Il s'agit donc à partir de l'arbre de syntaxe abstraite du programme de construire l'ensemble E des unités d'exécution,

accompagnées de leurs assertions, et l'ensemble AEX : l'algorithme correspondant est décrit en Annexe III.1.

Ensuite une ou toutes les séquentialisations possibles, pour un rythme donné des processus, pourront être construites selon la définition du § II.4: l' utilisation d'un module de calcul logique permet de passer d'une séquentialisation définie au rang i à la séquentialisation suivante par la vérification de la relation "inférieure" .

Nous ne détaillerons pas cette construction statique des séquentialisations car elle n'a pas donné lieu à une réalisation (cf §III.2).

III.1.2. Définition de la fonction INTERP

Cette seconde phase réalise l'interprétation d'une séquentialisation donnée S, par la machine virtuelle MVP pour un certain jeu de données, selon le schéma :

$$(S,m_0) \xrightarrow{tr_0} \cdots \xrightarrow{tr_{i-1}} (\underline{S}_i,m_i) \xrightarrow{tr_i} \cdots \xrightarrow{tr_k} (\ ,m_k)$$

où tr_i désigne la i ème transition de la machine MVP qui met en oeuvre la i ème composition parallèle d'unités d'exécution de la séquentialisation S

$\underline{S}_i$ désigne le reste de la séquentialisation à activer :
si $S = (<U_0,H_0>; <U_1,H_1>; \ldots;<U_k,H_k>)$ à la fin du (i-1) ème cycle $\underline{S}_i$ désigne la séquence $(<U_i,H_i>; \ldots;<U_k,H_k>)$

m_i désigne l'état mémoire du programme qui résulte de la (i-1) ème transition

m_0 est l'état mémoire initial défini par les données du programme
et m_k est l'état final qui contient les résultats.

L'interprétation est terminée lorsqu'aucune transition de la machine n'est plus déclenchable :

. soit le dernier ensemble U_k de la séquentialisation a été activé et le programme termine normalement en délivrant ses résultats,

. soit le programme s'arrête sur un état de blocage et la séquentialisation n'est pas terminée.

Le rôle de la machine MVP est donc à chaque déclenchement de transition :

- de séquentialiser l'ensemble des actions $A_1 A_2 \ldots A_n$, qui seront ensuite exécutées sur la machine séquentielle MS: l'action A_i réalise l'interprétation de l'unité d'exécution du i ème processus à activer; cette action est éventuellement l'action vide lorsque le i ème processus est "en avance" sur les autres processus ou s'il est bloqué en attente d'une consommation (cf § II.4),

- de déterminer pour chaque action A_j la séquence $seq_j = a_{j1} \ldots a_{jq}$ des actions élémentaires de la machine MS : il s'agit des actions atomiques qui composent l'unité d'exécution représentée par l'action A_j.

Enfin la troisième phase du processus d'interprétation consiste en l'exécution, par la machine MS, des séquences d'actions élémentaires produites par la fonction INTERP à chaque cycle, à partir de l'état initial défini par les données du programme.

Décrivons maintenant les principes de mise en œuvre de ces trois phases.

III.2. Principes de mise en œuvre

III.2.1. Construction des séquentialisations

Nous avons vu au paragraphe II que la construction d'une séquentialisation ne peut être réalisée de manière statique que pour les programmes où les types de communications utilisés ne tiennent pas compte des valeurs des données échangées.

Or le processus d'interprétation doit être le même pour tous les types de programmes: la construction de la séquentialisation n'est donc pas réalisée a priori mais de manière incrémentale à chaque pas d'interprétation de la machine MVP.

Ainsi à la fin d'un cycle l'ensemble U des unités à activer au cycle suivant est élaboré à partir des valeurs courantes des assertions, et non pas de leur expression symbolique comme dans le cas de la fonction COMPIL.

Nous appelons CONSTR cette opération de construction qui a pour profil :

$$UE^n \times PC_i \times H_i \longrightarrow (U_{i+1} , UE'^n)$$

où UE^n est le n-uplet d'unités d'exécution susceptibles d'être activées au cycle i+1 (n étant le nombre de processus); au 1 er cycle d'interprétation il est constitué des unités initiales e0 de chacun des processus ; pour les autres cycles il est composé de la prochaine unité à activer pour chaque processus : celle qui "satisfait" la trace d 'exécution du processus

PC_i désigne la valeur de l'ensemble des ports du sytème résultant du cycle n° i

H_i est le n-uplet des horloges des processus définies par le cycle n° i : elles indiquent la date de fin des actions activées au cours de ce cycle

U_{i+1} désigne l'ensemble des unités qui seront activées au cycle i+1

UE'n désigne le n-uplet des prochaines unités susceptibles d'être activées au cycle i+2 (celles qui satisfont la trace d'exécution de leur processus).

Les deux résultats de la fonction CONSTR sont élaborés de la manière suivante :

- U_{i+1} est un sous-ensemble extrait de UE^n composé des unités effectivement activables , c'est à dire dont la pré-assertion a la valeur (vrai, ... ,vrai) et telles que l'horloge du processus correspondant ait la valeur minimum de l'ensemble Hi; la valeur de chaque terme de la pré-assertion est calculée à partir de la

valeur courante du port de communication concerné, donnée par l'état PC_i.

Lorsque plusieurs unités activables de l'ensemble UE^n sont exclusives, les choix réalisés pour constituer l'ensemble U_{i+1} sont ceux définis au § II.2:

. soit le choix peut être arbitraire et une seule est choisie, de manière aléatoire, pour faire partie de l'ensemble U_{i+1},

. soit il s'agit d'unités réalisant l'une une production et l'autre une consommation sur le même port : ces deux unités sont modifiées ainsi que nous l'avons décrit au §II.2 de manière à ce que la production et la consommation soient toutes les deux activées .

Notons que lorsqu'une unité activable n'est pas retenue pour faire partie de l'ensemble U_{i+1}, elle peut ne plus l' être après l'activation de l'unité choisie à sa place (cas d'un port de type "égalité" dont l'ensemble consommable ne contient qu'une valeur par exemple).
Si l'unité de l'ensemble UE^n associée au processus P_k n'est pas activable ce processus est "mis en attente" jusqu'à ce qu'un autre processus du système lève cette attente en rendant "vraie" la pré-condition de consommation par l'activation d'une opération de production ;

. UE'^n est constitué d'au plus n unités d'exécution qui sont pour chaque processus (non encore terminé):

. soit l'un des successeurs possibles de l'unité sélectionnée dans l'ensemble U_{i+1}, choisi au hasard dans l'arbre d'exécution du processus,

. soit l'unité écartée de l'ensemble U_{i+1} parce qu'exclusive avec d'autres,

. soit l'unité non retenue parce que sa pré-assertion n'était pas vérifiée dans le cas où le processus est mis en attente.

Ainsi réalisée la fonction CONSTR utilise deux des composantes de la machine MVP : l'état courant des ports de communication et les horloges des processus.

Nous décrivons maintenant la mise en œuvre de la fonction INTERP et de la machine virtuelle associée.

III.2.2. Réalisation de la fonction INTERP

Chaque cycle de la machine MVP active la fonction CONSTR qui définit un ensemble U d'unités à activer : l'interprétation de ces unités est réalisée de la manière suivante :

. construction d'une séquence A1, ... ,Ak , où chaque "action" Ai réalise l'interprétation d'une unité d'exécution ; l'ordre dans cette séquence est arbitraire puisque, par définition, les unités à activer sont indépendantes,

. pour "chaque" action Ai construction de la séquence des actions élémentaires qui composent l'unité correspondante et qui seront exécutées par la machine séquentielle MS

. gestion des horloges des processus : élaboration de l'ensemble H qui résulte des activations réalisées au cours d'un cycle .

Détaillons la construction de l'ensemble H_i défini au cycle n° i à partir des valeurs de H_{i-1} et de l'ensemble U_i des unités déterminées par la fonction CONSTR.

H_i est le n-uplet des horloges $<h_i^1, \ldots, h^n>$ tel que :

. si l'horloge h_{i-1}^k du processus P_k a la valeur minimum de l'ensemble H_{i-1} et qu' une unité du processus P_k est retenue dans U_i alors $h_i^k = h_{i-1}^k + \delta$, où δ est la durée de l'unité retenue,

. lorsque le processus Pk est "en attente" de consommation son horloge doit progresser jusqu' à atteindre la valeur des horloges des processus activés au cours de ce cycle : cette date est la plus petite date à laquelle une production est susceptible d'être réalisée et de lever l'attente; par conséquent h_i^k prend la valeur minimum des horloges de H_{i-1} associées aux processus dont une unité est activée dans U_i,

. si le processus P_k n'a pas terminé l'action précédente son horloge est "en avance" sur celles des autres processus donc $h_i^k = h_{i-1}^k$, si au moins un autre processus d'horloge minimum est activable au cycle n° i,

. si l'ensemble des processus d'horloge inférieure à celle du processus P_k sont bloqués en attente d'une consommation, le processus P_k doit être activé, si sa prochaine unité est activable; dans ce cas $h_i^k = h_{i-1}^k + \delta$ où δ est la durée de l'unité activée au cycle n° i.

Définissons maintenant l'état mémoire de la machine MVP. Etant donné que l'exécution proprement dite est réalisée par la machine support MS, la fonction INTERP n'utilise pas la mémoire locale à chaque processus défini par le programme; l'ensemble des mémoires locales aux processus sera défini au niveau de la machine MS.

Par contre l'état des ports de communication partagés par les processus est nécessaire à la fonction CONSTR ainsi que nous l'avons vu au § III.2.1 : il sera défini par la machine MS qui réalise les opérations de communication et utilisé par la fonction CONSTR.

L'etat mémoire de la machine MVP est donc défini par le couple $< PC , H >$ où PC désigne l'état des ports de communication et H est le n-uplet des horloges des processus.

III.2.3. Réalisation de la phase d'exécution séquentielle par la machine MS

La machine séquentielle, abstraite, MS constitue le noyau de l'interprète. Ses actions sont les six actions atomiques définies par la sémantique au chapitre II : affectation, produire, consommer , conditionnellle élémentaire, action vide et l'action prod-cons, auxquelles il faut ajouter la réalisation des entrées/sorties.

L'exécution d'une action élémentaire a pour effet:
. la modification de l'état mémoire local d'un processus et/ou de l'état des ports de communication dans le cas d'une opération de communication,
. l'incrémentation de l'horloge du processus concerné de la durée de l'action réalisée (durée définie en paramètre par l'utilisateur).

L'etat mémoire de la machine MS est donc défini par la donnée du triplet

$< PC , \bigcup_{i=1,n} m_i, H >$ où m_i désigne la table des liaisons variable-valeur pour les variables du processus i.

A chaque cycle d'interprétation la machine exécute une séquence d'actions définie par la fonction INTERP. Les transitions sont donc de la forme :

$$s\ x\ e \xrightarrow{a} s'\ x\ e'$$

où s désigne une séquence à activer,
s' désigne la séquence restant à activer après l'exécution de l'action a,
a est une des six actions énoncées ci-dessus,
e et e' sont les états mémoire consécutifs de la machine.

IV. CONCLUSION

L'objectif de ce chapitre était de présenter les principes d'interprétation des programmes et leur mise en œuvre fonctionnelle en s'appuyant sur un modèle formel d'exécution.

Rappelons que le but de l'interprétation est d'offrir à l'utilisateur une simulation de l'exécution de son programme pour permettre une validation intuitive de la solution en termes de processus communicants en regard de la définition initiale sous forme de relations de communication.
Dans cette approche la simulation des communications constitue l'objectif central de l'interprétation : nous avons donc naturellement été conduit à définir comme modèle de simulation une sorte d'échéancier des communications, autrement dit, en termes d'objet typé, une *suite temporelle* d'évènements de communication. Cette simulation, que nous appelons **"séquentialisation"**, est donc parfaitement cohérente avec l'interprétation des types de communication que nous avons définie au chapitre I §III. Notons simplement que chaque évènement de cette suite temporelle est en fait un "paquet" de communications indépendantes mais simultanées.

Bien sûr les processus ne sont pas faits que d'actions de communication et il est apparu naturel d'englober dans une même action "abstraite" une opération de communication et les actions "internes" qui lui font suite dans une dérivation du processus et ne

sont pas "observables" (en utilisant la terminologie CCS). D'où la notion d'"**unité d'exécution** " déjà utilisée dans [Per 85] et que nous pouvons ici formaliser grâce à la définition de la sémantique donnée au chapitre précédent, elle-même héritée des travaux sur la notion de *critère d'observation* [AB 84].

Une fois introduite la notion de "**trace d'exécution**" d'un processus, définie comme une séquence d'unités d'exécution, il restait à formaliser l'ensemble des combinaisons possibles des traces d'exécution des processus, lesquelles constituent l'arbre des comportements d'un programme. Nous avons pour cela utilisé les notions de **pré** et de **post-assertion** d'une unité d'exécution introduites dans [Per 85] en étendant leur signification aux "paquets" d'unités d'exécution activables simultanément. Grâce à ces assertions nous avons pu définir formellement une relation de causalité sur les communications (relation "implique" définition 8) et delà en déduire la définition de la notion de séquentialisation.

Remarquons, ainsi que nous l'avons fait lors de la définition de la sémantique, que le caractère instantané des communications simplifie l'expression du modèle mais ne constitue pas une restriction : on pourrait donc étendre ce modèle en envisageant des communications de durée non nulle, par exemple dans le cadre d'une implantation répartie des programmes.

Le type de modèle que nous obtenons est difficilement comparable aux modèles d'exécution habituellement proposés dans la littérature sur la sémantique des programmes parallèles dans la mesure où nous traitons explicitement les dates des évènements. Généralement (*) le modèle du parallélisme adopté est celui du non-déterminisme borné(**) et les processus sont définis soit comme des termes d'une algèbre munie d'une sémantique opérationnelle sous forme d'un système de transition (approche "CCS") soit comme des automates (approche "langages réguliers", [AN 82] par exemple). L'intérêt de ces modèles est un cadre mathématique permettant des raisonnements formels notamment pour la définition d'équivalences de comportement. Par contre l'introduction du temps semble, à notre connaissance, incompatible avec ce type de modèle.

(*) *borné* parce que l'on suppose que les vitesses respectives des différents processus sont dans un rapport borné.

(**) nous reviendrons sur les modèles de comportement dans le cadre de la définition d'un système de validation des programmes au chapitre V.

Notons cependant le modèle proposé pour TCSP dans [ReR 86] qui repose sur les traces
datées des communications mais là le temps est vu comme continu.

La seconde partie du chapitre a été consacrée à la présentation fonctionnelle de l'interprète sur la base de la notion de séquentialisation.
L'interprète réalise donc la simulation d'une séquentialisation. Nous avons vu que les séquentialisations peuvent être construites a priori pour une certaine classe de programmes, ce que l'on utilisera au chapitre V, mais que, pour couvrir tous les cas, l'interprète fonctionne en construisant pas à pas une séquentialisation.

Cette partie est technique; nous nous sommes toutefois attaché à décrire rigoureusement les principes de mise en œuvre en présentant de manière structurée le processus d'interprétation et en faisant explicitement référence aux définitions établies dans la première partie du chapitre.
Pour compléter cette présentation il reste maintenant à décrire la réalisation de l'interprète, ce qui fait l'objet du chapitre suivant.

CHAPITRE IV

REALISATION DE L'INTERPRETE

REALISATION DE L'INTERPRETE

Ce chapitre présente la mise en œuvre du logiciel d'interprétation dont nous avons énoncé les principes au chapitre précédent.

Ce logiciel s'insère dans un environnement de programmation appelé COMEDIE et développé en Le_Lisp-CEYX [Cha 85] [Hul 84].

Nous commençons, au paragraphe I, par présenter brièvement cet environnement de programmation. Puis nous décrivons l'architecture de l'interprète (§II). Nous présentons au paragraphe III les principales structures de données utilisées ainsi que la description de la syntaxe abstraite. Ensuite nous décrivons le module principal de l'interprète qui réalise la simulation du parallélisme (§V). Puis au paragraphe V nous détaillons la réalisation de l'interprétation des opérations de communication et nous terminons, au paragraphe VI, par la description de la visualisation graphique.

I. L'ENVIRONNEMENT DE PROGRAMMATION COMEDIE

Ce logiciel est bâti autour d'un éditeur syntaxique [Jul 85] et organisé ainsi que le décrit le schéma [figure 1].

Cet environnement de programmation offre principalement à l'utilisateur :

- des fonctions de construction incrémentale des programmes au moyen de schémas prédéfinis de processus, ports, types de communication, etc...
- des fonctions de contrôle de la sémantique statique : validité du réseau de communication du programme, contrôle et inférence de type "à la ML" [Mil 84], etc...
- les fonctions traditionnelles d'un éditeur syntaxique qui permettent à l'utilisateur d'ignorer totalement la structure des arbres de syntaxe abstraite qui représentent son programme, dans le même souci que le système MENTOR [Mel 83],
- des fonctions de sauvegarde des processus et des types de communication en bibliothèque,
- un traducteur des programmes sous la forme de programmes Ada [JKP 86],

et enfin les fonctions d'interprétation et de visualisation graphique des communications.

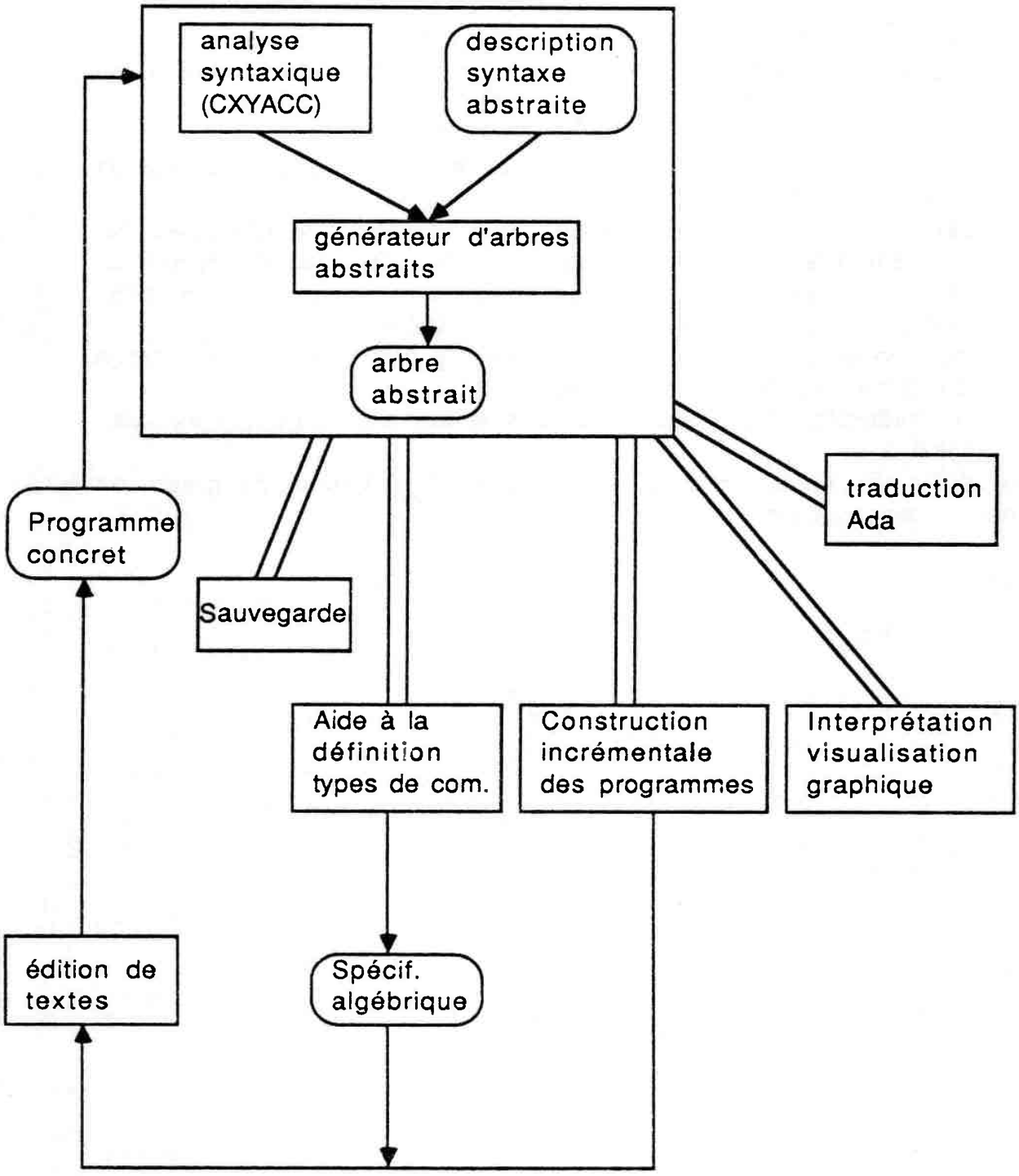


Figure 1 : l'environnement de programmation COMEDIE

II. ARCHITECTURE DE L'INTERPRETE

Le schéma donné ci-dessous, [figure 2], résume l'organisation de ce logiciel.

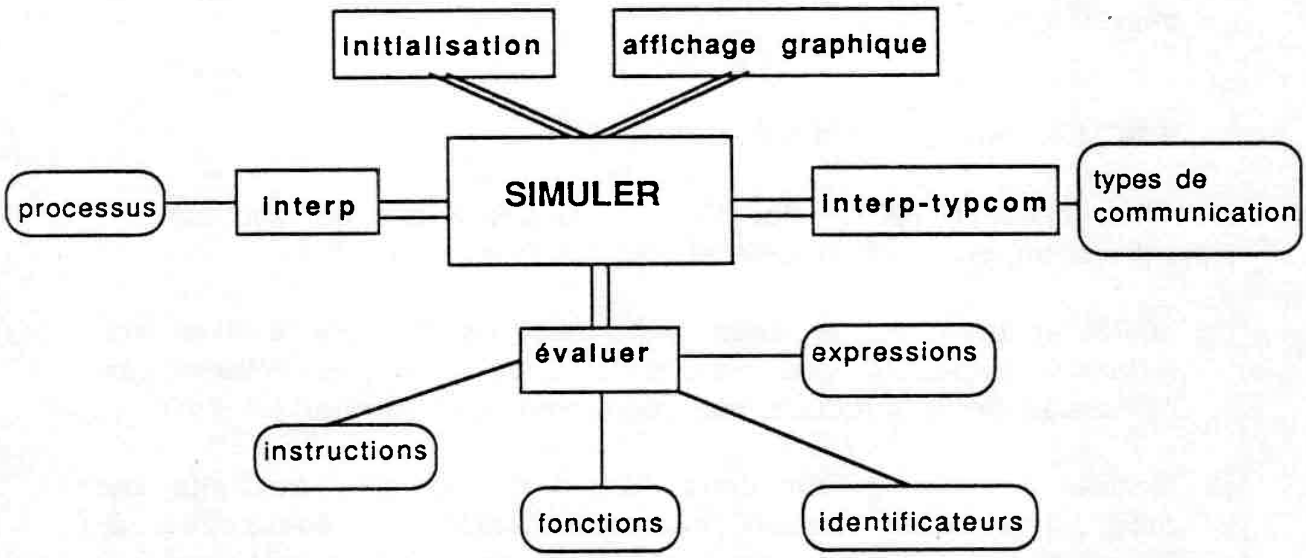


Figure 2 : Architecture de l'interprète

II.1. Le module SIMULER

Ce module constitue le noyau de ce système de simulation : il réalise la fonction CONSTR décrite au chapitre précédent (§III.2). Comme pour tout simulateur d'exécution, ce module consiste en une boucle itérée tant que la fin de l'exécution n'est pas atteinte. Chaque cycle réalise les fonctions suivantes :

- simulation du parallélisme : choix des processus pour lesquels une unité d'exécution doit être activée au cours du cycle, gestion des horloges des processus et des mises en attente,

- activation de la fonction "interp" pour chaque processus sélectionné, selon un ordre arbitraire (cf chap II §II.2).

La fin de l'exécution est atteinte lorsque :

- soit chaque processus de programme a terminé son exécution,

- soit les processus qui n'ont pas terminé sont dans un état d'attente d'une communication : il s'agit alors d'une situation d'interblocage.

II.2. Le module "initialisation"

Il est activé par la fonction SIMULER avant de commencer l'interprétation proprement dite et remplit quatre fonctions :

(1) instaurer un dialogue avec l'utilisateur de manière à fixer les rythmes respectifs des processus ; ce dialogue débute par l'affichage de la structure du programme à interprèter (cf §VI),

(2) réaliser la propagation dans l'arbre de syntaxe abstraite du programme d'un certain nombre d'attributs nécessaires à l'interprétation : nous y ferons référence dans les paragraphes qui suivent,

(3) créer et initialiser les structures de données nécessaires à l'interprétation, notamment :

- une liste des noms des processus, une liste de processus "prêts à être activés", une liste des processus "en attente",

- une horloge globale au système décrit et une horloge locale par processus (cf §III),

- un "compteur ordinal" par processus indiquant notamment l'adresse dans l'arbre d'un processus de la prochaine action à activer : la structure de cet objet sera décrite au paragraphe suivant.

(4) réaliser l'interprétation des déclarations du programme c'est à

dire créer les objets nécessaires pour représenter
. les variables internes déclarées par les processus,
. les objets de communication définis par la déclaration des ports de communication.
Les choix de représentation seront décrits au paragraphe III.

II.3. La fonction "interp"

Elle est attachée à l'opérateur "processus" de la syntaxe abstraite (cf §IV). Appliquée dans l'arbre du programme, à un noeud de type "processus", elle consiste à déterminer les noeuds (de type "instruction") qui composent la prochaine unité d'exécution à interprèter et à leur appliquer la fonction "évaluer" qui réalisera leur interprétation.
Le profil de la fonction "interp" est le suivant :

< état-mémoire > x co x hl → < état-mémoire>' x co' x hl'

où < état-mémoire > désigne l'état de la mémoire locale au processus et des ports de communication associés à ce processus,

co est l'objet "compteur ordinal" du processus qui désigne l'adresse dans l'arbre de l'unité à activer ; co' désigne l'adresse de l' unité suivante à activer.

hl désigne l'horloge locale au processus et hl' a la valeur de l'horloge hl incrémentée , par la fonction "évaluer", de la durée des actions réalisées

<état-mémoire>' est l'état-mémoire modifié par l'interprétation des instructions .

II.4. La fonction "évaluer"

Elle est attachée aux opérateurs de type "instruction", "expression", "indentificateur" et "fonction" de la syntaxe abstraite (cf §III).

Appliquée à un nœud de type "expression", "identificateur" ou "fonction", elle en réalise l'évaluation et retourne la valeur correspondante, sans modifier l'état de mémoire ni l'horloge du processus correspondant : en effet nous avons fait l'hypothèse, au chapitre II, que l'évaluation des expressions est instantanée.

Appliquée à un arbre de type "instruction" la fonction "évaluer" réalise l'interprétation de cette instruction : nous ne détaillerons pas les fonctions d'interprétation des instructions "classiques", par contre nous présenterons au §V la réalisation des opérations de communication "produire" et "consommer".

Notons simplement que cette fonction a pour profil :

< état-mémoire> x hl → < état-mémoire >' x hl

II.5. La fonction "interp-com"

Elle est attachée à l'opérateur "type de communication" et réalise l'interprétation des trois opérations caractéristiques d'un type de communication (cf. chap I, §III.1) :

- pré_cons : évaluation de la pré-condition de consommation
- val_cons : calcul de la valeur consommée,
- post_cons : évaluation du nouvel état de l'ensemble consommable suite à une consommation.

Cette fonction est donc activée lors de l'interprétation d'une opération "consommer" par la fonction "évaluer" : nous la détaillerons au paragraphe V.

II.6. Le module "visualisation graphique"

Il est activé à deux moments de l'interprétation :

- lors de la phase d'initialisation pour afficher la structure du programme à interprèter,
- lors de l'interprétation des opérations de communication pour visualiser les valeurs communiquées et l'état des ports de communication.

La réalisation de ce module sera présentée au paragraphe VI. Nous détaillons maintenant les principaux choix de représentation dans le langage "orienté objet" CEYX.

III. PRINCIPAUX CHOIX DE REPRESENTATION CEYX

III.1. Description de la syntaxe abstraite du langage

Elle a été définie par J.Julliand pour la réalisation de l'éditeur syntaxique. Nous en décrivons les principales caractéristiques en utilisant la terminologie MENTOR [DHK 80] que nous rappelons brièvement.

Les phylums et les opérateurs sont des types d'arbres. Les types sont structurés en une hiérarchie, sous forme d'arbre, par un mécanisme d'héritage simple : un phylum est un ensemble non vide d'opérateurs.
Les opérateurs sont les feuilles de la hiérarchie dont on prendra des instances; les ancêtres permettent de définir des propriétés communes : des attributs ou des opérations (ou *méthodes* dans la terminologie langages orientés objet). Les opérateurs sont d'arité fixe ou variable s'il s'agit de nœuds de type "liste".

L'union de types CEYX ("modelunion") permet de réaliser de l'héritage multiple.

Nous joignons en Annexe IV.1 la description complète de la syntaxe abstraite et présentons ici certains des principaux phylums et opérateurs.

La racine de la hiérarchie est le phylum "lu" : il permet de mémoriser les commentaires introduits dans le texte d'un programme; ses principaux descendants sont les phylums :

. "prog" : type de l'arbre qui représente un programme

. "decl-prog" : type des instances d'arbre qui décrivent les déclarations d'un programme parallèle : ses fils sont les phyllums "prog", "process" et "tca"

. "process" : type des instances d'arbre qui représentent les processus élémentaires

. "tca" : type des arbres associés aux déclarations de ports de communication

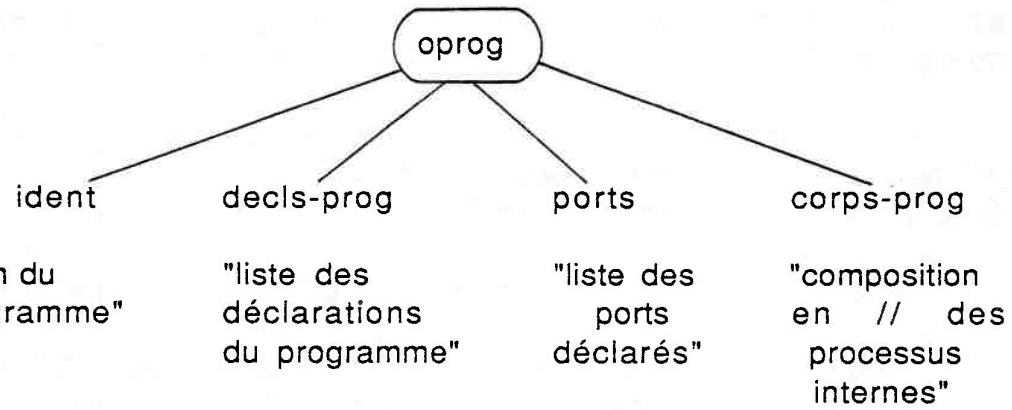
. "ident" : ensemble des opérateurs "oident" (identificateurs de variables), "fonct" appels de fonction) et "oid-int" (identificateurs intermédiaires utilisés dans la description d'un type de communication). Ce phylum est muni de deux attributs :
"val" : nom de l'identificateur
"vmodu" : attribut propagé lors de la phase d'initialisation : nom du processus où est déclaré l'identificateur.

A titre d'exemples décrivons la structure des instances d'arbre qui représentent un programme composé et un processus élémentaire.

La racine de l'arbre qui représente un programme composé est une instance de l'opérateur "oprog", fils du phylum prog, dont la structure est la suivante :

oprog : ident x decls-prog x ports x corps-prog → prog

On peut représenter graphiquement cette structure de la manière suivante:



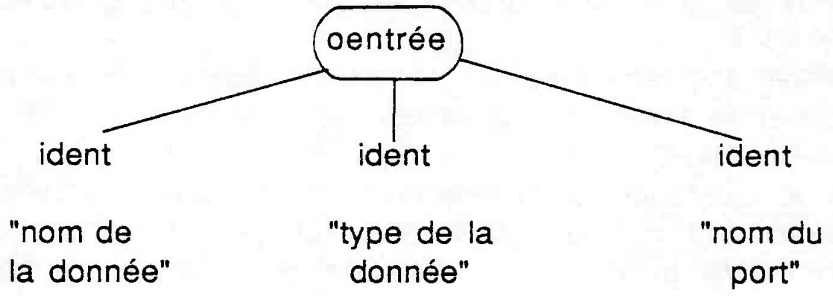
L'opérateur "oprocess" est le type des instances d'arbres associées à la définition d'un processus élémentaire.

Cet opérateur a la structure suivante :

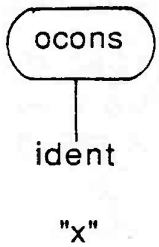
oprocess : ident x spécif x corps → process :

- le fils de type "ident" représente le nom du processus,
- le fils de type "spécif" décrit les déclarations de données communiquées dans les clauses "entrée" et "sortie" du processus (cf chap II, §I.1.1)
- le fils de type "corps" représente le corps du processus c'est à dire la liste des déclarations de variables locales et la liste des instructions.

Ainsi par exemple la déclaration d'une donnée communiquée en entrée du processus est décrite par une instance d'arbre dont la structure est la suivante :



L'instruction consommer(x), par exemple, est représentée par un arbre de type "ocons", structuré ainsi :



et muni de deux attributs : le nom du port de communication associé à la donnée x et le nom du type de communication associé à ce port ; ces deux attributs prennent leur valeur au cours de la phase "initialisation" et sont utilisés pendant la phase d'interprétation de l'opération consommer (cf §V).

Nous ne détaillerons pas davantage cette syntaxe abstraite. Notons simplement, à titre d'exemple, que la fonction "évaluer" attachée au phylum "exp", père de toutes les expressions arithmétiques et logiques, réalise l'interprétation de n'importe quelle impression, grâce au mécanisme d'héritage et aux facilités du langage LISP.

III.2. Principales structures de données et leur représentation

III.2.1. Représentation des variables et objets de communication déclarés par les programmes

Deux solutions étaient possibles pour gérer ces objets :

- définir une tables des symboles du programme à interprêter,
- utiliser la table des symboles de l'interpréteur LISP.

Nous avons choisi la seconde solution pour sa simplicité de mise en œuvre grâce à la notion de "package" offerte par Le_Lisp (cette notion est la base de l'implantation des classes du langage orienté objet CEYX).

Un "package" est un répertoire, éventuellement hiérarchisé, de symboles ; par exemple

#:Dalton:Joe et #:Dalton:Jack sont deux symboles du package "Dalton" ; dans le nom complet d'un symbole le signe ":" est le séparateur des noms des packages qui composent le chemin d'accès au package du symbole, depuis la racine de la hiérarchie désignée par le caractère " #".

Nous utilisons cette structuration pour représenter les variables déclarées dans les processus par des symboles Le_Lisp : par exemple

si "x" est une variable du processus P1, interne en programme P elle sera représentée par le symbole #: P: P1: x .

La représentation des ports de communication est particulière dans la mesure où un nom de port désigne un objet composé de trois champs (cf chap I, §III.1) :

- la suite produite SP
- la suite consommée SC
- l'ensemble consommable A.

Les suites peuvent être implémentées aisément par la structure de liste LISP mais cette structure est mal adaptée à la représentation de l'ensemble A, puisque l'on veut pouvoir accéder a priori à n'importe quel élément de cet ensemble . Nous avons donc choisi de représenter le contenu de chacun de ces objets dans la P-liste d'un symbole.

La P-liste d'un symbole est une table, représentée par une liste de couples (indicatif, valeur) dont les opérations d'accès sont : l'ajout d'un couple, la suppression, la modification, l'accès par l'indicatif. Cette structure correspond parfaitement au type "TABLE" que nous avons défini au chapitre I pour représenter l'ensemble consommable A.

Ainsi à un port de communication correspondent trois symboles :

: nom-proc : nom-port : SP
: " " : SC
: " " : A

où "*nom-proc*" : est le nom complet du processus où est déclaré le port de communication
"*nom-port*" : est le nom du port.

Remarque :
Cette représentation correspond aux ports définis dans le cadre de programmes réalisant des communications de type "bipoint' ou "diffusion" ou organisés selon un modèle de "convergence" ou de "divergence" "avec mélange" (cf chap I, § I.4)

Lorsque l'on veut distinguer l'origine de la production ("convergence sans mélange"), une suite produite par processus producteur est nécessaire. De même, dans le cas d'une situation de "divergence sans mélange", une suite consommable par processus consommateur doit être prévue. Ces extentions ne sont pas réalisées dans l'interprète actuel.

III.2.2. Principales structures de données manipulées par l'interprète

Les structures de données créées par le module d'initialisation et utilisées par la fonction SIMULER sont les suivantes :

(1) une liste des noms de processus élémentaires, nommée Liste-proc.

(2) une liste des noms de processus prêts à être activés, c'est à dire dont l'horloge a la valeur minimum (valeur de l'horloge globale) et qui ne sont pas en attente devant une opération de consommation : Liste-prets,

(3) une liste des noms de processus en situation d'attente : Liste-attente.

(4) l'horloge globale du système nommée HU (horloge universelle). Elle désigne un objet de type "hu", composé d'un champ (la valeur) et muni de deux opérations :

init : → hu : initialise la valeur à 0,

recaler : hu x liste-processus → hu', qui recale la valeur sur la valeur minimum des horloges des processus désignés dans la liste,

(5) les horloges locales des processus : elles ont pour nom : #:nom-proc:hl, où "nom-proc" est le nom complet du processus ; chaque horloge locale est un objet de type "hl" composé d'un champ (la

valeur) et muni des méthodes suivantes :

init : → hl : initialise la valeur à 0,

incrémenter : hl x valeur → hl',

faire-attendre : hl x hu → hl' : fait progresser la valeur de l'horloge locale jusqu'à la valeur l'horloge HU, lorsque le processus est en attente ,

(6) les compteurs ordinaux des processus :
ils ont pour nom #:nom-proc:co , où "nom-proc" est le nom complet du processus.
Chaque compteur ordinal est un objet de type "co" composé des champs suivants :

- un pointeur sur l'arbre qui représente la prochaine instruction à activer,
- le chemin suivi dans l'arbre qui représente la liste des instructions du processus : ce chemin est nécessaire pour élaborer la prochaine valeur du compteur ordinal,
- un booléen "attente" dont la valeur est 'vrai' si le processus est "en attente",
- un booléen "fin" à 'vrai' si la processus a terminé son exécution.

Les opérations définies sur ce type d'objet sont les suivantes :

init : nom-proc → co : initialise le pointeur avec l'adresse de la 1ère instruction du processus de nom "nom-proc",

instr-suivante : co x nom-proc → co' : fait pointer le compteur ordinal sur l'instruction suivante ,

fin-processus? : co → booléen : rend la valeur du booléen "fin",

attente? : co → booléen : délivre la valeur du booléen "attente",

mise-en-attente : co → co' : met le booléen "attente" à vrai ,

lever-attente : co → co' : met le booléen "attente" à faux .

A partir de la description de ces structures de données, nous pouvons maintenant décrire la fonction SIMULER.

IV. SIMULATION DU PARALLELISME : LE MODULE SIMULER

Rappelons le rôle de cette fonction, présenté au §II.1 :

- réaliser la simulation du parallélisme en définissant à chaque cycle la liste des processus à activer (Liste-prets), gérer l'horloge globale et les horloges locales des processus, gérer les mises en attente ,
- activer la fonction "interp" pour chaque processus sélectionné.

L'algorithme de ce module est le suivant :

```
. initialisation -- activation de la fonction "initialisation"
. tant que  Liste-prets ≠ vide  faire

      . pour  chaque processus de Liste-prets faire

        -- activer la fonction interp (qui modifie l'horloge du
                                                    processus)
        fait

      . lever l'attente des processus de Liste-attente qui ont été
        débloqués par l'activation d'une production ;
      . recaler l'horloge HU sur l'horloge minimum des processus
        qui ne sont pas en attente ;
      . faire attendre les processus en attente jusqu'à la valeur de
        HU ;
      . élaborer les listes Liste-prets et Liste-attente à partir de
        la liste Liste-proc, des valeurs des horloges locales des
        processus et des opérations fin-processus? et attente?
        associées aux compteurs ordinaux.

fait
```

Notons que la fonction qui élabore la liste Liste-prets réalise les choix entre les processus dont les prochaines unités à activer sont exclusives ainsi que nous l'avons décrit au chapitre précédent.

Notons que lorsque la fonction "interp" est activée pour un processus, le compteur ordinal désigne la prochaine opération de communication à activer (début d'une unité d'exécution, cf chap III, § II.1), et qu' au retour de cet appel il désigne la communication suivante dans la trace d'éxécution du processus.

Nous détaillons maintenant l'interprétation des instructions produire et consommer .

V. INTERPRETATION DES OPERATIONS DE COMMUNICATION

V.1. L'opération produire

Rappelons que l'arbre de syntaxe abstraite de type "prod" qui représente cette instruction a pour attributs (cf chap II §II.1) :

- le nom de la donnée communiquée et le nom du processus où elle est déclarée, pour pouvoir élaborer le nom du symbole Le_Lisp qui représente cette variable,
- la liste des noms des ports de communication associés à cette donnée communiquée : il s'agit des noms complets de ces ports afin de pouvoir élaborer le nom des symboles respectifs dont les P-listes contiennent la suite produite SP et l'ensemble consommable A.

L'interprétation de l'instruction produire(x) consiste à ajouter à la suite SP et à l'ensemble A de chaque objet de communication associé à la donnée x, la valeur de x.

De manière à représenter les suites SP et SC et l'ensemble A, la P-liste des symboles correspondants contient deux éléments particuliers :

- l'élément d'indicatif "min" dont la valeur est le rang minimum des termes de la suite représentée (ou l'indicatif minimum des

entrées de la table s'il s'agit de l'ensemble A)
- l'élément d'indicatif "max" dont la valeur est le rang maximum des termes de la suite (ou l'indicatif maximum dans le cas de l'ensemble A)

Exemple : soit un port de nom p déclaré dans le processus P1 et de type "égalité"

Les 3 symboles associés à ce port sont :

: P1: p : SP
: P1: p : A et # : P1: p : SC

Supposons que trois valeurs "a", "b", "c" ont été produites dans cet ordre et que la valeur "a" a été consommée.
Ces 3 symboles contiennent les P-Listes suivantes :

: P1 : p : SP → ((min,1)(max,3)(1, a)(2, b)(3, c))

: P1 : p : A → ((min, 2)(max,3)(2, b)(3, c))

: P1 : p : SC → ((min,1)(max,1)(1, a))

L'algorithme de la fonction "évaluer" qui réalise l'opération produire est le suivant :

- évaluer la valeur du symbole qui représente la variable x : soit v cette valeur

- pour chaque nom de port de l'arbre "prod" faire

  . reconstituer le nom des symboles dont les P-listes contiennent la suite SP et l'ensemble A. Soient sp et a ces symboles.
  . ajouter à la P-liste de sp la valeur v avec comme indicatif la valeur (max(sp)+1)
    -- on note max(s) la valeur désignée par l'indicatif max
    -- dans la P-liste de s
  . ajouter à la P-liste de a la valeur v avec comme indicatif la valeur (max(sp)+1)
  . incrémenter de 1 les valeurs désignées par l'indicatif max dans les P-listes de sp et a.

  fait

V.2. L'opération consommer

L'arbre de type "cons" qui représente une instruction consommer est porteur des attributs :

- le nom de la donnée communiquée,
- le nom du processus où elle est déclarée,
- le nom complet du port associé à la donnée,
- le nom du type de communication associé au port .

L'interprétation de l'instruction consommer(x) (via le port p) est la suivante :
si la pré-condition de l'objet désigné par p est "vrai"

alors affecter à x le résultat de l'évaluation de la fonction "val-cons", du type de communication associé à p ;
modifier l'objet désigner par p par l'application de l'opération "post-cons"

sinon mettre en attente le processus qui active cette opération.

Ainsi la réalisation de cette interprétation nécessite l'activation de la fonction "interp-cons" (cf §II) qui réalise l'évaluation des trois opérations caractéristiques d'un type de communication : pré_cons, val_cons, post_cons.

La définition algébrique de ces opérations peut-être complexe (cf l'exemple du type "croissante" donné au chapitre I, §III et l' exemple du type "ascenseur" joint en Annexe V.2) et nécessiter l'introduction d'identificateurs et/ou de fonctions intermédiaires.

Prenons l'exemple simple du type de communication "égalité" dont les trois opérations caractéristiques sont :

pré_cons (<SP, A, SC>) = non vide? (A)
val_cons (<SP, A, SC>) = premier (A)
post_cons (<SP, A, SC>) = retirer (A, min,(A))

Les opérations vide?, premier, retirer et min sont des opérations de type TABLE (défini au chapitre I) et sont représentées par des opérateurs de la syntaxe abstraite de type "tampon" ou "valeur", selon que le résultat de l'opération est une table ou la valeur d'un élément

(vide? est de type "expression logique") (cf la syntaxe abstraite jointe en Annexe IV.1).

La fonction 'interp-com" est donc attachée à chaque opérateur des phylums "tampon" et "valeur", ainsi qu'à l'opérateur "vide?".

Le mécanisme d'évaluation de ces fonctions est celui des fonctions récursives : en effet les fonctions intermédiaires supplémentaires sont décrites de manière récursive à partir des opérations de base du type TABLE (cf le type de communication "croissante" du chapitre I et le type "ascenseur" en Annexe IV.2). Nous ne détaillerons pas ici ce mécanisme.

Nous donnons l'algorithme de la fonction "évaluer" définie pour l'opérateur "cons":

- élaborer le nom du symbole qui représente la donnée communiquée x : soit x ce symbole.
- élaborer le nom des symbles dont les P-listes contiennent la suite SC et l'ensemble A : soient Si et a ces symboles.
- se positionner sur l'arbre de définition du type de communication utilisé.
- appliquer la fonction "interp-com" au noeud qui représente la pré-émolition, en substituant au paramètre formel "A" la valeur de la P-liste de a : soit bool le résultat de cette évaluation.

- si bool alors -- la pré-condition est vérifiée

  . appliquer "interp-com" sur le noeud qui représente l'opération "val_cons", avec comme paramètre effectif la valeur de la P-liste de a ; affecter le résultat de cette évaluation au symbole x et l'ajouter à la P-liste du symbole sc ;

  . appliquer "interp-com sur le noeud qui représente l'opération "post_cons", avec comme paramètre effectif la valeur de la P-liste de a ; affecter le résultat à la P-liste de a ;

sinon appliquer l'opération "mise-en-attente" au compteur ordinal du processus concerné

fsi

Il nous reste à décrire le module de visualisation graphique.

VI. LA VISUALISATION GRAPHIQUE

Ce module a été réalisé dans le cadre d'un DEA [Egl 87] à l'aide su système GKS (Graphique Kernel System) [Ber 86].

Ce module est activé à deux types d'instant au cours de l'interprétation :

- lors de la phase d'initialisation pour afficher la structure du système parallèle,
- à chaque activation d'une opération de communication .

La représentation graphique de la structure d'un système parallèle est du même style que celle proposée par le langage OCCAM [May 83]. Un exemple de cette visualisation est donnée par la figure 3 :

- un processus est symbolisé par un cercle plein contenant son nom,
- une flèche orientée dans le sens producteur → consommateur représente une liaison inter-processus,
- Les noms du port et du type de communication associés à cette liaison sont imprimés dans un rectangle,
- plusieurs couleurs sont utilisées pour diférencier les canaux du réseaux de communication.

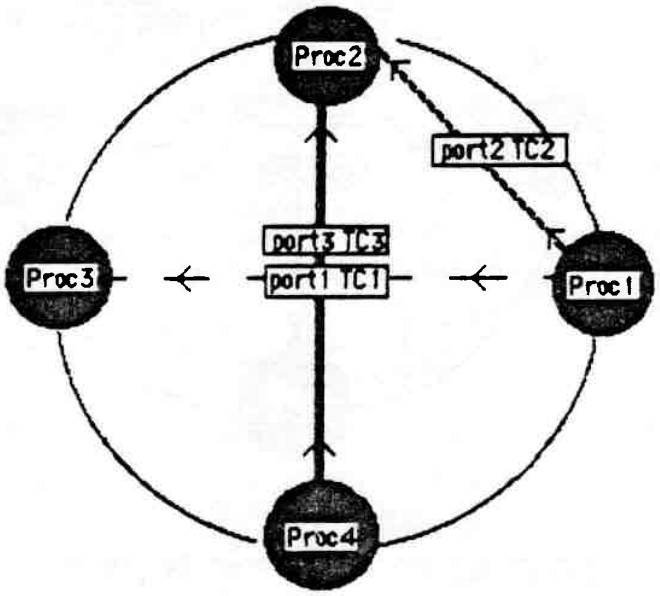


Figure 3 : Affichage de la structure d'un système parallèle

La visualisation des communications est réalisée de la manière suivante :

- chaque réalisation d'une opération "produire" ou "consommer" affiche dans le shéma qui symbolise la structure du système la valeur de la donnée communiquée (cf [figure 4]). Les valeurs sont affichées à coté du rectangle contenant les noms du port et du type de communication utilisés pour la communication : une valeur produite apparait à droite de ce rectangle et une valeur consommée à gauche,

- la valeur affichée est rémanente jusqu'à l'opération suivante,

- une touche permet à l'utilisateur de suspendre l'exécution et de la reprendre,

- une seconde touche doit permettre de remplacer l'écran de la figure 4 par celui de la figure 5 pour visualiser l'état d'un port de communication particulier.

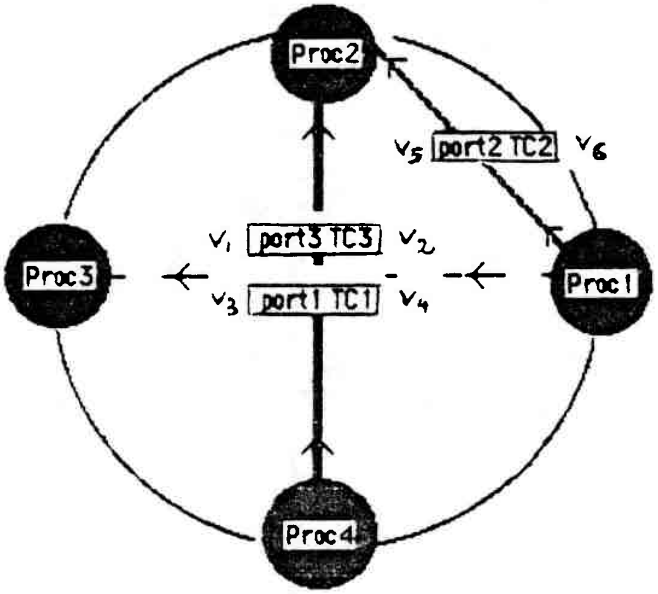


Figure 4 : Visualisation des communications

Note : v1, v2, ... désignent les valeurs des données communiquées dans le schéma [figure 4]

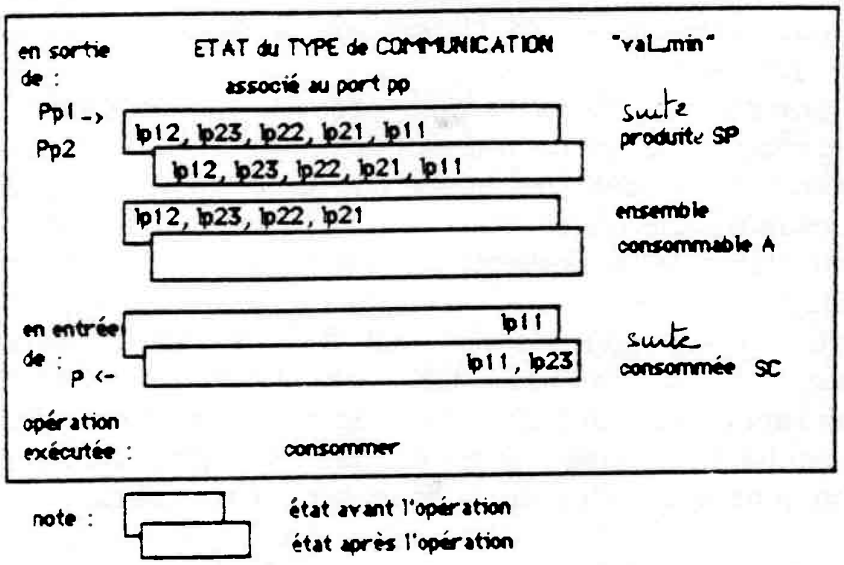


Figure 5 : Affichage de l'état d'un port de communication

Dans la fenêtre, schématisée [figure 5], sont représentés les états de la structure de donnée <SP, A, SC>, associée à un port de communication, qui correspondent à la dernière opération réalisée : états avant l'opération et après l'opération.
Sur l'exemple ci-dessus, la dernière opération réalisée est une consommation (de la valeur lp23) et le nouvel ensemble consommable A est vide (caractéristique du type de communication utilisé).

VII. CONCLUSION

Nous venons dans ce chapitre de donner une description technique de notre réalisation logicielle.
Nous nous sommes limité, pour être concis, à une présentation des principales caractéristiques de l'interprète, à savoir :

- son architecture,
- les grands traits de la syntaxe abstraite du langage,
- les principaux choix de représentation en CEYX,
- l'interprétation des primitives de communication,
- la visualisation graphique.

Afin de compléter cette présentation, nous proposons un bilan sur cette réalisation.
Ce logiciel, s'il est opérationnel, est à l'état de prototype. Les modifications à lui apporter pour en faire un produit "fini" concernent l'enrichissement pour simuler des systèmes hiérarchisés divers, l'amélioration des performances et l'amélioration de la visualisation graphique. Détaillons chacun de ces points.

Actuellement nous nous limitons à l'interprétation des programmes dont les processus internes forment une hiérarchie de type "bipoint", "diffusion", "divergence avec mélange" ou "convergence avec mélange" (cf chapitre I § I.4). Pour rendre compte de la divergence "avec éclatement" où l'on veut pouvoir distinguer les consommations spécifiques à chaque processus consommateur il faut :

- modifier la structure de données associée aux ports concernés: une suite consommée SC par processus consommateur doit être représentée,
- modifier l'interprétation de l'opération "consommer" : les opérations *pré_cons*, *val_cons* et *post_cons* doivent être évaluées pour la suite SC spécifique du processus qui active l'opération "consommer".

De même pour permettre la structure de convergence "sans mélange" où l'on veut distinguer l'origine des valeurs communiquées il faut prévoir une suite produite et une suite consommable par processus producteur et modifier en conséquence l'interprétation de l'opération *produire* dans de tels cas. Bien entendu les modifications syntaxiques prévues au chapitre II §I.4 pour distinguer chaque type type de divergence ou de convergence devront être intégrées.

D'autre part la simulation des programmes est peu performante lorsque les opérations des types de communication utilisés sont complexes (exemple le type "ascenceur" joint en annexe IV.2) et que le nombre de communications au niveau de chaque port devient élevé (plus de 10 valeurs dans l'ensemble consommable A). Cette dégradation des performances provient de la définition récursive des fonctions intermédiaires introduites pour décrire les opérations *pré_cons*, *val_cons* et *post_cons* des types de communication. La solution à ce problème consiste à optimiser la représentation de la structure de données associée aux ports de communication et à compiler les fonctions d'interprétation qui sont attachées aux opérateurs utilisés pour représenter les opérations des types de communication.

En ce qui concerne la visualisation graphique les améliorations possibles sont:

- raffiner l'affichage de la structure du programme de manière à visualiser des structures éventuellement complexes : en effet actuellement un nombre élevé de processus et de connections rend l'affichage peu lisible,
- faire cohabiter sur un même écran le schéma de la structure du programme ([figure 4]) et la visualisation détaillée des communications sur un port donné ([figure 5]).

Notons que ces objectifs, initialement prévus, n'ont pas été réalisés en raison de la difficulté à interfacer les programmes GKS écrits en FORTRAN avec le langage Le_Lisp.

Enfin à titre d'information mentionnons que les fonctions d'interprétation correspondent à environ 800 lignes de Le_lisp-CEYX, non compris la description de la syntaxe abstraite ni les procédures GKS.

CHAPITRE V

VALIDATION DES PROGRAMMES

VALIDATION DES PROGRAMMES

I. INTRODUCTION

Ce chapitre a pour objectif d'établir les principes d'un logiciel de validation basé sur l'interprète défini dans les deux derniers chapitres. Nous nous limitons à l'étude d'une propriété de vivacité, en l'occurrence l'**abscence de blocage**, et nous restreignons à une certaine classe de programmes, intuitivement les programmes pour lesquels la relation de causalité sur les communications (cf chapitre III, §I et II) ne dépend pas des valeurs communiquées et peut donc être établie de manière statique.

Rappelons que seules les opérations de consommation peuvent être bloquantes, l' opération de production n' étant pas soumise à une pré-condition : cette propriété est caractéristique des modèles où la communication est asynchrone et ne constitue donc pas une restriction abusive.

Nous avons défini au chapitre III un modèle d'exécution des programmes, appelé **"séquentialisation"**, et avons défini le fonctionnement de l'interprète comme la mise en œuvre de l'**une** des séquentialisations possibles d'un programme, pour un état initial donné et un certain rythme des processus composant le programme.

Intuitivement montrer qu'un programme satisfait une propriété $\mathcal{P}$ consiste à montrer que cette propriété est vérifiée par l'ensemble des séquentialisations possibles. En l'occurrence établir qu'un programme

satisfait la propriété d'abscence de blocage revient à vérifier que pour tout état associé à une exécution du programme, cette exécution peut progresser. Utilisant notre modèle de comportement cette vérification consiste à montrer que pour toute séquentialisation S d'un ensemble d'unités d'exécution :

- soit S est une séquentialisation de l' ensemble E des unités du programme (le programme est terminé)
- soit il existe un sous-ensemble U de E contenant au moins une unité activable c'est à dire tel que la post-condition de S "implique " la pré-assertion de U (cf chapitre II, §II, définition 8).

Pour la catégorie de programmes évoquée ci-dessus, où les ports utilisés correspondent à des types de communication dont l'opération "CONSOMMER" ne porte pas sur les valeurs, le graphe de la relation "implique" dépend uniquement de entrelacements des accès aux ports de communication.
Les types "égalité" et "aléatoire" sont des exemples de tels types de communication mais le type "croissante" n'en est pas un (cf chapitre I §III).

L'ensemble de ces types de communication est caractérisé par une pré-condition de consommation dont la valeur est :

- soit vrai : il n'y a alors pas de blocage possible ,
- soit un prédicat sur le cardinal de l'ensemble consommable A des objets du type TYPECOM. (cf des exemples de tels types en Annexe V)

Ainsi, pour des programmes qui n'utilisent que de tels types de communication, la pré-assertion d'une unité d'exécution :

- soit est toujours vérifiée,
- soit ne dépend que du nombre de productions et de consommations et de leur interclassement.

Nous envisageons donc une validation par simulation, a priori exhaustive, de l'ensemble des exécutions d'un programme: nous verrons que la définition d'une **relation d'équivalence** sur les séquentialisations permet de réduire le nombre des exécutions à examiner et d'envisager une utilisation interactive de l'interprète à cette fin de vérification.

En particulier, pour la classe de programmes définie ci-dessus, l'ordre selon lequel plusieurs communications, de même type et simultanées sur un port, sont séquentialisées est sans conséquence sur la propriété de non-blocage. Intuitivement on considérera comme équivalentes deux exécutions qui réalisent les mêmes entrelacements d'accès aux ports de communication, à une relation de commutation près. Par exemple si l'on note P!x la production par le processus P d'une valeur sur le port x et P?y la consommation d'une valeur sur le port y, les séquences d'actions s1 et s2 ci-dessous décrivent des comportements équivalents du programme (P//Q//R) où les processus P et Q "produisent" sur le port x et le processus R "consomme" via ce port:

s1 = P! x; Q! x; R? x; Q! x; P! x; ....
s2 = Q! x; P! x; R? x; P! x; Q! x; ...

Nous commençons donc, au paragraphe II, par définir formellement cette notion d'équivalence sur les séquentialisations d'un programme et la situer parmi les différentes notions d'équivalence proposées dans la littérature .
A partir de cette définition nous envisageons, au paragraphe III, la "couche" à ajouter à l'interprète actuel pour réaliser la vérification interactive de la propriété d'abscence de blocage. Enfin dans un dernier paragraphe nous faisons un tour d'horizon de la littérature en matière de vérification de systèmes parallèles, pour situer notre présentation.

II. EQUIVALENCE DE SEQUENTIALISATIONS

Rappelons de manière informelle la définition de la notion de séquentialisation établie au chapitre III, §II.
Une séquentialisation d'un programme est une séquence d'activation de "paquets" d'unités d'exécution de la forme:

$$(<U_0,H_0>; <U_1,H_1>; \ldots ; <U_i,H_i>)$$

où U_j est un ensemble d'unités non exclusives activées simultanément et qui débutent par une communication,

H_j désigne le n-uplet des horloges des processus du programme; la

k ème horloge indique la date de fin de l'unité du processus n° k activée dans le paquet U_j ou dans un ensemble U_m antérieur (si l'unité activée par U_m n'est pas terminée à la date d'activation de U_j).

Cette séquence est telle que:

- il existe une relation de causalité sur les "paquets" d'unités d'exécution établie à partir des pré et des post-assertions de ces unités et de leurs durées respectives,
- la "projection" de cette séquence sur chacun des processus est une des traces possibles du processus.

Dans les chapitres II et III nous avons défini explicitement les dates des actions dans l'objectif de l'interprétation et en particulier pour rendre compte de la simultanéité d'évènements. Dans ce paragraphe nous cherchons à mettre en évidence une équivalence de comportement: intuitivement deux exécutions sont équivalentes si elles induisent la même relation d'ordre sur les évènements que sont les communications. Ainsi dans cette approche les dates des évènements sont secondaires.

Par exemple on veut considérer comme équivalentes les séquentialisations

$S1 = (<U_0,H_0>; <U_1,H_1>; ...;<U_i,H_i>)$

et $S2 = (<U_0,H'_0>; <U_1,H'_1>; ...;<U_i,H'_i>)$

où pour tout j, $H'_j = H_j + \delta$ avec δ constante :
S1 et S2 sont deux interprétations d'un même programme avec un décalage de δ unités de temps.

On veut de plus, ainsi que nous l'avons vu en introduction, que l'ordre dans lequel sont activées deux opérations simultanées de même type sur un même port ne soit pas significatif puisqu'il n' a pas d'influence sur l'éventualité de blocage ou de non de blocage ultérieur.
Rappelons que deux opérations simultanées sur le même port et de type respectif "produire" et "consommer" sont regroupées en une seule action "prod-cons": elles forment un seul évènement de manière à éviter les

risques de "famine" dûs à la priorité de l'opération "produire" sur l'opération "consommer" (cf chapitre II, § II).

Nous allons donc nous intéresser aux traces de communication définies par les séquentialisations, en "effaçant" les actions internes des processus qui ne sont pas significatives pour notre problème.
Avant de définir formellement la notion d'équivalence, nous présentons un certain nombre de définitions intermédiaires.

II.1. Définitions préliminaires

Notons A_k l'ensemble des unités d'exécution du k ^{ème} processus du programme. On note ε l'unité d'exécution vide.
Notons C_k l'ensemble des instances d'opérations de communication défini par le processus n° k.

Définition 1 (idem définition 8a chapitre III):

On appelle **projection** de la séquentialisation $S = (<U_0,H_0>; <U_1,H_1>; ...;<U_i,H_i>)$ sur le processus n°k la séquence d'unités d'exécution

$$\Pi_k(U_0; U_1; ...; U_i).$$

La fonction Π_k, de profil $(A_1 \cup \{\varepsilon\} \times ... \times A_n \cup \{\varepsilon\})^* \longrightarrow A_k^*$, est définie par:

. $\Pi_k(\text{ Par}(\text{ uex}_1, ..., \text{uex}_n)) = \text{uex}_k$ si $\text{uex}_k \neq \varepsilon$

$= \wedge$ (le mot vide) sinon

. $\Pi_k(U_0; U_1; ...; U_i) = \Pi_k(U_0; U_1; ...; U_{i-1})\ \Pi_k(U_i)$.

Exemple

Soient deux processus P1 et P2 composés en parallèle. Leurs ensembles d'unités d'exécution sont respectivement :

A_1 = {e10, e11, e12, ...}

A_2 = {e20, e21, e22, ...} où e10 et e20 sont les unités initiales des processus (cf chapitre III, §II).

Soit S = ($<U_0,H_0>$; $<U_1,H_1>$; $<U_2,H_2>$) avec

U_0 = Par(e10, e20)

U_1 = Par(e11, e21)

U_2 = Par(e12, e22).

Alors $\Pi_1(U_0; U_1; U_2)$ = e10 e11 e12

$\Pi_2(U_0; U_1; U_2)$ = e20 e21 e22.

Définition 2 :

On appelle **communication** d'une unité d'exécution u l'opération de communication définie dans l'arbre de syntaxe abstraite qui représente cette unité:

- s'il s'agit d'une opération "produire" sur les ports x, y, ... on note cette action ! x, y, ... (pour alléger l'écriture et par analogie avec CSP)
- s'il s'agit d'une opération "consommer" via le port x, on note ?x cette action
- si l'unité u ne contient pas d'opération de communication, communication(u) = $\wedge$ (le mot vide).

Note : rappelons que nous ne nous intéressons pas aux valeurs communiquées et que seul le sens de la communication (émission ou réception) est utile dans le contexte de notre étude.

exemple :
Si l'unité u a pour structure abstraite <prod(a,x).aff(a,b)> alors
communication(u) = !x.
Dans l'exemple ci-dessus communication(e10) = communication(e20) = $\wedge$.

Définition 3 :

On appelle **trace de communication** (ou **histoire**) de la séquentialisation S = ($<U_0,H_0>$; $<U_1,H_1>$; ...; $<U_i,H_i>$) sur le processus n°k la séquence notée $hist_k(S)$ et définie par :

$$hist_k(S) = tr_k (\Pi_k(U_0; U_1; ...; U_i)) .$$

La fonction tr_k, de profil $A_k^* \longrightarrow C_k^*$ est définie par :

$tr_k(u)$ = communication(u), si u est une unité d'exécution
$tr_k(u_1\ u_2\ \ldots\ u_j) = tr_k(u_1\ u_2\ \ldots\ u_{j-1})\ tr_k(u_j)$.

Définition 4 :

On appelle **trace de communication** (ou **histoire**) de la séquentialisation $S = (<U_0,H_0>; <U_1,H_1>; \ldots; <U_i,H_i>)$
le n-uplet noté hist(S) et défini par :

$$hist(S) = < hist_1(S), \ldots , hist_n(S) >.$$

exemple

Supposons que les "communications" des unités e11,e12,e21 et e22 ci-dessus sont respectivement : !x, ?y, !y et ?x.
Les ensembles d'évènements de communication des processus P1 et P2 sont alors :

C1 = { !x, ?y, ...} et C2 = { !y, ?x, ...}.

On en déduit :

$hist_1(S) = tr_1(\Pi_1(U_0; U_1; U_2))$ = !x ?y
$hist_2(S) = tr_2(\Pi_2(U_0; U_1; U_2))$ = !y ?x
$hist(S) = < hist_1(S), hist_2(S) >$ = < !x ?y , !y ?x > .

Remarques

1. On pourrait de même définir des "histoires" sur les ports de communication en s'intéressant aux projections des séquentialisations sur les ports .
   Une notion analogue d' "histoire" est définie dans [HW 87]; les auteurs définissent une axiomatisation pour montrer la "linéarisabilité" des histoires des accès à des objets partagés. Notons que dans notre cas l'histoire des accès à un port de communication est forcément séquentielle de par la construction des séquentialisations.

2. Une notion analogue à celle d'histoire d'une séquentialisation est

étudiée dans [Fau 87]. L'auteur de cet article définit pour les programmes CSP deux types de comportements:
- les comportements "vectoriels" : à un instant donné du temps, supposé discret, un "paquet" d'actions est réalisé,
- les comportements synchrones par entrelacements des actions.
Une relation d'équivalence est définie sur les comportements et on montre que pour toute exécution "vectorielle" il existe une exécution synchrone équivalente : intuitivement deux comportements sont équivalents s'ils ont les mêmes ensembles d'évènements et s' ils induisent les mêmes relations de simultanéité et de précédence sur ces évènements. L'auteur montre alors que les exécutions asynchrones sont entièrement caractérisées par leurs "projections" sur les processus : notre démarche sera ici la même.

II.2. Relation d'équivalence sur les séquentialisations d'un programme

II.2.1. Définition

Pour toutes séquentialisations S1 et S2 d'un programme P = P1//...//Pn , S1 ~ S2 ⇔ hist(S1) = hist(S2).

Intuitivement deux séquentialisations sont équivalentes si l'ensemble de leurs traces de communications sur chacun des processus sont identiques : il s'agit donc d'une équivalence dite *de traces.*

Afin de situer notre choix résumons les principales notions d'équivalence proposées en matière de sémantique de programmes parallèles.

Le problème des équivalences de programmes parallèles a principalement été étudié dans le cadre des algèbres de processus introduit par Milner avec CCS [Mil 80]. L'avantage de ce cadre algébrique est de permettre de raisonner directement dans la syntaxe : un calcul de processus tel que CCS, SCCS [Mil 83], Meije [Bou 84], TCSP [BHR 84] ou RCSP [Ada 85] par exemple, est une algèbre de termes munie d'une sémantique opérationnelle sous la forme d'un système de transition et munie d'une égalité sémantique qui correspond à une congruence sur les systèmes de transition.

On peut répertorier trois grands types d'équivalence :

(1) les équivalences **de traces** : la plus simple et la plus "faible" des équivalences , que nous notons $\sim_t$, est la suivante : deux processus sont $\sim_t$ s'ils ont ont même ensemble de traces; la trace d'un processus p est la séquence des actions déclenchées dans une dérivation de p.
Par exemple les deux processus CCS , p = (a. b. nil) et q =(b. a. nil), ont même ensemble de traces = {ab, ac} et donc p $\sim_t$ q.

(2) les **bisimulations** : elles définissent la plus "forte" des équivalences que nous noterons $\sim_b$. Soit $\mathcal{R}$ une relation binaire sur les termes ; cette relation est une bisimulation si pour tout processus p et q tels que p $\mathcal{R}$ q, et pour toute transition p $\xrightarrow{a}$ p' , il existe une transition q $\xrightarrow{a}$ q' telle que p' $\mathcal{R}$ q', et et pour toute transition q $\xrightarrow{b}$ q" , il existe une transition p $\xrightarrow{b}$ p" telle que p" $\mathcal{R}$ q". on définit alors l'équivalence $\sim_b$ par p $\sim_b$ q s'il existe une bisimulation $\mathcal{R}$ telle que p $\mathcal{R}$ q : $\sim_b$ est elle-même une bisimulation (cf [AD 86]) et c'est une relation d'équivalence; on la symbolise généralement par le diagramme suivant:

$$\begin{array}{ccc} p & \xrightarrow{a} & p' \\ \sim_b & & \sim_b \\ q & \xrightarrow{a} & q' \end{array}$$

Cette équivalence forte est une congruence : on peut substituer à un terme un autre terme "fortement" équivalent dans n'importe quel contexte. Intuitivement p $\sim_b$ q si p et q ont le même arbre de dérivation et pas seulement les mêmes séquences de dérivation comme dans le cas de l'équivalence *par traces* . Par exemple les deux processus p et q ci-dessus ne sont pas $\sim_b$:

a.b.nil + a.c.nil $\not\sim_b$ a.(b.nil + c.nil)

(3) l'équivalence **observationnelle** : l'équivalence "forte" ci-dessus prend en compte toutes les actions des processus. Or on peut vouloir ne tenir compte que des actions *observables* et s'abstraire des actions *internes* non identifiables par l'observateur (terminologie CCS).

On considère alors la relation $p \overset{s}{\Rightarrow} q$, où intuitivement s est une dérivation de p menant à q dans laquelle on a effacé les actions invisibles τ.
L'équivalence observationnelle $\sim_o$ est alors la bisimulation la plus générale associée au système de transition $\overset{s}{\Rightarrow}$ ce que l'on représente par le diagramme :

$$\begin{array}{ccc} p & \overset{s}{\Rightarrow} & p' \\ \sim_o & & \sim_o \\ q & \overset{s}{\Rightarrow} & q' \end{array}$$

Cette équivalence permet en particulier d'identifier les processus p et τ.p mais ce n'est pas une congruence.

Nous verrons au §V des notions d'équivalence intermédiaires entre l'équivalence observationnelle et l'équivalence par traces, visant notamment à traiter le problème du blocage.

En ce qui concerne notre étude le modèle que nous avons retenu n'est pas défini de manière algébrique et les seules notions que nous pouvons appréhender sont celles de "trace" et de relation de simultanéité et de précédence sur les évènenents : d'où la notion d'équivalence, notée ~ que nous venons de définir et que nous illustrons par l'exemple ci-dessous.

II.2.2. Exemple

Reprenons l'exemple du programme de calcul de factorielle traité au chapitre III (§II.5).
Toute séquentialisation de ce programme est de la forme :

$$S0;\ \underset{i}{*}\ (\ <U_{i1},\ H_{i1}>\ ;\ Si\ ;\ <U_{i3},\ H_{i3}>)$$

où S0 est une des séquentialisations possibles de l'ensemble {e11, e12, e13, e21, e22, e23} des trois premières unités respectives des processus P1 et P2.

Montrons que toutes les séquentialisations S0 possibles sont équivalentes, quelles que soient les durées des unités mises en jeu.

Soient $\delta 1$ et $\delta 2$ les durées respectives des unités e11 et e21 les premières activées.

Les trois formes possibles pour S0 sont les suivantes:

(1) <u>cas $\delta 1 = \delta 2$</u>

Les unités e11 et e21 sont activées simultanément à la date $H_0=<0,0>$ et se terminent à la même date $t = \delta 1 = \delta 2$.

Dans ce cas $S0 = (\ <U_1, H_1>;\ <U_2, H_2>;\ <U_3, H_3>)$ avec

U_1 = Par(e11, e21) et $H_1 = H_0 + <t, t> = <t, t>$

U_2 = Par(e12, e22) et $H_2 = H_1 = <t, t>$: en effet e12 = prod(n1,O1) et e21 = prod(n2,O2) et l'activation de ces unités est intantanée

U_3 = Par(e13, e23) et $H_3 = H_2 = <t, t>$: comme e13 = cons(n2,O2) et e23 = cons(n1,O1), l'activation de ces unités est instantanée.

(2) <u>cas $\delta 1 < \delta 2$</u>

$S'0 = (\ <U'_1, H'_1>;\ <U'_2, H'_2>;\ <U'_3, H'_3>;\ <U'_4, H'_4>\ )$

avec

U'_1 = Par(e11, e21) et $H'_1 = H_0 + < \delta 1, \delta 2 > = < \delta 1, \delta 2 >$

U'_2 = Par(e11, ε) et $H'_2 = H'_1$:

en effet l'unité e21 n'est pas terminée à la date $\delta 1$ et seul le processus P1 peut engager une nouvelle action à cette date; l'unité e12 réalise donc la première production sur le port O1 à la date $\delta 1$

U'_3 = Par(ε, e22) et $H'_3 = < \delta 2, \delta 2 >$:

en effet après la réalisation de l'unité e12 le processus P1 est en attente de consommation de la donnée n2 sur le port O2; cette attente est levée à la date $\delta 2$ où a lieu la production sur le port O2 par le processus P2 (unité e22) : l'horloge du processus P1 progresse donc jusqu'à cette date

U'_4 = Par(e13, e23) et $H'_4 = H'_3 = < \delta 2, \delta 2 >$:

les deux consommations ont lieu simultanément à la date $\delta 2$ et sont instantanées.

(3) cas $\delta 1 < \delta 2$: symétrique du précédent

$S''0 = (<U''_1, H''_1>; <U''_2, H''_2>; <U''_3, H''_3>; <U''_4, H''_4>)$

avec

U''_1 = Par(e11, e21) $H''_1 = H'_1 = < \delta 1, \delta 2 >$

U''_2 = Par(ε, e21) $H''_2 = < \delta 2, \delta 2 >$

U''_3 = Par(e12, ε) $H''_3 = < \delta 1, \delta 1 >$

U''_4 = Par(e13, e23) $H''_4 = < \delta 1, \delta 1 >$.

Les traces de communication de ces trois séquentialisations sont respectivement:

hist(S0) = $< tr_1(\Pi_1(U_1; U_2; U_3)), tr_2(\Pi_2(U_1; U_2; U_3))>$
= < !n1 ?n2 , !n2 ?n1 >

hist(S'0) = $< tr_1(\Pi_1(U'_1; U'_2; U'_3)), tr_2(\Pi_2(U'_1; U'_2; U'_3))>$
= < !n1 ?n2 , !n2 ?n1 >

hist(S''0) = $< tr_1(\Pi_1(U''_1; U''_2; U''_3)), tr_2(\Pi_2(U''_1; U''_2; U''_3))>$
= < !n1 ?n2 , !n2 ?n1 >

Ces trois formes de séquentialisation, pour l'ensemble d'unités considéré, sont donc équivalentes.

Remarques sur l'exemple

1- On aurait pu de même traiter les séquentialisations complètes de

l'ensemble des unités du programme et montrer qu'elles sont toutes équivalentes , quels que soient les rythmes des processus : pour cet exemple une simulation par l'interprète constitue donc une validation.

2- Cet exemple est particulier dans la mesure où les exécutions des processus P1 et P2 sont forcément synchronisées sur le rythme du processus le plus lent et cela en raison du type de communication utilisé : "égalité".
L'utilisation, par exemple, du type "aléatoire" permettrait au processus le plus rapide de "prendre de l'avance" dans ses calculs par rapport au second processus; dans le cas où P1 est le plus rapide et où le port O2 est de type "aléatoire" le processus P1 consommera plusieurs fois la même valeur de la donnée n2, tant qu'une nouvelle valeur ne sera pas produite par le processus P2. Il n'y aura donc pas d'attente du processus P1, hormis pour la consommation de la première valeur de n2.

3- Toutefois cet exemple du programme de calcul de factorielle est intéressant puisqu'il montre l'utilité de la notion d'équivalence définie ci-dessus pour réduire le nombre des exécutions à simuler , lorsque le facteur temps n'est pas discriminant pour la présence ou l'abscence de blocage.

4- Nous savons que la définition d'un modèle en termes de traces des communications est mal adapté à la vérification de l'abscence de blocage: en effet deux débuts d'exécution peuvent réaliser les mêmes communications et conduire ou ne pas conduire à une situation de blocage.
Exemple:
Selon le choix activé, le processus P1 ci-dessous pourra être bloqué après l'exécution de l'opération de communication notée © en commentaire:

```
select vrai → consommer(x)  -- ©
   ou  vrai → consommer(x)  -- © ; consommer(x) ; ...
fin_select
```

Dans le cas du second choix , la seconde opération "consommer(x)" ne sera pas activable si l'exécution de la première opération a rendu la

pré-condition de consommation fausse.
Ces deux choix induisent des séquentialisations différentes du programme dont fait partie le processus P1 et dans de tels cas l'interprète devra réaliser l'exploration de toutes les branches du choix.

III. UTILISATION DE L'INTERPRETE POUR LA VERIFICATION DE LA PROPRIETE D'ABSCENCE DE BLOCAGE

Ce paragraphe se limite à la présentation d'idées et ne prétend pas définir formellement un système de vérification de la propriété d'abscence de blocage.

Nous avons vu que nous nous limitons aux programmes ne faisant intervenir que des types de communication tels que la pré-condition de consommation est soit 'vrai' soit un prédicat sur le cardinal de l'ensemble consommable A.
Rappelons que l'interprète construit une des séquentialisations possibles du programme en élaborant à chaque pas un ensemble d' unités à activer (cf chapitre III § III.2.1); cette opération de construction (appelée CONSTR au chapitre III) a pour données :

- pour chacun des n processus du programme, la prochaine unité d'exécution à activer c'est à dire celle qui satisfait le schéma de contrôle du processus (ou l'ensemble des unités possibles si elles font partie d'un choix non-déterministe et que leurs gardes sont vérifiées)
- la valeur de l'horloge du processus
- l'état des r ports de communication du programme.

Les résultats de cette opération sont:

- un ensemble noté U d'au plus n unités activables déterminées à partir des valeurs des horloges et des pré-conditions de ces unités; certaines de ces unités sont éventuellement sélectionnées de manière arbitraire lorsque plusieurs choix sont posssibles (cas d'un choix non-déterministe ou d'unités de processus distincts exclusives)
- pour chacun des n processus, la ou les unités susceptibles d'être

activées au cycle suivant.

Ainsi pour un rythme fixé des processus et un état initial donné, l'ensemble de séquentialisations d'un programme peut être élaboré par l'interprète, moyennant d'envisager tous les successeurs possibles de l'état défini à chaque cycle.

Nous étudierons , au paragraphe II.1, pour quelles configurations de programmes l'exploration exhaustive de l'ensemble des séquentialisations peut être réduite, par le biais de la relation d'équivalence.
Ensuite nous envisagerons, au paragraphe II.2, les interactions de l'utilisateur visant à réduire le coût apparent de cette validation par simulation.

III.1. Réduction de l'ensemble des séquentialisations

L'ensemble des simulations possibles d'un programme , pour un état initial donné et un rythme fixé des processus, résulte de choix réalisés par l'interprète de deux natures différentes:

- les choix indéterministes définis par le texte du programme,
- les sélections arbitraires entre les unités exclusives pour l'accès à un port de communication.

Nous étudions successivement les conséquences de ces deux types de choix sur la véfification de la propriété de non blocage.

III.1.1 Les choix non-déterministes du texte de programme

Chacune des alternatives qui définit une ou plusieurs opérations de communication doit être envisagée.
Si l'une d'elles conduit à une situation de blocage l'ensemble des autres alternatives doit être examiné avant de décider de l'état de blocage du programme.

Rappelons que l'un des intérêts de la notion de type de communication est d'"encapsuler" les différentes alternatives de consommation : ainsi les choix figurant dans le corps des processus portent-ils normalement sur les *calculs* et non les *communications* (cf les exemples traités notamment dans [Per 85]).

L'opération de choix non-déterministe ("select") n'est donc généralement pas à la source du nombre des séquentialisations d'un programme.

III.1.2. Les sélections entre unités exclusives activables simultanément

Trois cas de conflit d'accès à un port de communication sont possibles:

- deux processus distincts sont prêts à réaliser simultanément l'un une production et l'autre une consommation sur le même port : ces deux communications sont activées consécutivement mais de manière atomique avec priorité à la production,
- deux processus veulent activer chacun une opération de production (structure de convergence),
- deux processus sont prêts à réaliser simultanément une consommation (structure de divergence).

Toutes les combinaisons de ces trois cas sont possibles lorsque le nombre de processus est supérieur à deux.

Dans le premier cas il n'y a pas de choix réalisé puisque les deux opérations sont activées.

Dans le second cas le choix d'activer une production avant l'autre est sans influence sur l'éventualité de blocage; notons qu'il n'en serait pas de même si l'on s'intéressait aux valeurs produites et à la propriété de correction partielle par exemple.

Seul le cas de processus en conflit pour réaliser une consommation est à étudier.

On peut écarter le cas où le port utilisé est associé à un type de communication dont la pré-condition de consommation est 'vrai' (cf les types "très-aléatoire", "égalité-presque -partout", ... joints en Annexe).

Dans le cas d'un port pour lequel la pré-condition de consommation est un prédicat sur le cardinal de l'ensemble A, deux possibilités sont à envisager. Notons k le nombre de processus en conflit:

- les k opérations de consommation peuvent être réalisées de manière consécutive: il n'y a donc pas de blocage possible et les k! entrelacements possibles sont équivalents (traces de communication identiques).
  Un seul choix sera donc examiné par l'interprète.

- moins de k opérations peuvent être activées consécutivement, les premières activations rendant les autres opérations impossibles: tous les entrelacements conduisant à des traces de communication différentes devront être examinés.

III.2. Interactions de l'utilisateur

L'intervention de l'utilisateur est nécessaire pour guider l'interprète dans l'étude des simulations à envisager pour au moins trois types de décisions :

- décider si le choix du rythme des processus intervient sur la construction des séquentialisations ; à partir de l'essai d'un certain nombre de configurations et des traces de communication correspondantes délivrées par l'interprète, l'utilisateur doit être en mesure de cerner l'ensemble des combinaisons "type" des rythmes des processus,

- fournir à l'interprète un invariant portant sur les pré et post-conditions des unités d'exécution définies par une itération, de manière à limiter le nombre des répétitions définies par la condition d'arrêt de l'itération,

- aider l'interprète à n'envisager que les cas nécessaires lors des choix arbitraires (interprétation des choix non-déterministes ou choix entre unités exclusives) : les traces de communication exhibées par l'interprète doivent permettre à l'utilisateur de décider des branches des choix à écarter.

IV. TOUR D'HORIZON BIBLIOGRAPHIQUE

La vérification des programmes parallèles concerne l'étude de leurs propriétés que l'on peut classer en deux catégories:

- les propriétés de **sûreté** ("safety") ou invariantes : elles expriment que le "pire" ne se produira jamais; citons par exemple la correction partielle, l'exclusion mutuelle, ...

- les propriétés de **vivacité** ("liveness") qui expriment dualement que

le système doit atteindre un état satisfaisant; on classe par exemple dans cette catégorie la terminaison, l'abscence de blocage, l'équité, etc .

Nous commençons par relater les travaux "théoriques" dans le domaine avant de présenter les réalisations effectives de systèmes de preuves.

!V.1. Etudes théoriques

Les premiers travaux sur la vérification des programmes parallèles ont consisté à étendre les méthodes de preuve établies pour la programmation séquentielle. Citons par exemple les travaux de Owicki [OwG 76] qui définissent une extension de la "logique de Hoare" aux programmes parallèles avec données partagées.

De nombreuses études ont ensuite porté sur l'extension de cette même logique aux systèmes de processus communiquant par échange de messages selon le modèle CSP; les références en ce domaine sont les travaux de Apt : [AFR 80], [Apt 83], etc ... . Une synthèse est proposée dans [LaS 84] . Le point commun de ces méthodes de preuve est une approche compositionnelle conduite en deux temps :

- preuve des différents processus du programme,
- puis "coopération" de ces preuves par l'intermédiaire d'un système de règles d'inférence caractéristiques de l'expression du parallélisme considérée.

Citons également, dans cette même approche et utilisant une logique classique, les systèmes de preuve "basés sur les traces" des messages ("trace-based") : [MC 81], [GSW 87] par exemple. Enfin mentionnons, comme exemple d'extension de la logique de Hoare pour des programmes parallèles avec communication *asynchrone* , le système de preuve défini dans [Per 85] auquel nous ferons référence dans le paragraphe de conclusion.

Très vite le besoin d'exprimer des contraintes temporelles sur l'évolution de l'état du programme ont conduit à l'introduction des logiques dites *temporelles.* On trouve dans [AFE 87] une synthèse de l'utilisation de ces logiques pour la spécification, la sémantique et la vérification des programmes parallèles.

La première variante de la logique temporelle est la logique du temps *linéaire* ("linear time logic" ou LTL) initialement proposée dans [Pnu 77] : intuitivement chaque instant du temps a un unique successeur; plus précisément les structures sur lesquelles est interprétée cette logique

sont les *suites* . La seconde variante est la logique du temps *arborescent* ("branching time logic" ou BTL) [BMP 81] : chaque instant du temps peut avoir plusieurs successeurs et la structure correspondante est celle d'*arbre*. Enfin un troisième type de logique temporelle permet de raisonner sur les exécutions des programmes en termes d'ordres partiels ("partial orders temporal logic" ou POTL) [PW 84].

On montre dans [Pnu 85] qu'il n'y a pas de dichotomie entre les structures linéaire et arborescente et que chacune est adaptée à une approche différente de la formalisation des systèmes parallèles. Pour résumer la logique LTL correspond aux sémantiques par traces alors que la logique BTL correspond à l'approche "observationnelle" ou "bisimulation".

On trouve dans [Wo 88], une synthèse de l'utilisation des trois types de logique cités ci-dessus pour la vérification des programmes. Par exemple on représente généralement un programme non déterministe comme un modèle de la logique BTL et ses exécutions comme des séquences extraites de l'arbre de son interprétation; mais un tel programme peut également être vu comme un modèle générateur de la logique LTL (c'est à dire un automate de Buchi) et ses exécutions des modèles de la logique LTL.

Enfin citons d'autres travaux qui n'utilisent pas la logique temporelle mais la théorie des automates : dans [AS 85] et [MP 87] les propriétés sont décrites et vérifiées par des automates.

IV.2. Les réalisations logicielles

Les principales réalisations peuvent être classées en deux grandes catégories :

- les systèmes de validation par test,
- les systèmes de validation exhaustive.

Dans la première catégorie citons le logiciel VEDA [JMG 87] qui simule des programmes ESTELLE [CDG 86] pour la vérification de protocoles de télécommunication. Cet outil permet de vérifier que la spécification est satisfaite à chaque étape du test en simulant des évènements externes à partir d'un paramétrage du comportement du médium de communication (délai d'acheminement, taux d'erreur, etc...).

Les logiciels de la seconde catégorie réalisent une vérification

effective des programmes, à la condition qu'ils soient décrits par un automate fini, soit par l'exploration exhaustive de l'ensemble des exécutions soit par la production d'un automate réduit équivalent à l'automate initial .

Un premier exemple est celui du système CESAR (et de ses successeurs XESAR, ...) [Que 82] développé à Grenoble. Ce logiciel produit un graphe d'états (réseau de Pétri interprété) à partir de la description du programme soit dans un langage algorithmique "à la" CSP soit dans le langage ESTELLE; les propriétés de comportement souhaitées sont spécifiées dans une logique temporelle arborescente et évaluées par parcours exhaustif du graphe. Citons un logiciel similaire : le système EMC [CES 86]. Citons également le système MEC [AD 86] développé à l'université de Bordeaux qui repose la composition d'automates.

Bien sûr la limitation de ces systèmes est la taille des automates produits (actuellement plusieurs dizaines de milliers d'états) et de nombreux travaux portent sur les équivalences d'automates. Par exemple le système AUTO [Ver 87] est développé autour du calcul de processus Meije [Bou 84] à l'INRIA : ici l'automate est un terme du calcul Meije et l'équivalence utilisée pour réduire la taille de l'automate est l'équivalence observationnelle. Un système comparable est le logiciel Aldébaran développé à Grenoble [Fer 88].

Enfin mentionnons un travail original de simulation de programmes décrits dans un langage dérivé de SCCS qui prend en compte le temps et des probabilités de comportements : le système Ellipse [Car 88]; ce système, développé en Prolog, réalise une analyse des performances des comportements spécifiés et permet d'étudier les protocoles de communication.

V. CONCLUSION

Ce chapitre concerne le vaste problème de la vérification des programmes parallèles et aurait pu donner lieu à lui seul à une thèse.

L'objectif visé, suite à la réalisation de l'interprète, était l'enrichissement de ce logiciel pour permettre une validation interactive des programmes avec, a priori, comme support théorique le système de preuve défini dans [Per 85].

Ce système de preuve repose sur un modèle d'exécution des programmes

appelé "sérialisation" : une *sérialisation* est un programme séquentiel non-déterministe qui "sérialise" l'ensemble des unités d'exécution d'un système parallèle. Cette mise en séquence est construite uniquement à partir des pré et post-assertions des unités d'exécution et ne traite pas les contraintes temporelles ainsi que nous le faisons avec la notion de *"séquentialisation"* : le modèle du parallélisme sous-jacent à la notion de *sérialisation* est donc celui du non-déterminisme borné. Nous avons défini, au chapitre II, une sémantique différente rendant compte explicitement de la durée des actions atomiques et le modèle de *sérialisation* n'est donc pas cohérent avec cette sémantique ni avec le logiciel d'interprétation associé.

Par conséquent si l'intérêt de l'utilisation de l'interprète à des fins de validation était évidente, la définition formelle de cette validation restait à établir. Nous avons proposé dans ce chapitre quelques idées pour y parvenir, sur la base du modèle défini pour l'interprétation.

Remarquons tout d'abord que le fait de nous limiter à la classe des programmes où les communications peuvent être définies de manière statique correspond à la restriction adoptée par les systèmes de preuve concrets : tous traitent de programmes à états finis.

Notre objectif étant d'obtenir une vérification exhaustive de la propriété de non-blocage, et pas simplement une validation par test, nous avons focalisé notre attention sur la recherche d'une équivalence qui permette de réduire le nombre des exécutions à simuler. L'équivalence de traces que nous avons définie, bien que permettant avec interaction de l'utilisateur un certain nombre de simplifications, n'est certainement pas la mieux adaptée à l'étude de la propriété de non-blocage. Une étude plus approfondie des travaux autour de TCSP [BHR 84] et des sémantiques "par échec" ("failure équivalence") [BR 85] [ReR 86] devrait déboucher sur une notion plus adéquate d'équivalence. Toutefois l'implantation d'une telle équivalence ne nous semble pas, à l'heure actuelle, évidente.

Une perspective plus ambitieuse serait de remettre en cause le modèle d'exécution et de viser la définition d'une algèbre de processus traitant explicitement du temps et de la communication asynchrone .

Enfin notons que nous nous sommes ici limités à la propriété d'abscence de blocage et que le problème des autres propriétés reste ouvert et nécessiterait une étude approfondie des moyens de description des propriétés : logique temporelle, automates, ...?

CONCLUSION

CONCLUSION

Pour conclure ce document nous proposons de revenir sur les objectifs de cette thèse et de faire le point sur les problèmes auxquels nous avons apporté une réponse, sur ceux qui restent ouverts et sur les prolongements possibles de ce travail.
Notre objectif était d'apporter une contribution à l'étude de la programmation parallèle dans deux domaines :

- celui de la spécification des problèmes et de la conception des programmes,
- celui de la sémantique des langages de description des systèmes de processus communicants et de la validation de ces systèmes.

Pour cela, à partir des propositions du projet COMETE principalement décrites dans [Per 85], nous avons mené diverses études qui ont abouti aux résultats suivants :

BILAN

Nos propositions d'une part se situent au plan formel et d'autre part concernent une réalisation logicielle qui s'appuie sur des principes formellement définis.

Dans le premier chapitre de cette thèse nous définissons un environnement formel pour la spécification des problèmes et la conception des programmes. Il s'agit d'une part de la formalisation en termes de types abstraits de données des notions développées par le projet COMETE et d'autre part de l'étude de la validité de l'expression algorithmique des solutions en regard de la spécification. Nous montrons dans ce chapitre que les notions de *suite temporelle* et de *relation de communication* sont adaptées à la conception de systèmes parallèles qu'ils soient de nature *transformationnelle* ou de nature *réactive*. Nous montrons ensuite que le concept de *type de communication* permet une représentation algorithmique valide des solutions : la définition de la sémantique des types de communication

constitue la base de l'interprétation des communications que nous donnons au chapitre suivant.

Dans le second chapitre nous nous sommes attaché d'une part à présenter clairement les propositions du projet COMETE pour l'expression du parallélisme au niveau d'un langage et d'autre part à définir une sémantique rigoureuse qui prenne explicitement le temps en compte.
Notons que depuis le début de notre travail le langage a évolué de sa forme "impérative" vers un style "fonctionnel" dont les caractéristiques sont présentées dans [Ju 89].

La sémantique que nous avons définie constitue à notre avis un apport dans la mesure où elle ne réduit pas l'expression du parallélisme à celle du non-déterminisme borné. D'autre part le formalisme d'expression, issu des propositions de [Plo 81] et [Bou 87], nous semble élegant par rapport aux présentations classiques de sémantique opérationnelle sous la forme de systèmes de transition; remarquons que l'expression obtenue rejoint la "sémantique naturelle" de [Kah 87]. Nous sommes toutefois conscients de la distance entre notre approche et les travaux théoriques sur les modèles mathématiques du "vrai parallélisme" tels que [BC 88], [DK 83], [NPW 81] ou [Rei 85], pour ne citer qu'eux.

Dans les trois autres chapitres nous mettons en œuvre les résultats formels des deux premiers chapitres, l'objectif étant de définir un outil logiciel de simulation des programmes et d'ébaucher l'étude d'un système de validation.

Tout d'abord nous proposons au chapitre III un modèle d'exécution des programmes que nous appelons *séquentialisation*. Pour la définition de ce modèle nous avons utilisé les résultats des travaux sur la notion de critère d'observation ainsi que le travail de G.R.Perrin pour l'aspect "assertions".

Le logiciel d'interprétation que nous décrivons dans le chapitre IV constitue notre réalisation pratique qui vient compléter l'environnement de programmation COMEDIE. Cette réalisation correspond au début de notre travail de thèse; outre le "savoir-faire" en matière de "programmation objet" que nous a apporté l'écriture de l'interprète, cette réalisation a permis d'orienter notre travail vers l'étude de la sémantique des langages parallèles.

La définition d'un système de validation des programmes était le prolongement naturel de cette réalisation. Dans ce mémoire nous présentons quelques idées pour une telle définition, dans le cadre de la vérification de la propriété de non-blocage. Leur mise en œuvre fait

partie de nos perspectives que nous décrivons ci-dessous.

PERSPECTIVES

Les prolongements de notre travail, dans une perspective à brève échéance, se situent dans deux directions.
D'une part la poursuite de l'étude d'un système de validation des programmes que nous avons abordée ici constitue un objectif de travail immédiat dont la première étape concerne l'étude d'une équivalence appropriée au modèle défini pour l'interprétation. Nous avons cité à ce sujet les travaux autour de TCSP [BR 85], [ReR 86] notamment; l'étude approfondie des réalisations de logiciels de preuve récentes [Fer 88], [Ver 87] devrait également ouvrir de nouvelles voies. De plus au-delà de l'étude de la propriété de non-blocage, l'étude d'autres types de propriétés constitue un thème de recherche ouvert .
D'autre part notre travail, déconnecté de l'évolution du projet COMETE pendant la phase de rédaction de ce manuscrit, doit maintenant rejoindre les propositions récentes du projet. Le langage actuel défini dans [Ju 89] repose sur les idées suivantes :

- le parallélisme est vu comme l'application d'une fonction sur des suites temporelles,
- l'expression est séparée en une partie fonctionnelle qui décrit les processus définissant les valeurs des suites et une partie opérationnelle pour rendre compte du temps,
- la communication est exprimée par des opérations sur une structure de données (notion de type de communication).

Ce langage sera développé à l'aide d'outils formels tels que "TYPOL" et "CENTAUR". Nos résultats sur, la sémantique en particulier, devraient pouvoir être repris dans le cadre de ce nouveau langage et conduire à de nouveaux développements.

BIBLIOGRAPHIE

BIBLIOGRAPHIE

[AB 84] : D.AUSTRY, G.BOUDOL : "Algèbre de processus et synchronisation". Theor. Comp. Sci. 30 (1984). p 91-131

[AB 85] : J.M. ADAMO : "Introduction du temps asynchrone dans RCSP" Rapport Université de Lyon (1985)

[AD 86] : A.ARNOLD, A.DICKY : "An algebraic characterization of transition system equivalences". Techn. Rep. I-8603. Université de Bordeaux I. (1986)

[Ada 80] : "Formal definition of the Ada programming Language". Rapport Honewell Inc. Cii Honewell Bull (1980)

[ADJ 77] : J.A. GOGUEN, J.W. TATCHER, E.G. WAGNER, J.B. WRIGHT "Initial Algebra Semantics and continuons algebras". JACM vol 1. Janvier 77 . p 68-95.

[AFE 87] : E.ANDUREAU, L.FARINAS del Cerro, P.ENJALBERT : "théorie de la programmation et logique temporelle". Techn. Science Inf. 6,6 (1987) p 527-540 et 7,2 (1988) p 181-200.

[AFR 80] : K.APT, N.FRANCEZ, W.P. de ROEVER : "A proof System for Communicating Sequential Processes". TOPLAS, ACM vol.2 n° 3 (1980) .p 359-385

[AN 82] : A.ARNOLD, M.NIVAT : "Comportements de processus". Colloque AFCET "Les Mathématiques de l'Informatique" p.35-68 (1982)

[Apt 83] : K. APT : "Formal Jusitification of a Proof System for

Communicating Sequential Processus".
JACM, vol.30, n° 1 (janvier 1983) p.197-216

[AS 85] : B. ALPERN, F. SCHNEIDER : "Verifying Temporal Properties without using Temporal Logic".
Rap. n° TR 85-723, Dept. of Computer Science, Cornell University (Décembre 1985)

[Bak 69] : J.W. de BAKKER : "Semantics of programming languages". Advances information systems science. Plenum Press, 2, p.173-227

[BC 88] : G. BOUDOL, I.CASTELLANI : "Concurrency and Atomicity" Theor. Comp. Sci. n°59 (1988)

[BCD 87] : P. BORRAS, D. CLEMENT, T. DESPEYROUX, J. INCERPI, G. KAHN, B. LANG, V. PASCUAL : "CENTAUR : the system".
Rapport INRIA n° 777 (décembre 1987).

[Ber 85] : G. BERTHOMIEU : "le langage LCS et son interprète. Une implantation expérimentale de CCS". Actes du 1er Colloque C^3. Angoulême 1985. Ed. A. Arnold.

[Ber 86] : P. BERTRAND : "Présentation des normes graphiques GKS". Rapport INRIA n° 70 (1986).

[BHR 84] : S. BROOKES, C.A.R. HOARE, A.W. ROSCOE : "A Theory of Communicating Sequential Processes". JACM 31. (1984).

[BMP 81] : M. BEN-ARI, Z. MANNA, A. PNUELI : "The Logic of Nexttime ". 8^{è} ACM Symp. on Principles of Programming Languages. p.164-176 (janv. 1981).

[Bof 87] : F. BOUSSINOT :" Une sémantique du Langage Esterel" .
Actes du 2 ème Colloque C^3. Ed. A. Arnold. Mai 1987. p 65-79.

[BOU 85] : G. BOUDOL : "Calcul de processus et vérification".
Actes du 1er Colloque C^3. Angoulême 1985.

[BOU 87] : G. BOUDOL : "Communication is an abstraction". Actes du 2 ème Colloque C^3. Angoulême 1987. p.45-63.

[BR 85] : S.BROOKES, W. ROSCOE : "An improved failures model for Communicating Processus". Lecture Notes in Computer Science 197. (1985).

[Bü 62] : J.R. BUCHI : "On a Decision Method in Restricted Second Order Arithmetic". Proc Int. Cong. Logic, Method and Phil. Sciences.1960. Standford univ. Press 1962. p 1-12.

[Car 86] : D. CAROMEL. "Conception méthodique et progressive des relations de communication". Rapport de DEA - Université Nancy I -1986.

[Car 88] : V. CARCHIOLO : "A Simulator Tool for the behavioral timed language LIPS". Proc. of the Nineteenth annual Modeling and Simulation Conference. Pittsburgh 1988.

[CDG 86] : J.P COURTIAT, P. DEMBINSKI, R. GROZ, C. JARD : "ESTELLE: un langage ISO pour les algorithmes didtribués et les protocoles". Tecn. Science Inf. 6,2. (1986).

[CES 86] : E.M. CLARKE, E.A. EMERSON, A.P. SISTLE : "Automactic Verification of finte state concurrent systems using temporal logics specifications". ACM Trans. Prog. Lang. Syst. 8 . p.244-263. 1986

[CFM 83] : I. CASTELLANI, P. FRANCESCHI, U. MONTANARI : "Labelled Event Structures : a Model for Observable Concurrency". In formal Description of Programming Concepts 2 (D. BJORNER Ed) (1983). p 383-400.

[CH 74] : R. CAMPBELL, A. HABERMANN : "The specification of process synchronisation by path expressions". Lecture Notes in Computer Science . p 89-102, Vol. 16. Springer Verlag 1974.

[Cha 85] : J. CHAILLOUX : "Manuel de référence LE_LISP 15 ". Rap. tech. INRIA Février 1985

[Cha 86] : S. CHANUDET : "Une expression descriptive des types de communication" . Rapport de DEA - Université de Franche Comté - Septembre 1986.

[CPH 87] : P. CASPI, D. PILAUD, N. HALBWACKS, J.A. PLAICE : "LUSTRE : a declarative language for programming synchronous systems". Proceedings of the 14th ACM Symposium on Principles of Programming languages - ACM. Janvier 87.

[DeF 79] : J.C DERNIAME, J.P FINANCE : "Types Abstraits de Données : Spécification , utilisation et réalisation". Ecole d'été de l'AFCET - Monastir 1979. Rap. CRIN 79.E.57.

[Des 86] : T. DESPEYROUX : " Typol : A formalism to implement Natural Semantics". Rapport Inria n° 94 (1986).

[DHK 80] : V. DONZEAU-GOUGE, G. HUET, G. KAHN, B. LANG : "Programming environments based on structured editors : the MENTOR experience". Rapport INRIA n° 26 (1980)

[DK 83] : P. DARONDEAU, L. KOTT : "On the observationnal Semantics of Fair Parallelism". Lecture Notes in Computer Science n° 154.1983. (ICALP83)

[Egl 87] : M.C. EGLIN : "Contribution à la réalisation d'un interprète de processus communicants". Rapport de DEA. Universite de Franche-Comté (1987).

[EJ 88] : M.C. EGLIN, J. JULLIAND : "Interprétation parallèle asynchrone de systèmes d'équations récurrentes" . Rapport de recherche CRIN et LIB. Soumis à publication (1988)

[Fer 87] : J.C. FERNANDEZ : "Aldébaran : un système de vérification par réduction de processus communicants". Thèse de doctorat. Université J. Fournier. Grenoble. 1988.

[Fau 87] : H. FAUCONNIER : "Sémantique asynchrone et comportements infinis en CSP". Theor. Comp. Sci. n° 54 (1987)

[FIJ 8O] : J.P. FINANCE - J. JARAY : "Toward a methodology to specify and construct concurrent programs" . Rapport CRIN 80-P-09.

[Fin 79] : J.P. FINANCE : "Etude de la construction des programmes: méthodes et langages de spécification et de résolution de problèmes". Thèse d'Etat . Université Nancy I . 1979.

[FiO 83] : J. P. FINANCE, M. S. OUERGHI : " On the Algebraic specification of Concurrency and Communication". Proc ICPP 1983.

[Gau 80] : M. C. GAUDEL : "Génération et preuve de compilateurs basées sur une sémantique formelle des langages de programmation". Thèse d'Etat. Inst. Nat. Polytec. de Lorraine (1980)

[GBG 84] : P. LE GUERNIC, A. BENVENISTE, T. GAUTHIER : "Signal : un langage pour le traitement du signal". Traitement du signal, n°1, 1984.

[GKS 86] : Manuel de référence TELMAT du système graphique GKS.

[GSW 87] : D. GRIES, F. SCHNEIDER, J. WIDOM : "Completeness and Incompleteness of Trace-Based Network Proof System". Proc. 14 th ACM Symp. on Programming languages. p 27-38. Janv.87

[GuH 78] : J.V GUTTAG, J.J HORNING : "The algebraic specification

of abstract data type". Acta Informatica . vol. 10. 1978.

[HaP 85] : D. HAREL, A. PNUELI : "On the Development of reactive systems"- Nato Asi Séries : "Logic and Models of Concurrent Systems"- Springer Verlag 1985.

[Hdn 83] : M. HENNESSY, R de NICOLA : "Testing Equivalences for Processes" . Lecture Notes in Computer Science n° 154 (1983).

[HM 85] : M. HENNESSY, R. MILNER : "Algebraic Laws for Non-determinism and Concurrency". JACM, vol 32, n°1. p137-161 (1985).

[Hoa 78] : Car. HOARE : "Communicating Séquential Processes". CACM 21 (1978).

[HW 87] : M. P HERLIHY, J.M. WING : "Anxious for concurrent Objects". Proc. 14 th ACM Symposium on Principles of Programming Languages (1987).

[Hul 84] : J. M. HULLOT. Manuel de référence CEYX. Version 15. Rapport INRIA (1984).

[JJa 88] : J. JARAY : "Timed specifications for the development of real-time systems". Symp on formal techniques in Real-time and Fault Tolerance Systems. Warwick - Septembre 88.

[JJu 81] : J. JULLIAND : "Expression des communications entre processus d'un programme parallèle par des types abstraits".Thèse 3ème cycle. Université de Franche Comté . Mai 1981.

[JJu 83] : J.JULLIAND : "Spécification algébrique de la communication entre processus parallèles" - TSI vol. 2 n° 4 - 1983.

[JKP 86] : J. JULLIAND, A. KOUKAM, GR. PERRIN : "Une méthode de dérivation de programmes Ada". Rap. CRIN n° - 1986.

[JMG 87] : C. JARD, J.F MONIN, R. GROS : "Development of Veda, a prototyping tool for Distributed algorithms". IEEE Trans. Softw. Eng. 14,3 (1987).

[JMP 86] : J. JULLIAND, M. MARMONIER, G.R PERRIN : "La conception des programmes dans le projet COMETE". Rap. CRIN n° 87-R-047 - 1986.

[JP 81] : J. JULLIAND - GR PERRIN : "Design and Development of Concurrent Programs" - Congrès Compar - Nüremberg . Juin 1981.

[Jor 84] : P. JORRAND : "PF2 : Functionnal Parallel Programming based on term substitution". Rapport de Recherche LIFIA, 15. (1984)

[Jul 89] : J. JULLIAND : "A functionnal language for parallel programming". Rapport crin 89R-001. (1981).

[Jul 85] : J. JULLIAND : "COMEDIE : manuel d'utilisation". Rapport CRIN 85-R-075.(1985)

[JuM 86] : J. JULLIAND - M. MARMONIER : "COMEDIE : outils de développement de systèmes parallèles". 3ème congrès Génie Logiciel - Versailles -Mai 1986.

[JuP 85] : J. JULLIAND - GR PERRIN : "Towards a methodology for Concurrent Programming" . Int. Conf. on Parallel Computing - Berlin 85. Elsevier Science Publishers. (1985)

[Kuh 87] : G. KAHN : "Natural Semantics". Rapport Inria n° 601. (1987).

[Kel 76] : R.M. KELLER : "Formal verification of parallel programs". CACM 19. p 371- 384.

[Lam 85] : L. LAMPORT. "An axiomatic Semantics of Concurrent

Programming Languages". Nato Asi Series, Vol FI3. Logics and Models of concurrent Systems. Ed. K.R. Apt. (1985)

[Lam' 85] : L. LAMPORT : "On Interprocess Communication". Rapport DEC. (Dec. 85).

[LAS 78] : P. LAUER, W.W. SHIELDS : Abstrait spécification of resource accessing disciplines : adequacy, starvation, priority and interrupts". SIGPLAN Notices. Vol 13 n° 12. (1978).

[LaS 84] : L. LAMPORT, F;B. SCHNEIDER : "The Hoare Logic of CSP and all that". ACM Trans. Prog. Lang. Syst. 6,2 (1984).

[Lec 86] : P. LE CERTEN : "Conception et mise en oeuvre d'un langage impératif pour la programmation parallèle". Thèse de doctorat. Université de Rennes 1. (avril 1986).

[Liv 78] : C. LIVERCY : "Théorie des programmes. Schémas, preuves, sémantique". Dunod Informatique (1978).

[LIZ 75] : B. LIZKOV, S. ZILLES : "Specification techniques for Data abstractions". Proc. Int. Conf on Reliable Software (1975).

[LTS 79] : P. LAUER, P. TORRIGIANI, MW. SHIELD : "COSY. A Specification Language Based on Paths and Processes". Acta Informatica n°12, p 109. 158. (1979)

[May 83] : D. MAY : "The Occam Language". Sigplan Notice 18,4. (1983)

[MC 81] : K. M. CHANDY, J. MISRA : "Proofs of Networks of Process". IEEE Transactions on Software Engineering, Vol. SE. 7, n°4 (juillet 81)

[Mel 83] : B. MELESE : "Mentor : l'environnement Pascal". Cours INRIA ."Les éditeurs dirigés par la syntaxe". Aussois.

(Avril 83).

[Mey 85] : B. MEYER : "Etapes sur le chemin du génie logiciel". Thèse d'état - Université de Nancy I - 1985.

[Mil 80] : R. MILNER : "A Calculus of Communicating Systems". Lecture Notes in Computer Science. Vol. 92. Springer Verlag 1980.

[Mil 83] : R. MILNER : "Calculi for Synchrony and Asynchrony". Theor. Comp. Sci. n° 25 (1983). p 267- 310.

[Mil 88] : R. MILNER : "Operationnal and Algebaic Semantics of Concurrent Processes". Rapport n° 88-46, Laboratory for Fondations of Computer Science. Université d'Edimbourg. (Fev . 88).

[MP 87] : Z. MANNA. A. PNUELLI : "Specification and Verification of Concurrent Programs by ∀ - Automata". Proc 14 th ACM Symp. on principles of Programming Languages. p 1-12. Janv 87.

[NPW 81] : N. NIELSEN, G. PLOTKIN, G. WINSKEL : "Petri nets, Event Structures and Domaines". Theor. Comp. Sci. n°13 (1981) . p 85-108.

[OwG 76] : S.OWICKI, D. GRIES : "An Axiomatic Proof technique for parallel programs". Acta Informatica 6. (1976).

[Pai 78] : C. PAIR : "La conception de programmes" - RAIRO Informatique Vol. 13 - n° 2 - 1979.

[Per 85] : GR. PERRIN : "La communication : un outil pour la spécification, la construction et la vérification de systèmes parallèles".
Thèse d'Etat - Université de Nancy I - Octobre 1985.

[Plo 81] G. PLOTKIN : "A Structural approach to operational Semantics". Report Daïmi FN. 19. Computer Science Dpt. Aarhus Univ. (1981)

[Pnu 77] A. PNUELI : " The Temporal Logic of programs". Proc. of the 8th Symp. on Foundations of Computer Science. Providence. Nov. 1977. p 46-57.

[Pnu 85] A. PNUELI : "Linear and Branching Structures in the Semantics and Logics of Reactive Systems". Lecture Notes in Computer Science n° 194. (1985). (ICALP 85)

[PW 84] : A. PNUELI, P. WOLPER : "A Temporal Logic for reasoning about Partially Ordered Computations". Proc. 3 rd ACM Symp. on Princ. of Distributed Computing. Vancouver 1984. p 28-37.

[Que 82] : J. P. QUEILLE : "Le système CESAR : Description, spécification et analyse des applications réparties". Thèse de docteur Ingénieur. Univ. Scient. et Médicale de Grenoble (Juin 1982).

[Ra 70] : M.O. RABIN : "Weakly Definable Relations and Special Automata". Proc Symp. on Mathematical Logic and Foundations of Set Theory. Y. Bar. Hillel ed. p1-23. (1970).

[Ray 81] : M. RAYNAL : "Contribution à l'étude de la coopération entre processus dans les langages et les systèmes informatiques". Thèse d'Etat - Université de Rennes - 1981.

[Rei 84] : W. REISIG : "Partial Order Semantics versus Interleaving Semantics for the CSP-like Languages and its impact on Fairness". Lecture Notes in Computer Science n°172 (1984) (ICALP 84).

[Rei 85] : W. REISIG : "On the Semantics of Petri Nets". In Formal Models in Programming. North-Holland (1985). p 347-372.

[Rem 82] : J.L REMY : "Etude de systèmes de Réécriture Conditionnelle et applications aux types abstraits

algébriques" . Thèse d'Etat. Inst. Nat. Polytech. de Lorraine - 1982.

[ReR 86] : G.M. REED, A.W. ROSCOE : "A Timed Model for Communicating Sequential Processes" . Lecture Notes in Computer Science n° 226. p 314-321. (1986)

[Ros 85] : A. W. ROSCOE : "Denotational Semantics for Occam". Lecture Notes in Computer Science n° 197. p 306- 329. 1985.

[SiF 83] : J. SIFAKIS : "Property - preserving homomorphisms of Transition Systems". In Logics of programs. Lecture Notes in Computer Science n°164. (1983).

[Tof 88] : C. TOFTS : "Temporal Ordering for Concurrency" . Rapport LFCS. Université d'Edimbourg. (1988).

[Ver 87] : D.VERGAMINI : "Vérification de réseaux d'automates finis par équivalence observationnelle : le système AUTO ". Thèse de Dotorat. Université de Nice. 1987.

[VW 86] : M. Y VARDI, P. WOLPER : "Automata - Theoretic Techniques For Model Logics of Programs". Journal of Computer and System Science. Vol n°2. p 183-221 (Avril 86).

[Win 83] : G. WINSKEL : "Synchronisation Trees". Lecture Notes in Computer Science n° 154. (ICALP 1983).

[Wo 88] : P. WOLPER : On the Relation of Programs and Computations to Models of Temporal Logic. Rapport Université de Liège.
(Fev. 88)

ANNEXES

ANNEXE 0 : DEFINITION DE LA GESTION D'UNE UNITE DE DISQUE

1. Spécification du problème

L'énoncé du problème [Abs 79] peut se résumer ainsi : la gestion du disque doit servir toute requête d'accès; l'accès à une piste est défini par son numéro ; les pistes sont numérotées en ordre croissant de l'extérieur vers l'intérieur du disque; deux requêtes de la même piste doivent être ser- vies selon leur ordre d'émission.

Ce problème peut être ainsi spécifié :

<u>données</u> : d , suite infinie de requêtes d'accès :
$d = (d_0, d_1, \ldots, d_i, \ldots)$

<u>résultats</u> : s , suite infinie des services :
$s = (s_0, s_1, \ldots, s_j, \ldots)$

telle que

il existe une bijection ϕ du domaine $S = \{0, 1, \ldots, i, \ldots\}$ de s sur le domaine $D = \{0, 1, \ldots, i, \ldots\}$ de d vérifiant :

$\forall\ j, j' \in S,\ s_j = s_{j'}$ et $j < j' \Rightarrow \phi(j) < \phi(j')$

<u>Figure 1</u> : énoncé initial du problème

A cette spécification correspond une solution triviale qui consiste à servir les requêtes d'accès selon leur ordre d'émission. La bijection ϕsolu- tion est l'"identité" :

$$\forall\ j \in S\ ,\ \phi(j) = j\ .$$

Cette solution, théoriquement correcte, ne prend pas en compte la contrainte sous-jacente à ce type de problème : l'**efficacité** de la solution.

L'énoncé initial du problème doit donc être renforcé avec la contrainte suivante : les déplacements mécaniques du bras d'accès aux pistes doivent être minimaux.

Toute solution à ce nouveau problème passe par la définition d'une application ϕ qui prenne en compte l'**état** du système **au moment** de choisir la requête à servir.

Une telle spécification définit le terme général de la suite résultat, de la façon suivante :

<u>données</u> : d, suite infinie de requêtes d'accès :
$d = (d_0, d_1, \ldots, d_i, \ldots)$

<u>résultats</u> : s, suite infinie des services :
$s = (s_0, s_1, \ldots, s_i, \ldots)$

<u>définition</u> : $\forall j \in S$ (domaine de s), $\exists i \in D$ (domaine de d)
<u>tel que</u> $s_j = d_i$ <u>et</u> $i = \phi(E_j)$
<u>avec</u> E_j état courant du système
<u>et</u> ϕ application telle que "la contrainte de déplacements minimaux du bras soit vérifiée"

A partir de cette spécification, la **conception** d'un système solution du problème réside dans la définition de l'état "courant" E_j et de l'application ϕ .

L'état du système est décrit par la donnée de trois arguments :
- la position courante du bras, c'est-à-dire le numéro dela dernière piste accédée
- la direction courante de déplacement du bras : vers l'intérieur ou l'extérieur du disque (notée dir)
- l'ensemble des requêtes non encore servies (noté P).

La définition de la prochaine requête à servir, pour l'état E_j, devient alors : $i = \phi(s_{j-1}, dir_j, P_j)$

La fonction ϕ qui minimise les déplacements, connue sous le nom de "stratégie de l'ascenceur", est définie de la manière suivante :

- s'il existe des requêtes dans la direction courante de déplacement du bras, le prochain service satisfait celle concernant la piste la plus proche,

- la direction du déplacement est inversée si la position

courante du bras est la première ou la dernière piste ou bien s'il n'existe pas de requête concernant une piste accessible dans la direction courante et s'il en existe dans la direction opposée,

- enfin, deux requêtes concernant la même piste sont servies selon leur ordre d'arrivée.

Nous n'explicitons pas davantage cette fonction ϕ : le but de cette présentation est seulement de montrer le type de définition auquel nous sommes conduits, qui s'exprime en termes d'une **relation de communication** vérifiée par la suite produite des requêtes, et la suite consommée des services.

Pour nous focaliser sur la construction d'une solution, sans nous perdre dans la complexité du problème, nous traitons une variante d'une partie de ce problème : supposons fixée la direction courante (par exemple vers l'intérieur) et supposons ne traiter que les requêtes concernant les pistes dans cette direction. La spécification de ce problème simplifié prend la forme suivante :

$$i = \phi(s_{j-1}, P_j)$$

où la fonction ϕ peut être définie par l'énoncé de la figure 2 ci-dessous à l'aide des notations suivantes :

soit n un numéro de piste de N, multi-ensemble de numéros de piste,

$SUP(n, N) = \{m \in N / m > n\}$

$INF(N) = \{n \in N / m \in N \Rightarrow m \geq n\}$;

soit I un ensemble non vide d'entiers naturels, on note :

MIN(I) un élément de I tel que $j \in I \Rightarrow j \geq MIN(I)$

MAX(I) un élément de I tel que $j \in I \Rightarrow j \leq MAX(I)$

<u>données</u> : d, suite infinie de requêtes d'accès
d= $(d_0, d_1, \ldots, d_i, \ldots)$ dont on note D le domaine
<u>résultats</u> : s, suite infinie des services
$s = (s_0, s_1, \ldots, s_i, \ldots)$ dont on note S le domaine
telle que
$\forall j \in S\ s_j = d_i$ avec
$i = MIN(\{k \in D / d_k \in INF(SUP(s_{j-1}, P_j))\})$
-- P_j est l'ensemble des requêtes non servies au
-- "moment" du jème service : on ne considère
-- que les requêtes versl'intérieur du disque

-- on note $s_{-1} = n_0$, le plus petit numéro de piste.

Figure 2 : Enoncé du problème simplifié

Pour expliciter cet énoncé il reste à définir la suite P des ensembles de requêtes non servies : c'est ici qu'interviennent les instants des éléments de la suite d des demandes d'accès et le la suite s des services.

2. La relation de communication "asc"

Ainsi complété l'énoncé se présente alors comme la définition d'une relation de communication vérifiée par la suite de demandes d et la suite de services s, définies comme des suites temporelles. Intuitivement une suite temporelle est une suite de valeurs datées, c'est-à-dire de couples <valeur, instant> : on note sj.valeur et sj.instant les 2 composantes du jème terme de la suite s.

Cet énoncé est complètement décrit dans la figure 3 ci-dessous: cette définition utilise les opérations du type "suite temporelle" définies au chapitre I(prem, conc,entre,...)...

données : r suite temporelle des requêtes d'accès (de domaine R et de type no-piste)

résultats : s : suite temporelle des services (de domaine S et de type no-piste) telle que (r,s) ∈ *asc*, relation de communication caractérisée par la relation τ sur R*S, et définie par :

$\forall j \in S, \exists i \in R$ tel que
- . s_j.valeur = d_i.valeur
- . s_j.instant ≥ d_i.instant
- . τ(d,s,i,j) avec τ définie par

définitions intermédiaires

. τ(d,s,i,0) ≡ (i = prem(P0))

-- prem: délivre le premier terme d'une suite

. P_0 = plus-gd(n_0,Q_0)

Q_0 = jusqu'à(s_0.instant)

-- plus-gd(v,s): rend la suite extraite de s composée des
-- termes supérieurs à la valeur v
-- n0 position initiale du bras

. $\forall j \in S - \{0\}$

τ(d,s,i,j) ≡ (i = prem(Pj))

P_j = plus-gd(s_{j-1}.valeur, Qj)

Q_j = reste(Q_{j-1}) conc entre(d,s_{j-1}.instant,s_j.instant)

-- Qj : suite des valeurs de d d'instants inférieurs à
-- l'instant du jème service non encore servies
-- reste: rend une suite privée de son premier terme
-- conc : concatène deux suites
-- entre : extrait d'une suite temporelle la suite des
-- valeurs d'instants compris entre les deux dates
-- paramètres.

plus-gd(*s*,*v*)= si v < der(s) alors plus-gd(début(s),*v*)* der(s)
sinon plus-gd(début(s),*v*) fsi

Figure 3: première solution: la relation de communication *asc*

Cet énoncé est une définition *statique* qui admet de nombreuses

solutions s. En effet la définition des instants de la suite s n'est pas explicitée : la seule contrainte s_j.instant ≥ d_i.instant ne suffit pas à construire la suite s. On obtient une définition constructive en prenant en compte la suite des requêtes émises par le système, c'est-à-dire la suite des instants auxquels le gestionnaire du disque est prêt à servir une demande. Cette suite de requêtes dépend des deux suites d et s et des carctéristiques de l'unité de disque(vitesse de déplacement,...). (cf chapitrel)

Nous n'expliciterons pas ici la définition constructive obtenue en intégrant la suite des requêtes,notée c, et qui est similaire de l'énoncé [figure3] en substituant dans les définitions des suites P et Q les dates de c aux dates de s.

3. Représentation de la relation de communication "asc"

La relation de communication "*asc*" définie au paragraphe précédent est représentée par le type de communication TYPECOM (no-piste, ascenceur) caractérisé par les opérations suivantes :

```
pré_cons (<SP, A, SC>) = si non vide? (SC) alors non vide? (A)
   non vide? (Sup(accès(SC, max(SC)), A)) fsi

val_cons (<SP, A, SC>) = si vide? (SC) alors
   accès (A, min(inf(sup(n0, A)))) sinon
   accès (A, min(inf(sup(extraire(SC, max(SC)), A)))) fsi

post_cons (<SP, A, SC>) = si vide? (SC) alors
   retirer (A, min(inf(sup(n0, A)))) sinon
   retirer (A, min(inf(sup(extraire(SC, max(SC)), A)))) fsi
```

où *n0* représente la plus petite valeur possible de no-piste,

inf et *sup* sont des fonctions intermédiaires, dont la définition doit être ajoutée à la spécification du type "*asc* ", et qui représentent les fonctions INF et SUP introduites dans la spécification (de même que les opérations *min* et *max* du type TABLE(cf chapitre I §I.2) réalisent les fonctions MIN et MAX).

La définition du type de communication *"lift"* qui réalise la solution au problème complet initial est donné en Annexe IV.2.

Annexe IV.1 :

Syntaxe abstraite du langage (Description CEYX)

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;; syntaxe abstraite de notre   ;;;;;
;;;;; langage comete               ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;; le super univers racine de toute la hierarchie des univers;;;;;
;;;;; pour definir les commentaires sur chaque noeud           ;;;;;
(deftree lu (dc~string "")(fc~string ""))
     ;;;;; dc:com pre ; fc : com post

;;;;; programme et declarations    ;;;;;

(defmodel l-string(Predicate fl-string))
(de fl-string
  ;liste platte de nom d'objets
  (l)
    (while (and l(stringp(car l)))(nextl l))
    (null l))

(deftree {lu}:decls-prog)
(deftree {decls-prog}:decl-prog)
(defcons {decls-prog}:odecls-prog sons~(List decl-prog))
(deftree {decl-prog}:prog (sem-ok~symbol nil)
          (tc-v~l-string nil)(tc-ut~l-string nil))
  ;*-v objet * visibles; *-ut  : objet * utilises
  ;*=(tc,var,typ,entree,sortie,port)
(deftree {lu}:nomvars)
(deftree {nomvars}:ident (val~identificateur)(vmodu~identificateur))
     ;;;;; ident est un sous-type de nomvars avec les attributs
     ;;;;; val :nom de l'identificateur
     ;;;;; vmodu: nom du module(prog ou proc) où est declare l'idf
(deftree {lu}:sisole)
(deftree {sisole}:decls)
(defcons {prog}:oprog sons~(Vector ident decls-prog ports))
(defcons {ident}:oident)
(defcons {ident}:oid-int) ;identificateur pour les types de com.
(modelunion mdecl(metavar decl))
(defcons {decls}:odecls sons~(List mdecl))

(defmodel {string}:identificateur(Predicate fidentificateur))
(de fidentificateur    (id)
    (or(omatchq ident_minus id)(omatchq ident_majus id)))
     ;;;;; ident_minus et ident_majus defini dans spes.l ;;;;;

;;;;; declaration de variables ;;;;;


(deftree {decls}:decl)
(deftree {decl}:dvars)
(deftree {dvars}:dvar)
(deftree {lu}:type)
(deftree {type}:typpredef)
(modelunion mdvar(metavar dvar))
(defcons {dvars}:odvars sons~(List mdvar))
(defcons {dvar}:odvar sons~(Vector nomvars type-ident))
(defcons {nomvars}:onomvars sons~(List ident))

;;;;; declaration de types ;;;;;
(deftree {decl}:dtypes)
(deftree {dtypes}:dtype)
(modelunion mdtype(metavar dtype))
(defcons {dtypes}:odtypes sons~(List mdtype))
(defcons {dtype}:odtype sons~(Vector ident type-ident))
```

```
(defcons {typpredef}:oentier)
(defcons {typpredef}:oreel)
(defcons {typpredef}:obooleen)
(defcons {typpredef}:ocar)
(deftree {lu}:intervals)
(deftree {intervals}:interval)
(defcons {interval}:ointerval sons~(Vector cst-num cst-num))
(defcons {intervals}:ointervals sons~(List interval))
(deftree{type}:typcons)
(defcons {typcons}:otab sons~(Vector intervals type-ident))
(deftree {sisole}:champ)
(defcons {champ}:ochamp sons~(Vector nomvars type-ident))
(defcons {typcons}:ostruct sons~(List champ))
(modelunion type-ident(type ident metavar))
;;;;; ce type-ident permet d'accepter un op{rateur de phylum ident ;;;;;
;;;;; pour tous les fils de phylum type @ condition de remplacer type ;;;;;
;;;;; type par type-ident ;;;;;
(modelunion cst-ident(ident cst-num cst-alpha  arithsuite))

;;;;; declaration de processus ;;;;;

(deftree {sisole}:specif)
(deftree {sisole}:corps)
(deftree {lu}:entres)
(deftree {lu}:sorties)
(deftree {entres}:entre)
(deftree {sorties}:sortie)
(deftree {lu}:ports)
(deftree {decl-prog}:process (sem-ok~symbol nil)(var-v~l-string nil)
          (var-ut~l-string nil)(typ-v~l-string nil)(typ-ut~l-string nil)
          (fonct-v~l-string nil)(fonct-ut~l-string nil)(entree-v~l-string nil)
        (entree-ut~l-string nil)(sortie-v~l-string nil)(sortie-ut~l-string nil)
          (port-v~l-string nil)(port-ut~l-string nil)(vmodu~identificateur))
;;;;;; vmodu :nom du prog englobant
(defcons {process}:oprocess sons~(Vector ident specif  corps))
(defcons {specif}:ospecif sons~(Vector entres sorties))
(modelunion mentre(metavar entre))
(modelunion msortie(metavar sortie))
(defcons {entres}:oentres  sons~(List mentre))
(defcons {sorties}:osorties sons~(List msortie))
(defcons {entre}:oentre sons~(Vector nomvars ident ident))
(defcons {sortie}:osortie sons~(Vector nomvars ident nomvars))
(deftree {ports}:port)
(defcons {ports}:oports sons~(List port))
(defcons {port}:oport sons~(Vector nomvars ident))

;;;;; corps d'un processus ;;;;;

;(deftree {lu}:instrs-p)
;(deftree {instrs-p}:instr-p)
;(deftree {instr-p}:instrs)
(deftree {lu}:instrs(vmodu~identificateur))
(defcons {corps}:ocorps sons~(Vector decls instrs ))
(modelunion minstr(metavar instr))
(defcons {instrs}:oinstrs sons~(List minstr))
(deftree {instrs}:instr)
;(defcons {instr}:ouexec sons~(Vector nomvars));inutile
(defcons {instr}:oprod sons~(Vector ident))
(deftree {instr}:ucons(vnomtc~identificateur))
(defcons {ucons}:ocons sons~(Vector ident))
```

```
(defcons {instr}:oselect sons~(List cdegardee))
(deftree {sisole}:cdegardee)
(defcons {cdegardee}:ocdegardee sons~(Vector garde instrs))
(modelunion garde(log ocons))

;;;;; declaration de fonctions ;;;;;
(deftree {decl}:dfonct  (sem-ok~symbol nil)(var-v~1-string nil)
          (var-ut~1-string nil)(typ-v~1-string nil)(typ-ut~1-string nil)
          (fonct-v~1-string nil)(fonct-ut~1-string nil)
         (pf~1-string nil))
(defcons {dfonct}:odfonct sons~(Vector ident dvars ocorps type-ident))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;; EXPRESSIONS ARITHMETIQUES ET LOGIQUES  ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(deftree {sisole}:exps)

(deftree {lu}:arithmetique (vmodu ~identificateur) )
(deftree {lu}:logique (vmodu~identificateur) )
(deftree {logique}:cst-bool)
(modelunion exp(arith log))
(modelunion arith (arithmetique ident destab deschamp metavar opint tampon valeur))
(modelunion log (logique ident destab deschamp metavar opint tampon valeur))
(defcons {arithmetique}:o+ sons~(Vector arith arith))
(defcons {arithmetique}:o- sons~(Vector arith arith))
(defcons {arithmetique}:o* sons~(Vector arith arith))
(defcons {arithmetique}:odiv sons~(Vector arith arith))
(defcons {arithmetique}:omod sons~(Vector arith arith))
(defcons {arithmetique}:ohasard sons~(Vector arith arith));ajoute pour le handler
(deftree {arithmetique}:uarith)
(defcons {uarith}:ou+ sons~(Vector arith))
(defcons {uarith}:ou- sons~(Vector arith))
(defcons {logique}:o= sons~(Vector arith arith))
(defcons {logique}:o<> sons~(Vector arith arith))
(defcons {logique}:o>  sons~(Vector arith arith))
(defcons {logique}:o< sons~(Vector arith arith))
(defcons {logique}:o>= sons~(Vector arith arith))
(defcons {logique}:o<= sons~(Vector arith arith))
(deftree {logique}:bool2)
(defcons {bool2}:oet sons~(Vector log log))
(defcons {bool2}:oou sons~(Vector log log ))
(defcons {logique}:ounon sons~(Vector log))

(defcons {cst-bool}:ovrai)
(defcons {cst-bool}:ofaux)
(deftree {arithmetique}:cst-alpha( val~string))
(defcons {cst-alpha}:ocst-alpha)
(deftree {arithmetique}:cst-num (val~integer))

(defcons {cst-num}:ocst-ent)
(defcons {cst-num}:ocst-reel)
;;;;; l'attribut associe a cst-reel est un integer !... neutre;;;;;
```

```
(deftree {sisole}:destab)
(deftree {sisole}:deschamp)
(modelunion destab-ident(destab ident))
(modelunion destab-deschamp-ident(deschamp destab ident))
(defcons {exps}:oexps sons~(List arith))
(modelunion exps-arith(exps arith))
(defcons {destab}:odestab sons~(Vector destab-ident exps-arith))
   ;;;;; pour les tabeaux de tableaux ;;;;;
(defcons {deschamp}:odeschamp sons~(Vector destab-deschamp-ident destab-ident))
   ;;;;;; pour la structure des champs

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;; CORPS D'UN PROCESSUS OU D'UNE FONCTION                ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;(deftree {sisole}:corps)
;(defcons {corps}:ocorps sons~(Vector instrs-p ))

;(deftree {instr-p}:simpl-p)
;(deftree {instr}:instr-hab)
;(modelunion minstr-p(metavar instr-p))
;(defcons {instrs-p}:oinstrs-p sons~(List minstr-p))
(defcons {instr}:ocom sons~(Vector nomvars));sans doute inutile
;(defcons {instr-p}:otantque-p sons~(Vector log instrs-p))
;(defcons {instr-p}:osialorsinon-p sons~(Vector log instrs-p instrs-p))
;(defcons {instr-p}:osialors-p sons~(Vector log instrs-p))
(defcons {instr}:olire sons~(Vector nomvars))
(defcons {instr}:oecrire sons~(List arith))
;(modelunion minstr-hab(metavar instr-hab))
;(defcons {instr-hab}:oinstrs-hab sons~(List minstr-hab))
(modelunion minstrs(metavar instrs))
(defcons {instr}:osialors sons~(Vector log minstrs))
(defcons {instr}:osialorsinon sons~(Vector log minstrs minstrs))
(defcons {instr}:otantque sons~(Vector log minstrs))
(defcons {instr}:oaffec sons~(Vector destab-deschamp-ident exp))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;; DECLARATION D'UN TYPE  DE COMMUNICATION     ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(deftree {lu}:udef(vmodu~identificateur))
(deftree {udef}:defs)
(deftree {udef}:def)
(deftree {defs}:s-def)
(deftree {lu}:ints) ; liste d'intermediaires
(deftree {ints}:int);intermediaire
(defcons {decl-prog}:otca sons~(Vector ident log defs ints))
(defcons {ints}:oints sons~(List int));liste d'intermediaires
(modelunion choix-exp(choix exp))
(defcons {int}:ointid sons~(Vector ident choix-exp ident));intermediaire ident
  ;(Vector nom-inter. definition type)
(deftree {lu}:choix)
(deftree {ident}:fonct)
(defcons {lu}:oprofil sons~(Vector nomvars ident)) ;(Vector domaine co-domaine)
(defcons {int}:ointop sons~(Vector fonct choix-exp oprofil));intermediaire operat
(defcons {choix}:ochoix1  sons~(Vector log choix-exp))
(defcons {choix}:ochoix2 sons~(Vector log choix-exp choix-exp))
```

```
(deftree {lu}:tampon )
(deftree {tampon}:tfval)
(deftree {tampon}:tnval)
(deftree {tampon}:opint (val~identificateur) (vmodu~identificateur));
;val:nom de l'operation
;vmodu : nom du tca ou est declaree la fonction intermediaire?
(defcons {opint}:oopint sons~(List arith))
(deftree {sisole}:l-arg)

(defcons {l-arg}:ol-arg sons~(List ident))
(defcons {defs}:odefs sons~(List def))
(defcons {def}:odef sons~(Vector log s-def))
(defcons  {s-def}:os-def sons~(Vector mvaleur mtampon))
(modelunion mtampon(metavar tampon opint choix))
(modelunion mvaleur(metavar valeur opint choix))

(deftree {lu}:valeur )
(deftree {valeur}:vfval)

(defcons {vfval}:opremier sons~(Vector tampon))
(defcons {vfval}:odernier sons~(Vector tampon))
(defcons {valeur}:oextrait sons~(Vector tampon arith))
(defcons {valeur}:oomega)
(defcons {tfval}:odecapite sons~(Vector tampon))

(defcons {tnval}:oenleve sons~(Vector  tampon arith))
(defcons {tnval}:osuffixe sons~(Vector  tampon arith))
(defcons {tfval}:ofin sons~(Vector tampon))
(defcons {tfval}:otue sons~(Vector tampon))
;;;;; compl{ment aux expressions logiques ;;;;;

(defcons {logique}:ovide sons~(Vector tampon))
(defcons {logique}:odans sons~(Vector tampon  arith))
(deftree {tampon}:neutre(val~nomsuite)(vmodu~identificateur))
;;;;;;;; vmodu:  nom du processus de la donnee associee au tampon   ;;;;;;;
(defcons  {neutre}:oneutre) ;;;;;operateur sans effet sur la suite
(defmodel {string}:nomsuite (Predicate fnomsuite))
(de fnomsuite(x)
    (or(equal x "A")(equal x "SP")(equal x "SC")
       (equal x "a")(equal x "sp")(equal x "sc")))
;;;;; complement aux expressions arithmetiques ;;;;;

(deftree {arithmetique}:arithsuite)
(defcons {arithsuite}:ocard sons~(Vector tampon))
(defcons {arithsuite}:osup sons~(Vector tampon))
(defcons {arithsuite}:oinf sons~(Vector tampon))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;; meta-variables et schemas predefinis ;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defmodel {string}:metaident(Predicate fmetaident))
(de fmetaident
    (id)(let((l(explode id)))
    (and(eq(car l)(cascii '$))
        (omatchq identificateur(string(implode(cdr l)))))))

(deftree {lu}:metavar (val~metaident))
(defcons {metavar}:ometavar)
```

Annexe IV.2 :

Description algébrique du type de communication "ascenseur".

(résultat du décompilateur CEYX —> spécification algébrique)

```
TYPCOM lift[elt];
OPERATIONS
   init      :              --> lift;
   produire  :(lift,elt)--> lift;
   consommer :(lift)      --> lift;
DEFINITIONS
t∈ lift tq t=<SP,A,SC>; e ∈ elt;SP,A,SC ∈ ensemble
   init(t)       =<creer(),creer(),creer()>
   produire(t,e)=<ajoute(SP,e),ajoute(A,e),SC>
      non (vide(A)); ==>
   consommer(t,e)=<SP,NA,ajoute(SC,e)>
      e  = extrait(A,next);
      NA = enleve(A,next);
   INTERMEDIAIRES
   next:nat=
   si card(SC) = 0 alors
      ipg(A)
   sinon
      si dir = interieur et non (vers_interieur) ou dir = exterieur et
      vers_exterieur alors
         isup(A,dernier(SC))
      sinon
         iinf(A,dernier(SC))
      fsi
   fsi;
   vers_exterieur:bool=
   si card(SC) = 0 alors
      faux
   sinon
      isup(A,dernier(SC)) <> 0
   fsi;
   exterieur:nat= 1;

   vers_interieur:bool=
   si card(SC) = 0 alors
      vrai
   sinon
      iinf(A,dernier(SC)) <> 0
   fsi;
   interieur:nat= 0;

   dir:nat=
   si card(SC) <= 1 alors
      interieur
   sinon
      si dernier(SC) > extrait(SC,sup(SC) - 1) ou dernier(SC) = extrait(SC
       sup(SC) - 1) et dir = exterieur alors
         exterieur
      sinon
         interieur
      fsi
   fsi;
```

```
iinf(A,x):ensemble,elt --> nat=
si non (vide(A)) alors
   si card(A) = 1 alors
       si premier(A) < x alors
          inf(A)
       sinon
          0
       fsi
    sinon
       si premier(A) = x alors
          inf(A)
       sinon
          si iinf(decapite(A),x) = 0 alors
             si premier(A) < x alors
                inf(A)
             sinon
                0
             fsi
          sinon
             si premier(A) < x et premier(A) >= extrait(A,iinf(decapite(A),
             x)) alors
                inf(A)
             sinon
                iinf(decapite(A),x)
             fsi
          fsi
       fsi
   fsi
sinon
   0
fsi;

isup(A,x):ensemble,elt --> nat=
si non (vide(A)) alors
   si card(A) = 1 alors
       si premier(A) > x alors
          inf(A)
       sinon
          0
       fsi
    sinon
       si premier(A) = x alors
          inf(A)
       sinon
          si isup(decapite(A),x) = 0 alors
             si premier(A) > x alors
                inf(A)
             sinon
                0
             fsi
          sinon
             si premier(A) > x et premier(A) <= extrait(A,isup(decapite(A),
             x)) alors
                inf(A)
             sinon
                isup(decapite(A),x)
             fsi
          fsi
       fsi
   fsi
sinon
   0
fsi;
```

```
ipg(A):ensemble --> nat=
si card(A) = 1 alors
   inf(A)
sinon
   si premier(A) >= extrait(A,ipg(decapite(A))) alors
      inf(A)
   sinon
      ipg(decapite(A))
   fsi
fsi;

INTYPCOM lift
```

Annexe V :

Exemples de types de communication "ne portant pas sur les valeurs".

- Type de communication "égalité":

Ce type exprime une relation d'égalité, telle que celle mise en évidence au paragraphe 1-4-2-, entre les suites produite et consommée: c'est-à-dire que toute valeur émise est reçue une et une seule fois, et l'ordre des réceptions est identique à l'ordre des émissions. Ce type est caractérisé par:

pré-cons(<SP,A,SC>) = non vide?(A)
val-cons(<SP,A,SC>) = premier(A)
post-cons(<SP,A,SC>) = décapiter(A)

- Type de communication "aléatoire":

Ce type exprime une relation telle que chaque valeur reçue est la dernière émise à cet instant: ainsi, certaines valeurs ne sont pas reçues et d'autres peuvent l'être

leurs fois, selon le "rythme" aléatoire de chacun des essus producteur ou consommateur.

Pour cela on définit une nouvelle opération sur le type semble-indexé":

cer : ensemble-indexé * entier_positif --->
ensemble-indexé

supprime tous les éléments d'indice inférieur ou égal à tier positif donné. Cette opération est définie par:

cer(s,i) = si i=minimum(s) alors décapiter(s)
sinon si dans?(s,i)
alors effacer(décapiter(s),i)
sinon si i<minimum(s)
alors s
sinon effacer(s,i-1)
fsi
fsi
fsi

des cas particuliers intéressants sont:

nir(s) = effacer(s,maximum(s)-1)
r(s) = effacer(s,maximum(s))

finit alors le type "aléatoire" par:

pré-cons(<SP,A,SC>) = non vide?(A)
val-cons(<SP,A,SC>) = dernier(A)
post-cons(<SP,A,SC>) = finir(A)

- Type de communication "très-aléatoire":

Ce type est une généralisation du précédent en tant une consommation avant la première opération de tion. Il est défini par:

pré-cons(<SP,A,SC>) = vrai
val-cons(<SP,A,SC>) = si vide?(A) alors Ω
sinon dernier(A)
fsi
post-cons(<SP,A,SC>) = finir(A)

- Type de communication "rafraichie":

Ce type exprime une relation semblable à celle décrite par le type "aléatoire", mais où toute valeur reçue doit être "rafraichie": c'est-à-dire qu'entre deux consommations successives de la donnée communiquée il doit y avoir au moins une nouvelle production de valeur de cette donnée.

pré-cons(<SP,A,SC>) = non vide?(A)
val-cons(<SP,A,SC>) = dernier(A)
post-cons(<SP,A,SC>) = tuer(A)

- Type de communication "égalité-presque-partout":

Ce type exprime que toute valeur émise est reçue une fois, et une seule, si l'opération de production survient "assez tôt". Plus précisément, il y a égalité des suites produite et consommée, si ce n'est que le ième élément produit est remplacé par "ω" dans la suite consommée, si la production de cet élément est en retard sur la ième consommation.

pré-cons(<SP,A,SC>) = vrai
val-cons(<SP,A,SC>) = extraire(A,cardinal(SC)+1)
post-cons(<SP,A,SC>) = effacer(A,cardinal(SC)+1)

- Type de communication "rafraichie-avec-trous":

Ce type exprime qu'au cas où la donnée communiquée n'aurait pas été rafraichie, la valeur reçue est ω ; dans le cas contraire la dernière valeur produite est consommée:

pré-cons(<SP,A,SC>) = vrai
val-cons(<SP,A,SC>) = si vide?(A) alors ω
sinon dernier(A)
fsi
post-cons(<SP,A,SC>) = si vide?(A) alors A
sinon tuer(A)
fsi

- Type de communication "égalité-avec-trous":

Différent de "l'égalité-presque-partout", ce type est tel que toute valeur émise est reçue; "entre-temps" le processus consommateur peut recevoir la valeur ω, lorsqu'aucune nouvelle valeur n'est émise.

pré-cons(<SP,A,SC>) = vrai
val-cons(<SP,A,SC>) = si vide?(A) alors ω
sinon premier(A)
fsi
post-cons(<SP,A,SC>) = si vide?(A) alors A
sinon décapiter(A)
fsi

- Type de communication "à-retard-borné":

Dans la situation décrite par ce type, toute valeur reçue est celle émise le plus récemment. Cependant le "retard" pris par le processus producteur reste borné: c'est-à-dire que le nombre d'occurrences de consommation moins le nombre d'occurrences de production est borné par un entier "p":

pré-cons(<SP,A,SC>) = non vide?(A)
val-cons(<SP,A,SC>) = dernier(A)
post-cons(<SP,A,SC>) = si cardinal(SC)-maximum(A)<p
alors finir(A) sinon tuer(A)
fsi

- Type de communication "échantillon":

La relation décrite est telle que le processus consommateur prélève la plus récente valeur produite, une seule fois, en n'ignorant pas plus de "p" valeurs produites consécutivement.

On définit le prédicat suivant sur le produit cartésien ensemble-indexé * entier_positif:

```
P(s,i) =    minimum(s)<i<maximum(s)
        et non dans?(s,i)
        et j ∈ [i+1,maximum(s)]  ==> dans?(s,j)
```

-- définit le plus grand entier "i" dans le domaine de "s", s'il existe, correspondant à une consommation antérieure
--

On définit la fonction suivante de "ensemble-indexé" dans "entier":

```
q(s) =  si existe(i) telque P(s,i) alors i
        sinon minimum(s)-1
        fsi
```

-- définit le plus grand entier correspondant à une consommation antérieure
--

Le type de communication "échantillon" est alors défini par:

```
pré-cons(<SP,A,SC>) = non vide?(A)
val-cons(<SP,A,SC>) = si maximum(A)-q(A)<p
                      alors dernier(A)
                      sinon extraire(A,q(A)+p)
                      fsi
post-cons(<SP,A,SC>) = si maximum(A)-q(A)<p
                       alors tuer(A)
                       sinon effacer(A, q(A)+p)
                       fsi
```

- Type de communication "élastique":

La relation décrite par ce type de communication est telle que toute valeur émise est reçue une fois, et une seule, dans l'ordre des émissions, sauf lorsque le processus consommateur prend un retard supérieur à "p": dans ce cas toutes les valeurs émises sont "sautées" jusqu'à la dernière.

```
pré-cons(<SP,A,SC>) = non vide?(A)
val-cons(<SP,A,SC>) = si cardinal(A)<p
                      alors premier(A) sinon dernier(A)
                      fsi
post-cons(<SP,A,SC>) = si cardinal(A)<p
                       alors décapiter(A) sinon tuer(A)
                       fsi
```

- Type de communication "circulaire":

La relation décrite par ce type de communication est semblable à la précédente, mais où, en cas de retard du processus consommateur, la valeur reçue est la plus anciennement émise "modulo" un cycle de taille "p":

```
pré-cons(<SP,A,SC>) = non vide?(A)
val-cons(<SP,A,SC>) =
   extraire(A,minimum(A)+k(maximum(A)-minimum(A),p)*p),

   avec k(x,p) = si p>=x alors 0
                 sinon k(x-p,p)+1
                 fsi

post-cons(<SP,A,SC>) =
   moduler(A,minimum(A)+k(maximum(A)-minimum(A),p)*p)
```

où l'opération "moduler"

```
moduler : ensemble-indexé * entier_positif --->
                                        ensemble-indexé
```

est définie sur tout objet "s" de type "ensemble-indexé" et tout entier_positif "i" par:

```
moduler(s,i) = si dans?(s,i)
               alors moduler(disparaitre(s,i),i-p)
               sinon s
               fsi
```

- Type de communication "pile":

La relation décrite par ce type est telle que la plus haute valeur, parmi les valeurs consommables, est consommée une seule fois et retirée de cet ensemble:

```
pré-cons(<SP,A,SC>) = non vide?(A)
val-cons(<SP,A,SC>) = dernier(A)
post-cons(<SP,A,SC>) = disparaitre(a,maximum(A))
```

NOM DE L'ETUDIANT : Madame MARMONIER Marina

NATURE DE LA THESE : DOCTORAT DE L'UNIVERSITE DE NANCY I en INFORMATIQUE.

VU, APPROUVE ET PERMIS D'IMPRIMER

NANCY, le 21 FEV. 1989 n° 348

LE PRESIDENT DE L'UNIVERSITE DE NANCY I

M. Boulangé