Propositions pour un langage d'écriture de programmes répartis. Expression du contrôle de la communication entre processus distribués

Jean-Michel Guillot

THESE

présentée à

Université Scientifique et Médicale de Grenoble

Institut National Polytechnique de Grenoble

pour obtenir le grade de

DOCTEUR DE 3ème CYCLE
"Informatique"

par

Jean-Michel GUILLOT

PROPOSITIONS POUR UN LANGAGE D'ECRITURE DE PROGRAMMES REPARTIS. EXPRESSION DU CONTROLE ET DE LA COMMUNICATION ENTRE PROCESSUS DISTRIBUES.

Thèse soutenue le 6 décembre 1979 devant la Commission d'Examen :

S. KRAKOWIAK *Président*

P. DECITRE
M. GIEN
C. KAISER
J. MOSSIERE
} Examinateurs

UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

Monsieur Gabriel CAU : Président
Monsieur Joseph KLEIN : Vice-Président

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

PROFESSEURS TITULAIRES

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

.../...

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

.../...

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

.../...

PROFESSEURS ASSOCIES

MM.	CRABBE Pierre	CERMO
	SUNIER Jules	Physique

PROFESSEURS SANS CHAIRE

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

.../...

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

MAITRES DE CONFERENCES ET MAITRES DE CONFERENCES AGREGES

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

.../...

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

.../...

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

MAITRES DE CONFERENCES ASSOCIES

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

MAITRE DE CONFERENCES DELEGUE

M.	ROCHAT Jacques	Hygiène et hydrologie (pharmacie)

Fait à Saint Martin d'Hères, novembre 1977

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Année universitaire 1977-1978

Président : M. Philippe TRAYNARD

Vice-présidents : M. René PAUTHENET
M. Georges LESPINARD

PROFESSEURS TITULAIRES

MM.	BENOIT Jean	Electronique - automatique
	BESSON Jean	Chimie minérale
	BLOCH Daniel	Physique du solide - cristallographie
	BONNETAIN Lucien	Génie chimique
	BONNIER Etienne	Métallurgie
*	BOUDOURIS Georges	Electronique - automatique
	BRISSONNEAU Pierre	Physique du solide - cristallographie
	BUYLE-BODIN Maurice	Electronique - automatique
	COUMES André	Electronique - automatique
	DURAND Francis	Métallurgie
	FELICI Noël	Electronique - automatique
	FOULARD Claude	Electronique - automatique
	LANCIA Roland	Electronique - automatique
	LONGEQUEUE Jean-Pierre	Physique nucléaire corpusculaire
	LESPINARD Georges	Mécanique
	MOREAU René	Mécanique
	PARIAUD Jean-Charles	Chimie - physique
	PAUTHENET René	Electronique - automatique
	PERRET René	Electronique - automatique
	POLOUJADOFF Michel	Electronique - automatique
	TRAYNARD Philippe	Chimie - physique
	VEILLON Gérard	Informatique fondamentale et appliquée

* en congé pour études

PROFESSEURS SANS CHAIRE

MM.	BLIMAN Samuël	Electronique - automatique
	BOUVARD Maurice	Génie mécanique
	COHEN Joseph	Electronique - automatique
	GUYOT Pierre	Métallurgie physique
	LACOUME Jean-Louis	Electronique - automatique
	JOUBERT Jean-Claude	Physique du solide - cristallographie

.../...

MM.	ROBERT André	Chimie appliquée et des matériaux
	ROBERT François	Analyse numérique
	ZADWORNY François	Electronique - automatique

MAITRES DE CONFERENCES

MM.	ANCEAU François	Informatique fondamentale et appliquée
	CHARTIER Germain	Electronique - automatique
	CHIAVERINA Jean	Biologie, biochimie, agronomie
	IVANES Marcel	Electronique - automatique
	LESIEUR Marcel	Mécanique
	MORET Roger	Physique nucléaire - corpusculaire
	PIAU Jean-Michel	Mécanique
	PIERRARD Jean-Marie	Mécanique
	SABONNADIERE Jean-Claude	Informatique fondamentale et appliquée
Mme	SAUCIER Gabrielle	Informatique fondamentale et appliquée
M.	SOHM Jean-Claude	Chimie Physique

CHERCHEURS DU C.N.R.S. (Directeur et Maîtres de Recherche)

M.	FRUCHART Robert	Directeur de Recherche
MM.	ANSARA Ibrahim	Maître de Recherche
	BRONOEL Guy	Maître de Recherche
	CARRE René	Maître de Recherche
	DAVID René	Maître de Recherche
	DRIOLE Jean	Maître de Recherche
	KLEITZ Michel	Maître de Recherche
	LANDAU Ioan-Doré	Maître de Recherche
	MATHIEU Jean-Claude	Maître de Recherche
	MERMET Jean	Maître de Recherche
	MUNIER Jacques	Maître de Recherche

Personnalités habilitées à diriger des travaux de recherche (décision du Conseil Scientifique)

E.N.S.E.E.G.

MM.	BISCONDI Michel	Ecole des Mines St. Etienne (dépt. Métallurgie)
	BOOS Jean-Yves	Ecole des Mines St. Etienne (Métallurgie)
	DRIVER Julian	Ecole des Mines St. Etienne (Métallurgie)

.../...

MM.	KOBYLANSKI André	Ecole des Mines St. Etienne (Métallurgie)
	LE COZE Jean	Ecole des Mines St. Etienne (Métallurgie)
	LESBATS Pierre	Ecole des Mines St. Etienne (Métallurgie)
	LEVY Jacques	Ecole des Mines St. Etienne (Métallurgie)
	RIEU Jean	Ecole des Mines St. Etienne (Métallurgie)
	SAINFORT	C.E.N. Grenoble (Métallurgie)
	SOUQUET	U.S.M.G.
	CAILLET Marcel	Ecole des Mines St. Etienne (Chim. Min. Ph.)
	COULON Michel	Ecole des Mines St. Etienne (Chim. Min. Ph.)
	GUILHOT Bernard	Ecole des Mines St. Etienne (Chim. Min. Ph.)
	LALAUZE René	Ecole des Mines St. Etienne (Chim. Min. Ph.)
	LANCELOT Francis	Ecole des Mines St. Etienne (Chim. Min. Ph.)
	SARRAZIN Pierre	Ecole des Mines St. Etienne (Chim. Min. Ph.)
	SOUSTELLE Michel	Ecole des Mines St. Etienne (Chim. Min. Ph.)
	THEVENOT François	Ecole des Mines St. Etienne (Chim. Min. Ph.)
	THOMAS Gérard	Ecole des Mines St. Etienne (Chim. Min. Ph.)
	TOUZAIN Philippe	Ecole des Mines St. Etienne (Chim. Min. Ph.)
	TRAN MINH Canh	Ecole des Mines St. Etienne (Chim. Min. Ph.)

E.N.S.E.R.G.

MM.	BOREL	Centre d'études nucléaires de Grenoble
	KAMARINOS	Centre national recherche scientifique

E.N.S.E.G.P.

M.	BORNARD	Centre national recherche scientifique
Mme	CHERUY	Centre national recherche scientifique
MM.	DAVID	Centre national recherche scientifique
	DESCHIZEAUX	Centre national recherche scientifique

Qu'il me soit permis de remercier,

S. KRAKOWIAK et J. MOSSIERE pour la formation, les conseils, l'amitié qu'ils m'ont offerts, pour le fait d'avoir accepté, le premier de présider mon jury, le second d'y participer,

P. DECITRE, M. GIEN, C. KAISER qui ont accepté de participer à mon jury, m'ont donné de multiples conseils,
C. CLEMENCET, J. L. CHEVAL, P. LAFORGUE, J. RAYMOND, S. ROUVEYROL du projet SORTILEGE mais également J. BRIAT, B. CASSAGNE, J. CHASSIN de KERGOMMEAUX pour des discussions multiples, des conseils judicieux et pour la montagne,

G. DUFFOURD et l'équipe de D. IGLESIAS pour la réalisation pratique de cet ouvrage,

et tous ceux qui m'ont aidé dans ce travail.

Que tous, au delà de l'aspect formel et trop répété des mots, trouvent ici l'expression de ma reconnaissance et de mes remerciements.

PRESENTATION
-o-

Pour présenter cette thèse et la situer il nous faut décrire le projet SORTILEGE qui lui a servi de cadre. Le thème de ce projet est la conception et la réalisation d'outils d'écriture, d'exécution d'applications distribuées.

Le terme de répartition est très général et doit être explicité et défini dans notre cas particulier. On peut l'appliquer aussi bien à une structure mettant en jeu plusieurs microprocesseurs, et qui tient dans une boîte, qu'à un réseau de calculateurs couvrant un pays ou même un continent. Plutôt que de développer une étude sur tous les aspects du terme nous nous limiterons au type d'architecture que vise SORTILEGE, c'est-à-dire les réseaux de petits et moyens calculateurs. On couvre, là encore, une variété très grande de matériels et d'applications :

- un réseau peut être destiné à une application unique ou à une vaste gamme d'applications,
- il peut être constitué de machines autrefois indépendantes connectées pour des applications somme toute marginales par rapport à l'ensemble des traitements,
- il peut avoir été conçu pour des applications dont l'aspect distribué est fondamental.

Pour définir les objectifs de SORTILEGE nous sommes partis des constatations suivantes :

1- Il se crée de plus en plus de réseaux spécialisés de mini ou micro-calculateurs. Ceci est particulièrement vrai dans des applications de contrôle de processus industriels, bancaires, en bureautique etc, et d'une façon générale partout où certains traitements coordonnés doivent mettre en jeu différents lieux, différents services.

2- Il est sans doute toujours possible de ramener ces traitements à un calculateur central. Ce n'est pas forcément souhaitable : pour des raisons de panne, d'éloignement qui implique des coûts et des risques de transmission pas forcément négligeables. Enfin, il se peut que la plupart des traitements soient locaux à un service, une usine, une agence.. Dans ce cas, les utilisateurs préfèreront disposer de moyens de calculs locaux, facilement accessibles, et connectés entre eux pour des applications plus générales.

Pour toutes ces raisons, il semble que les réseaux de mini-calculateurs soient appelés à se multiplier et en même temps à se banaliser, qu'en d'autres termes on les considèrera comme un type de matériel particulier et non comme un agglomérat de matériels distincts.

La conséquence logicielle de cette évolution ne peut être qu'une évolution parallèle, c'est-à-dire la conception de systèmes-réseaux adaptés à ce type de matériel nouveau. On ne doit plus considérer un réseau comme un ensemble de fils entre des systèmes mais plutôt comme un système tournant sur un ensemble de calculateurs connectés.
De plus, la multiplicité de ce type de matériel doit amener à concevoir des logiciels portables, dépendant peu de la configuration choisie. De la même façon que des logiciels centralisés doivent être portables d'une machine à l'autre.

Le manque d'outils logiciels généraux adaptés à l'architecture réseau force actuellement la plupart des projets industriels à repenser et réécrire tous les niveaux de système nécessaire à l'exécution de leurs applications réparties spécialisées. Le but de SORTILEGE est de fournir des outils d'écriture et d'exécution nécessaires et communs à ces applications, c'est-à-dire :

- un langage d'écriture
- un noyau d'exécution répartie fournissant des mécanismes de communication, de contrôle entre les éléments des applications supportées.

C'est le premier point qui fournit le sujet de cette thèse, particulièrement les outils de contrôle et de communication introduits dans le langage pour faciliter la tâche du programmeur.
Pour situer notre propre travail, nous présentons dans une première partie une étude de propositions récentes sur des langages ou simplement des mécanismes de contrôle d'exécution.
Une seconde partie présente nos primitives qui sont ensuite utilisées dans une proposition de langage (partie III).
Une quatrième et dernière partie, essaie d'illustrer nos propositions sous forme d'un petit système.

PREMIÈRE PARTIE : ETUDE BIBLIOGRAPHIQUE

ONZE APPROCHES POUR STRUCTURER, ÉCRIRE, CONTRÔLER DES APPLICATIONS RÉPARTIES À ACTIVITÉS PARALLÈLES

---

1. PRELIMINAIRE

L'étude d'une dizaine de propositions récentes nous sert de pretexte pour aborder l'analyse des applications distribuées. Certains de ces onze langages ne sont pas ouvertement présentés par leurs auteurs comme devant servir à la réalisation de programmes répartis. Cela n'aura guère d'importance pour plusieurs raisons :

- le thème de la répartition est aujourd'hui présent dans l'esprit de tout informaticien. On ne peut actuellement proposer de nouveaux langages, de nouvelles primitives d'expression sans en tenir compte. Toutes ces propositions sont récentes et chacune a vraisemblablement une implémentation distribuée possible pour son auteur;
- proposer un nouveau langage, le plus souple et le plus puissant possible c'est aussi cacher de nombreuses contraintes techniques à l'utilisateur, y compris celles concernant la répartition. La notion de liaison par exemple, est retirée des préoccupations du programmeur d'application et confiée au système support. La notion même de site d'exécution n'apparaît pas dans les propositions les plus récentes. Ce sera l'un des buts de cette étude, que d'examiner dans quelle mesure l'expression d'une application peut être indépendante de son support matériel et des sites qui le composent.

1.1 - Thèmes de l'étude

Plus particulièrement concernée par des problèmes de structuration, de contrôle, de communication, cette analyse comporte trois volets d'importance inégale. Les deux premiers décrivent la composition des applications et celle de leurs éléments, le troisième décrit rapidement l'environnement d'exécution des programmes. On étudiera donc successivement :

Chap.2 . La structure globale proposée pour les applications, réduite à :
- . la définition d'éléments de décomposition du programme (s'il en existe),
- . la description des activités qui s'y déroulent, leur rapport avec les composants introduits ci-dessus,
- . le type de communication envisagé entre activités, entre composants.

Chap.3 . La structure des composants, la façon dont s'y déroulent les activités existantes, leur contrôle.

Chap.4 . Un complément concernant la structure générale des applications et leur vie : lancement, évolution éventuelle de la structure, notion de liaison, contrôle d'erreur.

D'un volet à l'autre, les choix ne sont bien sûr pas indépendants, par exemple tel choix dans la définition du mode de communication influera sur le contrôle interne d'un composant. On essaiera de souligner ces implications tout au long de notre analyse.

Un aspect considérable de l'écriture des applications est passé sous silence c'est.. le langage d'écriture. C'est un domaine qui sort quelque peu des préoccupations de cette thèse, de plus, il est fréquent que les propositions étudiées se limitent à des considérations de structure et à la présentation de quelques primitives. Dans ce cas, les auteurs indiquent que cette structure, ces primitives peuvent être ajoutées à un langage existant, le plus souvent PASCAL.

1.2 - Support de l'étude

Les propositions qui interviendront ici sont les suivantes (on souligne le nom que l'on utilisera) :

ADA : [GREEN*i.d*]
Communicating Sequential Processes (CSP) de Hoare [HOARE 78]
Distributed Processes (DP) Brinch Hansen [BRINCH 77]
Programming Language in The Sky (PLITS) J. Feldman [FELDMAN 76] [FELDMAN 77a][FELDMAN 77b]
Synchronisation par compteurs P. Robert, J.P. Verjus [ROBERT 77]
Expressions de chemin Haberman [HABERMAN 75]
Moniteur d'Exécution Répartie (MER) Decitre André [ANDRE 78]
Un langage anonyme, extension aux moniteurs : G. Pacini [PACINI]
SIGOR N. Dang, G. Sergeant [DANG 77]
SESAME [CHEVAL 76] [MOSSIERE 76]
NODAL : langage du réseau SPS du CERN [ALTABER 77]

Ces éléments interviendront dans toute l'étude, expliqués en même temps que les concepts ou outils qu'ils utilisent : commandes gardées, moniteurs etc.. On trouvera en annexe un tableau résumant leurs caractéristiques, et indiquant les passages où ils sont décrits.

2. STRUCTURE D'UNE APPLICATION

2.1 - Problèmes généraux de structure et décomposition

Qu'elle soit centralisée ou répartie, l'analyse d'une application pose à son concepteur des problèmes de décomposition. Le soin que l'on a d'écrire le plus simplement possible un programme complexe amène à isoler des parties distinctes, à faire une décomposition fonctionnelle de l'application. Ce travail fait, on retirera de l'application l'image d'un réseau de composants, chacun étant décrit par :

- les actions qu'il réalise et que peuvent demander d'autres éléments,
- la liste des autres composants que lui-même utilise.

On a ainsi morcelé le travail de programmation et si les divers éléments sont bien distincts, on peut les écrire individuellement sans connaître la réalisation pratique des autres constituants. Il suffira que cette réalisation respecte la définition formelle donnée.
L'avantage est considérable si l'application est complexe ou si elle est écrite par plusieurs programmeurs.
Ce sont là des constatations déjà anciennes et l'on verra quelles réponses sont apportées par la littérature (2.2).
Sur la structure matérielle ainsi dégagée, il faut implanter les processus qui seront l'activité de l'application. Il y a là encore, une décomposition à faire : que représente chacun des processus, comment se sert-il des différentes unités ? (2.3).
Troisième aspect important d'une proposition de structuration d'application : comment gérer la communication entre entités, que l'entité considérée soit l'unité d'écriture ou le processus (2.1.2).
Enfin, de quels outils peut-on disposer pour exprimer, globalement dans l'application, la synchronisation des activités (2.4). C'est l'étude de ces quatre points :

- éléments de décomposition fonctionnelle de l'application
- définition de l'activité
- communication (en liaison avec le point précédent car en découlant obligatoirement)
- contrôle global de l'activité

qui constitue les pages suivantes.

2.2 - Décomposition d'une application, notion de module

La décomposition d'une application peut prendre des formes très variées. La recherche d'une structuration pratique et propre intervient d'ailleurs plus dans des disciplines d'écriture que dans les langages eux-mêmes. Mais ces disciplines d'écriture finissent invariablement par trouver une matérialisation dans les langages. On a ainsi écrit des procédures en assembleur avant que des langages leur donnent une formulation explicite.
Chacun des langages que nous étudions propose, ou plutôt impose, une certaine unité d'écriture. Cette unité implémente une fonction ou un ensemble de fonctions liées logiquement; on peut dire qu'elle représente une ressource logique et les différentes actions liées à l'utilisation de cette ressource. Il s'agit là de considérations fonctionnelles mais qui sont le plus souvent accompagnées d'autres impératifs :

- l'interface avec l'extérieur doit être la plus simple possible pour faciliter la description de l'unité,
- la réalisation, pour la même raison, ne doit pas importer à ses utilisateurs,
- enfin, par souci de sécurité, le mauvais fonctionnement (erreur) d'une unité doit avoir le moins possible de conséquences pour les autres.

On est ainsi conduit à rechercher une structure dont les unités ont peu d'objets communs (variables par exemple) et ne sont utilisés qu'à travers des voies de communication (voies d'accès) bien spécifiées et protégées. D'où l'utilisation, très répandue (9 de nos 11 langages) de la notion de module comme unité d'écriture d'application. Même si ce terme recouvre beaucoup d'interprétations possibles, toutes les propositions définissent un module comme :

- un ensemble de services
- un ensemble de données partagées par les services.

Les services implémentent les actions liées à la ressource logique représentée par le module. On ne peut accéder à un module que par des requêtes aux services. L'accès par l'extérieur aux variables est proscrit.
Les types d'unités non modulaires proposés par IGOR et NODAL sont décrits en 2.4.2.

Cas particulier des applications réparties

Nous avons là un premier exemple dans lequel les problèmes de répartition ont finalement peu d'importance dans le choix des structures ou plutôt des unités. Ce type d'application nécessite a priori des unités très indépen-

dantes, pour limiter les délais de communication; l'absence de mémoire partagée rend indésirable l'existence de variables communes à plusieurs unités. Mais ce sont des problèmes déjà abordés pour simplifier et rendre plus fiable toute application, centralisée ou répartie. Paradoxalement, deux langages non modulaires, permettant le partage de variables entre activités distantes, sont destinés à des applications réparties : ce sont IGOR et NODAL.

La notion d'unité de décomposition est d'autant plus importante pour les applications réparties qu'elle correspondra le plus souvent avec celle d'unité de répartition : le plus grand élément de programme qui doit résider entièrement sur un site pour que son exécution soit possible matériellement.

2.3 - Implantation de l'activité. Communication

2.3.1. Identification de l'activité

A quoi identifie-t-on un processus dans une application ? C'est là un choix qui aura de nombreuses conséquences sur le type de communication choisi entre unités, mais également sur la structure interne de celles-ci.

Très grossièrement, deux attitudes sont possibles :

- un processus représente un utilisateur de l'application. Il s'exécute suivant ses besoins dans les unités du programme et se propage à travers elles;
- un processus est identifié à une fonction ou à une ressource. Il s'exécute à l'intérieur de l'unité représentant cette ressource.

Dans le premier cas l'application est un réseau d'unités que parcourent les processus exécutant les unités, et dont on verra plus loin les interactions.

2.3.2. Influences sur la structure des unités et la communication

2.3.2.1. Processus se propageant dans l'application : schéma procédural

Le choix d'une activité plus directement liée à l'utilisateur force à définir des outils de propagation des processus. Pour comprendre ceux-ci, il faut se rappeler que chaque unité représente un service pour l'unité qui l'utilise. L'exécution est lancée dans une unité qui, pour effectuer le travail demandé, requiert un service à une autre, en reçoit des résultats puis continue le traitement.

Ce mode d'utilisation n'est pas unique [BRIN 76] mais c'est sans doute le plus répandu. Il est alors naturel d'utiliser des outils de changement de contrôle du type procédural. On est ainsi amené à décrire une unité comme une collection de procédures groupées autour de variables communes (voir 3.2.1).
La propagation des processus se fait par appel procédural, et la communication est alors liée aux paramètres des procédures.
Les processus distribués de Brinch Hansen, les expressions de chemins, les moniteurs, SESAME utilisent cette technique.

2.3.2.2. Processus résidents. Communication par messages

Les contraintes sont moins fortes sur la structure des unités servant de résidence à des processus. On pourra toutefois noter :

- que plusieurs processus peuvent partager une unité, ce qui posera des problèmes de désignation et de contrôle,
- que si l'unité choisie est de type modulaire les processus d'unités différentes ne disposent pas de moyens implicites de communiquer.

Il faudra définir un moyen de communication entre unités qui sera généralement celui de message.
ADA, CSP, PLITS, MER disposent de ce mécanisme.
La notion de message est également sujette à variations d'un langage à l'autre. On reprend ses diverses interprétations en 3.3.3.
Déduisons, simplement pour le moment, que dans ce schéma une unité est considérée de l'extérieur comme une boîte noire :

- comportant une ou plusieurs voies d'arrivée de messages
- exécutée par au moins un processus.

L'expédition d'un message pourra se faire :

- vers un processus donné
- vers une voie d'accès
- vers un processus donné via une voie d'accès.

Les choix sont multiples et dépendent en partie de la structure que le concepteur donne aux unités. Nous y reviendrons en abordant les problèmes de désignation (3.3.3).

2.3.3. Comparaison des deux approches. Remarques

Une comparaison quantitative des deux méthodes est exposée par J. ESTUBLIER dans [ESTUBLIER]. Pour notre part, nous ne saurions a priori préférer l'une ou l'autre. Le simple fait que sur huit langages étudiés, deux utilisent exclusivement les messages, deux la structure procédurale et les autres un compromis des deux, prouve l'intérêt des deux méthodes. (Les expressions de chemin, les compteurs et la proposition de G. PACINI sont exclus de ce décompte puisqu'ils concernent presque exclusivement le contrôle).

Pour ce qui est de l'écriture d'applications réparties, on peut faire quelques remarques :

- Le schéma des processus résidents présente une analogie évidente avec l'architecture réseau; les processus, processeurs virtuels disposent d'une mémoire propre et communiquent par l'échange de messages. Cette analogie et le fait que l'information soit réellement transportée par des messages entre sites peuvent constituer un avantage. Il ne semble pourtant pas être déterminant. Les deux types de schéma peuvent être implémentés sur un réseau, et aucun de nos auteurs ne cite cet aspect pour justifier son choix.

- L'appel d'une procédure distante ne doit pas être différent de celui d'une procédure du même site; puisque l'absence de mémoire commune rend difficile le passage de paramètres par nom ou référence ceux-ci sont généralement proscrits. Ces préoccupations de répartition rejoignent les efforts faits pour la fiabilité des systèmes centralisés. Dans tous les cas, l'accès à des variables d'autres unités risque de faciliter la propagation d'erreurs.

2.4 - Contrôle global de la synchronisation des activités

On introduit ici très rapidement le problème du contrôle lorsqu'il est exprimé pour l'application dans son ensemble.
Il existe peu de moyens de contrôle global dans la littérature et ils dépendent étroitement des structures choisies.

2.4.1. Environnement modulaire

Dans ce schéma, l'application est constituée d'un réseau de modules indépendants. Les communications se font par appel procédural ou par échange de messages, suivant le choix du concepteur du langage. C'est dire que dans le premier cas, la résolution des conflits entre processus est à la charge de modules, puisque ces conflits se jouent dans l'accès aux procédures ou aux variables.

Dans le schéma des processus résidents, l'interaction entre processus d'unités différentes est limitée à l'échange de messages; c'est une technique qui porte en elle-même un choix de synchronisation : il est en effet admis qu'un processus désirant recevoir un message reste bloqué tant que celui-ci n'est pas disponible. Le concepteur peut, par contre, choisir de suspendre ou de continuer l'exécution du processus émetteur si son message n'est pas consommé. Dans tous les cas, le système support assure la cohérence de la gestion des messages (conflits d'accès de plusieurs processus sur une même voie de communication par exemple. Un message ne pourra être consommé que par un processus à la fois).

L'outil principal de la gestion des messages est la file d'attente : elle servira à ordonner les requêtes (les messages) des différents utilisateurs d'un module. Elle résoudra donc implicitement tous les problèmes de conflits :

- entre émetteur et destinataire
- entre différents émetteurs.

Dans ce schéma, on laisse donc peu d'intiative au programmeur pour le contrôle global des conflits entre processus.

On reviendra sur les files d'attente et les interactions entre processus dans le chapitre suivant (3.3.3).

2.4.2. Cas particuliers de IGOR et NODAL

IGOR et NODAL ont une particularité commune, ils ne sont pas modulaires. Dans les deux cas l'application est exprimée en un seul programme. Le programmeur y définit des sections de code qui peuvent être envoyées pour s'exécuter sur d'autres sites. Pour IGOR il s'agira de procédures, pour NODAL de groupes de lignes de code. Ces sections de code s'exécutent en parallèle avec l'activité qui les a créées, ce sont elles qui forment les différentes unités composant une application.

Contrairement aux modules d'autres langages, ces unités peuvent partager des données, données globales du programme par exemple. Les contrôles exercés sur ces accès sont très simples et peu contraignants. Ainsi, dans IGOR plusieurs activités peuvent modifier simultanément une variable. (Elles peuvent cependant communiquer également par messages).
A côté de ce contrôle relativement faible sur les conflits d'accès, l'application (identifiée au processus père) peut contrôler l'exécution des processus fils, leur permet de se synchroniser entre eux (rendez-vous d'IGOR) et peut se synchroniser avec eux (WAIT de NODAL). Le contrôle de l'application sur ses différents éléments est donc plus souple et programmable. C'est une possibilité qui fait défaut aux langages modulaires; quant à l'aspect négatif que constitue le partage de variables, il ne semble pas primordial et peut être évité par le programmeur, dans IGOR au moins.

3. STRUCTURE, EXECUTION ET CONTROLE D'UNE UNITE

3.1 - Problèmes étudiés

Nous entrons dans le détail de ce que nous avons jusqu'ici manipulé globalement sous le nom d'unité. Ces unités sont :

- les procédures d'IGOR,
- les groupes d'instructions de NODAL,
- les modules des autres langages.

Nous ne les avons considérées jusqu'ici que comme des boites noires disposant d'une interface. Suivant le type d'activité choisie cette interface est composée :

- de points d'entrée de procédures (schéma procédural)
- ou des voies de réception de messages (schéma des processus résidents)

Il est pratique, particulièrement dans le premier schéma* de formaliser chaque voie d'interaction entre unités comme correspondant à un service que peut rendre l'unité. On pourra donc définir avec ce terme une unité comme :

- une ressource virtuelle offrant à ses utilisateurs (les autres unités) un ensemble de services.

Ainsi, un module gérant la mémoire d'un calculateur peut offrir à son environnement deux fonctions (deux services) :

- allouer
- désallouer de (l'espace mémoire).

Notre premier soin sera de discerner comment un service est rendu à son demandeur et comment une unité peut être structurée en fonction des services qu'elle doit rendre. On pourra pour cela analyser :

- la façon dont une requête est prise en compte,
- l'exécution des actions traitant une requête (exécution du ou des processus et définition du code traitant la reqûete),
- l'envoi de requêtes à d'autres services d'autres unités (l'exécution d'un service peut en effet nécessiter l'utilisation d'un service extérieur).

*Remarque : L'identification service-voie n'est pas aussi évidente dans certaines propositions utilisant la communication par message. Nous reviendrons sur ce point.

Ceci concerne la réalisation purement algorithmique des actions à entreprendre pour rendre les services demandés. L'écriture d'une unité doit cependant tenir compte de contraintes précises sur l'exécution des services :

- des règles de synchronisation qui définissent les services que l'on ne peut rendre simultanément,
- des règles de séquencement indiquant, suivant l'état de l'unité, les services que l'on peut rendre à un moment donné,
- des règles d'ordonnancement qui peuvent compléter les précédentes en précisant quelles préférences on donne dans le choix d'un service à exécuter si plusieurs sont possibles.

Ces règles peuvent provenir soit de la logique de la ressource implémentée, soit de la façon matérielle dont elle a été implémentée. Ainsi, il tient de la logique d'une ressource fichier que l'on n'autorise pas simultanément écriture et lecture.
Par contre, on peut interdire l'exécution simultanée de deux actions parce qu'elles partagent et modifient toutes deux une variable d'état.

Nous allons donc analyser la structure des unités d'écriture d'application à travers les problèmes suivants :

- prise en compte des requêtes,
- structuration des codes de traitement,
- exécution des processus intervenant dans ce traitement
- envoi de requêtes à d'autres unités, désignation de celles-ci,
- contrôle de l'exécution des services.

Cependant, les choix que peut faire le concepteur d'un langage sur le type d'unité, et surtout le type d'activité de l'application, a une importante influence sur ces différents problèmes. Il ne nous paraît donc pas possible de traiter simultanément l'étude du schéma procédural et celle du schéma des processus résident. De l'une à l'autre, le plan idéal que nous venons de décrire variera quelque peu. On verra ainsi que, dans le premier schéma, "prise en compte et structuration" sont liées et que dans le second on ne pourra guère dissocier la prise en compte des requêtes et le contrôle.

3.2 - Structure des unités construites sur un schéma procédural

Les langages que nous pensons pouvoir rattacher à ce schéma sont les Distributed Processes (DP), la proposition d'extension des moniteurs de G. PACINI, IGOR, SESAME. On y ajoute les expressions de chemin et la proposition des compteurs qui, quoique s'intéressant surtout au contrôle, se basent sur une structure procédurale des unités.
ADA présente un problème de classification car il rappelle les interactions procédurales dans l'aspect extérieur des modules, mais la structure interne de ceux-ci est propre au schéma des processus résidents. On l'étudiera donc en 3.3.

3.2.1. Logique du schéma procédural

Nous avons, dans le chapitre 2, rattaché très rapidement le premier type d'implantation des processus (propagation dans les unités) à la notion de procédure. Ce n'est pas forcément général. On peut tout à fait concevoir des unités proposant des points d'entrée aux processus sans que ceux-ci reviennent à l'unité dont ils proviennent. On peut donc admettre que l'interaction entre unités soit exprimée par des goto et n'impliquent pas de retour. Du moins pas de retour en séquence. OURS[BRIAT 76] offre pour une communication par message un schéma de ce type. C'est la seule proposition récente dans ce cas, du moins parmi celles dont nous avons connaissance. On peut dans doute attribuer la préférence des concepteurs pour le schéma procédural aux raisons suivantes :

- Le recours aux méthodes d'analyse descendante conduit à considérer un service offert par une unité comme une instruction d'une machine abstraite. Il est donc naturel de poursuivre en séquence après l'exécution de cette instruction.
- L'appel d'une procédure rendant un service conserve le contexte de l'appelant au retour. Celui-ci dispose donc tous les éléments pour traiter la réponse. Une autre implémentation où une réponse n'arrive pas en séquence mais dans un autre élément de l'unité, et où elle peut arriver à n'importe quel moment, impose la recherche du contexte dans lequel la demande a été faite (il peut y en avoir eu plusieurs s'il y a des activités multiples).

Pour toutes ces raisons, on ne s'intéressera qu'aux interactions du type appel procédural. Et celui-ci amène la notion de procédure et indique donc une certaine forme de structuration des modules, c'est cette structuration que nous étudions en 3.2.2.

Remarque

Nous allons encore une fois un petit peu trop vite en disant que l'appel habituellement nommé procédural impose la structure des procédures. On pourrait fort bien imaginer un module comme un programme non structuré parsemé de primitives indiquant un point d'entrée ou un point de retour. Cela n'a inspiré personne, sans doute pour des raisons évidentes de clarté.

3.2.2. Procédures : Ecriture et éxecution

Nous voici restreints à cette définition d'unité qui est la suivante : un ensemble de procédures auxquelles il faut ajouter des éléments de mémorisation d'état : les variables globales.

On connaît bien les éléments d'écriture des procédures :

- un en-tête
  - décrivant le nom de la procédure,
  - précisant éventuellement si elle rend une valeur typée : fonction ,
  - décrivant les paramètres formels de la procédure , valeur d'entrée : paramètres par valeur, valeurs rendues en résultats : paramètres par résultat variables appartenant au demandeur et pouvant être accédés par la procédure : paramètres par nom ou référence.
- un ensemble de variables locales, variables de travail de la procédure
- éventuellement la description de procédures imbriquées et propres à cette procédure,
- la liste d'instructions décrivant les actions de la procédure.

Remarques

1) Si la procédure est écrite en langage de bas niveau (ne comportant pas la notion explicite de procédure), cette structuration peut se simplifier (pas de variables locales par exemple) et est le résultat de conventions d'écriture. Ce n'est généralement pas le cas des langages étudiés.

2) Nous avons vu en 2.3.3 que le passage de paramètres par nom ou référence n'est guère souhaitable pour des raisons liées à la configuration : procédures distantes et surtout pour éviter la propagation d'erreurs.

Cette structure procédurale a un avantage important : on associe clairement l'accès à un service et l'algorithme de celui-ci.
L'en-tête définissant l'interface de la procédure est décrit en association avec les instructions qui exécutent ce service. Les services sont nettement séparés. D'autant plus séparés dans IGOR que l'unité n'est autre que la procédure,mais la remarque reste valable sur l'ensemble d'un programme IGOR.
Il y a peu de choses à dire sur la structure d'une procédure qui ne soit déjà largement connue,de même qu'est connue la façon dont s'y déroule l'activité :

- début d'exécution avec
  - . apport des paramètres (via un message en application répartie au moins lorsque l'appel est distant),
  - . construction d'un bloc de variables locales (gérées dynamiquement en pile ou ce qui en tient lieu pour une exécution distribuée)
- exécution de l'algorithme qui peut comporter des appels à d'autres procédures,
- fin de l'exécution avec libération du bloc de variables locales et si nécessaire envoi des paramètres par retour. Retour en séquence de l'appel dans la procédure appelante.

Ce déroulement n'est remis en cause par aucun auteur. Les problèmes se posent lorsqu'il existe des activités concurrentes dans l'application et que l'on doit contrôler leur exécution cohérente. Ainsi, plusieurs processus peuvent appeler la même procédure ou plusieurs procédures de la même unité. Il se peut que ces accès concurrents soient possibles, mais très souvent pour des règles de séquencement, de synchronisation (3.1) ils seront contrôlés. Le paragraphe suivant étudie différentes méthodes pour exprimer ce contrôle.

3.2.3. Contrôle des activités

Il y a deux méthodes pour exprimer le contrôle : l'intégrer au code des services, ceux-ci se contrôlent alors réciproquement,ou l'exprimer à part et donc administrer l'exécution des services de l'extérieur. Il peut sembler illogique d'analyser le contrôle à partir de sa localisation, nous allons voir que le choix de la localisation impose cependant de nombreuses caractéristiques de ce contrôle.

3.2.3.1. Contrôle exprimé à l'intérieur des services

Quels sont les outils dont peut disposer un programmeur pour exprimer dans chaque service l'interaction avec les autres ?
A priori un processus exécutant un service pourra :

- bloquer l'exécution d'un ou plusieurs autres processus
- libérer d'autres processus
- se bloquer

Les actions de blocage ou de libération pourront se faire sur des processus exécutant le même service (contrôle de la réentrance) ou d'autres services. Elles ne comportent jamais la désignation explicite du processus à bloquer ou libérer, mais il faut admettre qu'on ne peut guère connaître a priori le nom des processus concurrents. Elles ne pourront donc se faire qu'à travers des entités propres au contrôle (sémaphores, événements, conditions des moniteurs), c'est-à-dire indirectement.
Il serait possible d'indiquer le nom d'un service que l'on désire libérer ou bloquer soit dans son accès, soit en cours d'exécution. Dans ce deuxième cas on ne connaîtrait pas forcément le lieu du blocage éventuel et l'état des variables risquerait d'être incohérent, aussi évite-t-on cette méthode.
Le premier cas est utilisé dans les moniteurs mais implicitement par l'exclusion mutuelle de l'exécution, encore n'a t-on aucun moyen d'empêcher le lancement d'un service une fois l'exclusion mutuelle levée.
On en est donc réduit aux mécanismes de contrôle indirects, et l'on peut faire deux remarques à leur sujet :

Note : cette partie paraîtra peut être un peu longue au lecteur mais la notion de D-module dépend beaucoup du choix de séparation du contrôle que nous avons faite. Aussi avons-nous essayé d'analyser très complètement ce domaine.

- ils nécessitent l'utilisation de primitives explicites (de test de blocages, de libération),
- pour l'exécution de ces primitives, on a besoin d'un début d'exécution. A plus forte raison si l'exécution de ces primitives requiert le test de variables locales ou d'état.

Ceci signifie :

- que le contrôle est fortement mêlé à l'algorithme, entre deux actions, après l'évaluation de certaines conditions etc,
- qu'il faut assurer que lorsqu'un processus se bloque il laisse les variables dans un état cohérent : choix judicieux des points d'arrêt dans l'exécution.

L'expression du contrôle sous cette forme est donc délicate et pas forcément très claire au lecteur du programme. En contrepartie on peut penser qu'il sera assez efficace car exprimé au moment opportun dans l'algorithme des services.

On conclut ce paragraphe par l'étude de deux aspects particuliers, les mécanismes employés et l'affinement du contrôle.

Mécanismes

Parmi les mécanismes de contrôle, nous avons cité les sémaphores, les événements, les conditions des moniteurs. Les deux premiers sont sans doute trop élémentaires pour être proposés dans des langages évolués mais ils servent souvent à implémenter des mécanismes plus sophistiqués. En ce qui concerne la proposition des moniteurs de HOARE, elle sert de référence à la plupart des propositions de langages procéduraux modulaires et a atteint un incontestable succès, SESAME et SIMONE utilisent ce mécanisme. La proposition de G. PACINI se sert de cette base, elle sera d'ailleurs étudiée dans le sous-titre suivant. On peut citer comme avantage des moniteurs, la notion de condition qui met relativement bien en avant la correspondance de différents points de contrôle d'un service à l'autre.

Autre mécanisme, la condition WHEN de Brinch Hansen dans DP qui permet de suspendre l'exécution d'un processus tant qu'une ou plusieurs conditions booléennes ne sont pas réalisées. Ce mécanisme proche des commandes gardées de DIJKSTRA permet de définir plusieurs alternatives, chacune en regard d'une condition. La première condition devenant vraie détermine le traitement suivant le déblocage. Si plusieurs sont vérifiées le choix est non déterministe. On exprime donc ainsi clairement les points d'arrêt et les diverses actions qui peuvent suivre. Les points de déblocages sont,

eux, disséminés dans toutes les instructions où l'on modifie les variables utilisées dans les commandes gardées.

Affinement du contrôle

Nous avons introduit en 3.1 une idée de décomposition du contrôle en règles de séquencement, de synchronisation et d'ordonnancement. Force nous est de constater que cette distinction n'est pas apparente dans les langages étudiés. De toute façon l'inclusion du contrôle dans les services rend difficile la séparation du séquencement et de la synchronisation : comment distinguer un blocage lié à l'impossibilité de rendre un service et un autre protégeant l'accès à certaines variables ? Comment, sinon peut-être par des commentaires dans le programme ?
L'intérêt de la proposition de G. PACINI est de tenter une telle distinction. On peut associer des expressions booléennes à l'entrée de chaque procédure, ces expressions indiquent dans quelles conditions l'exécution est possible. C'est une façon de formaliser le séquencement des actions suivant l'état d'un module. Des points d'arrêt, proches des conditions des moniteurs, permettent à l'intérieur des procédures de compléter ce séquencement ou d'exprimer la synchronisation (les deux liés ici). Mais la file d'attente liée à une condition peut être gérée explicitement (FIFO ou ordonnée par priorités). Si c'est une gestion par priorité on indique celle-ci au moment du blocage. L'opérateur de déblocage libère évidemment le processus le plus prioritaire. C'est un outil d'ordonnancement limité mais d'emploi très facile.

3.2.3.2. Contrôle exprimé hors des services

Logique de cette approche

Dans les deux propositions qui ont adopté ce type d'écriture (expressions de chemin, compteur) les modules sont composés :

- des variables et des procédures habituelles
- d'un programme de contrôle, non algorithmique ou du moins* n'ayant pas accès aux variables du module et écrit dans un langage différent de celui des procédures. En d'autres termes ce contrôle ne partage rien avec les procédures.

* Dans leur écriture, les expressions de chemin peuvent ressembler à un algorithme, mais leur compilation en terme de sémaphores morcelle cet algorithme général en autant de prologues et d'épilogues qu'ils y a de procédures. Voir plus loin ces notions.

Il est facile de discerner les conséquences d'une telle approche :

. Le contrôle ne dispose pour toutes données que de la connaissance
  - des événements début et fin d'exécution des procédures,
  - du nombre de demandes sur chacune (pour les compteurs seulement)

  Aucun autre événement pouvant intervenir dans le cours de l'exécution des services ne peut être connu. En effet, on devrait avoir recours à des entités partagées (condition, variables..) ce qui est exclu des langages étudiés.

. Inversement, et pour les mêmes raisons, le contrôle ne s'exerce que sur l'entrée des procédures, bloquant ou libérant l'accès. (Le blocage ou la libération de la fin des procédures n'aurait a priori guère d'intérêt).

. Enfin, puisqu'il n'existe pas d'activités résidentes dans le module, ce sont les processus demandeurs qui exécutent les actions liées au contrôle. Pour cela le programme qui l'exprime est compilé sous forme de prologue et d'épilogue pour chacune des procédures concernées

Avant de discuter l'emploi de ces outils nous allons décrire rapidement les deux propositions.

Compteurs

Le contrôle dispose pour toute donnée de trois séries de compteurs gérés automatiquement pour chaque procédure :

- le nombre d'exécutions commencées depuis le lancement de l'applicatio
- le nombre de fins d'exécutions
- le nombre de demandes d'exécution.

Nous les nommerons pour une procédure i : CDEB(i), CFIN(i) et CDEM(i).

Cette procédure est active si CDEB(i) > CFIN(i)

le nombre de processus l'exécutant est CDEB(i) - CFIN(i)

le nombre de processus bloqués au début de son exécution est CDEM(i) - CDEB(i).

A partir de ces compteurs on exprime pour chaque procédure les conditions qui doivent être vérifiées pour son exécution.

Pour indiquer que i est en exclusion mutuelle avec j :

(CDEB(j) - CFIN(j) = 0) ou (CDEB(i) - CFIN(i) = 0)

Pour indiquer que i et j s'exécutent tour à tour en exclusion mutuelle : i puis j puis i etc.

Prédicat d'entrée de i : (CDEB(i) = CDEB(j)) et (CFIN(j) = CDEB(j))

Prédicat d'entrée de j : (CDEB(i) = CDEB(j)+1) et (CFIN(i) = CDEB(i))

ect.

Expressions de chemin

Plus connu que le précédent, ce mécanisme sert souvent de base pour la comparaison de nouveaux mécanismes de contrôle. La description qui suit s'inspire de [BEKKERS].

On dispose d'un langage d'écriture qui permet d'indiquer l'ordre dans lequel les procédures doivent être exécutées et dans quelle mesure des exécutions parallèles sont possibles.

définition de l'ordre d'exécution :

p; q; r c'est-à-dire p entièrement puis q puis r

exécution sélective d'une procédure parmi plusieurs :

p, q, r p ou q ou r mais l'une seulement

exécution simultanée :

{p} *p est réentrante*

{p,q} *p et q peuvent s'exécuter parallèlement et sont réentrants*

boucle de contrôle :

path ..... end le contrôle exprimé entre path et end une fois exécuté est répété

ex : path p;q end : exécution de p puis q puis p etc

Ces primitives ont été enrichies par la suite (degré de réentrance) mais sous cette forme elles nous donnent une bonne idée des principes généraux. Précisons qu'elles peuvent être combinées pour former des expressions plus complexes.

ex : p; (path p; (q,r) end, z)

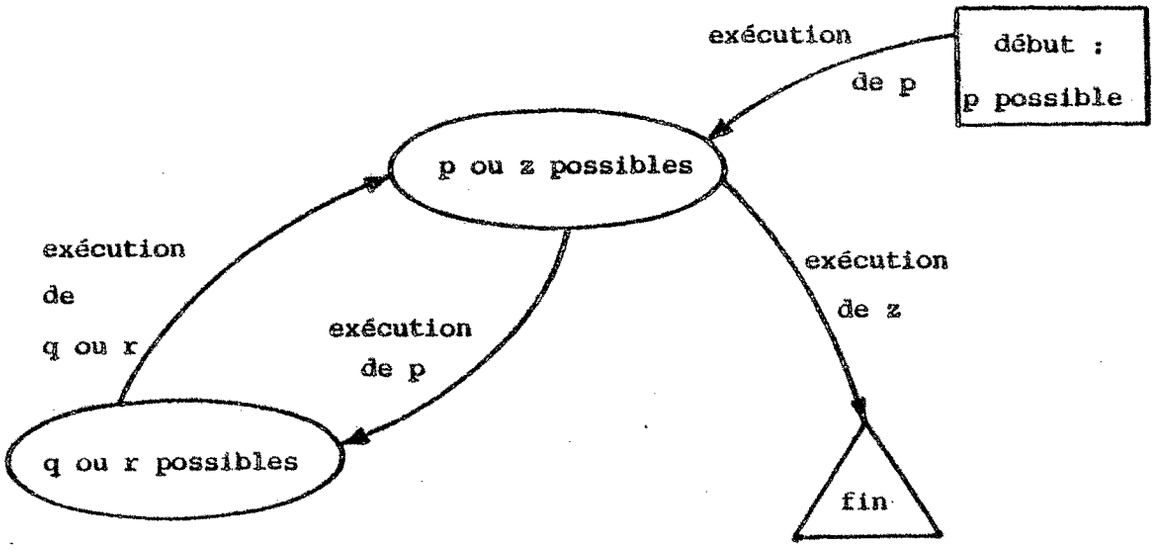
p puis une boucle sur (p puis q ou r)

on termine lorsqu'au lieu de la boucle on exécute z.

Ces directives concernent uniquement l'ordre d'exécution des procédures en général et non pas leur ordre d'exécution pour chaque processus utilisateur. C'est également le cas pour les compteurs.

Notons enfin que ce formalisme peut mettre relativement bien en évidence l'automate d'état fini qui définit les changements d'état du module et les procédures qu'il peut exécuter pour chaque état.

Pour l'exemple ci-dessus :



Les deux propositions (compteurs et expressions de chemin) de par leurs traits communs peuvent être commentés globalement.

Discussion

Les deux propositions ont en commun, leur expression séparée du contrôle c'est-à-dire un certain nombre d'avantages :

- le contrôle isolé des actions est sans doute plus lisible que dans les propositions étudiées en 3.2.3.1;
- il joue sur la notion de procédure (de service) dans son ensemble c'est-à-dire sur des considérations fonctionnelles homogènes avec la description externe du module. On "voit" clairement que lecture et écriture sur un fichier ne sont pas possibles simultanément. Dans d'autres propositions un processus lecteur s'interrompt sur le test d'une variable indiquant une écriture en cours. La correspondance est moins évidente;
- les domaines du contrôle (ordonnancement etc) sont relativement distincts :
  - . dans les expressions de chemin on peut indiquer l'ordre (;), l'alternative (,) la réentrance ({}). Mais on n'a pas la possibilité d'indiquer une préférence (ordonnancement),
  - . les compteurs s'ils ne distinguent guère séquencement et synchronisation permettent de connaître la taille des files d'attente ce qui peut guider un choix. Exemple : exécution de p s'il n'y a pas trop de processus bloqués sur q, sinon q etc.

Elles ont également un certain nombre de désavantages qui sont dûs essentiellement aux choix initiaux :

- peu de connaissance sur l'état du module (pas d'accès aux variables)
- non prise en compte de l'identité des processus
- forme non algorithmique (compteurs) ou peu souple (peu de primitives dans les expressions de chemins).

On donne ici deux exemples de ces inconvénients, une comparaison plus poussée entre expressions mélangée et séparée du contrôle est faite dans [BEKKERS].

Manque de connaissances : Il est naturel de penser que les informations (paramètres) accompagnant une requête sont importantes dans la modification de l'état du module, donc pour la poursuite de l'exécution. Supposons qu'un allocateur gère N ressources identiques et que chaque utilisateur puisse demander plusieurs ressources à la fois. Ce nombre exprimé en paramètre n'est pas connu du contrôle, qui ne peut pas non plus consulter la variable indiquant le nombre de ressources disponibles. Entre autre problème: , comment indiquer au contrôle qu'il ne reste plus de ressources libres et que donc l'accès à la procédure d'allocation doit être suspendu ? La solution proposée est la suivante :

. on crée une procédure supplémentaire, qui ne fait aucune action
. l'allocateur de i ressources comporte i appels à cette procédure. Le contrôle a alors le moyen de comptabiliser les allocations. On fera de même pour les libérations de ressources.

En résumé, pour résoudre ce problème assez courant il aura fallu créer deux procédures additionnelles et les gérer. Le contrôle en sera d'autant plus complexe.

Identité de processus : Nous avons vu que les règles de séquencement s'appliquent aux procédures et ne concernent pas les processus utilisateurs. Supposons qu'un module gère l'accès à un fichier : il offre les procédures

OUVRIR, *LIRE*, *ECRIRE*, *FERMER*. Limitons-nous à l'aspect séquencement d'un accès en écriture, il peut s'écrire avec les expressions de chemin de la façon suivante : *OUVRIR* ; *(path ECRIRE end, FERMER)*
soit *OUVRIR*, puis un nombre quelconque d'écritures puis *FERMER*. Si un processus autre que celui qui a exécuté *OUVRIR* appelle *FERMER*, le contrôle n'a pas de moyens de l'en empêcher. A l'intérieur de *FERMER* on peut détecter

l'erreur mais comment en informer simplement le contrôle afin d'autoriser à nouveau les écritures et la fermeture au processus initial ? La solution réside encore dans l'utilisation de procédures annexes et c'est un facteur de complexité du contrôle.
Ayant examiné les différents aspects du contrôle dans le schéma procédural, nous allons pouvoir conclure l'étude de celui-ci par la description des appels de procédure.

3.2.4. Appel et désignation de procédures

Les formules d'appel et de désignation proposées dans la littérature sont relativement proches.
En règle générale le nom des procédures externes appelées est connu dans l'écriture des unités ainsi que le type des paramètres. L'appel est exprimé par l'indication du nom de la procédure suivi de la définition des paramètres réels. Il n'est jamais soumis à un contrôle d'activité, ce qui paraît normal puisque celui-ci dépend de l'état de l'unité appelée, non de celui de l'appelante. Par contre, les auteurs insistent sur la nécessité du contrôle des types de paramètres, afin de vérifier que l'interface fournie par une procédure est bien respectée lors des appels. On pourra effectuer ce contrôle :

- à la compilation puis à l'édition de liens,
- ou dynamiquement lors de l'appel.

On décrit ici la désignation et le contrôle de types dans SESAME, où ils ont fait l'objet d'une étude particulière.

3.2.4.1. Désignation et contrôle de type dans SESAME

Caractéristiques générales du langage

SESAME est un langage de type procédural permettant la construction de modules (ensemble de procédures + variables d'état). En ce qui concerne l'expression du contrôle des activités SESAME reprend le schéma de concurrent PASCAL. C'est-à-dire que les modules peuvent être :

- des "process" : un programme associé à un processus où celui-ci est créé
- des classes : modules sans contrôle
- des moniteurs : modules disposant d'un contrôle par :
  - exclusion mutuelle des procédures
  - synchronisation par conditions.

Le langage est issu de PASCAL, modifié pour inclure :

- la notion de module
- la notion de métavariables (paramètres de compilation)
- le contrôle par moniteur
- un mécanisme de traitement d'erreur (voir §4.3)
- la désignation et l'appel décrits ci-dessous.

Désignation et appel

Chaque procédure du module accessible de l'extérieur est indiquée par le mot clé ext précédant sa déclaration;on indique ensuite son nom, ses paramètres formels, ses variables locales etc.

Toute procédure extérieure utilisée dans un module est déclarée dans ce module :

. *par ref procédure <nom du module> .. < nom de la procédure> si l'on connaît déjà l'identité de cette procédure.*

. *par dummy procédure <nom local>*
dans le cas contraire, ou si l'identité de la procédure peut changer d'une utilisation du module à l'autre.

Dans les deux cas, on décrit ensuite les paramètres formels. Dans le corps du programme ces procédures sont appelées normalement par le nom cité, nom composé pour le premier cas, nom local pour le second.

Après compilation une phase de connexion permet de résoudre les problèmes de désignation encore non résolus en précisant le nom réel des dummy procédures. Ceci se fait par des directives telles que :

```
with <nom de module> do
    begin
    <nom local de dummy> := <nom de module> .. <nom de procédures>
                                de la procédure réelle
    end;
```

Contrôle des types des paramètres

A l'intérieur même d'un module le compilateur vérifie que l'appel des procédures est conforme à leur déclaration. Si ces procédures sont extérieures, la vérification se fait par rapport à la déclaration locale (ref. procédure ou dummy..).

La vérification de la compatibilité des types entre module d'une procédure et modules utilisant cette procédure se fera à la connexion à partir des données suivantes :

- Les types non simples (autres que entier, caractère..) communs à plusieurs modules doivent être définis dans une structure particulière, l'environ. Celui-ci est simplement une suite de lignes où les types communs sont déclarés de la même façon qu'ils pourraient l'être dans un module.

- Dans un module utilisant un environ, on référence cet environ par env <nom de l'environ> parmi les déclarations de type propres au module. A ce niveau cette primitive joue le rôle d'un include de PL1, on insère les lignes de l'environ dans ce programme.

- Dans la déclaration des procédures extérieures, ou dans l'en-tête des procédures extériorisées (accessibles de l'extérieur), les types non simples des paramètres formés doivent être définis par
  <nom de l'environ qui les contient> .. <nom du type dans l'environ>
  Le compilateur génère donc dans le code produit la suite de noms, et le connecteur vérifiera ensuite que les suites de noms correspondent dans les différents modules.

- Le système est protégé de la malveillance des utilisateurs car tous les éléments de programmation : modules, sources, environs, modules binaires sont conservés par une base de données spécialisée : le bibliothécaire.
  Celui-ci maintient la cohérence des objets conservés et détruit, par exemple, lors d'une modification d'environ, tous les modules binaires dont la compilation a utilisé cet environ.
  On ne pourra donc :
  1) Compiler un module avec un environ
  2) Modifier l'environ
  3) Compiler un autre module utilisant le même nom d'environ
  4) Connecter les deux modules.

  En effet, le premier résultat de compilation aura été détruit lors de la phase 2.

Cette description de SESAME clôt la section 3.2. qui analysait les différents aspects du schéma procédural. Nous allons maintenant faire la même analyse pour le schéma des processus résidents.

3.3 - Structure des unités construites sur le schéma des processus résidents

3.3.1. Quelques interprétations possibles

Rappelons certaines caractéristiques de ce schéma :

- L'exécution d'une unité est confiée à un ou plusieurs processus attachés à cette unité.
- Deux unités différentes ne partagent pas de données.
- La communication est faite par l'envoi de messages que l'on destinera à une file d'attente; il peut y avoir plusieurs files d'attentes associées à une unité.

En résumé, suivant qu'il y a plusieurs processus, chacun spécialisé ou non dans tel service, suivant le choix d'attribution des files d'attentes : à tous les processus globalement, ou à un seul exclusivement, on s'adressera à un module en précisant :

- le nom d'une file
- ou le nom d'un processus
- ou même le nom d'une file et celui d'un processus.

On le voit, le double choix suivant, une ou plusieurs voies d'accès, un ou plusieurs processus, rend toute définition générale très vague. Pour faciliter notre analyse nous écarterons le cas où plusieurs processus se partagent une même unité puisqu'aucun langage de notre échantillon ne fait ce choix. Nous nous baserons donc sur la définition suivante :

> Une unité est exécutée par un processus qui reçoit des messages sur une ou plusieurs voies (ou files d'attente).

Nous avons écarté l'hypothèse d'une communication indirecte par message, via des boites à lettres par exemple, pour les raisons suivantes :

- soit il n'existe qu'un seul processus susceptible de retirer des messages, cette boîte à lettre n'est alors rien d'autre qu'une file d'attente des messages vers ce processus et l'on est ramené à notre définition,
- soit plusieurs processus peuvent retirer les messages; on peut alors considérer qu'ils rendent un service équivalent et puisque nous nous intéressons ici à la structure d'une unité l'analyse suivante est valable pour chacun d'entre eux.

Comme nous l'avons fait pour le schéma procédural, nous allons tenter de dégager les caractéristiques essentielles du schéma des processus résidents. Nous le ferons à travers les aspects suivants :

- Structuration des codes et implémentation des services; contrôle de l'activité. Nous serons amenés à étudier les deux domaines conjointement.
- Détail des interactions entre unités : émission de requêtes, désignation, gestion des files, notion de message.

3.3.2. Structure des unités et contrôle

3.3.2.1. Structure et notion de service

En abordant le schéma procédural, il a été relativement aisé de montrer que le choix de l'implantation de l'activité imposait pratiquement une structure bien précise des unités. Le type de changement de contexte le plus souple étant l'appel procédural il était naturel de structurer les unités comme un ensemble de procédures. Le contrôle se déduisait de cette structure même si l'on pouvait l'exprimer à l'intérieur ou à l'extérieur des procédures.

Dans le schéma des processus résidents, il est moins facile d'interpréter les choix des concepteurs comme découlant de contraintes logiques. Plus exactement, il n'apparait pas de structure type de ce schéma. Aussi, nous allons plutôt passer en revue les structures choisies et essayer d'interpréter la démarche des concepteurs.
Grossièrement, deux approches sont utilisées :

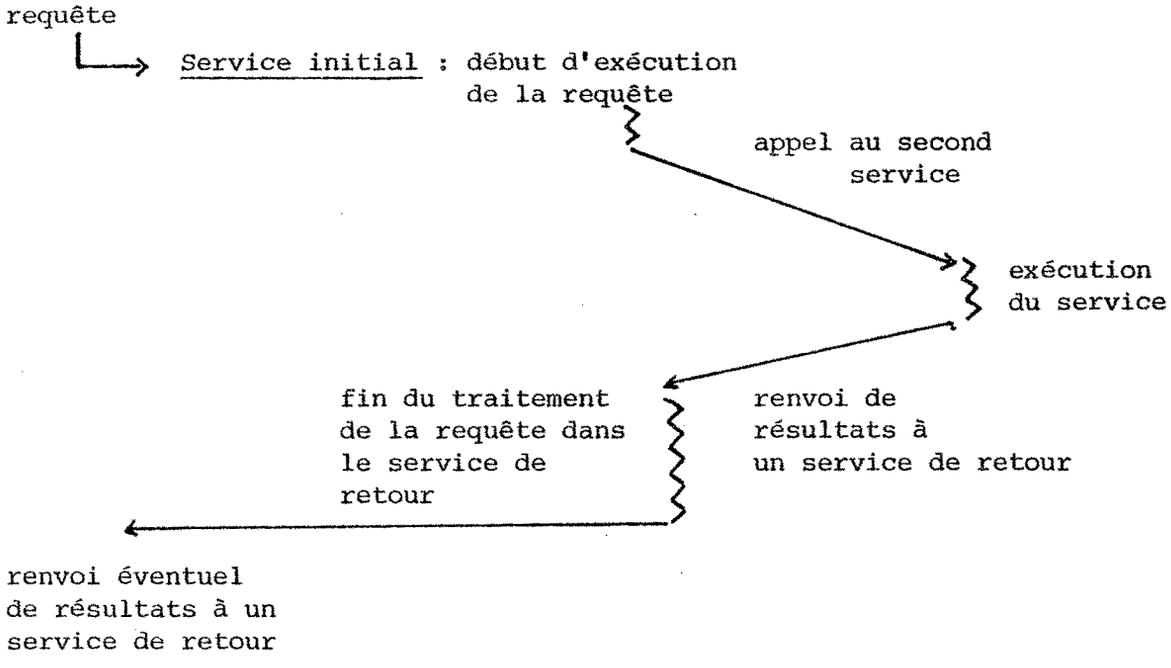
- l'unité est structurée en services distincts, (MER)
- l'unité ne contient qu'un algorithme général : l'algorithme du processus résident. Les différents services ne correspondent à aucune structure particulière (ADA, CSP, PLITS)

MER procède de choix originaux qui nous amènent à le décrire séparément, le reste de la section 3.3. concernera ADA, CSP et PLITS et les traitera globalement.

3.3.2.2. Structure et contrôle de MER

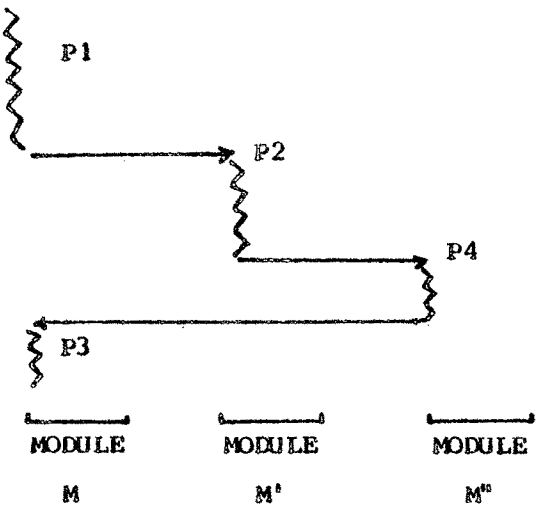
Un module MER est composé de services écrits comme des procédures de PL1. PL1 sert d'ailleurs de base au langage. Tout service comporte des paramètres qui définissent la structure des messages traités.
Si ce projet n'a pas été abordé dans le schéma procédural, c'est parce que l'exécution d'un service est parallèle avec celle du service demandeur. Il s'agit donc d'un appel asynchrone.
Si des résultats doivent être retournés au demandeur, ou s'il faut être informé de la fin d'exécution du service, les paramètres de la demande comportent le nom d'un service de retour. Le service demandé expédiera donc ces résultats en paramètres d'une requête au service de retour.

L'exécution d'un service comportant l'utilisation d'un second service se déroulera ainsi :



Le traitement d'une requête est donc souvent scindé en deux services d'un même module. Mais :

- on peut prévoir plusieurs services de réponse suivant les résultats rendus : ex : exécution normale + détection d'erreur
- on peut indiquer en paramètre des noms de services d'autres modules : exemple :
  - . P1 envoie une requête à P2 (retour à P3)
  - . P2 envoie une requête à P4 (retour directement à P3)



Le contrôle est limité à la contrainte d'exclusion mutuelle des services d'un même module. Il n'est pas possible d'interdire explicitement l'accès à certains services suivant l'état du module.
En résumé, MER offre une structuration en modules qui identifie bien voie d'arrivée des requêtes et traitements spécifiques. Le contrôle d'exécution est simple et implicite. Le programmeur n'a donc pas à s'en soucier.

3.3.2.3. Structure et contrôle dans ADA, CSP et PLITS

Ces trois langages gèrent très différemment les messages, mais ont des structures très semblables.
En ce qui concerne la gestion des messages :

ADA possède des files d'attentes définies à l'avance (ENTRY)

PLITS offre une seule file réelle mais l'indication d'une clé permet de "multiplexer" cette file (TRANSACTION)

CSP offre autant de voies qu'il y a d'utilisateurs.

On reviendra sur ces points en 3.3.3, il nous suffit pour l'instant de noter que dans les trois langages, le processus indique explicitement sur quelle(s) voie(s) il attend un message. Si aucun message n'est disponible, le processus se bloque en attente.
Dans les trois langages, l'unité est composée des variables d'état et d'un algorithme général qu'exécute le processus résident. Celui-ci au cours de son exécution consomme les messages en attente, en sélectionnant éventuellement les voies d'accès (ADA, PLITS) ou les expéditeurs (CSP). Ceci signifie :

- que les différentes actions correspondant à différents services ne forment pas des structures séparées,

- et que le contrôle des actions à exécuter est exprimé par l'algorithm global lui-même.

  Ainsi : . Le processus consomme les messages de voies explicitement choisies. Cette sorte de "cueillette" guidée correspond à ce que l'on a nommé séquencement.

  . L'aspect séquentiel de l'algorithme évite tout problème d'exclusion mutuelle (synchronisation).

Ainsi, à cause du choix de la structure (algorithme général) le contrôle est disséminé dans tout le programme. Le programmeur peut cependant essayer de structurer son unité en ayant recours le plus possible à des procédures séparées. L'algorithme général se trouve alors réduit aux traits généraux concernant les grandes phases de la vie du processus. Les actions plus élémentaires étant alors décrites dans les procédures. Mais cette technique n'est pas imposée par les mécanismes proposés.

Nous étudions le détail de ces mécanismes dans la section suivante.

3.3.3. Emission, réception de requêtes, notion de message

Pour faciliter l'analyse des trois langages sur ces points nous les décrivons séparément, tout en mettant en évidence leurs originalités, c'est-à-dire leurs différences.

3.3.3.1. Le langage PLITS

PLITS propose des unités de type modulaire. Un module se compose, outre les variables et l'algorithme du processus résidant, de la définition de "types modules". Ceux-ci sont des images de modules (paramétrables) à partir desquels peuvent être créés par requête programmée de nouveaux modules. On peut ainsi créer de nouvelles unités pendant le cours de l'exécution de l'application.

Chaque module peut mettre un terme à son activité (mort du module) par une requête programmée.

L'envoi d'un message à un autre module se fait par une primitive send où l'on indique :

- le nom du module destinataire
- le message envoyé (voir plus bas)
- éventuellement une clé (transaction) qui permettra une réception sélective.

Cet envoi effectué, le processus expéditeur continue son exécution. Le traitement de la requête peut donc se faire parallèlement à l'exécution du demandeur ou même après sa mort.
La réception d'une requête est programmée explicitement grâce à l'instruction receive. On y indique le nom de la variable message qui reçoit les valeurs transmises. Cette primitive est bloquante en l'absence de message. On peut éventuellement restreindre le choix du message à consommer parmi ceux en attente en exprimant :

- des conditions booléennes sur la valeur ou l'existence de champs du message,
- le nom d'une clé, qui doit être identique à celle indiquée à l'expédition.

Un message est composé d'une suite de couples (NOM, valeur typée). Les noms sont communs à tous les modules et indiqués en tête de ceux-ci avec le type des champs auxquels ils correspondent. On peut ainsi contrôler la cohérence de la définition de types associés à chaque nom.
Le nombre de couples d'un message peut varier et l'on dispose de primitives pour :

- ajouter, retirer, modifier un couple
- tester la présence d'un couple.

Les valeurs transportées dans les messages, peuvent être de type classique (entier, caractère etc) ou de type transaction ou module. On peut ainsi donner le nom d'une clé à un module pour des échanges ultérieurs, ou établir des liaisons dynamiques.

3.3.3.2. Proposition des CSP

L'originalité de ce langage réside dans la contrainte suivante : non seulement l'expéditeur d'un message nomme le destinataire, mais celui-ci, à la réception nomme l'expéditeur. Les noms des processus communiquants sont connus dès l'écriture, ce qui exclu toute liaison dynamique, et rend difficile la réutilisation des unités dans plusieurs applications différentes.
L'envoi d'une requête d'exprime par

<nom du destinataire> ! <liste de valeurs expédiées>

L'expéditeur est bloqué tant que le message n'est pas consommé (contrairement à PLITS qui permet une exécution indépendante des deux processus communicants). La liste des valeurs expédiées peut varier d'une instruction d'envoi à une autre. Pour un émetteur et un destinataire donnés le format des messages est donc variable.

La directive de réception est :

<nom de l'expéditeur> ? <liste de variables>

elle estbloquante tant que le message n'est pas parvenu au processus. Le type des variables indiquées, leur nombre, doivent correspondre au type et au nombre des valeurs expédiées. La vérification ne peut être que dynamique.

Si plusieurs messages de plusieurs expéditeurs ou de formats différents sont susceptibles d'arriver au processus à un moment donné, on exprimera cette attente multiple par une commande gardée. Celle-ci se compose de plusieurs alternatives composées d'une expression booléenne et/ou d'une directive d'attente de message. A chaque alternative est associée une séquence d'instructions.

La première alternative dont l'expression est vraie, et si nécessaire pour laquelle un message est disponible, indique la séquence à exécuter. Si plusieurs remplissent ces conditions, on exécute une des séquences au hasard. Si l'alternative indiquait une réception de message, celui-ci est bien sûr consommé.

Les commandes gardées qui ne comportent aucune instruction d'échange avec l'extérieur correspondent à des instructions conditionnelles habituelles. C'est d'ailleurs dans ce but qu'elles ont été initialement conçues par DIJKSTRA.

Exemple 1

x ≥ y → z := x | y ≥ x → z := y rend dans z le maximum de x et y

alternative 1 — instruction — alternative 2

Exemple 2

aiguillage : le processus p met en communication soit i soit j avec k
(i si indic est vrai, j sinon)

```
programme de p
n : integer; indic : boolean;
...
indic; i ? (n) → k!(n) |
not indic; j?(n) → k!(n)
```

3.3.3.3. Contrôle et communication dans ADA

On se limite à ces traits d'un langage très complet qui serait, sinon, trop long à décrire.

ADA offre une amélioration notable de la formalisation externe des unités en leur donnant un aspect procédural. Chaque unité présente à ses utilisateurs un ensemble de entry connus par un nom et une liste de paramètres formels par valeur ou résultat. Chacun correspond à un service possible. Les requêtes à ces services ont, pour l'utilisateur toutes les caractéristiques d'un appel procédural :

- indication des paramètres réels (qui doivent correspondre au type des paramètres formels),
- indication du nom de l'entry
- continuation en séquence après l'appel, avec les résultats de l'interaction rendus dans les paramètres par résultat.

La structure interne des unités ne reproduit cependant pas la structure procédurale, c'est comme dans PLITS et CSP un algorithme dans lequel le processus résident consomme suivant ses besoins les requêtes en attente sur les entry. Celles-ci lui apparaissent comme des files d'attente classiques.

La primitive de réception de message est accept, elle est accompagnée du nom de l'entry référencé et d'une liste de variables ou expressions qui reproduisent la liste des paramètres de l'entry.

Exemple

```
x : char;
entry READ (PAR : out char);
...
x := (lecture physique d'un caractère);
accept READ (x : out char);
```

Cette unité rend un par un à ses utilisateurs les caractères qu'elle lit sur le support physique. *Out* indique un paramètre par résultat. L'appel de *READ* chez un utilisateur pourrait s'écrire : *READ(car);*

Une deuxième forme de accept permet d'exécuter une séquence d'instructions entre accès au message et retour des paramètres par résultat.

Exemple : pour obtenir le même résultat que ci-dessus on aurait pu écrire :

```
entry READ (PAR : out CHAR);
...
accept (x : out char) do
                      x := (lecture physique d'un caractère)
                      end;
```

Enfin ADA dispose d'une commande gardée (select), dont la logique est identique à celle des CSP, à l'indication de l'expéditeur près.

En résumé, avec la notion d'entry, avec la directive do de l'instruction accept et avec l'instruction select on dispose d'outils qui semblent efficaces et pratiques. L'aspect procédural offert aux utilisateurs des entry est particulièrement intéressant. On pourra cependant regretter que cette structuration apparente des services vus de l'extérieur ne corresponde pas à une meilleure structuration interne.

On termine avec cette description de ADA, à la fois l'étude des langages à activité résidente et celle de la structure des unités d'écriture en général.
Le prochain chapitre complète ce que nous avons pu dire de la structure générale des applications distribuées, de leur exécution, de leur contrôle, de leur liaison.

4. COMPLEMENTS SUR LA CONSTRUCTION ET L'EXECUTION DES APPLICATIONS

Ce chapitre aborde des sujets qui sortent quelque peu du cadre de cette thèse ou des points esquissés rapidement dans la description d'un langage. Il en sera de même dans le chapitre de la seconde partie qui donne quelques idées sur l'environnement d'exécution des D-modules. Mais dans les deux cas il nous a paru souhaitable de "situer" cette exécution des applications. On s'intéressera aux sujets suivants :

- Vie des applications : lancement, évolution de structure, fin de l'exécution
- Problèmes de liaison
- Contrôle d'erreur

4.1 - Vie d'une application

Ce problème concerne plutôt celui de la vie de ses composants. On peut en effet supposer que l'application se termine lorsque son dernier processus disparait. (A moins qu'une erreur de l'un d'entre eux provoque l'interruption de toute activité dans certains langages). Quand la création de processus intervient-elle ? Quels sont ceux qui sont créés dès le lancement de l'ensemble du programme ?

On peut utiliser différentes techniques :

- la création d'un processus père unique qui créera d'autres processus par des instructions spéciales (IGOR, NODAL) etc,
- la création de plusieurs processus qui pourront également en créer d'autres (ADA, PLITS),
- la création de tous les processus de l'application dès le lancement (concurrent Pascal, SESAME, CSP).

En ce qui concerne la disparition de ces processus elle peut correspondre

- à l'arrivée en fin de leur algorithme (SESAME, ADA, IGOR, NODAL, DP),
- à une requête programmée (ADA (requête abort), PLITS).

On verra ainsi s'éteindre peu à peu les activités de l'application. On peut faire une remarque importante : ces onze propositions n'introduisent aucune corrélation entre la disparition d'un processus fils et celle de son père, elles ont délaissé le schéma du fork-join ou parbegin parend qui synchronisait le père sur la fin du fils. Si cette synchronisation est à faire on utilisera les mécanismes de communication généraux.

Les processus ne sont pas forcément seuls à apparaître ou disparaître pendant la vie de l'application. On peut se poser les mêmes problèmes en ce qui concerne les unités.
Il est évident que dans le schéma des processus résidents, elles ont la même existence que le processus qui les possède. Même si, comme dans PLITS, on dispose de modèles pour créer autant d'unités soeurs qu'on le désire.
Dans le schéma procédural, les unités sont généralement créées en même temps que l'application et disparaissent avec elles, mais il nous faut prendre quelques précautions :

- On pourrait imaginer la création de nouvelles unités en cours de traitement, mais cela poserait des problèmes de liaison. Celle-ci est en effet le plus souvent statique, exception faite des passages de procédures en paramètre de certains langages.
- On peut supposer que dans le schéma des expressions de chemin, qu'une unité disparaît quand on est en "bout du chemin". L'unité n'étant plus accessible par la suite.
- Enfin, dans SESAME et tous les langages utilisant les moniteurs, les modules réservés à un processus n'ont plus de raisons d'être lorsque ce processus disparaît.

4. 2 - Problèmes de liaison

Nous tentons ici d'étudier l'utilisation respective de la liaison statique et de la liaison dynamique entre unités. Nous appelerons :

- liaison statique, la liaison avec des objets connus à l'avance, qu'elle soit faite avant le lancement ou à l'exécution,
- liaison dynamique, la liaison avec des objets dont on ne connaît pas à priori l'identité.

Le schéma procédural est le domaine privilégié de la liaison statique, seul le passage de procédures en paramètre peut fournir un outil de liaison dynamique. Mais cette liaison est limitée à la durée de l'exécution de la procédure qui a reçu ce paramètre. Aucune structure de donnée n'est jamais proposée pour stocker le paramètre-procédure plus longtemps.
Il se poserait, de toute façon, des problèmes d'existence de contexte lors de l'appel différé d'une procédure imbriquée. Il se peut en effet que la procédure qui la contient ne soit plus active et que donc ses variables globales n'aient pas d'existence.

Le stockage de l'identité d'une voie d'accès est par contre courant dans le schéma des processus résidents. On dispose alors de variables de type adéquat qui mémorise ces identités. Par exemple on peut mémoriser le nom d'un module dans PLITS (puisque c'est à un module que l'on envoie un message). On peut également mettre en communication différents descendants d'un processus dans IGOR.
La liaison dynamique entre processus est particulièrement utile dans certains cas :

- lorsqu'on peut créer de nouvelles unités, que l'on doit alors insérer dans le réseau des unités plus anciennes (PLITS),
- lorsque les réponses à une requête donnée peuvent être envoyées à plusieurs interlocuteurs différents suivant le demandeur (MER).

4.3 - Contrôle d'erreur

La détection et le traitement des erreurs ou anomalies à l'exécution sont peu abordés par les langages de notre étude. Seuls ADA, SESAME et MER proposent des mécanismes spécialisés. Ceci est d'autant plus regrettable que dans les systèmes répartis, les ruptures de communication, les pannes de sites distants s'ajoutent aux erreurs plus classiques : erreurs de programmation, conditions exceptionnelles telles que mauvais accès à un périphérique etc.
Quels sont les outils fournis par ces trois langages ?

MER propose une primitive SURVEY pour la scrutation des interactions. En l'utilisant le programmeur demande au moniteur d'exécution de le prévenir de la fermeture d'une interaction. Cette fermeture peut être due à une erreur, une rupture de liaison entre sites ou a une fermeture demandée par le programme distant. Dans toute requête il est également possible de prévoir un paramètre indiquant le nom d'un service de réponse en cas d'erreur.

SESAME permet à une procédure d'informer la procédure appelante d'une condition exceptionnelle. Ceci se fait par l'exécution de l'instruction escape où l'on indique en paramètre un code d'erreur entier. L'exécution de la procédure s'interrompt alors, et le contrôle passe à une séquence de traitement d'erreur écrite dans le corps de la procédure appelante. Pour faciliter ses traitements celle-ci a accès au code d'erreur transmis et à l'identité de la procédure qui s'est terminée anormalement.

ADA offre une entité particulière nommée exception. Certaines exceptions sont pré-définies (OVERFLOW,INVALID-ASSERTION etc), d'autres sont déclarées par le programmeur pour ses besoins propres. Exemple :

Problèmeligne : exception;

Des "handlers" d'exception associés à un bloc d'instruction ou une ou plusieurs unités indiquent les traitements à effectuer lorsque telle ou telle exception apparaît dans le bloc ou les unités.

Les exceptions apparaissent :

- soit implicitement : exceptions standards décelées par le matériel ou des tests automatiques
- soit par programme avec l'instruction :
  raise <*nom d'exception*>
  assert <*expression booléenne*> qui provoque une exception INVALID-ASSERTION si l'expression est fausse. Ce sont là quelques éléments de traitement d'erreur parmi un ensemble assez complexe.

ANNEXE A LA PREMIERE PARTIE

-o-o-

Tableau résumant les principales caractéristiques des langages étudiés. La colonne "optique répartie" indique si le langage est explicitement destiné à l'écriture d'applications distribuées. Elle ne préjuge pas de la faisabilité d'une répartition éventuelle pour les langages a priori destinés à des applications centralisées.

Proposition	Optique répartie	Mode de communication	Mécanisme de base pour le contrôle	Structure élémentaire	Liaison dynamique	Structure dynamique	Passages de l'étude où l'on décr le langage ou l'un de ses aspect:
ADA	NON	MESSAGE,TYPE RENDEZ-VOUS	COMMANDES GARDEES	MODULE	NON	OUI	3.3.3.3 /4.3
CSP(HOARE)	NON	MESSAGE,TYPE RENDEZ-VOUS	COMMANDES GARDEES	MODULE	NON	NON	3.3.3.2
DP(B.HANSEN)	OUI	APPEL PROCEDURAL	COMMANDES GARDEES	MODULE	NON	NON	3.2.3.1
PLITS (J.FELDMAN)	OUI	MESSAGES+FILE D'ATTENTE	ATTENTE SUR TRANSACTION	MODULE	OUI	OUI	3.3.3.1
COMPTEURS (ROBERT-VERJUS)	NON	APPEL PROCEDURAL	CONDITION EN REGARD DE CHAQUE PROCEDURE	MODULE	NON	NON	3.2.3.2
EXPRESSIONS DE CHEMIN (CAMBELL HABERMAN)	NON	APPEL PROCEDURAL	AUTOMATE DECRIT A L'EXTERIEUR DES PROCEDURES	MODULE	NON	NON	3.2.3.2
MER(DECITRE)	OUI	APPEL PROCEDURAL ASYNCHRONE	EXCLUSION SUR MODULE	MODULE	OUI	NON	3.3.2.2 /4.3
G.PACINI	NON	APPEL PROCEDURAL	TYPE MONITEUR SEPARATION SYNCRO ET ORDONNANCEMENT	MODULE	NON	NON	3.2.3.1
IGOR	OUI	ACTIVATION PROCEDURALE ASYNCHRONE + MESSAGE	RENDEZ-VOUS	PROCEDURAL	OUI	OUI	2.4.2
SESAME	NON	APPEL PROCEDURAL	TYPE MONITEUR	MODULE	NON	NON	3.2.4.1 /4.3
NODAL	OUI	VARIABLES PARTAGEES	ATTENTE FIN D'EXECUTION D'UN PROCESSUS DISTANT	SUITE DE LIGNES NUMEROTEES	NON	OUI	2.4.2

DEUXIÈME PARTIE

PRÉSENTATION DES D-MODULES

1. PRELIMINAIRE

On aura pu déjà saisir, dans la première partie, les caractéristiques que nous jugeons importantes dans un langage d'écriture d'applications. Sous les termes de puissance d'expression et de clarté, habituellement mis en avant, nous voudrions privilégier des traits essentiels tels que :

pour la clarté :

- la séparation des algorithmes réalisant les différents services offerts par une unité,
- la séparation entre ces algorithmes et le contrôle exercé sur eux.

pour la puissance d'expression, ou du moins la simplicité de celle-ci :

- l'usage d'un langage évolué qui libère le programmeur d'innombrables problèmes de détail,
- dans le même ordre d'idée, l'élimination de contraintes matérielles telles que la notion de site, de liaison entre unités d'exécution etc,
- enfin, l'utilisation de formes d'écriture éprouvées, habituelles aux programmeurs, même si ces formes cachent une réalité moins simple.

  Ainsi, le lecteur aura peut-être deviné que nous apprécions la notion de procédure pour ce qu'elle offre d'interface bien définie de simplicité d'appel, de bonne séparation des algorithmes.

  En revanche, la communication par message nous semble également intéressante car elle peut permettre des exécutions indépendantes entre demandeur et service demandé. De plus, la synchronisation implicite qu'elle suppose entre les deux nous paraît simple et sûre.

  Il y a donc là des contradictions que nous tentons de résoudre dans les pages suivantes.

Ce sont sur ces idées qu'ont été conçues les premières propositions concernant les D-modules. Nous épargnerons au lecteur l'exposé de leurs formes successives pour décrire l'état actuel des propositions. Deux exceptions à cette règle : l'introduction du parallélisme et celle de l'imbrication des services pour lesquels il nous a paru plus aisé d'expliquer les choix initiaux et puis ensuite, seulement, les extensions apportées.
Dans son ensemble, cette seconde partie présente les D-modules tels qu'ils sont définis actuellement, tels qu'ils sont implémentés ou en cours d'implémentation sur un réseau de SOLAR-16.
Comme pour l'étude bibliographique nous délaissons la description du langage lui-même pour nous intéresser uniquement aux problèmes suivants :

- Structure d'une application et structure d'une unité.
- Interactions entre unités, communication.
- Contrôle exercé sur l'exécution des services.

Quelques exemples d'école illustrerons ensuite les primitives introduites.

De nouveau, comme dans la première partie, un chapitre plus général décrit l'environnement d'exécution des unités (les D-modules).
L'aspect langage du projet sera abordé dans la partie suivante, en même temps que quelques extensions ou compléments aux primitives.

2. STRUCTURE D'UNE APPLICATION. STRUCTURE D'UN D-MODULE

Définissant des outils d'écriture d'applications, nous avons bien sûr à nous poser, d'abord, des problèmes de structure et de définition d'unités de décomposition.
Dans le second chapitre de l'étude bibliographique, nous avons décrit les contraintes qui peuvent guider le choix de l'unité :

- Interface la plus simple possible.
- Indépendance des unités pour en faciliter l'écriture, la modification, la mise au point etc.

A ces contraintes s'ajoutent celles dues plus précisément à l'aspect distribué des applications :

- Pas de mémoire commune entre sites d'exécution.
- Relative lenteur des communications, qui incite à rendre les unités les plus autonomes possibles.

Ces contraintes rendent souhaitable l'identité entre unité d'écriture et unité de répartition des programmes.

Nous avons été conduits assez naturellement à choisir une unité de type modulaire (D-module pour module distribué). Ces modules sont relativement indépendants les uns des autres et ne partagent pas de variables. Une application se présentera comme un réseau de D-modules.

Nous avons vu dans la première partie l'inconvénient de ce type de structure elle ne facilite pas le contrôle global d'une application, nous tenterons de résoudre le mieux possible ce problème par la suite.

Composants d'un D-module

Dans sa structure générale, un D-module se rapproche des moniteurs, des modules SESAME etc. Il se compose :

- d'un ensemble de variables internes (variables globales)
- d'un ensemble de services,
- d'un programme de contrôle.

Pour expliquer cette structure, nous définissons dans les sections suivantes

- la notion de service et le type d'activité qui s'y déroule,
- le type de communication choisi,
- le rôle du programme de contrôle.

2.1 - Rôle et structure d'un service

Un service est la représentation et la réalisation de l'une des fonctions offertes par le D-module. Il présente donc une interface et effectue les actions reprises par les demandeurs. Il est composé :

- d'un en-tête où sont décrits :
  - . son nom
  - . ses paramètres (définis par valeur ou résultat)
- d'un ensemble de variables locales
- d'une liste d'instructions, son algorithme, qui peuvent accéder aux variables tant locales que globales.

Les interactions entre D-modules sont limitées aux requêtes à des services.

2.2 - Activité - Communication

Cette ressemblance entre services et les procédures classiques est seulement syntaxique. En effet, un service est le lieu d'exécution d'un processus cyclique (serveur) recevant les requêtes sous forme de messages.
Ces messages :

- sont conservés dans une file d'attente (une par service) dans leur ordre d'arrivée,
- ont pour chaque service un format fixe : une liste de valeurs dont la description est donnée par la liste des paramètres par valeur du service. (L'utilisation des paramètres par résultat est donnée plus loin).

Le déroulement d'un cycle du serveur se fait en trois temps :

1) La consommation des messages est faite implicitement au début de chaque cycle. Il n'existe donc pas de primitive explicite de consommation et par là, pas de possibilité d'exprimer la sélection de messages particuliers dans la file.

2) Lorsque le message a été retiré de la file d'attente associée au service, le processus exécute l'algorithme de ce service. Les accès aux champs du message sont exprimés de la même manière que ceux aux variables locales du service ou globales. Les valeurs transmises par le demandeur font partie des variables locales du service, tout comme dans les procédures classiques.

3) A la fin du service, et avant de tenter une nouvelle consommation de message, le processus peut avoir à retourner implicitement des résultats en demandeur.

Ce sera le cas lorsque :

- il y a des paramètres par résultat,
- le demandeur est bloqué en attente de la fin de traitement de sa requête (primitive d'appel bloquante).

Dans ces deux cas, un message est automatiquement expédié au demandeur. Il contient les paramètres par résultat s'il y en a, est vide sinon.

Exemple

```
service lire (result tampon : array [1.. 80] of char);
    var i : integer;
    begin
    for i := 1 to 80 do
            tampon [i] := {prochain caractère};
    end;
```

tampon sera automatiquement expédié au demandeur après exécution de *lire*.

2.3 - Discussion sur le choix de la structure

Un service a donc une présentation hybride car on donne un aspect procédural à une réalité basée sur la notion de message, et l'exécution par un processus cyclique. Cette dualité nous apporte cependant des avantages importants :

- une association claire entre le texte du service et la voie de communication où sont reçues les demandes. Chaque service est ainsi clairement exprimé et séparé des autres;
- une forme courante d'écriture, bien connue de tout programmeur,
- l'économie d'une notion de processus-réseau pouvant se propager d'un site à l'autre, d'appel procédural en appel procédural.

Nous n'avons pas trouvé de contrepartie grave à ces avantages. La pire aurait pu être que le type d'activité et de communication choisi ait été en contradiction avec la forme des services. Mais, nous le verrons au fil de l'exposé, les structures et primitives que l'on présente s'accommoderaient fort bien d'une implémentation purement procédurale. C'est à notre sens la meilleure garantie de compatibilité entre la forme présentée et l'exécution réelle.

En résumé :

- Les services implémentent les différentes fonctions qu'offre un D-module à ses utilisateurs.
- Ils sont écrits séparément sous forme d'algorithmes indépendants.
- Leurs échanges avec l'extérieur se font par messages.
- Une liste de paramètres décrit pour chaque service le format des messages de requête (paramètres par valeur) et celui des messages envoyés implicitement en réponse (paramètres par résultat).

3. INTERACTIONS ENTRE D-MODULES. DEMANDES DE SERVICES

Après la description des services, nous abordons leur utilisation, c'est-à-dire les primitives exprimant l'envoi d'une requête. Cette demande prend la forme d'un envoi de message. Or nous avons vu dans l'étude bibliographique que l'interaction ainsi demandée pouvait prendre deux formes (du moins lorsqu'elle est basée sur la communication par messages) :

- requête blocante : le demandeur attend la fin du traitement de sa requête,
- requête non blocante : l'exécution du demandeur se poursuit sans attendre ni la consommation, ni le traitement du message.

On trouve également, dans la littérature, les termes : requêtes "synchrones" ou "asynchrones".
Nous avons choisi de fournir les deux types de demandes.

3.1 - Requête blocante

Elle permet au demandeur de rester bloquer en attente de la fin du traitement de sa requête. Elle s'exprime comme un appel de procédure habituel, dont elle a d'ailleurs toutes les caractéristiques pour le demandeur.

<nom de D-module> . <nom de service> (<paramètres réels>)

Exemple :

RUBAN.LIRE (80, tampon);

C'est le seul type de requête possible lorsque le service demandé admet des paramètres par résultat (qu'il faut donc attendre avant de poursuivre).
Elle peut cependant s'appliquer à tous les autres services.

3.2 - Requête non blocante

Elle permet l'exécution asynchrone du processus demandeur et de sa requête. Elle a la même forme que la demande synchrone, mais est précédée du mot-clé exec.

exec <nom de D-module> . <nom de service> (<paramètres réels>)

Elle ne peut s'appliquer à des services admettant des paramètres par résultats. Si le programmeur a à la fois des contraintes d'asynchronisme et des résultats à rendre, il devra passer par l'émission explicite de messages de résultats. Pour faciliter ceci, on admet le passage de noms de services en paramètres d'une requête.

2.2.3. Justification des primitives précédentes

Si la littérature présente aussi bien des requêtes synchrones qu'asynchrones, il est rare que les deux soient offertes à la fois.
Ce sont apparemment des choix généraux de structure qui imposent telle ou telle méthode, quitte à construire des relations synchrones par des primitives asynchrones ou inversement. Pour ce qui est des D-modules, nous ne voyons pas de raisons d'interdire l'une ou l'autre.
Les interactions blocantes correspondent bien à une certaine classe de problèmes : ceux où un processus a besoin d'un service indispensable à la continuation de son travail et ces problèmes sont nombreux.
Les requêtes non blocantes permettent de leur côté de profiter du parallélisme réel d'exécution sur une architecture répartie, par exemple dans les cas suivants :

- consultation simultanée de plusieurs bases de données distantes, ce qui diminue l'importance des délais de communication,
- lancement sur un événement de plusieurs activités en environnement industriels.

Elles peuvent s'appliquer également lorsque l'application a une architecture "pipe-line" pour envoyer un flot de données d'une première série de traitement à une seconde etc.
L'utilisation de cette primitive suppose que pour chaque couple expéditeur-destinataire l'ordre des messages soit conservé.
On fournit une troisième primitive de communication :

3.4 - Primitive TRANSFER

Cette primitive permet au service qui l'exécute de transmettre le message (la demande) qu'il est en train de traiter à un autre service. Elle s'écrit :

transfer <*nom de module*> . <*nom de service*>

Le service référencé doit admettre le même type de messages que le service exécutant l'instruction.
Cette primitive est utilisée quand les traitements doivent s'exécuter en deux phases non forcément consécutives dans le temps, ou lorsqu'il faut connaître certains paramètres avant de décider du traitement ou du contrôle à donner à celui-ci. On reviendra sur l'utilisation de cette primitive par la suite.

4. CONTROLE EXERCE SUR L'EXECUTION. INSTRUCTION ALLOW

4.1 - Expression séparée du contrôle

Il n'apparaît pas dans les services de mécanismes de contrôle, du moins de mécanismes qui permettent à un service de contrôler l'exécution d'autres services.

> *L'expression du contrôle de l'exécution est faite entièrement dans un algorithme séparé : le programme de contrôle exécuté par un processus nommé arbitre.*

On retient donc l'idée d'un contrôle externe mais on le rend plus puissant :

- en le rendant algorithmique,
- en l'écrivant dans le même langage que les services,
- en lui donnant accès aux variables d'état du module.

Les expressions de chemin et les compteurs, on l'a vu, n'offrent d'autres connaissances au contrôle que celles des débuts et fins de procédures; l'accès aux variables d'état offre une connaissance bien plus précise et permet au contrôle de "raisonner" dans les mêmes termes que les services. Cette possibilité permet à l'arbitre d'autres traitements tels que : initialisation des variables, contrôle de cohérence, traitements éventuels d'erreurs.

4.2 - Principes du contrôle

Le choix d'un contrôle externe, même doté de moyens importants tels que ceux que l'on vient de décrire, amène obligatoirement aux contraintes suivantes :

- il ne peut s'exercer que sur le début et la fin des exécutions de service. L'exécution étant cyclique ici il s'agira surtout d'intervenir au début des services. Tout autre intervention sur l'exécution d'un service se ferait sans connaissances précises de son état et est donc à proscrire;
- pour disposer de variables d'état cohérentes, ce contrôle ne peut s'exercer que dans des moments où l'exécution des services est interrompue, il est donc sans intérêt de laisser l'arbitre actif lors de l'exécution des services.

Ceci se traduira pour les D-modules par les deux idées suivantes :

- le contrôle des services s'exerce sur leur accès aux files d'attente qu'il autorise ou interdit,
- il y a exclusion mutuelle entre l'exécution des services d'une part et celle de l'arbitre d'autre part.

Il y a un corollaire à tout cela : ce n'est évidemment qu'à la fin d'un service que l'on peut réactiver l'arbitre.

Pour exprimer tout cela on offre la primitive *allow*.

4.3 - Primitive allow

Cette instruction est réservée à l'arbitre pour le contrôle de ses services, sa syntaxe est la suivante :

allow (<liste d'idf.de service>) until <expression booléenne>

On supposera pour le moment que les services s'exécutent en exclusion mutuelle.

Principe

. L'exécution d'un D-module commence par le lancement de l'arbitre, ce qui suppose le blocage des services.

. Lorsque l'arbitre exécute une instruction allow, il libère les services cités. L'un d'entre ceux qui possèdent des messages sur leur file d'attente, est lancé. L'absence de messages sur les services libérés repousse ce lancement à l'arrivée du premier message. L'arbitre est de toute façon bloqué.

. L'exécution du service lancé se poursuit jusqu'à la fin du service. Il évalue alors l'expression booléenne (ou expression de terminaison) associée à la primitive allow en cours.

. Si cette expression est fausse, un nouveau service possible est lancé, éventuellement le même. L'absence de message, là encore, ne fait que différer le lancement.

. Si l'expression est vraie, le blocage des services est maintenu et l'arbitre réactivé. Son exécution reprend à la suite de la dernière instruction allow exécutée.

On exécute donc au moins un service, même si l'expression booléenne est déjà vraie dès l'exécution de l'instruction allow qui la contient.

On peut décrire les différents états d'un D-module par le graphe d'état suivant :

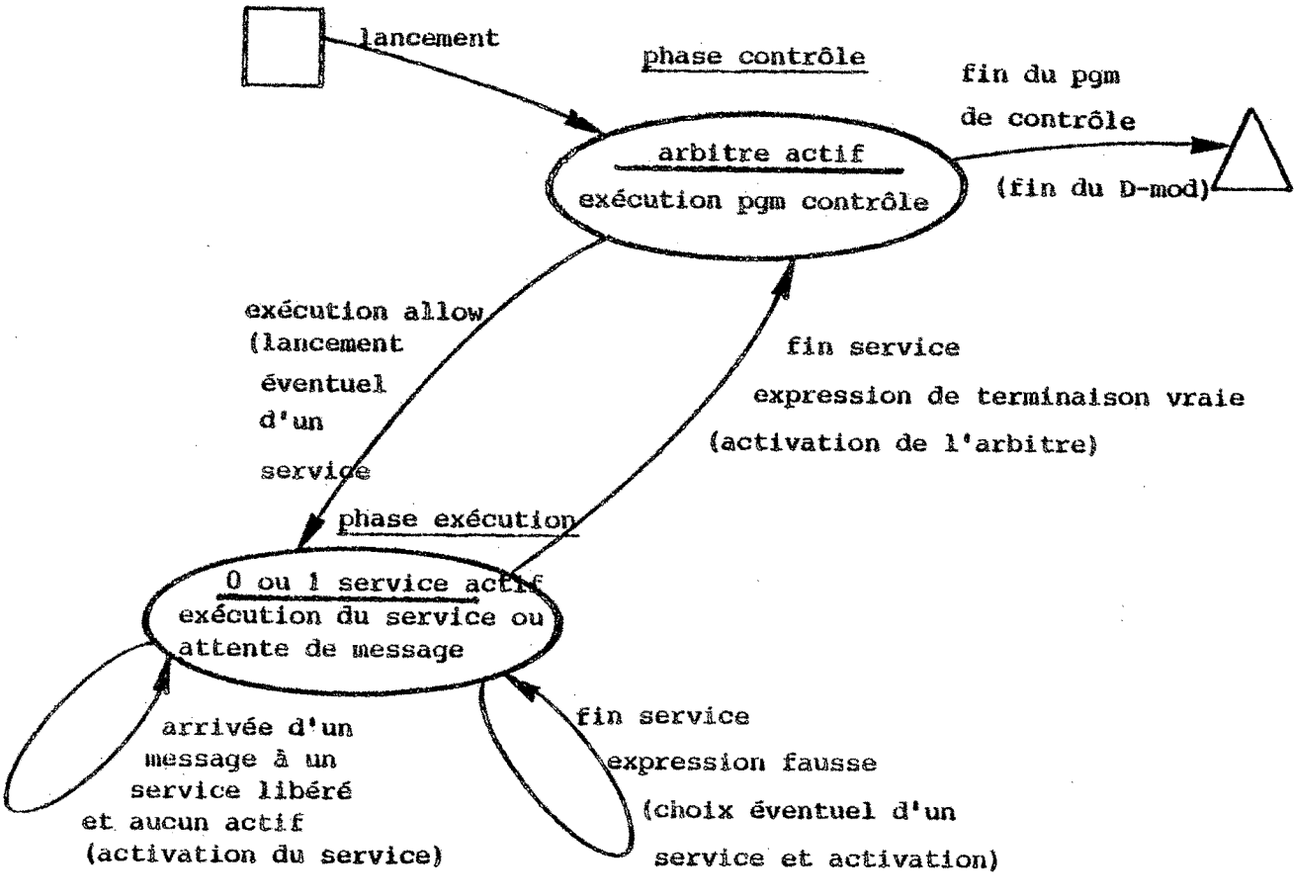


Figure 1. Graphe d'état d'un D-module.

On indique :

- pour chaque état ou phase, le processus actif et les actions en cours et
- pour chaque transition d'état, la cause de cette transition et éventuellement, entre parenthèses les actions l'accompagnant.

Figure 1. Graphe d'état d'un D-module

En d'autres termes :

- cette primitive allow définit un état du D-module, état dans lequel il peut rendre certains services,
- l'expression de terminaison indique les conditions dans lesquelles cet état n'est plus valable. L'arbitre doit être alors réactivé pour évaluer un nouvel état possible.

On peut ainsi programmer l'automate qui régit le comportement du D-module, chaque état est matérialisé par une instruction ALLOW. Pour reprendre des termes introduits dans la première partie, on dispose ainsi d'outils d'expression du séquencement des actions.

A titre d'exemple on peut implémenter un sémaphore de la façon suivante. Soit N > 0 la valeur d'initialisation du sémaphore.
(La construction repeat...endrepeat, qui n'appartient pas à PASCAL, crée une boucle sans fin).

```
D-module sem;
    var count : integer;
    service p;
        begin count := count-1 end;
    service v;
        begin count := count+1 end;
                        %pgm de l'arbitre%
    begin
    count := N;
    repeat
        allow (p,v)  until count = 0
        allow (v)    until count > 0;
    end repeat;
    end;
endmodule.
```

Lorsque count est strictement positif les deux opérations p et v sont possibles. Le passage de count à 0 force un changement d'état et la libération de v uniquement.

4.4 - Exécution concurrente des services

On lève ici la contrainte d'exclusion mutuelle dans l'exécution des services. Pour cela il faut étendre la signification de la primitive allow. Dans sa syntaxe générale, elle s'écrit :

allow (<liste de services>)
[: <définition des exclusions>]
until <expression booléenne>

[...] désigne un élément facultatif.
La partie centrale, qui exprime les contraintes de synchronisation est composée du symbole : suivi de la définition des exclusions mutuelles entre services. Celle-ci est matérialisée par une liste de groupes d'exclusion :

<définition des exclusions> ::= <groupe d'exclusion> [,<groupe d'exclusion>]
<groupe d'exclusion> ::= (<id de service>!<id de service>[!<id de service>])*

La présence de plusieurs services dans un même groupe indique une exclusion mutuelle entre tous ces services.
Un service peut apparaître dans plusieurs groupes.

Exemple 1.

allow (s1, s2, s3) : (s1!s2), (s2!s3)
until n = 3;

ici s1 et s3 peuvent s'exécuter en parallèle mais
s2 doit être exécuté seul.

Exemple 2. Dans l'exemple du sémaphore on indique l'exclusion mutuelle de p et v par :

allow (p,v) : (p!v) until count = 0;

Exemple 3. Exemple d'exclusion entre plus de deux services :

allow (s1,s2,s3,s4) : (s1!s3!s4)
until ...

On exécute soit s1, soit s3, soit s4 et s2 peut s'exécuter librement.

Puisque plusieurs exécutions simultanées sont maintenant possibles, le rôle de l'expression booléenne de terminaison devient le suivant :

- chaque service évalue cette expression après chaque cycle,
- si elle est fausse, l'exécution continue, et de nouveaux services peuvent être éventuellement lancés,
- si elle est vraie, tout nouveau lancement d'exécution est interdit et l'on attend que les services en cours se terminent. L'arbitre est alors activé (un nouveau passage à faux de l'expression ne change rien à ce blocage progressif).

Le graphe d'états d'un D-module devient le suivant:

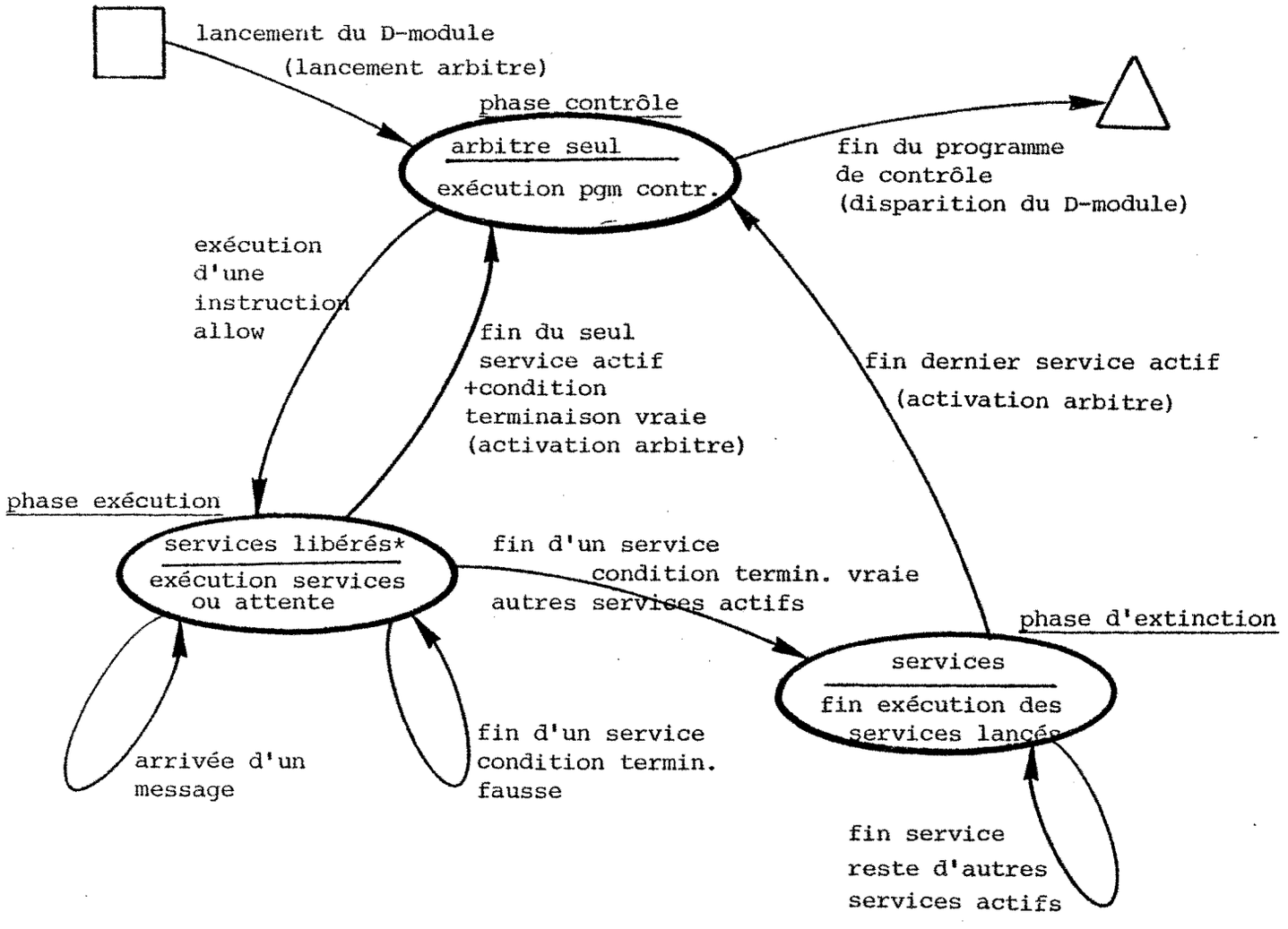


Figure 2. Graphe d'état d'un D-module lorsque l'exécution parallèle de services est possible
La formulation est expliquée en 4.3.

* ou éventuellement aucun.

5. EXEMPLES

On donne ici quelques exemples classiques, qui illustrent l'utilisation de nos primitives.

5.1 - Rendez-vous entre N processus

Le rôle du D-module ci-dessous est de permettre la resynchronisation de plusieurs activités : N processus doivent attendre d'être tous arrivés à un point donné de leur exécution pour pouvoir continuer celle-ci. Dans le code de chacun, ce rendez-vous sera implémenté par la requête synchrone :

synchro.rendez-vous

Le D-module *synchro* est le suivant :

```
D-module synchro;
    var co : integer;
    service block;
        begin co := co-1 end;
    service rendez-vous;
        begin co := co+1; transfer block end;
    % arbitre %
    begin
    co := 0;
    repeat
        allow (rendez-vous) until co = n;
        allow (block) until co = 0,
    endrepeat
    end
endmodule.
```

On trouve ici l'un des emplois possibles de la primitive transfer qui permet de prendre en compte une requête avant de la traiter complètement. Ici les appels à *rendez-vous* sont comptabilisés puis transmis à *block*. Lorsque tous les processus attendus sont bloqués sur *block* (co = n) on peut tous les libérer (... until co = o).

5.2 - Les cinq philophophes

Dans ce problème célèbre, cinq philosophes sont assis autour d'une table; chacun passe son temps à réfléchir ou à manger des spaghettis.C'est-à-dire qu'il boucle sur les actions :

- penser
- manger

Dans notre représentation programmée l'action "manger" sera, pour le philosophe i une requête blocante à un service mi.
L'exemple précise que par manque de fourchettes deux voisins ne peuvent manger simultanément. Pour indiquer cette contrainte on exprimera le programme de contrôle du D-module qui contient les services mi par la simple instruction :

```
allow (m1,m2,m3,m4,m5) :
        (m1!m2), (m2!m3), (m3!m4), (m4!m5), (m5!m1)
until false;
```

Cet exemple ne tient pas compte des problèmes de...famine. Deux philosophes peuvent en effet se liguer contre un troisième pour l'empêcher de manger. On reviendra sur cet exemple et pour éviter la famine par la suite.

5.3 - Lecteurs-Rédacteurs

Le D-module suivant implémente une ressource (un fichier par exemple) gérée sur le modèle lecteurs-rédacteurs avec les services :

ouvrirlec, lec, fermerlec
ouvrirecr, ecr, fermerecr.

Lorsqu'aucun utilisateur ne s'intéresse au fichier, les deux services *ouvrirlec* et *ouvrirecr* sont disponibles, ensuite et suivant le cas seront libérés (*lec* et *fermerlec*) ou (*ecr* et *fermerecr*).
Plusieurs utilisateurs peuvent coexister en lecture, un seul peut avoir accès en écriture.
Le texte des services est le suivant :

```
nblec : integer; %nombre de lectures en cours%
demecr, finecr : boolean;
service ouvrirlec;
     begin <opérations d'ouverture>
     nblec := nblec + 1;
     end :
service lec; begin... end;
```

```
service fermerlec;
begin <opérations de fermeture>
nblec := nblec - 1;
end;
service ouvrirecr;
     begin <...>
     demecr := true,
     end;
service ecr;
     begin... end;
service fermerecr;
     begin <...>
     finecr := true
     end;
```

Si l'on veut privilégier les lecteurs, c'est-à-dire autoriser les nouvelles lectures tant que l'une au moins est en cours, on pourra écrire le programme de contrôle ainsi :

```
begin
repeat
     nblec := 0; finecr := false
     allow (ouvrirlec, ouvrirecr) : (ouvrirlec!ouvrirecr) until true;
          % une seule exécution est nécessaire %
     if nblec > 0 then allow (ouvrirlec, lec, fermerlec)
                       : (ouvrirlec!fermerlec)
                       until nblec = 0
     else allow (ecr, fermerecr) until finecr;
endrepeat;
```

une demande sur ouvrirecr ne pourra être prise en compte tant qu'il y aura des lecteurs.

Si l'on veut que la première demande sur ouvrirecr interdise l'accès de nouveaux lecteurs on écrira le programme de contrôle suivant :

```
begin
repeat
     nblec := 0; demecr := finecr := false;
     allow (ouvrirlec, ouvrirecr) : (ouvrirlec!ouvrirecr) until true;
     if nblec > 0 then begin
                       allow (ouvrirlec, ouvrirecr, lec, fermerlec)
                       : (ouvrirlec!ouvrirecr), (ouvrirlec!fermerlec)
                       until demecr;
     if nblec > 0 then
     allow (lec, fermerlec) until nblec = 0;
     end;
     allow (ecr, fermerecr) until finecr;
endrepeat
end;
```

En admettant un accès à *ouvrirecr* pendant les lectures on reste à l'écoute des intentions d'écriture.

5.4 - Producteurs - Consommateurs

On réalise dans ce dernier exemple une boite à lettre utilisée par deux classes de processus : les producteurs et les consommateurs. Cette boîte à lettre : le D-module *bal* gère des messages de format fixe et dispose d'une capacité de mémorisation de n messages (le tableau tampon).
Pour un producteur la séquence de production d'un message est la suivante :

bal.debprod (ind)	*<demande d'un élément libre de tampon>*
<production>	*(rend l'indice)*
bal.finprod (ind)	*<indication que le message est disponible>*

La consommation d'un message donnera lieu aux actions suivantes :

```
....
bal.debcons(ind); <demande d'un message à consommer>
                  (rend l'indice du message)
<consommation>
bal.fincons (ind); <libération de l'élément de tampon qui contenait
                    le message>
```

Le D-module *bal* peut s'implémenter ainsi :

```
d-module bal;
    type indice = 1..n;
    var nplein, nvide : integer;
    marques : array [indice] of (vide, en-cours, plein);
    tampon : array [indice] of message;

    service debprod (result i : indice);
       begin i := {indice d'un élément libre de tampon};
       marques [i] := en-cours;
       nvide := nvide-1;
       end;

    service finprod (i : indice);
       begin marque [i] := plein;
       nplein := nplein+1;
       end;

    service debcons (result i : indice);
       begin i := {indice d'un élément plein de tampon};
       marques [i] := en-cours; nplein := nplein-1;
       end;

    service fincons (i : indice)
       begin marques [i]:= vide;
       nvide := nvide+1;
       end;

    %programme de contrôle%
    begin
    nplein := 0; nvide := n;
    allow (debprod, finprod) until (nplein > 0)or(nvide = 0);
    repeat
       if(nplein+nvide = 0) then
         allow (finprod, fincons) until (nplein+nvide > 0)
       else if nvide = 0 then
         allow (finprod, fincons, debcons) : (finprod!debcons)
         until (nvide > 0) or (nplein = 0)
```

```
        else if nplein = 0 then
          allow (finprod, fincons, debprod) : (fincons!debprod)
          until (nplein > 0) or (nvide = 0)
        else allow (debprod, debcons, finprod, fincons)
             : (debprod!fincons), (debcons!finprod)
             until (nvide+nplein = 0);
    endrepeat
    end
endmodule.
```

Le tableau *marques* mémorise l'état des éléments de tampon :

plein : contenait un message,

en-cours : en cours de remplissage ou de consommation

vide : libre

nplein et *nvide* servent respectivement de compteurs d'éléments pleins ou vides. Ce sont ces deux compteurs qui guident le contrôle selon les bases suivantes :

si *nplein* = 0 aucun message complet n'est disponible, on interdit donc *debcons*,

si *nvide* = 0 aucun élément de tampon n'est libre, on interdit *debprod*.

Ainsi : la première instruction allow exécutée à l'initialisation autorise les premières productions jusqu'à ce qu'un message soit disponible (*nplein* > 0) ou que tous les éléments de tampon soient réservés à des productions en cours.

En concurrence sur *nplein* (resp.*nvide*) *debprod* et *finprod* (resp.*debprod* et *fincons*) doivent être exécutés en exclusion mutuelle.

L'expression du contrôle peut sembler plus longue à priori que dans d'autres propositions mais les lignes précédentes montrent qu'elle est simple à concevoir puisque l'on raisonne uniquement sur les valeurs de *nplein* et *nvide*.

6. IMBRICATION DE SERVICES

6.1 - Critique de la structure définie

Nous n'avons jusqu'ici considéré le programme de contrôle d'un D-module, que dans sa fonction de contrôle et d'initialisation. Il ne lui est cependant pas interdit d'adresser lui-même des requêtes à des services d'autres modules, et de réaliser ainsi des actions complexes. A titre d'exemple, un programme utilisateur séquentiel peut être considéré comme un D-module réduit à un programme de contrôle.

Le problème que nous voulons souligner ici est le suivant :

- Lorsqu'un processus est un arbitre, il dispose de moyens importants d'exécution : il peut lancer des requêtes et recevoir des réponses sous forme de messages renvoyés aux services qu'il contrôle. Il peut d'une façon générale se synchroniser avec d'autres activités en se bloquant sur la libération de l'un ou plusieurs de ses services. Dans ce cas, la primitive allow joue le rôle d'une attente d'événements multiples.
- En comparaison, les outils dont dispose un service sont très limités : par exemple il ne peut recevoir de résultats que par paramètre par résultat. Toute autre méthode oblige :
  - à indiquer le nom d'un autre service, pour la réception de message-résultats,
  - à rendre le contrôle à l'arbitre qui libère le service de réception pour que celui-ci traite les résultats obtenus.

Cette technique provoque une multiplication des services et fait exécuter par le programme de contrôle des traitements qui ne dépendent que du service ayant lancé la requête.

On ne pourra donc, avec la structure décrite jusqu'ici, faire exécuter des actions complexes par les services sans multiplier le nombre de services et compliquer le programme de contrôle.

Remarque

Il y a de nombreux cas où la communication de résultats via des paramètres est insuffisante.

Ainsi, si le nombre de réponses à une requête donnée peut varier (nombre de personnes ayant telles caractéristiques par exemple) il paraît plus simple de les rendre sous forme d'un flot de messages.

6.2 - Solution proposée

Il semble qu'une définition récursive de la notion de service puisse apporte une solution assez naturelle à ces problèmes. On entend par définition récursive, la possibilité de définir des services à l'intérieur d'un service avec les règles de contrôle suivantes :

- Le programme de contrôle d'un D-module contrôle les services de niveau d'imbrication 0 du module.
- Si un service possède des services imbriqués il joue le rôle de programme de contrôle sur ces services.

Cette définition ne remet pas en cause l'indépendance des services de même niveau, mais permet de définir une hiérarchie entre services.
Ainsi on dispose dans chaque service d'outils de communication et de synchronisation identiques à ceux d'un programme de contrôle. Chacun de ces programmes de contrôle de quelque niveau qu'il soit n'a plus à tenir compte de la synchronisation interne d'un service qu'il contrôle.
On décrit en exemple ci-dessous, un service interfaçant l'utilisation d'un périphérique. Ses utilisateurs demandent la lecture d'enregistrements de 80 caractères qu'il doit demander un par un au D-module gérant directement le périphérique. On suppose, pour la simplicité de l'exemple, qu'il y a toujours des caractères à lire et qu'il n'apparaît jamais d'erreurs. Cela permet au service interface de lancer 80 demandes de lecture consécutives. Ce service peut s'écrire :

```
service liretampon (result buf:array [1..80] of char);
    var i : integer;
    service reçoitcar (c : char); %service de réception%
        begin i := i+1; buf [i] := c; end;
    begin %pgm de liretampon%
    for i := 1 to 80 do exec periph.lire (reçoitcar);
    %80 demandes de lectures avec résultat à reçoitcar%
    i := 0; allow (reçoitcar) until i = 80; %réception de 80 caractères
    end; %fin du service et renvoi implicite de buf%
```

6.3 - Utilisations de l'imbrication

Cette dernière extension nous paraît intéressante à plusieurs titres dont bien sûr d'abord les raisons qui nous ont poussées à l'introduire. Il semble que l'on augmente de façon notable la puissance d'expression des services, sans pour autant remettre en cause le principe de l'indépendance de services de même niveau.
L'arbitre contrôle l'exécution de services dont il n'a pas à connaître la synchronisation interne. Dans cette optique, les services imbriqués servent surtout de nouvelles voies de communication pour le service qui les contient.

On peut trouver une autre utilisation à ces services imbriqués : pour créer des activités parallèles coopérant pour réaliser le service global. Un service peut ainsi confier une ou plusieurs parties de son traitement à des services imbriqués en les activant par une primitive exec suivie d'un allow. On dispose alors de l'équivalent d'une instruction fork... join d'autres langages.

Nous prendrons pour illustrer cette possibilité l'exemple d'un service qui reçoit un tableau d'entiers, leur applique une fonction f rendant une autre valeur entière et édite ces résultats. On décompose l'action en deux sous-services: le premier *f* calcule les résultats de la fonction et envoie ces résultats au second : *sortie*, qui se charge de les convertir en chaînes de caractères et de les imprimer par un appel à un service extérieur *écrire*.

```
service calculédition (table : array [0.. 100] of integer);
    var
    compteur : integer;
    service sortie (aediter : integer);
        var chaîne : array [0.. 16] of char;
        begin
        chaîne := {conversion de aediter en caractères};
        exec écrire (chaîne); compteur := compteur+1;
        end;
    service f;
        var i,j::integer;
        begin
        for i := 0 to 1000 do
            begin j := {calcul de f (table [i])}; exec sortie (j) end;
        end;
```

```
begin %programme de calculédition%
compteur := 0;
exec f; %envoi d'un message vide pour lancer f%
allow (f, sortie) until compteur = 1001;
end;
```

7. UTILISATION ET CRITIQUE DES D-MODULE

7.1 - Etat des propositions

Les outils que nous proposons actuellement se réduisent aux primitives et structures présentées dans les pages précédentes. A défaut d'un langage complet, encore à l'étude, on pourra écrire des applications réparties avec :

- ces primitives et cette structure, c'est-à-dire des conventions de programmation,
- et un noyau d'exécution réalisant les primitives.

On peut, pour le moment, et en utilisant des langages de bas niveau (ASM 16, PL16) tester la validité de nos concepts; même si des éléments importants (contrôle de type, d'erreur, de paramètre) sont délaissés. Dans la partie III de cette thèse on essaie de définir un langage complet d'écriture des D-modules. Mais avant que d'aborder le langage, il est sans doute temps de reprendre les intentions que nous exposons dans le préliminaire de cette seconde partie et d'analyser nos propositions à la lumière de ces intentions. Il n'est pas inutile, non plus de "refaire un passage" sur nos primitives pour discuter de leurs caractéristiques et expliquer l'emploi que nous leur voyons. Ce sera l'objet des prochains paragraphes et l'occasion de répertorier les insuffisances visibles de ces propositions, insuffisances auxquelles la partie suivante de la thèse apportera quelques idées de réponse.

7.2 - Caractéristiques de la structure

Nous ne reviendrons pas sur le mode de communication retenu et les primitives qui le mettent en oeuvre : requêtes synchrones ou asynchrones. Ces points ne sont pas réellement nouveaux et ont été abordés amplement. L'originalité recherchée est plutôt celle de la séparation des codes : séparation entre services, séparation entre les actions et leur contrôle. Les deux points sont largement liés : le premier est une condition importante pour la réalisation du second. Il n'aurait guère été possible de contrôler de l'extérieur des actions différentes mais entremêlées. Les avantages de la séparation ont été également amplement exposés. On peut rappeler :

- la clarté de l'écriture,
- la facilité accrue des modifications.

Nous pensons avoir évité les principaux défauts qui entâchaient les propositions de séparation du contrôle faits antérieurement. Nous avons

particulièrement noté dans l'étude bibliographique les défauts suivants (I.3.2.3.2.) :

- peu de connaissance par le contrôle de l'état du module
- non prise en compte de l'identité des processus
- forme non algorithmique du contrôle (compteurs) ou peu souple (expressions de chemin).

En adoptant un compromis : expression séparée du contrôle, mais écrit dans le même langage et ayant accès aux variables d'état, nous pensons avoir résolu ces problèmes :

- le contrôle a donc accès aux variables globales et on évite ainsi la multiplication de services annexes (voir les exemples donnés en I.3.2.3.2);
- le contrôle ne prend toujours pas en compte l'identité des processus, mais,avec quelques extensions au langage que l'on verra dans la partie suivante de la thèse, les services peuvent avoir connaissance de l'identité de l'expéditeur. Ils sont alors libres de modifier ou non les variables utilisées dans l'expression de terminaison. D'une façon générale le partage de variables permet au contrôle d'indiquer tout ce que le programmeur désire;
- le langage d'écriture du programme de contrôle étant le même que celui des services, ce contrôle dispose des même facilités d'expression.

On pourra toutefois nous faire l'objection suivante : si le contrôle et les services partagent les mêmes variables, la séparation souhaitée n'est pas complète : il y a en effet mélange au niveau des variables. On reviendra sur ce problème en 7.2.2.

Puisque allow est le seul outil du contrôle et que nous analysons ce contrôle cette instruction mérite que l'on fasse quelques remarques sur ses aspects les plus importants. Ce sont à notre avis :

- le non-déterminisme du lancement des services (ignorance de la présence de message, de l'ordre de leur arrivée etc),
- l'utilisation de variables communes avec les services (variables de contrôle en même temps que de travail),
- l'intervention du contrôle sur le seul accès à la file, donc sans connaissance du contenu des messages.

7.2.1. Non déterminisme

7.2.1.1. Problème de famine

Lorsque l'arbitre en exécutant une instruction allow libère certains services on ne connaît pas, a priori, quels sont ceux qui vont s'exécuter et quand.

- Il peut y avoir, ou ne pas y avoir, de messages.
- Suivant le mode d'écriture du noyau, le choix entre services possibles est variable.

La possibilité d'attendre plusieurs événements en même temps est un outil pratique, mais délicat :

- Quels sont les critères de choix, quand ce choix est-il possible ?
- Quels sont les risques éventuels de famine ?

L'exemple des cinq philosophes développé en 5.2 comporte des risques évidents de famine (le mot est ici très adapté) pour au moins un philosophe. Un choix simple : priorité à la demande la plus ancienne permet d'améliorer le service s'il y a régulièrement changement d'état, c'est-à-dire si l'arbitre est régulièrement libéré pour exécuter de nouveaux allow. A chacune de ces exécutions la demande possible la plus ancienne est choisie en priorité et on évite donc les attentes infinies.
Dans l'exemple des cinq philosophes, on exécute un seul allow qui suffit théoriquement pour régler une fois pour toutes l'exécution des services.

Pour éviter une famine il faut vérifier que deux philosophes ne se liguent pas contre leur voisin commun. Il est difficile de faire un tel contrôle. On peut, par contre, provoquer des changements d'état lorsque un philosophe aura mangé beaucoup plus souvent que son voisin. Pour cela on introduit un compteur pour chaque philosophe et on sort du allow lorsqu'un philosophe aura mangé cinq fois de plus que son voisin, on espère ainsi régulariser les accès aux services.

Exemple :

```
D-module spaghetti;
    var i : integer
        compteurs : array [0..4] of integer;
    ...
    service mi;  %action de manger pour le philosophe i%
        begin ... actions correspondant à manger...
        compteurs [i] := compteurs[i]+1;
        compteurs [(i+1) mod 5] := compteurs [(i+1) mod 5]-1;
        end;
    ...
```

```
        begin %pgm de contrôle%
        For i := 0 to 4 do compteurs[i] := 5;
        repeat
        allow (m0,m1,m2,m3,m4):(m0!m1),(m1!m2),(m2!m3),(m3!m4),(m4!m0)
        until (compteurs [0] <= 0) or (compteurs [1] <= 0)
              or (compteurs[2]<=0) or (compteurs[3]<=0)  or(compteurs [4]·
        endrepeat
        end
endmodule
```

On joue ici sur la différence entre le nombre d'exécutions d'un service mi et celui de son voisin *m(i+1) mod5)*. Si le premier s'exécute cinq fois de plus que le second, on sort du allow : c'est-à-dire qu'on change d'état, que l'on exécute de nouveau la même instruction allow. Si il y a une demande pour le service défavorisé et si elle attent depuis longtemps, elle aura priorité. On a ainsi quelques chances de revenir à un état stable. Sinon les exécutions de allow se feront par à-coups : la condition de terminaison restant vraie en permanence, il y aura pour chaque exécution de allow deux services au maximum qui s'exécuteront.
Cette solution suppose donc que les fréquences d'appel à manger des différents philosophes seront proches les unes des autres, mais elle peut donner une idée sur la façon d'éviter les famines.

7.2.1.2. Parallèle avec les commandes gardées

Le non déterminisme de allow le rapproche quelque peu des commandes gardées, du moins de celles où l'arrivée d'un message a une importance. Nous avons décrit en I.3.3.3 (partie étude bibliographique) les règles d'écriture et de fonctionnement des commandes gardées. Rappelons simplement que l'on peut combiner expressions booléennes et directives de consommation de message pour décrire les alternatives qui indiquent quel traitement effectuer dans quelles conditions.

Exemple dans ADA :

```
select
    when (x > 3) => accept lire; {actions A1}
    or
    accept écrire ; {actions A2}
endselect;
```

ce qui signifie

- si (x > 3) et un message en attente sur LIRE alors A1
- si un message sur écrire alors A2
- si les deux sont vérifiés : A1 ou A2
- sinon attendre.

On peut faire plusieurs constatations autour de cet exemple :

- l'activité est séquentielle, on ne peut exécuter A1 et A2 parallèlement, ce qui serait possible avec allow;
- le choix est non déterministe (si choix il y a) comme dans nos propositions;
- l'association, voie d'arrivée de message - actions, ici LIRE-A1 et ECRIRE-A2 peut changer d'une commande gardée à l'autre. Elle est fixe dans les D-modules;
- les actions sont mêlées au contrôle, à moins d'une discipline de programmation qui substitue A1 et A2 par des appels de procédures (c'est alors une façon de structurer le code);
- on mêle expressions booléennes et références à l'extérieur. Allow utilise les expressions booléennes mais comme expressions de terminaison et non comme "filtres" à la réception de messages;
- le select de ADA n'est pas répétitif comme allow, mais d'autres langages proposent des commandes gardées répétitives.

En résumé, les deux propositions peuvent être assez proches mais allow impose des règles d'écriture qui ne dépendent que du programmeur dans les langages à commandes gardées.

Pour avoir le même résultat avec les D-modules, on aurait du écrire :

```
...
service lire
    begin A1 end;
service écrire
    begin A2 end;
...
if x > 3 then allow (lire, écrire) : (lire!écrire)
                                        until true
else allow (écrire) until true;
```

7.2.2. Partage de variables avec les services

Les deux défauts primordiaux que nous avons trouvés aux expressions de chemins étaient :

- l'aspect non algorithmique du contrôle,
- le peu de connaissance du contrôle sur l'état du module.

On sait comment nous avons résolu le premier point, et que pour éviter le second défaut nous avons autorisé l'accès par l'arbitre aux variables d'état. Nous avons souligné au début de la section 7.2 que ce partage des variables d'état pouvait être considéré comme une entorse à la séparation contrôle-actions. Certains projets de recherche proposent de différencier variables de contrôle et variables de travail. Ceci n'est guère possible ni d'ailleurs souhaitable dans notre cas pour les raisons suivantes :

- notre programme de contrôle peut être amené à faire des initialisations ou à répondre à certains cas d'erreur;

- dans ce schéma de séparation de code, de l'information utile au contrôl est forcément fournie par les services; quelle forme lui donner sinon une adaptation de la notion d'événement ? Cela amène obligatoirement la prise en compte dans les services de notions de contrôle, donc une connaissance du type de contrôle exercé au dessus d'eux, ce qui n'est pas forcément souhaitable.

On préfère, en effet, ne pas trop faire intervenir le contrôle dans la réalisation des services. Cela permettra éventuellement une extension du nombre de services du module ou une modification du contrôle sans grande modification des services existants.

En contrepartie, le contrôle doit connaître assez précisément la réalisation des services pour déceler les variables sujettes à ses tests. C'est là un inconvénient que le programmeur peut résoudre par un compromis entre le petit nombre de variables à tester et l'indépendance des services vis à vis du contrôle. Cela obligera également à décrire le service par des spécifications précises, utilisées par le contrôle.

7.2.3. Défaut d'informations sur le contenu d'un message

C'est là, pour le contrôle, le principal défaut causé par son expression extérieure aux services. Plus particulièrement dans un certain nombre de cas où une prise en compte et satisfaction d'une requête ne sont pas forcément immédiatement consécutifs.
Deux exemples peuvent illustrer deux aspects de ce problème.

. *Nécessité de connaître l'existence de la requête pour la satisfaire*

L'exemple (décrit en 5.1) du rendez-vous de N processus montre :

- que les requêtes doivent être comptées
- puis ensuite satisfaites.

La première requête arrivée subit donc un début de traitement puis attend que la dernière soit également parvenue.

. *Nécessité de connaître la valeur d'un ou plusieurs paramètres pour décider du délai ou de l'ordre de satisfaction*

L'exemple choisi est celui d'un allocateur gérant N ressources identiques. Les demandes d'allocation expriment un nombre variable de ressources demandées. (Des cylindres de disque par exemple).
Il faut connaître le nombre de ressources requises pour décider d'un blocage ou d'une satisfaction immédiate de la requête.

Solution proposée

Cette solution est basée sur la troisième primitive d'envoi de requête : transfer. Cette instruction permet d'analyser une requête puis de la confier au service destiné au traitement définitif. Parallèlement, on peut mémoriser des informations utilisées par le contrôle pour libérer le service de traitement en temps opportun.

Ce procédé peut comporter des dangers :

- duplication de messages (on devra interdire deux transfer du même message),
- découpage du traitement d'une requête en deux services ou plus,
- complexité du programme de contrôle qui devra gérer le séquencement des phases du traitement.

Ses principaux avantages sont sa puissance, et sa compatibilité avec les autres primitives, le fait qu'il conserve le caractère synchrone ou asynchrone d'une demande. Vis à vis de l'extérieur, la manipulation du message est totalement transparente.

Pour atténuer ou faire disparaître les défauts de cette primitive, le programmeur devra faire ressortir soigneusement la particularité des services qui l'utilisent. Une bonne séparation de ces services d'avec ceux qui effectuent les traitements réels constitue même un avantage en matérialisant bien :

- les règles de répartition du travail,
- les priorités données à certaines requêtes par rapport aux autres.

Le paragraphe 7.2.4 en donne un exemple.

Remarque : consommation sélective de messages

Certaines propositions (PLITS par exemple) permettent à un processus de sélectionner les messages d'une file d'attente en exprimant des prédicats sur leur contenu. La complexité des mécanismes que ceci suppose ne justifie pas à notre avis le gain de puissance acquis. De plus, ces primitives expriment une sélection parmi les messages et non une préférence. L'absence de message répondant aux critères indiqués provoque, en effet, un blocage même si d'autre messages existent.

7.2.4. Exemple de sélection de requête par transfer

L'exemple choisi est celui du D-module gérant la console opérateur d'un système. L'exemple est limité à l'écriture de messages; parmi ceux-ci certains (concernant des alarmes) doivent être prioritaires. Le seul service offert à l'extérieur par le D-module est *écrire*, il trie les messages à imprimer suivant leur urgence. Les messages d'alarme sont transférés à *ecralarme*, les autres à *ecrusuel*. Sitôt qu'un message d'alarme est arrivé on le signale au contrôle.

```
D-module console op;
type typemsg = (usuel, alarme);
     ch80 = array  0... 79  of char;
var
     urgence : boolean; %indicateur de présence d'alarme%
service écrire (priorité : type msg; texte : ch80);
  %interface répartissant les messages suivant leur type, positionnement
   de urgence en cas d'alarme%
     begin
     if priorité = alarme then begin urgence := true;
                                     transfer ecralarme
                                     end
     else transfer ecrusuel;
     end;
service ecralarme (priorité : type msg; texte : ch80);
     begin  ecriture du message
     urgence := false;
     end;
service ecrusuel (priorité : typemsg; texte : ch80);
     begin  ecriture du message  end;
begin %contrôle%
urgence := false;
repeat
allow (écrire, ecrusuel) until urgence;
allow (ecralarme) until true; %1 seule passe%
endrepeat;
end;
endmodule;
```

Ecrire ne fait aucun traitement long, les messages sont donc examinés très vite après leur arrivée. Dès qu'un message d'alarme est reconnu, *urgence* devient vrai, ce qui interdit tout nouveau lancement de service.
Le message sera imprimé dès que le message en cours de traitement par *ecrusuel* aura été écrit. (Et bien sûr après réactivation de l'arbitre et libération par lui de *ecralarme*).

ANNEXE

STRUCTURE GLOBALE ET VIE D'UNE APPLICATION

Nous décrivons ici, comme nous l'avons fait dans le chapitre 4 de l'étude bibliographique, des points qui dépassent le sujet de cette thèse. Ce complément se justifie, cependant, par le souci de situer les notions décrites jusqu'ici, particulièrement :

- la structure d'une application, la liaison de ses composants
- son lancement, son évolution, sa fin
- les interactions qu'elle peut avoir avec son environnement.

Les solutions, les définitions que nous donnons ne peuvent être que schématiques; ce sont, en effet, autant de points qui méritent d'amples études et de nombreuses modifications.

A1 - Structure et liaisons dans une application

La structure d'une application, déjà esquissée en 2, peut être définie par un graphe dont les sommets sont les D-modules qui la composent et les arcs les liaisons entre D-modules. On nomme liaison la connaissance par un D-module d'un service qu'il peut utiliser. C'est-à-dire la possibilité d'envoyer une requête à ce service.
Les primitives que nous avons introduites pour l'écriture de ces D-modules n'imposent pas de relations hiérarchiques entre eux. On peut articuler une application en plusieurs niveaux d'abstraction : les requêtes ne peuvent être faites qu'à des modules de niveau inférieur. On peut tout aussi bien écrire une application où les modules s'utilisent mutuellement sans relations hiérarchiques quelles qu'elles soient.

Ce graphe par lequel est décrite une application peut éventuellement évoluer au cours de l'exécution. L'évolution du nombre des sommets (du nombre des D-modules) est traitée en A4. L'évolution du système d'arcs concerne la liaison qui est l'un des sujets abordés dans ce paragraphe.

Le lancement d'une application comportant plusieurs D-modules suppose que le graphe constitué initialement soit connexe. Les liaisons statiques (connues dès le lancement) permettent d'engendrer des liaisons dynamiques par le passage de noms de services en paramètres. Mais si l'application possède deux parties distinctes et non liées au lancement, il ne sera pas possible de véhiculer des noms de services pour établir des liaisons dynamiques entre les deux parties.
La faculté d'établir des liaisons dynamiques permet de résoudre aisément certains problèmes :

- retour d'un message ou d'un flot de messages en résultat à une requête
- problèmes de confidentialité : on peut rendre les noms des services d'une ressource, après une requête préliminaire au seul service connu de la ressource qui teste le bon droit de l'utilisateur. Exemple : seul le service OUVRIR d'un D-module gérant un fichier est connu des utilisateurs du D-module. Lorsque l'un d'eux envoie une requête à OUVRIR on lui rend les noms des services LIRE, FERMER à condition que l'accès au fichier lui soit permis.

A2 - Interaction entre l'application et son environnement

Système

Si le système d'exploitation n'est pas limité au noyau implémentant les primitives, on devra tenir compte des communications entre l'application et le système. Il paraît simple d'implémenter celui-ci comme une application particulière et de régler ses rapports avec les autres dans les termes normaux d'échange entre D-modules. Il suffira à une application utilisateur de connaître les noms des services système requis pour établir un dialogue. La solution la plus simple est pour le moment de considérer le système comme un seul D-module et portant le nom SYSTEME.

Autres applications

Nous avons pour le moment négligé le problème de la communication entre applications utilisateurs, c'est-à-dire surtout de la désignation de services d'autres applications. Cette communication n'est d'ailleurs pas forcément souhaitable. De toute façon, elle serait rendue possible très simplement par un passage à travers des requêtes système, à défaut d'être directe.

A3 - Lancement d'une application

Des dernières pages, il est facile d'extraire les informations que l'on devra fournir pour le lancement. Quel que soit le rapport de l'utilisateur à la machine : batch ou temps partagé il devra mettre en évidence :

- la liste des composants (D-modules),
- l'indication des lieux d'exécution des composants,
- éventuellement l'indication de liaisons à établir.

Ces indications peuvent être plus ou moins complètes :

- la liste des composants peut être complétée par le système qui trouve dans les premiers D-modules le nom de ceux qu'ils utilisent,
- l'indication de localité peut être plus ou moins précise : de la désignation explicite de sites à l'indication plus qualitative de caractéristiques souhaitées : taille mémoire, proximité de telle ressource etc.

Les réponses du système peuvent être également variées :

- refus ou acceptation suivant la disponibilité des sites choisis : manque d'espace mémoire, surcharge des communications...
- aide à la distribution de l'application à travers un dialogue où l'on indiquerait la charge du réseau, les ressources disponibles suivant les sites etc.

A4 - Vie et contrôle d'une application

Si la vie d'une application débute avec son lancement comment finit-elle? Comment vivent ses composants, c'est-à-dire les D-modules?
Pour ces derniers il est relativement aisé de dire qu'ils commencent et finissent avec leur arbitre, c'est-à-dire qu'une fois le programme de contrôle exécuté ils n'ont plus d'existence active et peuvent donc être éliminés.
Ceci pourra signifier qu'une application se termine avec le dernier de ses composants. Il est a noter que ceux-ci ne sont pas forcément présents dès le début de l'application. Un système réellement efficace devrait permettre, par programme ou commande, l'introduction de nouveaux éléments. Cette facilité sera d'autant plus utile que les ressources (mémoire par exemple) seront limitées sur le réseau. Il s'agit également là, d'une possibilité d'adapter l'application aux circonstances de son exécution : création d'une nouvelle unité sur tel site plutôt que sur tel autre etc.

Tout ceci demande un contrôle global de l'application contrôle qui n'a guèr été encore abordé par notre projet, ni d'ailleurs les décisions à prendre sur l'ensemble d'une application lorsque l'un de ses éléments est déficient erreurs de programmation etc. Quelques-uns de ces points font cependant l'objet de propositions dans la partie suivante. Ici, dans cette partie qui se veut représentative de l'état du projet,on ne peut qu'inciter les réalisations de systèmes supports à prévoir des requêtes qui permettront le contrôle programmé d'une application par l'un ou plusieurs de ses composants.

ex : - requête de création de D-modules
- requêtes de prise d'information sur l'état du système etc.

TROISIÈME PARTIE

DESCRIPTION D'UN LANGAGE

1. PRESENTATION

La partie précédente a privilégié la logique des primitives au détriment de l'expression proprement dite. Nous avons, en effet, pensé que pour mieux montrer les aspects originaux de nos mécanismes, il était préférable d'éviter les problèmes de syntaxe.
Nous demandons donc au lecteur de ne conserver de la partie précédente que l'esprit de la structure, du contrôle de l'activité.

Cette troisième partie présente une proposition de langage qui à défaut d'être complète, cherche du moins à être cohérente. Si elle est incomplète c'est en particulier parce que manquent les aménagements qui font un langage d'écriture de systèmes : interface avec la machine, gestion des interruptions, contrôle d'erreur. En revanche, tous les mécanismes introduits en partie II sont décrits par le langage.
Nous n'avons pas tenté de compléter le langage dans cette thèse pour éviter de figer des aspects du projet qui n'en sont qu'à leur début. Il est, par exemple, trop tôt pour définir précisément les règles d'exécution d'une application, s'il est possible de transférer en cours d'exécution des D-modules d'un site à l'autre. Egalement, on n'a pas encore spécifié précisément l'interface des applications avec le système, la désignation des ressources, la forme et la signification que l'on donne à des événements tels que rupture de communication, défaillance de périphériques etc.
Les aspects correspondants du langage sont donc laissés en suspens pour le moment.

Les pages qui suivent décrivent l'écriture des D-modules sous les aspects suivants :

- forme générale d'un D-module,
- matérialisation de la notion d'interface, liaison,
- description précise de l'écriture des services,
- désignation, utilisation des services,
- expression du contrôle,
- création dynamique de D-modules.

Nous n'avons pas désiré créer un langage de toute pièce mais préféré adapter un langage existant. Il ne s'agit cependant pas d'ajouter simplement nos primitives à un langage mais de tenter de le refondre suivant notre approche. Choisir un langage pour ses qualités de puissance d'écriture : PASCAL, le débarrasser de ce qui nous est inutile, le remodeler pour supporter nos mécanismes.

Si nous avons choisi PASCAL c'est essentiellement pour deux raisons :

- sa diffusion grandissante,
- sa simplicité, le petit nombre de ses primitives, c'est-à-dire sa logique claire.

Cette clarté se répercute dans l'écriture des compilateurs, ce qui les rend aisément modifiables et ce n'est pas là un mince avantage.
L'adaptation d'un langage n'est pas un simple aménagement, sa logique interfère avec les extensions qu'on lui apporte et l'on en verra de multiples exemples par la suite. C'est une autre raison du choix de PASCAL car ce langage, quoique procédural, est tout à fait compatible avec nos primitives.
Il est vraisemblable que le choix d'un autre langage aurait donné une image sensiblement différente aux services et D-modules.

Enfin, il est bon de préciser que les propositions faites en partie II restent entièrement valables quant à la définition des activités dans les services, à la signification des instructions d'envoi de requêtes, de contrôle, et au format des messages. On ne reviendra donc pas sur la signification des primitives déjà introduites, et on ne s'attachera qu'à leur écriture.

2. RAPPELS SUR PASCAL

On ne peut envisager ici, une description complète du langage mais on en rappelle les traits principaux.

2.1 - Structure d'un programme PASCAL

Un programme PASCAL comprend

- un en-tête qui comporte entre autres le nom du programme,
- une partie déclarative où l'on définit
  - . les étiquettes utilisées dans le programme principal,
  - . les constantes connues de tout le programme (globales)
  - . les types globaux,
  - . les variables globales,
  - . les procédures de niveau 0.
- une partie algorithmique : le programme principal (liste d'instruction

La structure même des procédures reprend celle du programme à la forme de l'en-tête près. On définit ainsi des niveaux d'imbrication de procédures et des règles de portée des données : une constante, un type, une variable, une procédure,déclarés dans une entité (programme, procédure) sont connus de cette entité et de celles que cette entité contient.

2.2 - Notion de type

PASCAL permet de définir des types de donnée nouveaux à partir des types simples primitifs (entiers, booléens, caractères etc).

- par restriction de l'éventail de valeurs possibles : types intervalles
  Exemple : UNADIX = 1..10;
  est le type : entier de 1 à 10;
- par construction à l'aide d'opérateurs : tableaux, enregistrements, ensembles.
  Exemple : table = array [1..100] of integer;

Ces opérateurs peuvent s'appliquer à des types déjà construits :

tabletable = array [2..10] of table

Enfin, il est possible d'introduire des types totalement nouveaux en précisant les valeurs qu'ils peuvent prendre : (type symbolique).

Exemple : jour = (lundi,mardi,mercredi,jeudi,vendredi,samedi,dimanche);
on crée le type jour en même temps que les valeurs que peuvent prendre les variables de ce type : lundi,mardi..

2.3 - Déclarations de variables

On déclare une variable par la double indication de son nom et de son type :

Exemples : vecteur : table;
délai : jour

on peut également définir le type de la variable directement

tableau : array [-100 .. +100] of char;

2.4 - Instructions

Le langage dispose d'un ensemble complet d'instructions :

instructions conditionnelles : if..,then... else, case... of
boucles : while.. do, for... to, repeat... until
affectation, appel de procédure...

3. DESCRIPTION DU LANGAGE

Lorsque l'on indique que PASCAL sert de base à notre propre langage il convient de préciser qu'il ne s'agit pas de tout PASCAL mais d'un sous ensemble ou plutôt de la logique du langage. PASCAL est conçu pour l'écriture de programmes sur un schéma procédural, ce qui ne correspond pas à la logique des D-modules. Par contre, il a été pourvu de primitives d'expression des données et du code qui sont indépendantes de toute interprétation procédurale ou autre.
Le premier pas de la spécification de notre langage ne peut donc être que l'étude précise des structures initiales retenues. C'est l'objet de la section 3.1. La section 3.2. en sera le complément, c'est-à-dire la description de nos apports propres.

3.1 - Utilisation de PASCAL

Nous n'avons pas cru nécessaire de restreindre les possibilités offertes par PASCAL quant au choix des constructeurs de types ou des instructions. Nos apports concernent plutôt la définition des activités, leur communication, leur contrôle. C'est dire que nous n'avons pas de contraintes sur la partie purement algorithmique du langage. Les constructeurs de types, les instructions sont donc conservés et étendus par certaines de nos primitives.
La similitude est grande entre la structure d'un programme PASCAL et celle d'un D-Module tel que nous l'avons décrit en partie II.
Les programmes écrits en PASCAL et les programmes de contrôle des D-modules se rapprochent par :

- leur structure imbriquée,
- leurs paramètres,
- la structure sensiblement voisine de leurs procédures et services.

Nous verrons par la suite que malgré cette similitude, nous ne pouvons nous contenter de substituer le mot-clé service à celui de procédure et obtenir aussi aisément un D-Module.
Un service est une entité beaucoup plus indépendante qu'une procédure. Nous verrons par la suite (3.2) comment nous pensons déclarer et manipuler les services. La notion de procédure sera, elle, absente des pages suivantes, cela ne signifie pas qu'elle soit définitivement abandonnée, mais nous n'avons pas voulu rendre trop complexe la description du langage en y faisant intervenir cette structure supplémentaire.

Nous conservons par contre le programme principal de PASCAL sous forme de programme de contrôle. Ce dernier est le lieu de lancement et d'initialisation du module, l'algorithme qui guide l'exécution des services. Cet algorithme décrit donc l'automate général du module, les services sont les actions plus précises exécutées par ce module. Aux termes près, et à l'activité séquentielle près, ce sont les caractéristiques du programme principal d'un programme PASCAL.

Nous conservons, pour ces raisons, la même forme pour l'expression du programme de contrôle : c'est-à-dire un bloc d'instructions (begin...end) à la suite des déclarations du module (types, variables, services etc). Il n'est pas introduit de primitive indiquant la spécificité de cet algorithme, sa situation particulière doit être suffisante pour que l'on puisse le discerner aisément.

En résumé, la base algorithmique tirée de PASCAL a la structure décrite par la figure II.1.

Cette structure demeurera celle des D-modules.
Nos extensions se situent dans l'en-tête, la déclaration des types, celle des variables.
Nous abordons ces extensions dans la section 3.2.

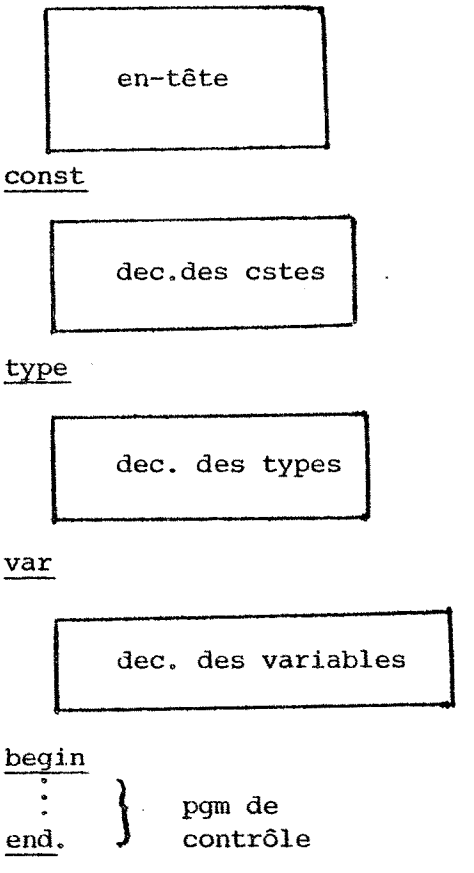


Figure II.1

3.2 - Primitives pour l'écriture des D-Modules

Sur le squelette retenu, décrit en 3.1, nous devons introduire les primitives propres à l'écriture des D-modules :

- notion de service : déclaration, structure, référence à des services extérieurs,
- instructions de contrôle (allow), d'envoi de requêtes (exec, transfer etc).

Nous essayons, dans le paragraphe qui suit, de dégager les caractéristiques qui nous semblent les plus importantes et ont donc largement dominé notre approche. Les paragraphes 3.2.2 et 3.2.3 décrivent précisément les extensions proposées.

3.2.1. Problèmes de bonne écriture des services

Ce sont ceux dont l'exposé n'était pas justifié dans une description logique (partie II) mais qui sont essentiels pour une réalisation pratique :

- le contrôle de la cohérence des définitions d'interfaces,
- la mémorisation d'identifications de services (liaison dynamique),
- le statut donné aux services : type de déclaration, gestion, etc.

3.2.1.1. Contrôle de cohérence

Nous avons jusqu'à présent décrit l'interface d'un service comme une liste de paramètres typés passés, par valeur ou résultat. Cette liste figure dans l'en-tête des services, ce qui dissémine les déclarations d'interface dans tout le module et rend difficile les vérifications de cohérence nécessaires dans les cas suivants :

a) Instruction transfer : cette instruction suppose que service émetteur et service destinataire possèdent la même interface (admettant des messages de même structure). Des définitions séparées rendent délicats les contrôles de compatibilité.

b) Interactions entre D-modules : il faut s'assurer, pour éviter d'éventuelles erreurs, que la déclaration d'un service dans un module, et les références qui lui sont faites dans d'autres sont bien compatibles.

c) Passage de services en paramètre : des identifications de services peuvent être passées en paramètre de requêtes. Il se pose dans ce type de liaison dynamique les mêmes problèmes de cohérence que dans les liaisons statiques.
Une vérification à l'exécution est possible mais coûteuse, il est préférable de faire cette vérification à la compilation, à la liaison ou au lancement d'une application.

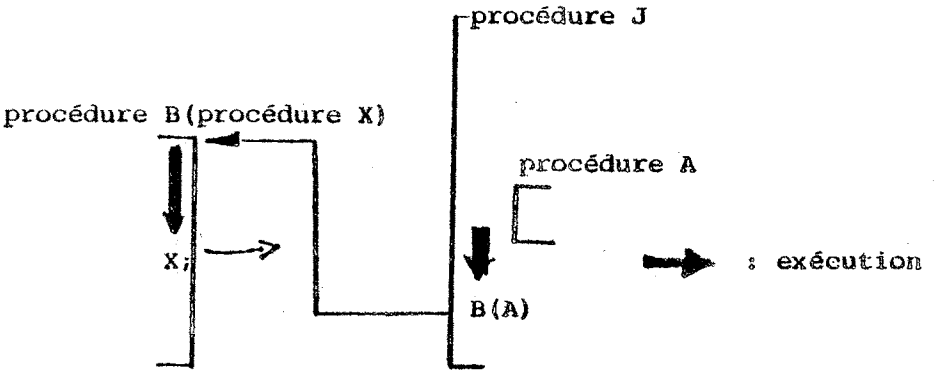
Pour faciliter le contrôle de la cohérence, au moins à l'intérieur d'un D-module, nous introduisons un type supplémentaire pour décrire le format des requêtes, c'est-à-dire le type et le nombre de leurs paramètres, il s'agit en quelque sorte d'un type interface.
La référence à l'un de ces types dans l'en-tête des services permet d'assure la compatibilité des interfaces de deux services par le simple fait qu'elles sont issues du même type. Les passages de services en paramètre, sont soumis à la règle suivante : un paramètre service doit être typé, c'est-à-dire que sa déclaration doit être accompagnée de son type d'interface. On donne des exemples en 3.2.2.

Les contrôles de cohérence entre D-modules se basent sur cette notion de type interface et sont décrits en même temps que l'en-tête des D-modules.

3.2.1.2. Mémorisation de l'identification de services

Le passage de services en paramètres rappelle celui des procédures en paramètres dans PASCAL. Son usage et les règles qui s'y appliquent en sont cependant assez différents et il n'est pas inutile de faire un parallèle entre les deux.
Le passage d'une procédure A en paramètre à une autre B permet, durant le temps d'exécution de B, des appels à A. Généralement, les langages procéduraux ne permettent pas à B de mémoriser l'identification de A pour des usages plus lointains. En effet, A peut être une procédure imbriquée, qui a accès aux variables locales des procédures qui la contiennent. Au moment de l'appel à B, les règles de l'appel procédural assurent que ces procédures sont en cours d'exécution, elles possèdent donc des variables locales. Si l'identification de A est mémorisée et si A est appelée lors d'une autre exécution de B, rien n'assure que ces variables locales auront encore une existence, c'est-à-dire que leur procédure sera encore en cours d'exécution.

Exemple :



Pendant l'exécution de B, J est en cours d'exécution et possède des variables que A peut utiliser. Un appel fait après l'exécution de B risque d'intervenir après l'exécution de J, ses variables locales n'auront aucune existence.
Le problème est différent dans les D-modules : une requête à un service n'est exécutée que lorsque l'arbitre l'a autorisée. Si le service est imbriqué, l'arbitre est le processus exécutant le service qui le contient, celui-ci est donc actif et dispose de variables locales.

De plus, la possibilité de mémoriser l'accès à un service est une facilité de programmation dans de nombreux cas (liste de processus à réveiller, liste de diffusion d'information, liaison dynamique etc). Nous reviendrons sur ses emplois, mais il faut noter qu'elle demande une structure de donnée supplémentaire : "le pointeur de service".

3.2.1.3. Statut des services

Nous avons indiqué en 3.1 que nous n'obtenions pas un langage SORTILEGE en remplaçant simplement le mot procédure par le mot service dans PASCAL. Il nous faut justifier cette affirmation.
Une procédure, au moins au sens classique du terme, est un outil de structuration des algorithmes, pas un objet au sens informatique courant du mot : entité manipulable, multipliable, que l'on peut combiner pour former d'autres objets etc. Il est sans intérêt d'écrire dans un même module trois procédures identiques, une seule si elle est réentrante suffit.
Un service, par contre, s'apparente plus à d'autres objets : les variables. Il n'est pas irréaliste d'écrire deux services identiques pour implémenter la gestion de deux imprimantes identiques. Il n'est pas impensable de disposer d'un tableau de n services : par exemple pour représenter les n consoles d'une configuration ou n disques etc.

Aussi, il nous semble d'un grand intérêt de pouvoir appliquer aux services les mêmes opérateurs de construction, de combinaison, qu'aux variables. Pour respecter la logique du langage, il est naturel d'introduire un déclarateur de type service, c'est-à-dire le moyen de créer des modèles de services. Ainsi, de la même façon que l'on peut créer des types tableaux entiers, nous permettons de créer des types services.
Et de la même façon que l'on peut définir des combinaisons de variables (tableaux..) nous permettons de combiner des services.
Cette généralisation est cependant soumise à quelques restrictions :

- certaines opérations permises sur les variables habituelles ne peuvent être appliquées aux services. Exemple : l'opération d'affectation d'un service à un autre;
- PASCAL et d'autres langages évolués permettent la création dynamique de variables à travers la notion d'enregistrement. Il est concevable, et il peut être utile, de permettre la création dynamique de services. Manquant d'exemples où cette possibilité s'avère indispensable nous ne l'avons pas retenue. De plus, elle se heurte à un problème de cohérence vis à vis de la logique de nos propositions : tout comme une variable créée dynamiquement, un service créé dynamiquement ne peut atteint que via un "pointeur". Si le langage permet la duplication (par affectation) du pointeur, il devient difficile de déterminer quel arbitre dirige le nouveau service. On pourrait interdire de telles affectations et rendre la création dynamique possible, mais nous ne l'envisageons pas ici. Nous ne fournirons donc pas de fonction de création de service;
- Compte-tenu de l'ordre des déclarations dans le langage (constantes, types, variables) on ne peut faire intervenir dans un modèle de service, les variables du module qui le contient. Ses instructions n'ont accès qu'aux données locales du service : paramètres et variables propres (locales). Ceci restreint dans une large mesure l'emploi de ces modèles pour la déclaration de services du module. Par contre, cette "étanchéité" entre le module et les modèles qu'il contient permet de s'en servir pour la création de nouveaux D-modules. Nous y reviendrons en 3.2.3.4.

Pour illustrer les diverses considérations de ce paragraphe, nous décrivons dans le suivant, les structures syntaxiques qui en sont issues, accompagnées d'exemples d'écriture.

3.2.2. Description des types nouveaux du langage

Ces types sont au nombre de deux et sont étroitement dépendants;

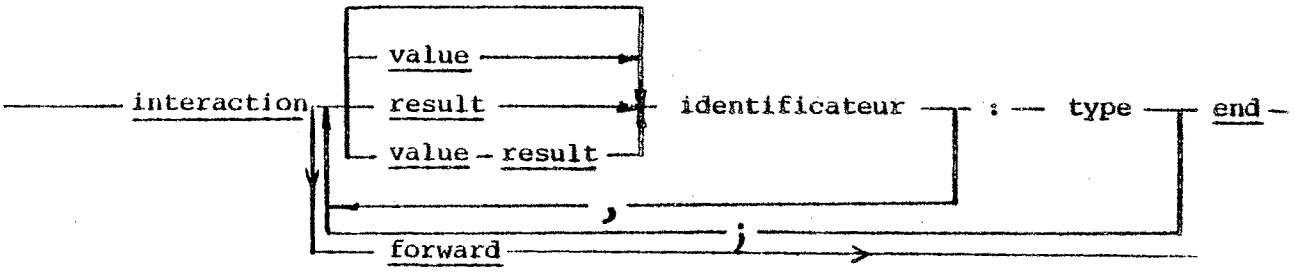
- l'un définit l'accès à un service,
- l'autre décrit les services.

3.2.2.1. Type interaction

Ce type doit être interprété comme la description des voies d'accès aux services. Il est construit avec une liste de champs qui sont les paramètres d'une classe de services. Il en décrit donc l'interface.
Un type voie d'accès est donc référencé par les services qui l'admettent pour interface. Une variable de ce type est réellement une voie d'accès, c'est-à-dire un pointeur vers un service admettant l'interface décrit par le type de la variable.

Syntaxe

Déclaration d'un type interaction :



Note : *On utilise ici, comme dans toute la suite, une description graphique du langage. (carte syntaxique : cf. le manuel PASCAL-SFER). Les identificateurs soulignés sont les mots-clés du langage.*

Explication

La déclaration d'un type interaction se compose d'une liste de champs (ou paramètres) typés, dont on indique le type de passage par valeur (explicitement ou par défaut, résultat, valeur résultat).

Exemples

```
type
...
ligne = array [0..79] of char;
messagein = interaction result msg : ligne end;
messageout = interaction msg : ligne end;
var
lecteur : messagein;
imprimante : messageout;
...
```

messagein et *messageout* sont deux types de requêtes, le premier rend un tableau de 80 caractères, le second est composé d'un tableau passé par valeur.

Lecteur et *imprimante* sont des variables pouvant désigner des services qui acceptent respectivement ces deux types d'interactions :

```
msg = interaction nombre : integer end;
requête = interaction question : integer; réponse : msg end;
```

requête décrit une structure de requête qui contient un entier et une variable de type *msg* (= un pointeur sur un service acceptant *msg* comme interface).

Il est possible que deux types interactions se fassent mutuellement référence. Pour permettre l'utilisation d'une interaction non encore définie, on introduit le mot-clé forward. Il indique que le nom qui le précède est celui d'une interaction décrite plus loin.

Exemple :

```
sens 1 = interaction forward;
sens 2 = interaction truc = integer; question : sens 1 end;
sens 1 = interaction c:char; réponse : sens 2 end;
```

Opérations sur les interactions

La suite de l'exposé montrera que les variables de type interaction sont utilisées dans l'envoi de requêtes. On peut les faire intervenir :

- dans des instructions d'affectation,

  Exemple : pour les types définis ci-dessus :

```
var
pt1, pt11 : sens 1;
...
begin pt1 := pt11
```

- dans les comparaisons,

  Exemple :

  if pt1 = pt11 then ...

La valeur initiale d'une variable interaction est nil; on peut modifier cette valeur par une affectation dont la partie droite est le nom d'un service.

La constante prédéfinie nil indique une valeur sans signification.

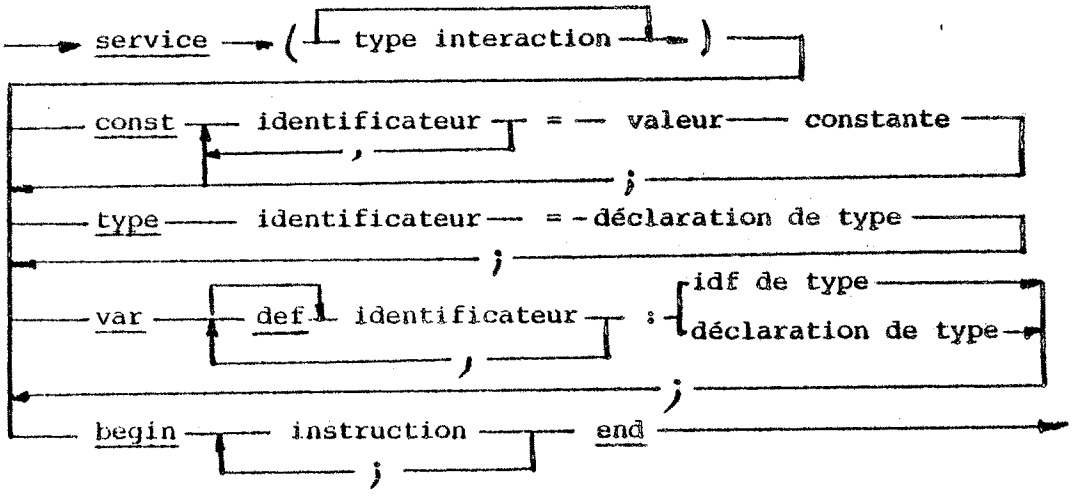
Exemple :

pt1 := nil; ... if pt11 = nil etc.

3.2.2.2. Type service

C'est la description d'un type de service, elle requiert; on l'a vu, la référence au type de requête accepté et définit les types, les variables locales (dont éventuellement des services), et l'algorithme du service.

Déclaration de type service



Explication

Un service se compose :

- du mot clé service suivi, entre parenthèses du nom d'un type interactio suivi :
- éventuellement de const puis d'une liste de déclarations de constantes, (qui ne peuvent être qu'entières ou de type caractère),
- éventuellement de type puis d'une liste de déclarations de types,
- éventuellement de var puis d'une liste de déclarations de variables, où l'on indique soit le type de la variable (idf de type) soit directement sa structure (déclaration de type),
- dans tous les cas d'un bloc d'instructions.

Le mot clé def devant une variable s'applique uniquement aux variables de type service. Il autorise la liaison du service avec l'extérieur.

Exemples

```
type ch 80 = array [1.. 80] of char;
ligne = interaction msg : ch 80 end;
...
var
imprimante : service (ligne);
     var i : integer;
     begin
     for i := 1 to 80 do {écriture de msg [i]}
     end;
...
```

La déclaration de nouveaux services parmi les variables d'un service permet l'imbrication.

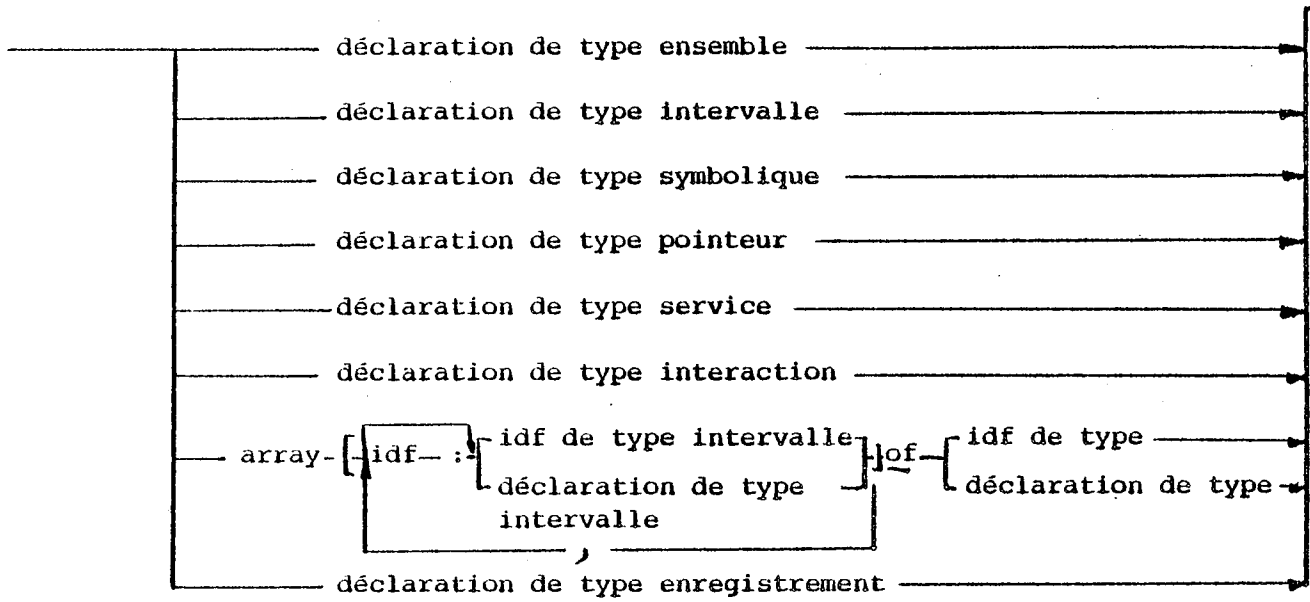
Exemple

```
type
ch 80 = array [1.. 80] of char;
ligne = interaction result msg : ch 80 end;
carac = interaction result car : char end;

modulelecture = service (ligne)
     var
     lecturelcar  : service (carac);
         begin ... end;
     begin
     ...
     end;
var
lecteurcarte : modulelecture;
```

On décrit un modèle de service : modulelecture contenant un service imbriqué *lecturelcar.lecteurcarte* est un service réel issu de ce modèle. Pour donner d'autres exemples, il est utile de rappeler que l'entité syntaxique déclaration de type est construite à partir de l'entité équivalente de PASCAL à laquelle on ajoute nos types interaction et service.

déclaration de type idf = identificateur



où les différents déclaration de... renvoient aux déclarations de types PASCAL.

On note simplement une modification dans la déclaration des tableaux : les indices peuvent être identifiés, c'est-à-dire que l'on peut donner un nom aux indices. Cet ajout trouve son utilisation dans les tableaux de services, lorsque des instructions de chaque service diffèrent selon l'indice du service.

Exemple de la gestion d'un ensemble de 5 unités de disques

```
type
sens = (lire, écrire);
bloc = array [1.. 256] of char;
ordre = interaction ecr-lec : sens; adresse : integer; value-result 5 :
                                                        bloc end;
...
var
disques : array [nodisque : 1..5]  of service (ordre),
    begin
        réalisation d'une lecture ou écriture sur l'unité nodisque
    end;
```

On déclare un tableau de 5 services identiques au numéro près de l'unité physique particulière qu'ils gèrent.

La définition formelle du langage autorise la combinaison de services dans de multiples structures : enregistrements, tableaux à une ou plusieurs dimensions, tableaux de tableaux, ensemble etc.
En pratique, on n'autorisera que les structures basées sur le constructeur tableau. La définition de services dans des ensembles ou des enregistrements présenterait en effet peu d'intérêt.

3.2.3. Ecriture d'un D-module. Accès aux services

Il nous reste à décrire :

- l'en-tête des D-modules,
- les références aux services dans :
  - l'envoi de requêtes,
  - l'expression du contrôle.

Ces points, pour être convenablement abordés, demandent un supplément d'analyse, et des choix qui n'avaient pas trouvé leur place jusqu'ici.

3.2.3.1. Harmonisation des notions de services et de D-module

Nous avons implicitement admis jusqu'ici que le D-module se confondait avec l'unité de compilation. Nous avons également supposé que le D-module est une entité particulière matérialisée dans le langage, distincte de celle de service. Or nous n'avons pas cessé de chercher une continuité, une homogénéité de nos concepts vers le bas : un service se conduit comme un D-module par rapport aux services qu'ils contient, ceux-ci peuvent en contenir d'autres etc. Nous montrerons par la suite que l'on peut créer des D-modules à partir de modèles de services. Pourquoi ne pas assurer cette continuité vers le haut ? C'est à dire supposer qu'un D-module peut faire partie d'un ensemble plus complexe, être manipulé, détruit, créé par un programme plus global qui définirait l'application. Nous ne pouvons guère faire de suppositions sur la description d'une application à ce stade du projet, mais nous devons offrir un maximum de souplesse pour des études ultérieures.
Aussi, nous supposons possible la création de nouveaux modules par un module existant et nous admettons qu'un D-module peut faire partie d'un ensemble plus complexe. On entend, sous ce dernier point que l'application peut être décrite par un programme qui manipule, crée, détruit les D-modules.

C'est donc une continuité vers le haut (l'application) et vers le bas (la création d'autres modules) que nous voulons assurer. Et la notion de service se prête bien à cette approche.
En effet, qu'est-ce qu'un service qui n'utilise que ses variables propres, sinon un D-module créé à chaque consommation de message et disparaissant à chaque fin de cycle ?
La possibilité de définir des modèles de services devient ainsi possibilité de créer des D-modules à partir de ces modèles. Les paramètres des services devenant paramètres de génération des nouveaux-D-modules.
Une manipulation équivalente mais à l'échellon de l'application et sur les D-modules est envisageable, mais, nous l'avons dit, ne rentre pas dans notre sujet. Cependant, il est naturel de décrire un D-module comme un service que l'application pourra utiliser, au gré du concepteur. Et même si celui-ci préfère une structure statique, au réseau, les paramètres des services/D-modules pourront servir au paramétrage de l'application à son lancement.

3.2.3.2. Cohérence des définitions d'interface

La section 3.2.1 a montré que l'on pouvait assurer cette cohérence dans un D-module par le biais du type interaction? Tous les services, tous les "pointeurs de services" font référence à un type interaction, il est facile de contrôler que les interfaces sont bien respectés (par la référence à une même interaction). Cette cohérence est maintenue dans les D-modules que l'on peut créer à partir de modèles de service. Ces modèles sont décrits dans le module initial et bénéficient du même contrôle à la compilation, dans leur écriture et dans leurs rapports avec le D-module père. Mais l'on ne résoud pas ainsi les problèmes de vérification entre D-modules écrits séparemment. Aussi, nous faut-il trouver un moyen d'assurer la cohérence et le moyen réside obligatoirement dans une vérification lors d'une phase de liaison ou de lancement. Pour rendre le contrôle possible, nous allons de nouveau extrapoler sur la notion de D-module/service. Si un D-module peut être assimilé à un service de l'application, celle-ci peut être assimilée à un D-module.
Les différents types communs aux D-modules, (interactions, etc) sont dans ce cadre, les types globaux de ce D-module application.
Notre façon d'écrire un D-module découle de cette approche : chaque D-module est précédé d'une liste de déclarations, sous-ensemble des déclarations de

constantes et types communs à toute l'application. Cette liste est précédée du mot clé common, pour souligner l'aspect partagé des déclarations qui la composent et qui sont écrits là comme une sorte de rappel des objets globaux d'un D-module application. Conservés dans le code objet, ils pourront servir à des comparaisons lors du lancement.

3.2.3.3. Désignation et accès aux services

Comment nomme-t-on et qui peut utiliser un service ?
Peut-on accéder par exemple à un service imbriqué dans un autre ?
Un strict décalage de la structure procédurale (qui guide après tout grossièrement la structure de nos services) nous interdirait l'accès direct par l'extérieur à un service imbriqué. Mais cette interdiction, dans le schéma procédural, est dictée par l'impossibilité d'exécuter une procédure imbriquée quand celle qui la contient n'est pas active. Cette impossibilité est levée, on l'a vu en 3.2.1, par les règles d'exécution de service.

Autre entrave possible à l'accès direct : les règles de propriété, également issues du schéma procédural. Les objets d'une procédure (variables etc..) sont inaccessibles aux procédures de niveau égal ou supérieur. Il n'est pas certain que ce choix, qui se conçoit pour une variable classique (entier etc) soit justifié pour un service. Nous n'avons jusqu'ici donné comme droit hiérarchique que le pouvoir de contrôler un service imbriqué, pas celui d'en disposer exclusivement. Il n'est pas impensable d'autoriser la connaissance, par l'extérieur, des services imbriqués. Le contrôle restant à la charge de l'arbitre possesseur du service.
L'utilisateur n'a à connaître que l'interface et les actions d'un service et peut tout ignorer de la réalisation exacte des actions et du contrôle exercé.
Les services connus de l'extérieur, quel que soit leur niveau d'imbrication sont indiqués par le mot clé def (cf. la carte syntaxique en 3.2.2.2).

Cette facilité peut rendre plus complexe la désignation d'un service. Il serait possible d'exprimer le chemin d'accès à un service par la liste de ceux qui le contiennent, suivie de son nom (comme on le fait pour l'accès aux champs d'enregistrements). C'est une écriture quelquefois lourde. Inversement, permettre que tous les noms de services soient connus partout peut multiplier les homonymes et donc rendre impossible la désignation, ou rendre difficile l'écriture sans homonymes. Nous avons

choisi un moyen terme en permettant que les noms de services soient tous connus partout et en admettant les homonymies. Lorsqu'une confusion est possible, le service est précisé par les noms d'un ou de plusieurs des services qui le contiennent.
Il faut également se souvenir que les services peuvent être combinés en tableaux et que ce fait doit être répercuté dans la désignation des services.

Les paragraphes suivants montrent la matérialisation de ces propositions, c'est-à-dire, en résumé :

- l'expression de la notion de D-module, sa matérialisation,
- le contrôle de la cohérence des déclarations de divers modules,
- la désignation des services.

3.2.3.1. En-tête d'un D-module

Un D-module est définit comme un service de l'application. Il a la possibilité de créer des D-modules à partir de ses modèles de service. Il n'existe donc pas de structure syntaxique D-module ; tout est ramené à la notion de service. On conserve cependant le terme D-module pour désigner une entité indépendante ne partageant aucune variable avec d'autres. En architecture distribuée, c'est l'unité de répartition des éléments d'une application.

La définition d'un D-module comprend :

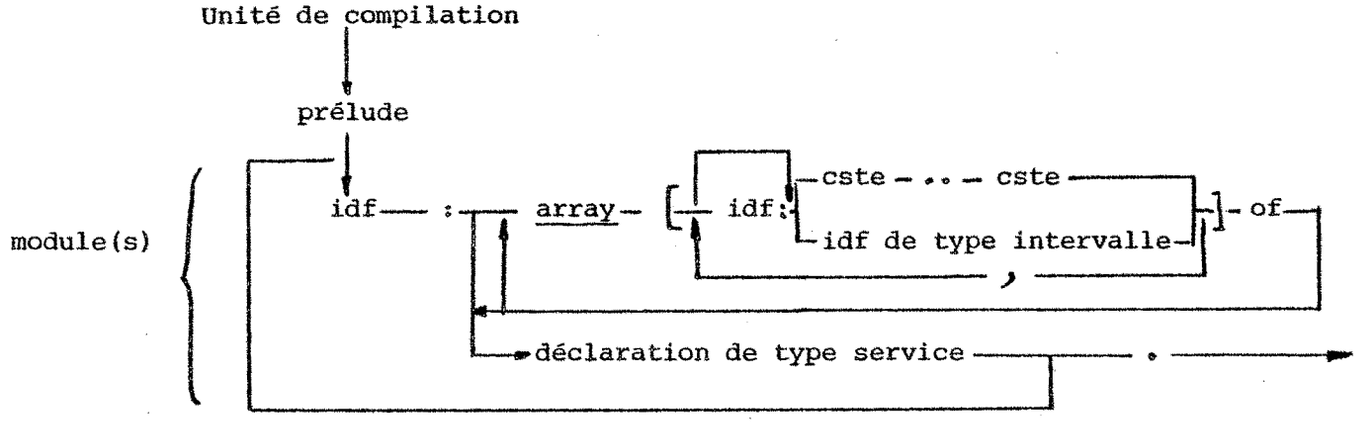
- un prélude
- un service : le D-module lui-même.

Le prélude (qui peut être vide) regroupe la déclaration :

- des types et constantes que le D-module partage avec les autres D-modules de l'application. Particulièrement les types interaction utilisés dans les échanges entre modules. Cette liste de déclarations est précédée du mot clé common;
- des services extérieurs utilisés par le D-module. Cette liste est précédée du mot-clé link.

Le prélude est destiné à remplacer la cohérence que fournirait une compilation globale de l'application. Les constantes, les types décrits sont les constantes et types globaux de l'application; ils sont déclarés dans le prélude comme une sorte de rappel des déclarations générales.

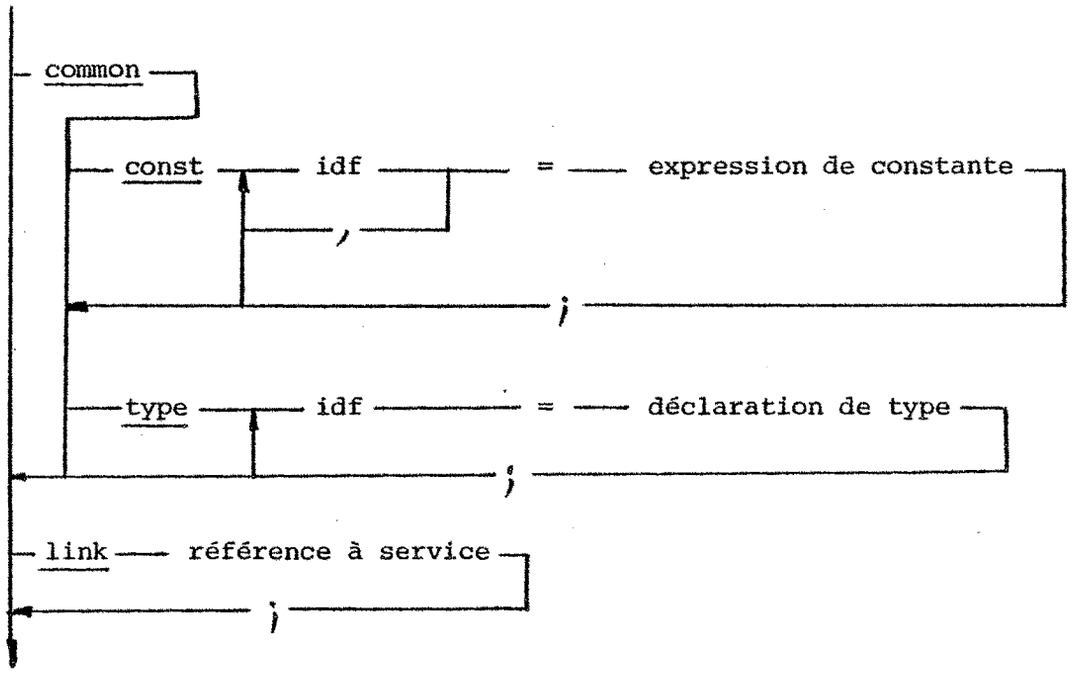
Il en est de même pour les services cités en référence. Ce sont les services d'autres modules qui sont utilisés par celui qui suit. On décrit leur en-tête, c'est-à-dire la façon dont ils sont organisés (tableaux etc), et le type d'interaction qu'ils admettent. On peut les déclarer séparément avec leur nom seul. On peut également rappeler la façon dont ils sont imbriqués dans d'autres si des confusions sont possibles.

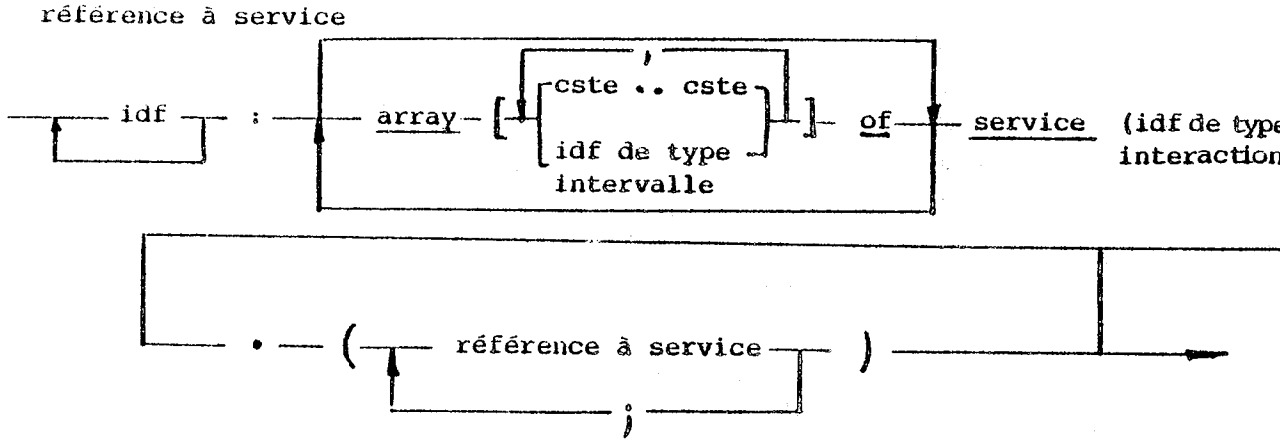


De ce premier schéma on peut déduire :

- qu'on peut déclarer une structure complexe de D-modules : tableaux à plusieurs dimensions, ou imbriqués, de D-modules identiques,
- qu'on peut déclarer à la suite plusieurs D-modules ayant le même prélude.

prélude :





Les déclarations de types et constantes communs sont identiques à celles faites dans le corps des services.
La notion référence à service demande plus d'explications. La partie link se compose d'une liste de références à des services extérieurs. Ces références peuvent être simples :

Exemple :

```
common
type envoisimple = interaction c : char end;
link
  printer : service (envoisimple);
  puncher, ligne : service (envoisimple);
...
```

Elles peuvent désigner des structures de plusieurs services.

Exemple :

```
link
pooltty : array [1.. maxtty] of service (envoisimple);
```

Elles peuvent enfin déclarer à la fois des services et ceux qu'il contiennent.

Exemple :

```
link
sgf : array [1.. maxfich] of service ()
      (service ouvrir (nomfich);
       service fermer (nomfich);
```

On désigne un ensemble de services structurés ainsi :

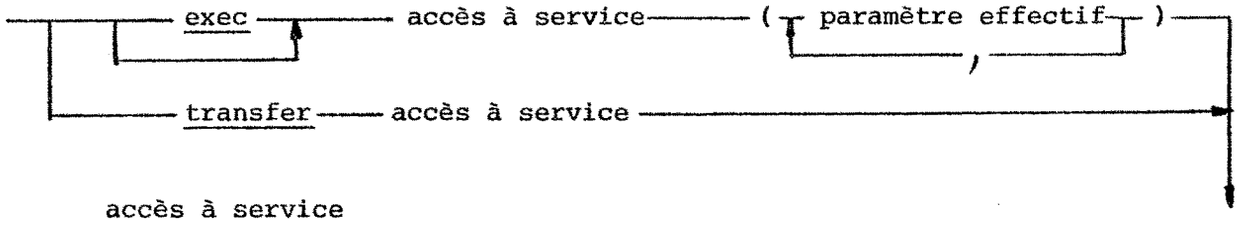
```
sgf : array [1.. maxfich] of service ();
        {
        var
        ouvrir : service (nomfich);
          {
        fermer : service (nomfich);
          {
        lire   : service (ordrelecture);
          {
        begin  %pgm de contrôle de chaque élément du tableau sgf%
          {
        end;
```

On voit que dans ce dernier exemple le simple énoncé d'un nom de service tel qu'ouvrir n'aurait pas suffit à déterminer le service référencé. Il y a en effet *maxfich* services ouvrir.

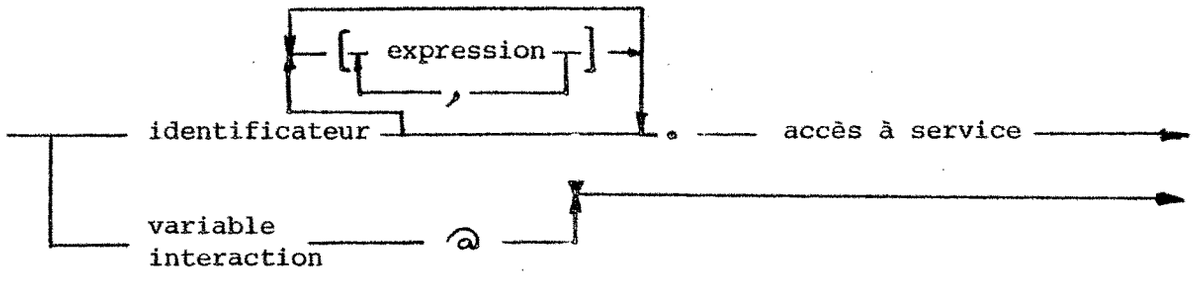
3.2.3.2. Envoi de requêtes

Les primitives d'envoi de requêtes : demande blocante, non blocante, transfer obéissant à la syntaxe suivante :

envoi de requête



accès à service



L'accès aux services est différent suivant que l'on connaît le service par son nom ou via une interaction. Dans le second cas on y accède par la variable interaction postfixée par @. Dans le premier, la désignation varie suivant qu'il s'agit d'un service extérieur ou intérieur.

Service extérieur : on reproduit le chemin d'accès décrit dans la partie link.

Service intérieur au module : on le nomme par son nom ou un chemin d'accès suivant qu'il y a ambiguité ou pas dans la désignation.

Exemple :

L'appel des services extérieurs décrits dans les exemples de 3.2.3.1. peut s'écrire ainsi :

```
    printer ('*');
exec puncher (x);
    transfer (ligne)
    exec pooltty [3] ('%');
    sgf [i] (...);
    transfer sgf [3]. ouvrir
```

Par convention, le fait de ne pas citer un ou plusieurs indices d'un tableau de services, indique que l'on s'adresse à l'un quelconque des services du tableau. On dispose ainsi d'un moyen d'autoriser la réentrance d'un service : on en crée autant de copies que l'on désire. Le degré de réentrance à un moment donné est le nombre des copies que l'instruction allow a libéré.

Exemple :

```
    compilateur : array [0.. 2] of service (...)
    ...
    exec compilateur (...);
```

Le service compilateur peut traiter 3 requêtes en même temps (Si les trois éléments de tableau ont été libérés).

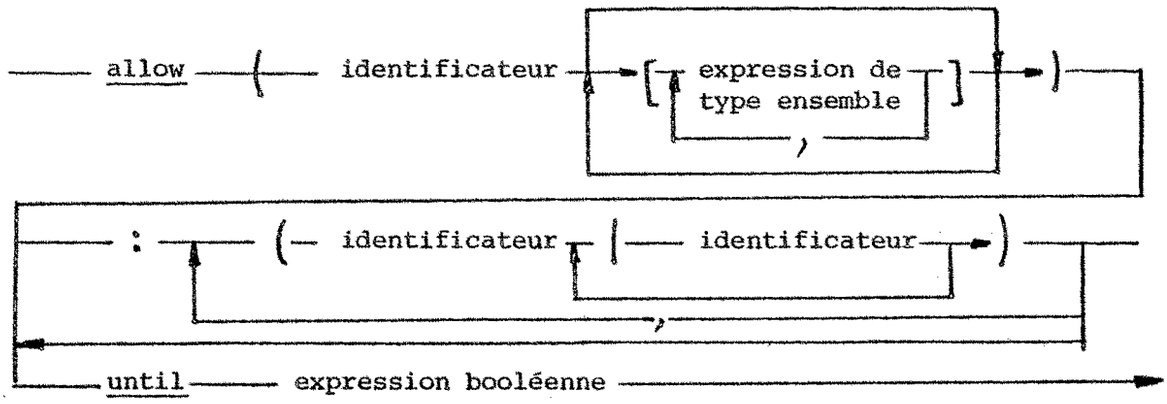
3.2.3.3. Instruction allow

Les diverses extensions apportés au langage jusqu'ici ne modifient que peu la syntaxe de la primitive allow; nous devons simplement tenir compte de la notion de tableau de services à travers les deux points suivants :

- on doit pouvoir énumérer des éléments de tableau,
- ces éléments peuvent être exclusifs entre eux.

La syntaxe adoptée est la suivante :

instruction allow :



Ce qui peut s'interpréter ainsi :
La primitive allow cite les service libérés

- par leur nom si ce sont des services simples,
- par leur nom et entre crochetsun ensemble d'indices si ce sont des services structurés en tableaux. L'entité expression-de-type-ensemble est issue de PASCAL. Ce peut être le nom d'une variable de type ensemble ou une constante de type ensemble ou une expression mettant en jeu des ensembles.

La partie de l'instruction préfixée par : regroupe des listes de services exclusifs entre eux. Lorsque l'on veut indiquer que les différents élements d'un tableau de services sont exclusifs, on cite deux fois le nom du tableau dans un groupe d'exclusion.
Exemple :

```
ens : set of 1.. 10;     %ensemble de nbres de 1 à 10%
s1  : array [1.. 10] of service (interface);
      ...
s2  : service (ligne)
      ...
```

```
begin
...
ens := (1,3,5 .. 10);  %ens := éléments 1,3,5,6,7,8,9,10%
allow (s1 [ens] , s2)                    (1)
until exp;
allow (s1 [ens  & (1..5)])               (2)
  • (s1!s1!s2)
until exp2;
```

A la ligne (1) on libère *s2* et les éléments de *s1* dont les indices appartiennent à *ens*.
A la ligne (2), on libère *s2* et les éléments de *s1* dont les indices appartiennent à l'intersection de *ens* et de l'ensemble des entiers de 1 à 5.
Les différents éléments de *s1* doivent s'exécuter en exclusion mutuelle, ainsi qu'en exclusion avec *s2*.

3.2.3.4. Instruction create

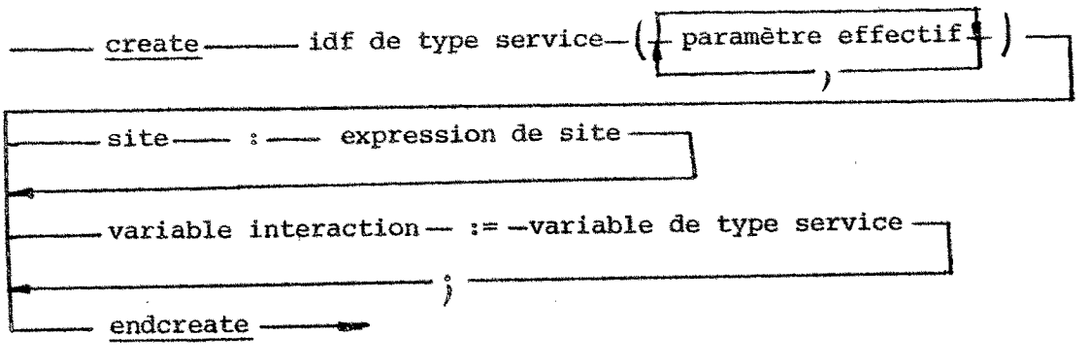
C'est elle qui permet de créer de nouveaux D-modules à partir de modèles de services. Elle doit résoudre les problèmes suivants :

- déclaration des paramètres effectifs du service s'ils existent,
- édition de liens avec le D-module père si celui-ci veut avoir accès aux différents services du nouveau D-module,
- définition du site d'exécution.

L'édition de liens avec le nouveau D-module ne peut être que dynamique.
La création du D-module doit s'accompagner du chargement d'un ensemble de "pointeurs" avec l'identification des services accessibles du D-module (signalés par le mot-clé def).

syntaxe

instruction create



Cette instruction précise :

- le nom du modèle et les paramètres effectifs du service/D-module créé
- le site de création, sous une forme que nous ne pouvons guère préciser dans l'état actuel du projet,
- les liaisons interactions appartenant au D-module créateur (les pointeurs de services) et les services du nouveau D-module. Cette suite peut ne pas être complète.

On peut noter que cette création pourrait se faire sur un modèle de tableau de services. Il est cependant préférable, pour des raisons de contrôle des créations, de créer les nouveaux D-modules un par un.

4. EXEMPLE

Recherche parallèle dans un arbre

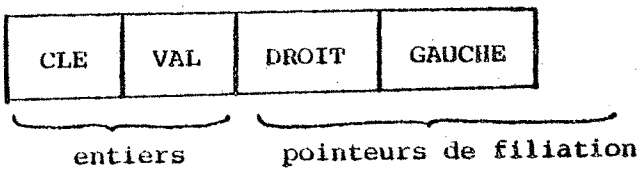
Cet exemple cherche à illustrer les capacités d'exécution parallèle et de réentrance de notre langage.

On implémente un service *question* acceptant des requêtes de format *ques* :

un paramètre entier par valeur : *clef*

un paramètre par résultat *réponse*.

Le rôle de *question* est de chercher dans un arbre binaire dont les enregistrements sont du format suivant :



la valeur associée à la clé *clef*.

En retour de question réponse contient la valeur ou -1 si on n'a pas trouvé la clé.

Pour réaliser cette recherche question la confie à un pool de services équivalents : *recherche*.

Une exécution d'un des services *recherche* se fait ainsi :

- comparaison de la clé du noeud passé en paramètre avec *clef*. S'il y a identité on place la valeur dans le paramètre par résultat de *question*,
- sinon on lance deux requêtes parallèles sur les deux sous arbres fils du noeud courant.

On rappelle que si on ne précise pas d'indice dans un envoi de requête à un tableau de services, ces services sont équivalents et partagent les requêtes.

```
type ques = interaction clef, result reponse : integer end;
     pointeur =  noeud
     noeud = record clé, val : integer; droit, gauche : pointeur end;
var
racine : pointeur; %pointeur sur la racine de l'arbre%
question : service (ques );
     type : oper = (add, sub);
      msgcø = interaction operation : oper end;
      ordrerech = interaction ptr : pointeur end;
         %type requête à recherche% .
```

```
var
compteur : integer; trouvé : boolean;
gere compteur : service (msgcø)
    ┌begin %assure l'exclusion mutuelle dans l'accès à compteur%
    │if oper = add then compteur := compteur+1
    │else compteur := compteur-1
    └end;

recherche : array [1.. 10] of service (ordrerech);

      begin
      if  (ptr ≠ nil) & (not trouvé) then
          begin
          if ptr  .clé = clef then begin trouve := true;
                                         resultat := ptrↂ.valeur;

                                   end
          else          begin %lancement de recherche sur les fils%
                  exec  gere compteur (add); exec recherche (ptrↂ.droit);
                  exec  gere compteur (add); exec recherche (ptrↂ.gauche)
                  end;
          end;
          gere compteur (sub);
      end;

      begin %programme de question%
      trouvé := false; reponse := -1;
      compteur := 1;
      exec recherche (racine);
      allow (gère compteur, recherche  (1.. 10) ) until compteur = 0;
      end; %emission implicite de réponse vers le demandeur%
```

Il ne s'agit pas d'une exécution parallèle complète, il ne peut y avoir en effet que 10 processus recherche actifs au maximum. Mais on contrôle mieux ainsi le degré de parallélisme des programmes.

La partie contrôle est basée sur les points suivants :

- on ne peut détruire les requêtes superflues lorsque la clé a été trouvée, mais on peut empêcher que ces requêtes n'en produisent d'autres. C'est le rôle de l'indicateur *trouvé* ;
- on s'arrête lorsque toutes les requêtes ont été exécutées :
  - si la clé est trouvée : quand les requêtes superflues sont traitées,
  - sinon, quand tous les noeuds ont été examinés.

Il faut donc compter les requêtes émises et décompter les requêtes traitées; c'est le rôle de compteur. Cet entier est géré par un service spécialisé pour éviter les conflits d'accès et les incohérences.

Il faut noter que cette utilisation récursive de *recherche* n'est possible que parce que les requêtes sont non blocantes, sinon on arriverait vite à une saturation.

QUATRIÈME PARTIE

EXEMPLE : ÉCRITURE D'UN PETIT SYSTÈME

1. PRESENTATION

Nous décrivons ici les traits principaux et la structure d'un petit système de temps partagé. Ce système offre à un utilisateur à sa console un langage de commande simple et la possibilité de lancer l'exécution de programmes interactifs ou background.

On supposera la configuration dotée de :

UDISK disques
NCONS consoles
une imprimante.

On suppose pour simplifier l'exemple que ce système est ouvert, c'est-à-dire que les utilisateurs partagent les fichiers, que l'on ne se pose pas de problèmes de confidentialité, sécurité etc.

Le système écrit sous forme de D-modules se compose des éléments suivants :

- gestion directe des périphériques :
  - . imprimante : gestion de l'imprimante
  - . TTY : gestion de l'ensemble des terminaux
  - . disque : gestion de chaque disque
- éléments du système :
  - . interpréteur : interpréteur du langage de commande
  - . sgf, sgfloc, fichier.. éléments du système de gestion de fichiers
  - . moniteur, guide : éléments du système d'exécution des programmes.

Les modules sont très autonomes et peuvent généralement être étudiés indépendamment les uns des autres.

Nous regroupons ci-dessous les déclarations des types communs, pour éviter leur répétition en tête de chaque module.
On rappelle en regard de chaque service, les principaux composants des messages qu'ils traitent.

```
const  NOPISTE : nbre de pistes par disque.
       NCONS   : nbre de consoles
       NSECT   : nbre de secteurs par piste
       NDISK   : nbre d'unités de disque
       NFICH   : nbre max. de fichiers par disque

type

  ch16 = array [0 .. 15] of char ; % nom de fichier %
  ch256 = array [0 .. 255] of char; % bloc disque ( secteur )%
  int128 = array [0 ..127] of integer; % idem considéré comme tableau
                                         d 'entiers %
  ch80 = array [0 .. 79] of char;  % ligne de terminal %

  typeaccès = (creer , ecr , lire , ajout ); % type d'accès à un fichier%
  typerel = ( interactif,background ); % mode d'exécution de pgm %

  %types interaction %

  vide = interaction end; %services sans paramètres %
  entiervaleur = interaction valeur : integer end ;
  entierrésultat = interaction value result resultat : integer end;
  outcons = interaction tampon : ch80 end; %ligne à écrire %
  incons = interaction result tampon : ch80 end ; %ligne à lire %
  ordrebatch = interaction nomfich : ch16; noutil : integer ;
                           relation : typrel  end;
              % envoi d'un travail pour exécution %
  echangedisque = interaction secteur : integer; sens : écr..lire ;
                           value result tampon : ch256 end;
  msgdetruire = interaction nomfich : ch16; value result code : integer
                end;
  msgfermer = interaction value result code : integer end;
  msgacces = interaction noctet : integer; value result tampech : ch256;
                value result code : integer; result eof : boolean end;
             %acces à un fichier %
  msgfichier = interaction unite : integer; nofich : integer end;
             %msg de lancement d'un module fichier%
```

```
msgouvrir = interaction nomfich : ch16; typdem : typacces;
                        value result code : integer;
                        result sacces : msgacces; result sfermer :msgfermer
                        end;
                        % demande d'ouverture de fichier %
lancelec = interaction descripteur : int128; unite : integer end;
                        %msg de lancement d'un module lecture %
```

2. MODULE TTY

2.1 - Commentaires sur le module tty

Composition : chaque console de numéro no est gérée par un service *console* [*no*] qui dispose de deux sous-services *lire* et *écrire*. Faute de conventions plus précises dans les rapports entre application et système support nous n'avons pas décrit la réalisation proprement dite des entrées-sorties.

Gestion des consoles : à l'initialisation du système, les consoles sont supposées inoccupées. Pour activer sa console et commencer à travailler, un utilisateur doit appuyer sur la touche "break". Pendant le travail cette même touche sert à annuler une commande en cours. Lorsque l'utilisateur indique à l'interpréteur qu'il cesse de travailler, celui-ci prévient le module *tty* par un message à *arrêt*. La console est alors désactivée.

Réalisation : on suppose :

a) Que l'utilisation de break est asynchrone par rapport aux entrées-sorties normales.

b) Que le système, à la réception de ce caractère prévient *tty* par un message à *break*, message qui contient le numéro de la console.

La seule donnée globale est *activité*: ensemble des consoles actives. *break* lui ajoute des éléments (et active les services correspondants), *arrêt* lui retire des éléments.

```
common
      . . . .
link
     contInterpreteur : service ( entiervaleur );

tty : service ( vide );

   var   activite : set of 1 .. NCONS ;

   console : array [no : 1 .. NCONS] of service(vide);
        var
        def lire : service (incons); begin <réalisation échange> end;
        def ecrire : service (outcons); begin <réalisation échange>end;
        begin
        allow (lire,ecrire):(lire!ecrire) until not(no in activite);
        end;

   def break : service (entiervaleur);    %valeur : integer %
        begin   % 1er break = activation console %
        if not (valeur in activite) then begin
                                          activite := activite + (valeur);
                                          exec console [valeur] ;
                                          end;
        transfer contInterpreteur ;  %informe l'interpreteur%
        end;

   def arret : service (entiervaleur);
        begin   % recoit les ordres de désactivation des consoles %
        activite := activite - (valeur);
        exec console [valeur ] .ecrire  ("bye");
        end;

   begin   % contrôle général %
   activite := () ;
   allow (console,break,arret) until false ; % boucle sans fin %
   end.
```

3. INTERPRETEUR DU LANGAGE DE COMMANDE

Ce D-module est l'interlocuteur normal des utilisateurs, il exécute les quelques commandes utilisateurs nécessaires pour le dialogue. On supposera que la plupart des commandes au système sont exécutées sous forme de programmes utilisateurs.
Nous ne traiterons donc ici que deux commandes particulières :

- la demande d'exécution d'un programme,
- la commande de fin de session.

La première précise le nom d'un fichier de cartes commandes et le type d'exécution : interactif ou background.
La seconde a pour résultat, la désactivation de la console de l'utilisateur.

Lorsque l'utilisateur lance l'exécution d'un programme interactif, c'est ce dernier qui va dialoguer avec l'utilisateur, le service qui gérait cet utilisateur se suspend donc jusqu'à la fin de l'exécution du programme.

Pendant une session un "break" signifie :

- arrêt de la commande en cours si on dialogue avec l'interpréteur,
- arrêt du programme en cours si on dialogue avec un programme interactif.

Structure

Elle est très proche de celle de tty : un service interpréteur pour chaque usager, un service recevant les breaks.

Structure de données : trois ensembles *1.. ncons : activité; breakreçu et suspendu* : la valeur i appartient à l'ensemble

activité : si un utilisateur est à la console i
suspendu : si cet utilisateur dialogue avec un programme
breakreçu : si cet utilisateur dialoguant avec l'interpréteur à émis un break pour interrompre une commande.

```
common
    . . . .
link
    arret : service (entiervaleur);
    monitor : service (ordrebatch);
    stopmonitor : service (entiervaleur);

conversation : service (vide);

    var activite,breakrecu,suspendu : set of 1 .. NCONS;

    interpreteur : array[nul..NCONS]of service(vide);
        var
       def finprog : service(vide) ; %recoit les msg de fin de programme%
          begin suspendu := suspendu - (nu) end;

       begin
       while nu in activite do
          begin
          %traitement des commandes avec:
          fin de session: exec arret (nu);
                          activite:=activité - (nu);
          exécution d'un programme (nomfich) interactif :
                          exec monitor (nomfich,nu,interactif);
                          suspendu := suspendu + (nu);
                          allow (finprog) until true; % une exécution%
          programme non intéractif:
                          exec monitor (nomfich,nu,background);
          end;
       end;

   continterpreteur : service (entiervaleur); % valeur : integer%
       begin
       if not (valeur in activite ) then  % début de session %
          begin activite := activite + (valeur);
          exec interpreteur [valeur ]
          end
       else if valeur in suspendu then transfer stopmonitor
            else breakrecu := breakrecu + (valeur);
       end;
```

```
begin     % programme de contrôle général %
activite := ( ); breakreçu := ( ); suspendu := ( );
allow (interpreteur, continterpreteur) until false
end.
```

4. GESTION DES DISQUES

Nous présentons ici les modules chargés des entrées-sorties sur disque. Chacun de ces modules est attaché à un disque et offre un seul service *accès* à ses utilisateurs.

Les paramètres à *accès* sont :

- un numéro de secteur absolu (les utilisateurs ignorent les notions de piste),
- un sens d'échange : lecture/écriture,
- un tampon de 256 caractères (1 secteur).

Pour optimiser les mouvements de la tête, on les gère suivant la méthode dite de l'ascenseur, c'est-à-dire que l'on classe les demandes par pistes accédées et que l'on balaye les pistes en épuisant pour chacune la liste des demandes en attente.

Cele se traduit dans l'écriture d'un module par la présence d'autant de services réalisant l'accès qu'il y a de pistes. Le service interface accès répartissant les demandes sur ces services.

Un tableau de compteurs : *àfaire*, mémorise le nombre de demandes par piste. Un compteur : *fait*, mémorise lui, les demandes réalisées sur la piste courante.

```
common
    . . . .
disque : array [nodisk : 0 .. NDISK-1] of  service (vide);   %d-modules%

    var
    fait,compteur,position,i:integer;
    afaire : array [0 .. NOPISTE-1]of integer;
    sens : (haut,bas);

    copiste : array [nop : 0 ..NOPISTE-1]of service (echangedisque);
         %paramètres: secteur, sens : (ecr,lire) , tampon %
         begin
         <réalisation de l'échange sur secteur dans la piste nop>
         fait := fait + 1;
         end;

    def acces : service (echangedisque);
         var  piste : integer;
         begin %transforme le numéro de secteur absolu en un numéro
                de piste et un numéro de secteur dans cette piste,
                transmet au service copiste concerné                 %
         piste := secteur div  NSECT ; secteur := secteur mod NSECT;
         afaire [piste ] :=  afaire [ piste ] +  1;
         compteur := compteur+ 1;
         transfer  copiste [piste ] ;
         end;

    begin  % contrôle %
    for i := 0 to  NOPISTE - 1 do afaire [i] := 0;
    position := -1; sens := haut ; fait := 0;
    repeat
       allow(acces) until true;     % attente 1ère demande %
       while compteur  ≠ 0 do
          begin   % recherche d'une piste %
          1: if sens = haut then for i:= position + 1 to NOPISTE-1 do
                                    if afaire [i]>0 then goto 2;
          sens := bas; %pas trouvé %
          for i := position downto 0 do
                                    if afaire[i]>0 then goto 2;
          sens := haut; goto 1;
```

```
        2: % ici on a forcément trouvé une piste %
        position := i;
         <positionnement physique de la tête >
        allow ( acces , copiste [position ])
              until afaire [position ] = fait ;
        afaire[position]:= afaire[position]- fait;
        compteur := compteur - fait ;
        fait :=  0;
        end;    %  while  %

    until false ;  % boucle sans fin %
end.
```

5. SYSTEME DE GESTION DE FICHIERS

Nous avons choisi un cadre simple :

- fichiers séquentiels,
- catalogue à un seul niveau,

pour ne pas rendre trop complexes nos algorithmes et privilégier l'étude de la structure du système.

5.1 - Principe

- Chaque disque peut porter jusqu'à NFICH fichiers.
- Un nom de fichier tient sur 16 caractères.
- Chaque disque contient :
  - . un catalogue des fichiers supportés (tableau de noms de fichiers un élément de ce tableau contient des blancs si le fichier d'indice i n'existe pas (adresse du catalogue : segment 0 et suivant),
  - . une suite de secteurs, où chaque secteur est le descripteur d'un fichier. Dans chacun de ces descripteurs on trouve : le nombre de secteurs du fichier, le nombre d'octets réellement utilisés du dernier et une liste de pointeurs sur les secteurs du fichier. (adresse des descripteurs : SECTDESC et suivant),
  - . une table d'occupation des secteurs du disque sous forme d'un bitmap des secteurs (adresse : BITMAP et BITMAP+1).
- A l'initialisation du système un catalogue général résidant est créé par concaténation des catalogues des disques.

5.2 - Structure du SGF

Les codes du système de gestion de fichier sont répartis en quatre types de modules :

- sgf : un seul module, conserve le catalogue général, sert d'interface pour l'utilisateur.
- sgfloc : un module par disque gère l'allocation de l'espace disque. Il gère globalement les fichiers et crée les modules de type fichier.

- fichier : un module de ce type est créé à la première ouverture d'un fichier et disparaît lorsque le dernier utilisateur ferme le fichier. Entre temps, il règle les accès au fichier suivant les règles des lecteurs-rédacteurs.

- lecture : un module de ce type est créé par les modules fichier pour chaque utilisateur en lecture.

En résumé :
Les modules sgf et sgfloc sont la partie statique du système de gestion de fichier.
On adapte ce système à la demande en créant un module fichier pour chaque fichier en cours d'utilisation. Ce module ordonne les accès, conserve le descripteur du fichier.
Puisque plusieurs utilisateurs peuvent ouvrir le fichier en lecture, on adapte encore une fois la charge en créant des modules lecture, ceux-ci conservent pour chaque utilisateur le contexte courant de sa lecture : pointeur courant etc.
Puisqu'il ne peut y avoir qu'une seule écriture à la fois, c'est le module fichier qui s'en charge directement.

5.3 - Ouverture de fichier

Une demande d'ouverture de fichier est adressée à sgf, qui en vérifie la validité et la transfère au sgfloc concerné.
Celui-ci, si le fichier n'est pas encore utilisé crée un module fichier et lui transfère la demande; si ce module existe la demande lui est simplement transférée.

Toute ouverture se fait donc via sgf, en résultat de cette ouverture on donne à l'utilisateur les noms des services d'accès et de fermeture du fichier, auxquels il s'adressera par la suite directement.

Paramètre *d'ouvrir* :

- *nomfich* : nom du fichier
- *typdem* : *(créer, lec, ecr, ajout)* type d'accès demandé
  créer : on crée un nouveau fichier
  lec : lecture
  ecr : écriture avec écrasement du contenu ancien
  ajout : écriture en séquence du contenu ancien

- *sgf* : code de retour (erreurs, etc)
  entre *sgf* et *sgfloc* : numéro de fichier local
- *saccès* et *sfermer* : identifications des services d'accès et de fermeture rendus en résultat *d'ouvrir*.

```
common
      . . . .
link  disque : array [0 .. NDISK - 1] of service   (vide)
                    ( acces : service (echangedisque ) );
    sgfloc : array [0 .. NODISK-1 ] of service (vide)
               ( ouvrir : service (msgfournir);
                 detruire : service ( msgdetruire )  );

sgf : service ( vide );

    var i,j,k : integer ; nsectcat : integer;
    catalogue : array [0 ..  NFICH * NODISK ] of  ch16 ;
    tampon : array [0 .. 15 ] of  ch16; % tampon intermédiaire pour
                                          initialisation    %

    def ouvrir : service  ( msgouvrir );
       begin  %  recherche fichier %
       i := 0;  j := -1 ;
       while i < NFICH * NODISK  do
           if catalogue [i] = nomfich then begin
                                          j:=i; i := NFICH * NODISK
                                          end
           else i := i + 1 ;
       if ( j=-1 ) & ( typdem ≠ creer ) then code := FICHIERINCONNU
       else if ( j ≠ -1 ) & ( typdem = creer ) then code := FICHIEREXISTE
       else begin  % demande correcte %
            if typdem = creer then begin
                                   j := <recherche d'une case libre> ;
                                   catalogue [j] := nomfich ;
                                   end;
            code := j mod NFICH ;    % numéro local %
            transfer sgfloc [j  div  NFICH ].ouvrir
            end;
       end;  % ouvrir %

    def detruire : service (msgdetruire); %nomfich:ch16;  code:integer %
       begin
       j :=<recherche du fichier>;
       code := j mod NFICH ;  % numéro local %
       transfer sgfloc [j  div  NFICH ].detruire;
       catalogue [ j ] :=  '    ';  % annulle l'entrée catalogue %
       end; %detruire%
```

```
begin    % *** contrôle *** %
%lecture du catalogue %
nsectcat := ( NFICH x  16) div  256; % nbre de secteurs par catalogue
                                       sur chaque disque            %
for i:= 0 to (NODISK x nsectcat ) - 1 do
     begin
     disque[i div nsectcat] . acces (          i mod nsectcat,lec,tampon);
             % = lecture d'un secteur de catalogue local %
     for j:= 0 to 15 do catalogue [ i x 16 + j ] := tampon [ j ] ;
     end;

allow (ouvrir,detruire) : (ouvrir ! detruire) until false ;

end.    % du module %
```

Description de sgfloc

services : *alloc*, *desalloc* : allocation libération de segments;

signalfin : invoqué par les services fichiers à chaque fermeture pour savoir s'il reste des utilisateurs;

détruire : destruction d'un fichier, libère ses segments. La distinction est différée si le fichier est en cours d'utilisation;

ouvrir : ouverture d'un fichier, création si c'est la première ouverture d'un module fichier.

structures de données

bitmap : table (implantée sous forme d'ensemble) d'occupation des segments du disque

compteur, *serviceouvrir*, *àdétruire* : chaque élément de ces est associé à un fichier et mémorise :

- le nombre d'utilisateurs courants,
- le service ouvrir du module fichier concerné
- une éventuelle destruction à effectuer lorsqu'il n'y aura plus d'utilisateurs.

```
common
      . . . .
link
     disque : array[0 .. NODISK -1] of service ( vide )
                ( acces : service ( echangedisque ) ) ;

sgfloc : array [nodisque : 0 .. NODISK ] of service  ( vide );

   var
   compteur : array [0  .. 127] of integer ;
   serviceouvrir : array[0 .. 127]of msgouvrir;
   adetruire : array[0 .. 127]of boolean ;
   bitmap : set of[0 .. 4095];

   alloc : service ( entierresultat );  % value result resultat :integer %

      var i , j : integer ;
      begin
      i := 0 ; resultat := -1 ;
      while i<4096 do  % recherche d'un bloc libre %
           if i in bitmap then begin resultat := i ;
                               bitmap := bitmap + (i) ;
                               i := 4096
                               end;
      end;

   desalloc : service ( entiervaleur ); % valeur : integer %

      begin  bitmap := bitmap + (valeur ) end ;

   detruire : service ( msgdetruire ); % nomfich , code %

      begin
      if compteur [code ] ≠ 0 then %fichier utilisé destruction différée %
          begin adetruire[code ] := true ; code := 0 end
      else <lecture du descripteur, libération des secteurs> ;
      code :=  0;
      end;
```

```
signalfin : service ( entierresultat ); % val. res. resultat : integer %

     begin
     compteur[resultat] := compteur[resultat] - 1;
     if compteur[resultat] = 0 then  % c'était le dernier utilisateur %
          if adetruire[resultat]then resultat := -1
          else resultat := 0
     else resultat :=  1;
     end; %signalfin %

ouvrir : service ( msgouvrir );

     type
          fichier = service (msgfichier );
           (voir description plus loin)

     begin
     compteur [code] :=  compteur [code] + 1;
     if compteur [code] =  1 then  %première ouverture%
         begin
         create fichier ( nodisque , code )
                serviceouvrir [code] :=  ouvrir
                endcreate;   % on mémorise le service ouvrir du
                               module   %
         adetruire [fichier] := false;
         end;
     transfer   serviceouvrir [code] ;
     end;  %  ouvrir  %

begin   %  pgm  de  contrôle  %
<initialisation de bitmap>
for  i := 0 to 127 do compteur [i] := 0 ;
allow (alloc,desalloc,detruire,ouvrir,signalfin)
   : ( alloc ! desalloc),(ouvrir!signalfin!detruire) until false;
end.
```

Description de fichier

initialisation : On lit le descripteur du fichier

contrôle : On ordonne les demandes suivant la méthode des lecteurs rédacteurs,
Le programme de contrôle se termine lorsque *fin* est positionné (dans *fermer*).

ouvrir : En lecture : crée un module nouveau à partir du modèle *unitélecture* (lui passe une copie du descripteur)
En écriture : suivant le type d'écriture met à jour le descripteur (éventuellement libère les blocs anciens si on écrase l'ancien contenu).
Dans tous les cas rend à l'utilisateur les noms des service d'accès et de fermeture concernés.

fermer : A la fin d'une écriture : copie et met à jour le descripteu
Dans tous les cas informe le *sgfloc* du disque de la fin de cet accès.
Suivant la réponse de *sgfloc* :

- plus d'utilisateurs : fin est positionné et le fichier éventuellement détruit si une demande de destruction le concernant a été faite,
- il y a encore des utilisateurs : on continue.

```
fichier = service ( msgfichier );  % unite : integer nofich:integer %

     var
     demecr,fin : boolean;
     descrip : int128;       % descripteur du fichier %
     i,depcour,sectcour : integer ;
     tampon : ch256;  % depcour, sectcour, tampon sont le contexte d'une
                        utilisation du fichier en écriture   %


     fermer : service ( msgfermer );

          begin
          if demecr then  % on était en écriture  %
               begin
               demecr := false;
               <mise a jour et sauvegarde du descripteur du fichier>
               end
          else nblec := nblec - 1;
          % on informe le sgfloc de la fermeture,si le rèsultat est
            négatif ou nul il n'y a plus d'utilisateur, (négatif :
            fichier à detruire) %
          i := nofich;  sgfloc[unité].signalfin(i);
          if i <= 0 then
               begin fin := true;
               if i = -1 then < destruction fichier=libération des
                                blocs>
               end;

     ecrire : service (msgacces);
          .
          .  réalisation écriture
          .

     ouvrir : service ( msgouvrir );

          type
```

```
unitelecture = service ( lancelec );
          % D_module créé à chaque ouverture en lecture %
     var
     fini : boolean;
               .../...

     lecture : service (msgacces);
          .
          .   réalisation de la lecture
          .
     fermlec : service (msgfermer);
          begin fini := true; transfer fermer end;
               % = informe le service fermer du module
                 créateur  %
     begin
     fini := false ;
     allow (lecture,fermlec) until fini;
     end;

var
i:integer;

begin  % corps d'ouvrir %
if typdem = lec then
     begin create  unitelecture (descrip,unite)
           sacces := lecture; sfermer := fermlec
           endcreate;
     nblec:=nblec+1
     end
else begin
     demecr := true;
     sacces := ecrire; sfermer := fermer ;
     if typdem = ajout then
          begin
          <positionner les variables  sectcour,depcour
           tampon pour décrire la fin actuelle du fichier>
          end
     else begin
          <libérer tous les blocs du fichier et positionner
           les variables pour une nouvelle écriture
            l'ancien contenu est donc détruit >
          end;
end  end;
```

```
begin   %  *** contrôle général *** %
demecr := false ; nblec := 0;
%lecture descripteur %
disque [unité]. acces ( SECTDESC + nofich, lec, descrip );
repeat
   allow ( ouvrir ) until true;  % attente première demande %
   if nblec > 0 then   % c'était une demande de lecture  %
        begin
        allow ( fermer,ouvrir):(fermer!ouvrir)  until demecr ! fin;
        if nblec ≠ 0 then allow (fermer) until nblec = 0;
        end;
   if not fin then allow (ecrire,fermer) until not demecr;
until fin
end.
```

6. EXECUTION DE PROGRAMMES

Nous nous limitons pour cette partie à donner les grands traits des modules qui la composent.
Leurs principes sont les suivants :

- Un module moniteur reçoit les demandes d'exécution. Lorsque les conditions s'y prêtent (taille mémoire par exemple), il lance l'exécution du programme en créant un D-module guide. Celui-ci lit le fichier de cartes contrôles dont le nom a été fourni par l'utilisateur.

- Un fichier de cartes contrôles est constitué d'un nombre quelconque de séquences (steps). Chaque séquence est composée d'affectations de fichiers suivies d'une demande d'exécution d'un programme.

- Le module *guide* mémorise les affectations, charge le programme puis lance son exécution. Chaque fois que le programme ouvre un fichier, il adresse une requête à son guide, qui la traduit en termes de fichier réel et la transmet à *sgf*.

- Lorsque le programme se termine il envoie une requête à *guide* qui traite éventuellement la séquence suivante.

- Un fichier peut être affecté : à un fichier réel, au terminal de l'utilisateur, à l'imprimante. Dans ce dernier cas, le module *guide* crée des fichiers temporaires qu'il fera imprimer à la fin du travail.

- A la fin du travail, on avertit l'interpréteur si le programme était interactif, on imprime les fichiers de spool après avoir réservé l'imprimante, puis le module *guide* prévient le moniteur et disparaît.

Cette partie du système nécessiterait, si on voulait la spécifier complètement, de nombreuses hypothèses sur la façon dont on gère la mémoire, sur la façon dont on y charge des programmes etc.
Nous n'avons pas voulu non plus la passer entièrement sous silence car la notion de module guide nous paraît intéressante. On pourrait l'étendre à la récupération des erreurs du programme etc.
Nous avons donc utilisé un compromis en n'en donnant que les grandes lignes.

```
execution : service ( vide );     %     d-module  %


   monitor : service (ordrebatch);

       .
       . allocation d'une zone mémoire pour le programme
       . création d'un module guide
       .

   finexec : service ( ... );

       .
       . libération de la zone mémoire du programme qui se termine
       .

   begin   %  contrôle  %
   boucle : |libérer ( monitor,finexec ) jusqu'à épuisement des
            |                                          ressources
            |libérer finexec  jusqu'à restitution d'un nombre suffisant
                                                       de ressources

   end.
```

Nous n'avons pas traité ici le traitement d'une demande d'abandon par l'utilisateur par manque d'hypothèses.

```
guide : service ( ... );

   finprog : service ( .... );
       .
       . appelé par le programme à la fin de son exécution;
       .
   ouvrir : service (msgouvrir );
       .
       . appelé avec un nom de fichier local par le programme.
       . recherche du nom local dans la table des affectations de fichiers
       . si c'est un fichier réel transmettre à sgf avec le nom réel
       . si c'est le terminal: rendre à l'utilisateur le nom du service
       .          concerné du module tty.
       . si c'est l'imprimante, transformer la requête en une requête
       .          de création de fichier que l'on gèrera comme un
       .          fichier temporaire.
       .
   begin   %   pgm  de  contrôle  %
   boucle :  lire les cartes d'affectation de fichier, les mémoriser
                  dans une table,
             lire la carte nommant le programme à exécuter, le charger
                  et le lancer,
             libérer finprog pour attendre la fin du programme
             on sort de la boucle quand toutes les commandes sont
             exécutées.
   lorsqu'on a exécuté les commandes:
        s'il y a des fichiers à imprimer :
             réserver  l'imprimante
             imprimer et détruire les fichiers.
             libérer l'imprimante.
        prévenir - l'interpréteur dans le cas d'un programme interactif
                 - le moniteur
   end.    %  le module disparait   %
```

CONCLUSION

Nous n'avons pu décrire dans cette thèse qu'un état intermédiaire de notre projet ou plutôt du langage de programmation que nous voulons proposer. Aussi, tout naturellement cette conclusion tentera-t-elle de faire le point sur le langage SORTILEGE et d'en esquisser les développements, les aménagements possibles. Il ne s'agit pas, nous le verrons, de modifications fondamentales mais d'une poursuite de notre travail. Cela justifie, nous l'espérons, l'écriture de cette thèse alors même que le projet n'est pas terminé.

Avant d'aborder ces extensions, nous allons tenter d'envisager les utilisations des D-modules puis de revenir sur les caractéristiques que nous désirions donner au langage pour discerner ce qui est atteint et ce qui ne l'est pas. Cela nous permettra de mieux préciser les apports futurs envisagés et de les justifier.

Utilisation particulière du langage

Le langage que nous avons présenté tout au long de cette thèse est le produit du projet SORTILEGE et a été conçu pour lui. Il suppose donc un certain noyau d'exécution et est conçu en fonction des primitives offertes par ce noyau.

Nous avons plutôt conçu ce langage pour améliorer l'accès au système que comme un langage général d'écriture. En ce sens il est vraisemblable que son transport sur des systèmes généraux existants poserait de nombreux problèmes. Peu de ces systèmes permettent à la fois à l'utilisateur :

- de créer de nombreux processus,
- de leur faire partager des données,
- d'assurer leur contrôle et la communication nécessaire.

Par exemple, un transfert sous le système BOS-D de la SEMS nous semble difficile, sous OS 360 ou SIRIS 8 il est presque impossible.

Pour ces raisons, les d-modules ne se posent pas en concurrents de PASCAL ou ALGOL 68 ou de tout autre langage général. Cependant, il est prévisible que l'apparition de langages tels ADA, que l'évolution des principes d'écriture de systèmes feront apparaître de nouveaux systèmes, plus ouverts, plus souples. Il est également possible d'implanter nos primitives sur des systèmes offrant la notion de machine virtuelle. Dans tous ces cas, une implémentation de notre langage sur ces systèmes serait alors possible et pourrait être utile dans les domaines suivants :

- production et mise au point de programmes destinés à être exécutés sur des réseaux spécialisés. Notre choix de portabilité permet en effet d'utiliser une machine de production, plus puissante et disposant de plus d'outils de mise au point que le réseau cible;
- simulation : il nous semble que notre langage peut être utilisé dans des applications de simulation, plus particulièrement dans la simulation d'architecture multi-processeurs, multi-calculateurs;
- enfin, et en général toutes les applications propices à des exécution parallèles : compilation, temps réel, calcul scientifique, etc...

Autre utilisation possible (en dehors des réseaux spécialisés) de la notion de d-module : les architectures multi-microprocesseurs spécialisées. La structure d'un d-module : variables partagées + ensemble de processus disposant chacun de données propres, rappelle certaines architectures où plusieurs processeurs partagent un espace de mémoire commune et disposent chacun d'un espace propre.

Il est hasardeux de tirer trop de conclusions de cette similitude, mais il pourrait être intéressant d'en étudier les implications :

- dans le domaine de la simulation,
- dans la réalisation de calculateurs spécialisés (contrôle de processus par exemple) où certaines primitives du langage pourraient être câblées.

Intérêt des d-modules dans l'écriture de systèmes

Nous revenons là au but initial des d-modules : l'écriture d'applications système réparties et la portabilité de ces applications.
Nous avons justifié dans notre présentation ce souci de portabilité par l'existence d'un nombre croissant de réseaux. Souvent constitués de calculateurs du même type, ces réseaux ne se différencient que par le nombre et la configuration des sites. Créer des applications portables d'un réseau à l'autre ne peut que faciliter la bonne utilisation de ces architectures et réduire les coûts de développement. Cela facilite également, et ce n'est pas négligeable, les variations de configuration. La notion de site n'apparaissant pas dans les d-modules, on peut si les ressources sont suffisantes faire tourner une même application sur 1, 2... N machines. Cela présente deux avantages importants :

- la souplesse des applications : on rend possible des modifications de configuration (ajout d'un nouveau noeud, retrait d'un site en cas de panne) sans modification des applications;
- compatibilité totale des logiciels entre une machine X isolée et un réseau constitué de machines X. Ce point nous paraît particulièrement important car rares sont les systèmes existants qui offrent une compatibilité entre configuration centralisée et configuration réseau.

Autres avantages importants que notre langage partage avec tous les langages modulaires :

- la possibilité de modifier un module sans modifier les autres modules de l'application, le réemploi facile des modules dans d'autres applications. On minimise ainsi les coûts du logiciel;
- une bonne adéquation à la notion de mémoire segmentée. Cette notion devient de plus en plus courante parmi les mini ou micro-calculateurs, il est intéressant de disposer de concepts logiciels adaptés. Deux processus peuvent partager le même espace, ou une partie de leur espace ou être totalement indépendants. Le matériel apporte donc une protection qui n'était souvent jusqu'ici que logicielle et qui s'adapte bien à la structure des d-modules et à leurs besoins de protection.

Extensions prévues ou souhaitables

Les compléments nécessaires pour faire de notre langage un outil d'écriture de systèmes concernent surtout les relations entre une application et ce qui lui est extérieur : périphérie, système et matériel support, autres applications...

Les points qui nous semblent mériter le plus d'intérêt et le plus d'étude sont :

- l'expression du contrôle global de l'application. Ce point offre de multiples aspects, par exemple la forme donnée à l'expression de ce contrôle. Nous avons ménagé dans la définition des d-modules une compatibilité vers le haut en faisant de chaque d-module un service de l'application. Il reste à définir comment l'on peut utiliser cette compatibilité pour exprimer le contrôle : "d-module application", contrôle limité au lancement de l'application (et alors quelles sont ses fonctions ?) ou valable durant l'exécution...

Autre aspect du contrôle : comment et sur quels critères sont implan les modules, cette implantation est-elle fixe ou variable dans le temps ? Quels sont les moyens de détection, de traitement des erreurs à quel niveau doivent-elles être traitées ?

- outils d'écriture et de conservation des programmes. La notion de modèle de module, d'en-tête de déclaration des types communs peuvent être lourds à manipuler et pourraient justifier l'introduction d'outils du type des environs de SESAME ou de l'INCLUDE de PL1. Dans ce ca un bibliothécaire faciliterait le travail du programmeur;
- enfin et c'est l'un des traits principaux d'un langage d'écriture de systèmes, il sera nécessaire de formaliser les notions que manipule un système : périphériques, interruptions, priorités etc. C'est-à-dire définir l'interface entre le logiciel et le matériel qui lui sert de support. On pourra par exemple introduire des moyens de nommer les interruptions et de définir quels sont les services destinés à les traiter. Ce rapprochement pourrait être explicité dans le code (comme dans ADA) ou par un langage externe.. (comme lors d'une affectation de fichier).

Il manque donc beaucoup d'éléments à ce langage, en ce qui concerne ses trait déjà définis rappelons que nous avons tenté de constituer un langage offrant un bon rapprochement entre requêtes et code, une expression claire des règles de synchronisation entre processus, et enfin un outil souple pour l'écriture d'applications elles-mêmes souples.

Nous espérons que la structure des d-modules et les primitives qu'ils offrent remplissent ces conditions.

BIBLIOGRAPHIE

ALTABER 77 — ALTABER, FRAMMERY, GAREYTE, JEANNERET, VAN DER STOCK
Contrôle en temps réel avec architecture distribuée.
Journées BIGRE, IRIA (Nov.1977).

ANDRE 78 — E. ANDRE, P. DECITRE
On providing distributed applications programmers with control over synchronization. Computer Network Protocols Symposium, Liège (Feb.1978).
Présentation de MER. Projet POLYPHEME. Centre Scientifique CII-HB. Université de Grenoble.

BARBER 76 — D.L.A BARBER
An european information network achievement and prospects.
3rd Conference Computer Communications (Aug.1976).

BEKKERS — Y. BEKKERS
A Comparison of two high level synchronizing concepts.
Dept. of Computer Science. The Queen University of Belfast.

BRIAT — J. BRIAT, X. ROUSSET DE PINA, J.P. VERJUS
Le projet OURS : ses principes. Congrès AFCET, Paris 1976.

BRINCH 77 — P. BRINCH HANSEN
Distributed Processes : a concurrent programming concept,
Computer Science Dept. University Southern California (1977).

CHEVAL 76 — J.L. CHEVAL, F. CRISTIAN, S. KRAKOWIAK, M. LUCAS,
J. MONTUELLE, J. MOSSIERE
Un système d'aide à l'écriture des systèmes d'exploitation,
Congrès AFCET Paris 1976.

CHEVAL 79 — J.L. CHEVAL, C. CLEMENCET, J.M. GUILLOT, P. LAFORGUE,
J. RAYMOND, S. ROUVEYROL
Un projet de système distribué : SORTILEGE. RR.150,
Université de Grenoble, janvier 1979.

CLEMENCET 79 — C. CLEMENCET, S. KRAKOWIAK
Structuration et contrôle de l'exécution dans les systèmes informatiques répartis. Journées BIGRE, Nancy 1979.

DANG 77 — M. DANG, G. SERGEANT
Expression of parallelism and communication in distributed network processing, Int. Conf. on Parallel Processing, Wayne State University USA (Aug.1077).

DIJKSTRA 75 — E.W.D DIJKSTRA
Guarded Commands, nondeterminacy and formal derivation of programs. C.ACM 18,8, pp.453-457 (Aug.1979).

DIJKSTRA 76 — E.N.D DIJKSTRA
A discipline of programming, Prentice Hall (1976).

ESTUBLIER 78 — ESTUBLIER
Processus cycliques et appel procédural. Thèse de 3e cycle, Grenoble 1978.

FARBER 72 — FARBER, LARSON
The system architecture of the distributed computing systems. Proc. of the Symposium on Computer Communication Network and Teletraffic, New-York (1972).

FELDMAN 76 — J.A. FELDMAN
A programming methodology for distributed computing (among other things). TR9, Computer Science Dept. Univ Rochester (Nov.1976).

FELDMAN 77a — J.A. FELDMAN
High level language constructs for distributed computing, Proc. 5th III Conf. (Implementation and Design of Programming Languages), Guidel (May 1977).

FELDMAN 77b — J.A. FELDMAN
Synchronizing distant cooperating processes, TR 26, Computer Science, Dept. Univ. Rochester (oct.1977).

GUILLOT 79 — J.M. GUILLOT
A synchronizing mechanism for distributed applications. IMAG RR.158, Université de Grenoble, février 1979.

GREEN 78/ADA — Rationale for the design of the GREEN programming language, a language designed in accordance with the Ironman requirements (Feb.1978).

GREEN programming language. Preliminary reference manual. Interim Draft. October 15, 1978. Honeywell Inc, CII-Honeywell Bull.

HABERMAN 75 — A.N. HABERMANN
Path expressions, Carnegie Mellon University (1975).

HEART 73 — F.E. HEART, S.M. ORNSTEIN, W.R. GROTHER, W.B. BARBER
A new minicomputer/multiprocessor for the ARPA network.
AFIPS 73, NCC Vol 12.

HELMS 78 — H.J. HELMS
Organization and technical problems of the European informatic Network. Congrès NATO Bonas (Aug.1978).

HOARE 74 — C.A.R HOARE
Monitors : an operating systems structuring concept, Comm. ACM 17, 10 (Oct.1974).

HOARE 78 — C.A.R HOARE
Communicating Sequential Processes. Comm. ACM, (Aug.1978).

KAISER 78 — C. KAISER, M. KRONENTAL, J. LANGET, S. NATKIN, S. PALASSIN
Système informatique réparti pour la conduite d'un atelier mécanique. Congrès AFCET Paris (1978).

KRAKOWIAK 78 — S. KRAKOWIAK, J. MOSSIERE
Problèmes et perspectives de l'enseignement des systèmes d'exploitation. Journées Informatiques, NICE, 1978.

METCALFE 76 — METCALFE, BOGGS
ETHERNET : Distributed packet switching for local computer networks (1976), CACM Vol. 19 n°7.

MOSSIERE 77 — J. MOSSIERE
Méthodes pour l'écriture des systèmes d'exploitation. Thèse d'Etat. Grenoble 1977.

PACINI — G. PACINI
Process Coordination and Scheduling. Instituto di Scienze dell'Informazione. Universita degli Studi di Pisa. Italia.

POUZIN 77 — L. POUZIN
Les réseaux informatiques. Séminaire d'informatique SI n°1, Laboratoire d'Informatique IMAG, Grenoble (Mar.1977).

ROBERT 77 — P. ROBERT, J.P. VERJUS
Towards autonomous description of synchronization modules, Proc. IFIP Conf.(1977).

ROBERTS 70 L.G. ROBERTS, B.D. WESSLER
Computer netwroks development to achieve ressource sharing. AFIPS Conference Proceedings, Sprint joint Computer Conference Montvale (1970).

SWAN 77 SWAN, FULLER, SIEWIOREK
SM* : A modular multi-microprocessor. AFIPS NRC (1977).