Université de NANCY I

U.E.R. sciences mathématiques

Centre de Recherche en Informatique de Nancy

C.R.I.N.

FLEXI: LANGAGE DE CONCEPTION D'APPLICATION DE CONDUITE DE PROCÉDÉS INDUSTRIELS

Contribution à l'Etude de la Communication entre Processus Coopérants

THESE

soutenue publiquement le **2 mars 1984**

À L'UNIVERSITÉ DE NANCY I

pour l'obtention du grade de

DOCTEUR DE 3ème CYCLE EN INFORMATIQUE

par

Abdelouahed ZAKARI

devant la Commission d'Examen :

Président : ..... *Jean-Claude DERNIAME*

Examinateurs : ..... *Jean-Pierre FINANCE*

Alain HAURAT

Jacques LONCHAMP

Guy-René PERRIN

Jean-Pierre THOMESSE

Université de NANCY I — U.E.R. sciences mathématiques

Centre de Recherche en Informatique de Nancy
C.R.I.N.

FLEXI: LANGAGE DE CONCEPTION D'APPLICATION DE CONDUITE DE PROCÉDÉS INDUSTRIELS

Contribution à l'Etude de la Communication entre Processus Coopérants

THESE

soutenue publiquement le **2 mars 1984**

À L'UNIVERSITÉ DE NANCY I

pour l'obtention du grade de
DOCTEUR DE 3ème CYCLE EN INFORMATIQUE
par
Abdelouahed ZAKARI
devant la Commission d'Examen :

Président : ..... *Jean-Claude DERNIAME*
Examinateurs : ..... *Jean-Pierre FINANCE*
Alain HAURAT
Jacques LONCHAMP
Guy-René PERRIN
Jean-Pierre THOMESSE

Je voudrais remercier tous ceux qui ont contribué directement ou non à la réalisation de ce travail.

J'exprime ici ma gratitude à Monsieur Jean Claude DERNIAME, Professeur à l'Université de Nancy I, Directeur du CRIN, pour avoir dirigé ce travail. Je le remercie vivement pour l'honneur qu'il m'a fait en présidant ce Jury, et pour tout ce qu'il m'a apporté au cours de mes études et mes recherches, et particulièrement dans mes débuts de chercheur stagiaire à la Régie RENAULT.

Je remercie Monsieur le Professeur Jean-Pierre FINANCE, pour les remarques et suggestions qu'il m'a adressées ; elles m'ont permis de compléter la présentation de ce travail.

J'adresse mes remerciements à Monsieur le Professeur Alain HAURAT, pour avoir trouvé le temps de venir participer à ce Jury, et s'être intéressé à mon travail.

J'exprime mes remerciements à Monsieur Jacques LONCHAMP, pour son attention et ses critiques à l'égard de mon travail, et les discussions fructueuses que nous avons eues.

J'exprime mes remerciements à Monsieur Guy-René PERRIN, pour ses critiques et suggestions, et pour les discussions enrichissantes qui m'ont permis de situer mon travail par rapport à divers niveaux de réflexion sur le parallélisme et la communication.

Je remercie vivement, Monsieur le Professeur Jean-Pierre THOMESSE ; il était à l'origine de mon travail, et l'a suivi depuis son début ; je lui suis reconnaissant pour son aide et son encouragement.

J'exprime en second lieu, mes remerciements à tous les camarades et amis de l'ATP Parallélisme.

Enfin, j'exprime mes plus vifs remerciements à Madame Danielle MARCHAND, pour la réalisation matérielle de cette thèse, et notamment pour tout le soin, la patience et la gentillesse qu'elle y a apportés.

=-=-=-=-=-

Sommaire

TABLES des MATIERES

CHAPITRE I

NTRODUCTION

I - 1 Le problème de construction d'applications temps réel réparties.

I - 2 Objectifs.

I - 3 Hypothèses.

I - 4 Présentation des travaux.

I - 1 LE PROBLEME DE CONSTRUCTION D'APPLICATIONS TEMPS REEL REPARTIES

Le domaine de traitement de procédés industriels se distingue par des caractéristiques pouvant avoir des répercussions sur l'architecture des systèmes informatiques proposés dans le cadre d'un système automatisé de commande/contrôle d'un ensemble d'installations données. Les applications dans ce domaine (systèmes logiques de commande) sont réalisées par des programmes de grandes tailles.

Le parallélisme exprimé dans ces programmes est induit par le parallélisme imposé par l'environnement (il s'agit de parallélisme réel dit aussi parallélisme de situation). En effet, le système de commande doit prendre en compte des données en entrée, provenant de procédés physiquement indépendants. Pour certaines entrées, il doit, en plus, répondre au bout d'un temps fini, relativement court (par l'envoi de données en sortie). Ceci en raison de contraintes fixées par le cahier des charges (elles sont appelées : contraintes temps réel). Les procédés commandés sont autonomes et opèrent en parfait asynchronisme vis à vis d'autres dans la même installation industrielle. Ceci nous amènent à considérer l'asynchronisme comme une caractéristique fondamentale des systèmes logiques de conduite de procédés.

Enfin, la répartition de ces systèmes peut être un choix pour mieux répondre à des contraintes imposées par ces installations, surtout dans le cas où elles sont géographiquement distribuées.

Le savoir-faire en matière de conception de logiciels, dans ce domaine, est encore fragmentaire. Les difficultés rencontrés sont nombreuses, car il faut tenir compte :

a- des problèmes attachés au type d'applications (relatifs aux caractéristiques ci-dessus) rendant difficile le passage de l'énoncé à la solution ;

b - des problèmes associés aux moyens utilisés (proposés), pour mettre l'application en œuvre, sachant qu'une solution a des qualités attendues (par exemple : structurée, évolutive).

On doit tenir compte des contraintes, ci-dessus, dans la nature des outils employés (par exemple ceux de communication).

Pour situer nos propos : considérons que les activités de construction de programmes peuvent être partagées en deux étapes majeures :

- la première part du cahier des charges (l'énoncé du problème) pour arriver à un ensemble de composantes formulant la solution (par exemple ces composantes sont reliées entre elles sous forme hiérarchique) ;

- la seconde part d'une telle description pour arriver à un ensemble de programmes écrits dans un langage donné sur une (des) machine (s) cible (s).

Concernant le point b ; on peut repousser le choix d'une forme hiérarchique (car elle va à l'encontre de l'asynchronisme) pour prendre celle en réseau ; on peut repousser le choix d'écriture de la communication à l'aide d'outils de bas niveau (ils vont à l'encontre de l'asynchronisme et de l'expression d'une solution structurée).

Concernant la première étape, nous énonçons quelques éléments d'aide à la décomposition d'une application basée sur le concept de communication.

Nos propositions se situent à la deuxième étape où nous donnons une démarche de conception modulaire, et par niveaux d'abstraction dans

laquelle s'insère le langage de description proposé. Il permet aussi bien le passage à la simulation qu'à la réalisation.

I - 2 OBJECTIFS

Nos objectifs peuvent être classés en deux catégories :

1 - comme l'application s'exprime par un réseau de composantes, la communication intervient dans la construction de cette application ; nous constatons alors que :

- d'une part, il faut voir dans le "concept communication" un aspect constructif et structurant par lui-même (c'est l'objet des chapitres VII et V dans un ordre chronologique d'activités de construction d'application) ;

- d'autre part, la communication étant un outil nécessaire dans la classe d'application considérée, il est important (relativement au point b ci-dessus) d'offrir à l'utilisateur des moyens d'expression des communications indépendantes de l'écriture des composantes elles-mêmes.

2 - Tenant compte des caractéristiques énoncées , nous proposons une démarche de construction de programmes, basée sur l'abstraction et la modularité ; elle s'appuie sur un langage de description appelé FLEXI. Les outils de communication proposés expriment des formes de communication prédéfinies. Ceci évite tout travail fastidieux de réécriture des communications dans un langage de programmation donné, et respecte les besoins de communication particuliers dans cette classe d'applications (par exemples : communication asynchrone, communication de

diffusion de messages, cf. chapitre IV). Le langage permet une conception des applications temps réel réparties, sans tenir compte à priori des contraintes physiques de répartition. Nous proposons aussi, à l'aide de ce langage d'établir le plus de contrôles statiques possibles.

I - 3 HYPOTHESES

Nous ne faisons des hypothèses qu'en ce qui concerne l'aspect fonctionnel des différents niveaux intervenant dans l'architecture du système de commande (centralisé ou réparti). Quant à l'aspect matériel, nous ne faisons aucune hypothèse sur la nature des machines (calculateurs) ou du réseau choisi (exemples : bus, anneau...). Le délai de transmission de message est supposé acceptable et le fonctionnement global du réseau est supposé non affecté par l'adjonction ou le retrait des sites. Au niveau application, la réception d'un message est perçue comme : "le message est reçu et il est en bon état", ou "le message est non reçu" (ou exclusif). La détection d'erreurs, retransmission et envoi d'accusé de réception, sont assurés par le système de communication ; dans ce cadre, des mécanismes de "récupération" d'erreurs au niveau application sont prévus (cf. chapitre VI).

Notre but, ici, n'est pas de définir un noyau ou un langage temps réel [SCE 80], [THO 80], mais d'utiliser ce qui existe sur le processeur disponible. Pour cette raison, nous ne faisons aucune hypothèse sur la nature du langage de programmation hormis l'accès de celui-ci au noyau.

I - 4 PRESENTATION DES TRAVAUX

Ils sont présentés en deux parties ; la première concerne l'intérêt qu'a suscité la communication dans divers travaux, aussi bien dans le domaine de spécification que celui de programmation, la deuxième concerne nos propositions.

Nous présentons dans la première partie :

- notre point de vue sur la communication telle qu'elle est apparue dans les langages ainsi que ses diverses expressions (cf. chapitre II) ;
- quelques travaux prenant en compte la communication dès le niveau de spécification (cf. chapitre III).

Nous présentons dans la deuxième partie nos propositions :

- il s'agit d'abord de donner les grandes lignes d'une approche adaptée à la classe d'applications choisie (cf. chapitre IV) ;
- ensuite, nous exposons nos propositions aux niveaux spécification et programmation (cf. chapitre V et VI) ;
- finalement, nous donnons des éléments d'aide à la décomposition de l'application sous forme d'un réseau de composantes communicantes (cf. chapitre VII). Cette étape se place chronologiquement avant celles exposées aux chapitres V et VI. Par le terme "spécification", nous entendons une résolution de problèmes à un niveau, ne prenant pas en compte des contraintes de langages de programmation, ou celles d'une machine cible particulière. Cette résolution peut être faite aussi bien par une démarche pragmatique [LEV 78] ; (c'est le cas de notre approche dans la deuxième partie) que par une démarche suivant l'une des trois lignes de recherches fondamentales citées dans [PAI 78] :

a - celle développée, en suivant un point de vue méthodologique (les travaux de HOARE, DIJKSTRA ... [HOA 74], [DIJ 75] ;

b - celle concernant les structures de données et types abstraits (les travaux de PARNAS, LISKOV, GUTTAG ... [PAR 72], [LIS 74], [GUT 77]) ;

c - celle sur la théorie des calculs (les travaux de MILNER [MIL 80]).

Ces dernières b et c, se retrouvent au chapitre III.

PARTIE I

ETUDE BIBLIOGRAPHIQUE DES EXPRESSIONS DE COMMUNICATION

Dans cette partie, nous présentons deux types d'outils d'expression de la communication, que nous situons à deux niveaux distincts :

- l'un où la communication est considérée comme étant décrite (ou spécifiée), de façon indépendante d'un langage de programmation cible (cf. chapitre III) ;
- L'autre où la communication, à l'opposé est écrite dans un langage de programmation donné. (cf. chapitre II). On ne dispose généralement, à ce niveau que d'une seule forme de communication (dans le sens où cette dernière induit par sa définition des règles particulières de synchronisation. Exemple : le rendez-vous).
  Ces deux niveaux correspondent à différentes étapes de construction de programmes.

L'objectif de cette partie est de montrer que toute forme de communication doit être définie dès le niveau de spécification.

CHAPITRE II

PRESENTATION DE LA COMMUNICATION DANS LES LANGAGES DE PROGRAMMATION

II - 1 Présentation du point de vue adopté.

II - 2 La famille de langages où la communication est orientée-ressource.

- 2.1 La ressource est une variable globale.
- 2.2 La ressource est une boîte aux lettres.
- 2.3 L'expression de la ressource est un moniteur.
- 2.4 Présentation d'un langage où la communication est "orientée-ressource" DPL.

II - 3 La famille de langages où la communication est "orientée-processus".

- 3.1 Le langage CSP.
- 3.2 Le langage ADA.

II - 4 Comparaison entre les deux familles.

II - 1 PRESENTATION DU POINT DE VUE ADOPTE

Il est actuellement commun d'exprimer un programme parallèle à l'aide de processus coopérants, par exemple dans les langages ADA, CSP ou DPL exposés ci-dessous (cf. [RAY 81] pour un historique sur la coopération entre processus).

Dans le domaine d'applications temps réel réparties, nous observons que suivant les problèmes posés, la communication peut être amenée à s'exprimer de manière synchrone ou asynchrone (dans ce dernier cas, le processus émetteur peut anticiper par plusieurs envois de messages). La notion de "rendez-vous" (par exemple, celle définie en CSP) est contraignante pour l'expression de ces problèmes.

On peut concevoir un système parallèle comme un ensemble de ressources et processus avec des relations entre ces derniers (exemples : concurrence d'accès à la ressource, exclusion mutuelle, expressions de synchronisation [HEL 80] , [PTH 82]).

Nous analysons les langages de programmation du point de vue de l'utilisateur (ou programmeur d'application) qui veut trouver dans un langage hôte une expression prédéfinie de communication satisfaisant à une "forme" de communication donnée correspondant à son besoin ; (par exemple une communication asynchrone où l'émétteur n'et jamais bloqué à l'émission).

Nous constatons qu'il existe deux famille de langages ; l'une où la communication est exprimée sous une forme, que nous appelons "orientée-ressources", l'autre sous une forme que nous appelons "orientée-processus".

II - 2 LA FAMILLE DE LANGAGES "ORIENTE-RESSOURCE"

Au niveau programmation, l'aspect prédominant dans ces langages est l'existence :

- d'une ressource partagée par les processus : il s'agit au niveau physique d'une mémoire partagée, et au niveau programmation d'une variable, soit globale pour les processus, soit locale à l' un d'eux ;
- d'un ensemble de règles d'accès à cette ressource utilisées par les processus : il s'agit de règles de synchronisation. Ces règles sont soit implicites (prédéfinies dans le langage), soit construi - tes par l'utilisateur.

Dans cette famille de langage, le nombre d'interlocuteurs dans une communication est quelconque.

Nous citons ci-dessous, des langages de cette famille en commençant par la forme de la ressource la plus primitive (la variable) et en terminant par la plus générale (le moniteur). Nous donnons ensuite la présentation du langage DPL.

2.1 La ressource est une variable globale

La communication n'existe pas sous forme prédéfinie dans le langage ; c'est à l'utilisateur de la construire à l'aide des structures de contrôles, et des règles de visibilité offertes par le langage.

Exemple : dans le langage SIMULA [DAH 70], la communication peut être réalisée entre deux processus X et Y par une variable globale aux

deux processus. L'instruction RESUME (X) placée dans Y, permet le passage de contrôle du processus Y au processus X.

La communication ici peut-être synchrone ou asynchrone suivant sa construction par l'utilisateur.

2.2 La ressource est une boîte aux lettres

La boîte est une suite d'éléments, produits par le processus émetteur, et consommés par le processus récepteur. (Nous supposons ici que cette suite est de longueur finie).

Les règles d'accès à la boîte sont les suivantes :

- le processus consommateur ne peut prélever (ou consommer) un élément si sa boîte est vide ; il est alors bloqué jusqu'à une nouvelle production ;
- le processus producteur ne peut déposer (ou produire) un élément si la boîte est pleine, il est alors bloqué jusqu'à une nouvelle consommation ;
- le producteur et le consommateur ne peuvent agir simultanément sur le même élément à produire ou à consommer (exclusion mutuelle).

La communication construite peut être synchrone ou asynchrone :

- si la boîte aux lettres peut contenir n éléments (la boîte est représentée par un "buffer" de longueur n) ; la communication est asynchrone ;
- si la boîte aux lettres ne peut contenir qu'un élément au plus, il s'agit alors d'une communication synchrone.

L'émetteur et le récepteur agissent donc suivant les règles du modèle dit producteur-consommateur [CRO 76]. Ces règles peuvent être construites par l'utilisateur (par exemple à l'aide d'évènements ou de sémaphores [CRO 76]), ou prédéfinies dans le langage : c'est le cas de la dernière version du langage LTR V3 ; une boîte aux lettres est déclarée localement à un processus (et est "exportable" [PAR 84]), elle est ensuite accédée par les procédures prédéfinies SEND (b, m) et RECEIVE (b, m) ; où b est un objet de type "boîte" et m (le message) de type figurant dans la déclaration de la boîte.

2.3 L'expression de la ressource est un moniteur

Le concept de moniteur (introduit par HOARE [HOA 74]) représente un cas généralisé dans les langages où la communication est orientée-ressource. Les règles sont l'exclusion mutelle au moniteur lui-même : l'accès est fait dans ce cas par emploi de procédures internes, et la synchronisation est assurée par l'utilisation de primitives, WAIT et SIGNAL employées dans le corps des procédures.

La communication peut être synchrone ou asynchrone suivant la construction du moniteur par l'utilisateur (programmeur d'application).

Malgré le caractère dynamique représenté par les corps des procédures, un moniteur constitue un élément passif au milieu des autres éléments actifs : les processus accèdant au moniteur.

Exemple : CONCURRENT PASCAL [BRI 75] : les primitives de synchronisation employées dans le corps des procédures sont : DELAY et CONTINUE (équivalentes aux primitives WAIT et SIGNAL de HOARE). La communication est réalisée par le moniteur.

2.4 Présentation d'un langage où la communication est orientée processus : "DPL"

Ce langage ("distributed processes" language) est défini par BRINCH HANSEN [BRI 78]. On distingue trois parties dans un processus écrit dans ce langage :

- une première partie concernant les déclarations de variables locales ;
- une seconde partie constituée de procédures communes (exécutables par les autres processus), chacune y est décrite par son nom, ses paramètres et son corps ;
- la dernière partie est le corps propre du processus ;

Exemple : l'exemple du producteur consommateur en DPL peut s'écrire suivant le schéma suivant : P est le processus producteur, Q est le processus consommateur. La ressource, une variable locale au processus Q, est accédée par le processus P par appel d'une procédure ("produire") déclarée dans la partie procédures communes de Q. La synchronisation est réalisée par des régions gardées, utilisées dans le corps de la procédure et dans le corps du processus Q. (Une région gardée s'exprime par : WHEN C : S END où C est une expression booléenne, S une section critique sur une variable partagée). L'appel dans Q est accepté (la communication est alors établie), si toute procédure activée dans Q est ou bien terminée, ou bien bloquée, et si en plus Q est bloqué (par exemple sur condition).

Les deux processus P et Q s'écrivent :

```
PROCESS P
% déclarations de variables locales %
   i : INT
:
% partie procédures communes %
:
BEGIN  % corps du processus %
:
CALL Q.Produire (i)
:
END
```

```
PROCESS Q
% déclarations de variables locales %
Temps plein : BOOL ;
J, Tampon : INT ;
:
% partie procédures communes %
PROC Produire (i : INT)

WHEN NOT  Tampon plein
       Tampon i := i ;
       Tampon plein := TRUE
END

:
BEGIN % Corps du processus %
Tampon plein := FALSE

      :
      WHEN  Tampon plein :
            J := Tampon ;
            Tampon plein := FALSE ;
      END
       :
END
```

L'exclusion mutuelle est ici assurée par définition de la procédure, les règles d'attente sur la production et la consommation sont assurées par la variable booléenne "tampon-plein" utilisée en région gardée dans Q. Le choix de mettre la ressource du côte consommateur (récepteur) pour

exprimer ce modèle producteur-consommateur est laissé à la charge de l'utilisateur, deux autres choix sont possibles :

. un second choix, duale au premier, consiste à mettre la variable tampon du côté émetteur, avec une procédure "consommer" déclarée dans ce dernier. Le consommateur appelle alors cette procédure pour prélever le message.

. Un dernier choix consiste à mettre un processus intermédiaire T entre P et Q. Dans le processus T, on déclare alors la ressource (c'est la variable "tampon") à laquelle peuvent accèder P et Q respectivement par des procédures "produire" et "consommer" déclarées dans T. Celui-ci représente alors un moniteur.

II - 3 LA FAMILLE DE LANGAGE OU LA COMMUNICATION EST "ORIENTEE-PROCESSUS"

L'aspect communication prend ici une expression significative dans le sens où il ne s'agit plus d'accèder à une ressource pour y déposer ou retirer des données, mais bien d'un échange d'informations défini entre "interlocuteurs". Cet échange s'exprime par des opérations d'envoi et de réception figurant à l'intérieur des corps des processus communicants. Ces opérations sont offertes à l'utilisateur dans le langage de programmation (hôte) sous formes prédéfinies (ce sont des expressions primitives d'envoi et de réception de messages). La synchronisation y est implicitement définie (par exemple dans CSP, c'est synchrone avec rendez-vous).

Nous présentons ci-après deux exemples de cette famille CSP et ADA.

3.1 Le langage CSP

CSP (Communicating Sequential Processes) est proposé par HOARE [HOA 78]. Un processus en CSP est séquentiel, il s'exprime à l'aide des commandes gardées de DIJKSTRA [DIJ 75]. Les processus sont autonômes (dans le sens où ils ne partagent pas de variables) ; chacun dispose de ses variables locales. Pour communiquer, et le processus émetteur (soit P), et le processus récepteur (soit Q) doivent effectuer simultanément (et respectivement dans leurs corps) les commandes complémentaires de sortie (soit Q ! m), et d'entrée (soit P ? n). Dans ce cas seulement la communication est établie, et le message envoyé ici par P, est reçu par Q. (m et n doivent être de même type). Autrement, il y a attente de l'interlocuteur ayant exécuté la commande d'entrée/sortie en premier. Ceci est la notion de rendez-vous de CSP. Les processus en CSP sont décrits statiquement, grâce à la commande parallèle "||". (Les symboles * [ ] et "$\longrightarrow$" désignent respectivement un cycle et une garde). L'exemple précédent du producteur-consommateur s'écrit simplement :

[P :: PRODUCTEUR || Q :: CONSOMMATEUR]

ces processus s'expriment à leurs tours par :

```
% le producteur %                    %le consommateur %
    i : integer                          j : integer
    * [ % produire une donnée dans i %   * [ ... ; P ? j ⟶ % consommer la donnée
      ... ; Q ! i ; ... ]                  reçue % ... ]
```

Ainsi, dans CSP l'expression de la communication est symétrique dans le sens où chaque processus doit désigner son interlocuteur.

L'emploi des gardes est cependant réservé aux commandes d'entrée (aspect dissymétrique).

Contrairement à la famille de langages précédente, on peut exprimer aisément une réception indéterministe d'un message parmi M envoyés.

Le processus Q peut s'écrire par exemple :

```
Q :: * [ ..
        P1 ? x ⟶ S1
        □
        P2 ? y ⟶ S2
       ]
```

Ceci signifie que si la réception est effectuée à partir de P1 alors les commandes désignées par S1 sont exécutées, et si la réception est effectuée à partir de P2 alors S2 est exécutée. Si les deux réceptions sont possibles, alors une d'entre elles est choisie de façon indéterministe.

Pour remédier aux problèmes de désignation et d'utilisation des sorties dans les gardes, SILBERSCHATZ [SIL 81] utilise le concept de port considéré logiquement comme un objet intermédiaire de communication (il constitue physiquement une sorte de mémoire "tampon". D'autres variantes sont proposées dans : [WAL 72], [KRA 78], [LIS 73], [KAH 81]). Le port est associé à un seul processus : son "propriétaire" ; celui-ci peut l'utiliser dans des commandes gardées d'entrée ou de sortie. Les autres processus "utilisateur" doivent désigner le port en dehors des gardes. (Le compilateur vérifie qu'une telle règle est respectée [SIL 81]). Cette dernière proposition de communication par un objet intermédiaire est nécessaire pour une expression modulaire des programmes et de leurs communications (ceci consiste à éviter toute désignation dans la communication).

3.2 Le langage ADA [ICH 79]

Un processus écrit en ADA (appelé TASK) dispose d'une partie déclarative : constituée de variables locales et de procédures, et d'une

partie constituant le corps du processus. Seules les procèdures sont visibles par les autres processus. (Dans ce cas les processus utilisateurs doivent déclarer le nom des processus serveurs ; par "USE"). Deux types de procédures peuvent être déclarées par un processus. Nous nous intéressons ici à l'un d'entre eux (réalisant la communication) où la déclaration de la procédure est faite par ENTRY. Dans ce dernier cas, si Q est le processus ayant déclaré une procédure PR en ENTRY, accepte explicitement l'appel de PR par une instruction ACCEPT qui définit également le traitement associé à PR (ou le service). L'instruction ACCEPT doit apparaître dans la séquence d'instructions du corps du processus Q. (Ceci est différent de DPL ; pour cette raison la communication en ADA, dite "point d'entrée", s'appelle aussi "communication par appel de procédure attendu").

L'appel exprimé dans un processus appelant s'écrit : CALL Q.PR (paramètres effectifs). L'attente de l'appel dans le corps du processus Q s'écrit :

ACCEPT PR (paramètres formels)

La communication est réalisée lorsque P et Q effectuent respectivement l'appel et l'attente exprimés ci-dessus ; les valeurs des paramètres effectifs déclarés en entrée (par IN) sont transférés à Q. Le traitement est ensuite effectué (le service). Les valeurs de paramètres résultats (définis par OUT dans l'ENTRY) sont transférées de Q vers P, et finalement les deux processus continuent leurs exécutions.

Sinon, le processus effectuant l'appel/acceptation de l'appel en premier est bloqué. C'est la notion de rendez-vous d'ADA.

Exemple : L'exemple proposé ici est celui d'un processus recevant des caractères et renvoyant des lignes de 80 caractères ; ceci est réalisé par deux points d'entrée appelés "Uncar" et "Uneligne" ; A l'aide d'une expression conjointe d'une structure de contrôle indéterministe (SELECT) et des attentes d'appels (ACCEPT). Ces dernières peuvent

être préfixées par une expression booléenne, à l'aide de WHEN. Le tout décrit alors une attente conditionnelle.

```
TASK card IS  % partie visible %
    TYPE Ligne IS ARRAY (1...80) OF CHARACTER ;
    ENTRY Uncar (Car : IN CHARACTER) ;
    ENTRY Uneligne (Tampon : OUT Ligne) ;
END Card ;

TASK Body Card IS % partie cachée %
Buffer : Ligne ; Compte : INTEGER RANGE 1.. 81 := 1 ;
BEGIN
    LOOP
        SELECT % expression indéterministe %
            WHEN Compte < = 80 ⇒ % choix.1 %
            ACCEPT Uncar (Car : IN CHARACTER) ;
              DO Buffer (Compte) := Car END ;
              Compte := Compte + 1 ;
        OR WHEN Compte > 80 ⇒ % choix.2 %
            ACCEPT Une ligne (Tampon : OUT Ligne) ;
            DO Tampon := buffer END ;
            Compte := 1 ;
        END SELECT
    END LOOP
END Card ;
```

Un compteur de caractères (initialisé à 1) est déclaré dans le corps du processus. C'est sur ce compteur que partent les attentes conditionnelles : si sa valeur est < = 80 alors, le premier choix est effectué (constitution de la ligne par adjonction dans le "buffer" du caractère reçu), sinon c'est le deuxième choix (restitution d'une ligne au processus demandeur).

3.3 Comparaison entre ces communications

Les communications en ADA et CSP sont toutes deux synchrones. Elles s'expriment au cours de l'exécution des corps des processus interlocuteurs (relation de communication directe d'un "interlocuteur-1" à un "interlocuteur-2"). La forme (appel de procédure attendu d'une part et commandes d'entrée/sortie d'autre part), et la "sémantique" du rendez-vous sont différentes. Dans ADA, le processus émetteur et le processus récepteur sont bloqués pendant le traitement demandé. Une forme élaborée de communication est présente dans ADA par l'utilisation des paramètres en entrée et en sortie (ceci représente une communication "bidirectionnelle" [OUZ -82.b] cf. chapitre IV).

Ainsi, dans les deux langages, une forme prédéfinie de communication synchrone bloquante à l'émission et à la réception.

Ainsi, pour des processus demandant un taux de communication élevé (asynchronisme), cette forme de communication est pénalisante [VAN 81]. Une forme de communication asynchrone, non bloquante à l'émission et bloquante à la réception est prédéfinie dans le langage PLITS [FEL 79].

II - 4 COMPARAISON ENTRE LES DEUX FAMILLES

La relation de communication dans la première famille, s'exprime d'une part par un accès du (corps du) processus émetteur à la ressource, et d'autre part du (corps du) processus récepteur à cette même ressource. Les "calculs" relatifs à la gestion de la ressource et aux synchronisations sont éparpillés dans le (s) processus et éventuellement dans le corps d'une procédure. Ceci va à l'encontre d'expressions modulaires des processus et de leurs communications. Enfin, toute communication est à

construire et une réception indéterministe est difficile à exprimer voire impossible [AND 81]. Dans la deuxième famille, la relation de communication est établie entre les corps des processus. Elle est donnée sous forme d'opérations dynamiques, et est prédéfinie dans le langage. Ceci permet en particulier d'exprimer aisément une structure de contrôle indéterministe d'actions de communication : une différence essentielle entre DPL et ADA, est que dans ce dernier, l'appel de procédure est attendu (il exprime une relation directe et dynamique de processus à processus et impose en plus le rendez-vous).

Enfin, nous remarquons de façon générale, qu'il n'existe pas dans les langages une expression séparée des communications et des calculs : en ce sens que, dès qu'on veut exprimer des communications plus élaborées que celle décrite dans une communication "bipoint" synchrone, la communication est alors construite, et est détaillée dans les corps des processus.

Il n'existe pas de description statique des chemins de communication indépendamment des processus (ceci à cause de l'existence d'un seul type de liaison : c'est le "bipoint"). cf. chapitre III.

Les formes de communication prédéfinies ne sont pas "riches" ; ou bien on dispose d'une communication synchrone (CSP et ADA avec rendez-vous), ou bien on dispose d'une communication asynchrone (PLITS [FEL 79]). Une expression de communication par des objets intermédiaires (tels que les ports) va dans le sens d'une programmation modulaire et évolutive, (car elle évite toute désignation [OUZ 82.b]) et est plus adaptée à une architecture répartie.

Nous présentons, dans le chapitre suivant, des approches situées au niveau spécification (indépendantes de langages de programmation) ayant adoptées les points précisés ci-dessus.

Nous présentons ensuite nos propositions concernant la communication au chapitre IV.

CHAPITRE III

LA COMMUNICATION EXPRIMEE AU NIVEAU SPECIFICATION

III - 1 Introduction.

III - 2 Les CCS de MILNER.

2.1 L'opération de composition.

2.2 Les opérations de communication.

III - 3 Les travaux de JULLIAND et PERRIN.

III - 4 Conclusion.

III - 1 INTRODUCTION

Ce chapitre résume deux travaux sur la spécification de processus parallèles et coopérants. D'une part, ceux de MILNER, et d'autre part ceux de JULLIAND et PERRIN. Comme notre propos n'est pas de les présenter complétement, nous nous attachons ici uniquement à la description des processus communicants, pour nous intéresser ultérieurement à leur réalisation.

En ce sens ces travaux sont présentés uniquement selon l'aspect de l'expression de la communication. Le but étant de montrer, dans le cadre d'une démarche de construction de programmes, que la communication doit être définie dès le niveau de spécification, et non dans une étape ultérieure (à la réalisation).

Deux points essentiels sont à signaler :

1 - la description statique des communications. MILNER présente un formalisme permettant cette description ;

2 - la description de façon isolée de chacun des processus impliqués dans la description statique. Ceci est traité dans les deux approches.

Pour ce dernier point, nous insistons particulièrement sur la description d'activités de communication internes à un processus ; en ce sens que nous voulons aussi (pour les applications temps réel réparties) décrire plusieurs formes de communication possibles dont un processus peut avoir besoin. Ceci est présenté dans les travaux de JULLIAND et PERRIN.

III - 2 LES "CCS" de MILNER [MIL 78]

(A calculus of communicating systems)

MILNER propose un modèle théorique pour la description du fonctionnement de systèmes parallèles. (Dans ce modèle, on exprime des calculs algébriques). Un système est décrit comme un ensemble d'"agents" communicants. A un agent est associé un "comportement" exprimant des possiblités de communication de l'agent. Chaque possibilité est supportée par un port étiqueté (ou "label") en entrée ou en sortie. Les ports d'entrée et de sortie sont respectivement étiquetés par des noms et co-noms (cf. figure 1) schématisés par :

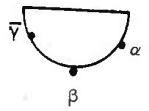


Figure 1 : présentation d'un comportement possédant des ports en entrée α et β ; et un port de sortie $\bar{\gamma}$.

Nous parlons, dans la suite de ce paragraphe, en terme de processus, sachant qu'un processus est un comportement de MILNER. (Dans ce modèle un processus peut être lui-même parallèle (cf. paragraphe 2.1). Les opérations décrites par MILNER sont définies sur les familles de processus (familles de comportements appelées algèbres de comportements).

Nous nous intéressons ici à deux types d'opérations :

- Celles portant sur les processus ; elles permettent une description statique. Nous présentons uniquement l'opérateur dit de composition ;
- celles propres à un processus : nous présentons celles de communication.

2.1 L'opération "|" de composition

C'est une opération portant sur deux processus de MILNER. Cela permet de réaliser une connexion entre les ports de sortie de l'un, et les ports d'entrée de l'autre, si ces ports sont complémentaires.

Exemple : soit p_1 et p_2 deux processus représentés schématiquement par :

Figure 2

La composition $p_1 \mid p_2$ donne un nouveau processus de MILNER schématisé par :

Figure 3

où les possibilités de communication sont suivant la connexion α, $\bar{\alpha}$ (les autres ports ne manisfestent pas de communication, car ils ne sont pas complémentaires).

2.2 Les opérations de communication

L'opération " ! " d'émission

Elle porte sur l'ensemble des valeurs pouvant être envoyées

sur un port de sortie (un port ne définit ici qu'un seul type de valeurs).

L'opération "!" de réception

Elle porte sur l'ensemble des valeurs pouvant être reçues sur un port d'entrée (même remarque que précédemment).

Si dans l'exemple précedent p_1 et p_2 échangent des valeurs entières, une émission dans p_1 s'écrit par exemple : $\vec{a}$! 5.

Une telle communication est synchrone avec rendez-vous.

Remarques :

Dans ce modèle on distingue donc deux points essentiels dans le concept de communication :

- les connexions, qui sont ici des chemins bi-points de transfert de messages (point de vue descriptif) ;
- les opérations de communication synchrones d'envoi et de réception qui expriment des actions de communication.

Nous présentons (chapitre V) nos propositions concernant le premier point, elles permettent une description, de façon prédéfinie, d'une connexion entre plusieurs interlocuteurs.

Quant au type de communication dans ce modèle, il est restreint au rendez-vous. Nous souhaitons pour la classe d'applications temps réel, de disposer de plusieurs formes de communications prédéfinies. Ce point de vue est exposé dans le paragraphe suivant. Nous présentons nos propositions de communication dans le domaine de traitement de procédés industriels au chapitre suivant.

III - 3 LES TRAVAUX de JULLIAND et PERRIN [JUL 81], [PER 82]

Le but principal de ces travaux est d'exprimer les caractéristiques des communications. Ceci a conduit particulièrement dans [JUL 81]

à vouloir exprimer des formes de communications variées synchrones ou asynchrones, "déterministe" ou "non-déterministe" (par déterministe on entend une communication dans laquelle les données produites sont consommées dans le même ordre).

Dans cette optique, l'outil retenu pour exprimer un processus définissant une forme de communication est le type abstrait algébrique [GUT 77]. La définition du type de communication est faite en trois parties :

- Spécification : elle est donnée par le quadruplet <T, F, A, R> avec :
  - . T : ensemble de types t, t_1, t_2, ..., t_n (ou sortes [FPA 81] dont t est la sorte d'intérêt) ;
  - . F : ensemble des opérations et leurs profils : $t \times t_j \longrightarrow t : f_i$ (f_i étant l'opération définie)
  - . A : ensemble d' axiomes ;
  - . R : ensemble de restrictions : une expression d'attente (ATTENTE) ainsi qu'un traitement d'erreurs (ERREUR) sont introduits à ce niveau.
- Réalisation : le type de communication est réalisé de la façon suivante :
  - d'une part, on considère la représentation de la variables d'échange, au niveau de laquelle on distingue une structure simple (permet de mémoriser une donnée) et une structure multiple (permet de mémoriser une suite ordonnée de données) ;
  - d'autre part, on considère la représentation des opérations. On distingue alors les opérations d'accès aux données (visibles à l'utilisateur) des opérations d'accès à l'état de la représentation (invisibles pour l'utilisateur).
- Vérification : il s'agit de vérifier que la réalisation satisfait à sa représentation.

Nous donnons ci-après la spécification d'un exemple simple de producteur-consommateur pour fixer les idées ; d'autres exemples sont donnés dans [JUL 81] sur des types de communication plus élaborés.

Exemple : soit la spécification d'une communication suivant le modèle de producteur-consommateur que l'on peut schématiser dans la représentation graphique de MILNER par :

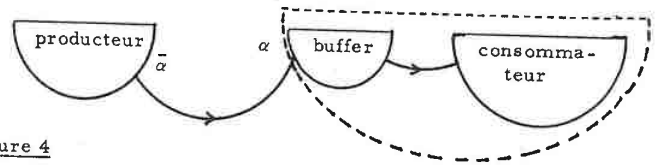


Figure 4

Le type de communication ici est défini par buffer, pour lequel la stratégie de communication est la suivante :

- la longueur de la variable d'échange étant un, le processus producteur ne peut produire une donnée tant que la précédente n'est pas consommée ;
- le processus consommateur consomme une seule fois toutes les données transmises dans le même ordre que celui de production et ceci jusqu'à la donnée DERNIER.

Cette communication est qualifiée de "déterministe synchronisée". C'est-à-dire que toutes les données produites sont consommées, et dans le même ordre avec en plus une de décalage.

La spécification est la suivante :

TYPECOM buffer, elt, BOOL

OPERATIONS

( ) ⟶ buffer : en ;

(buffer, elt) ⟶ buffer : produire ;

(buffer) ⟶ [elt, buffer] : consommer ;

OU (buffer) ⟶ elt : valeur ;

(buffer) ⟶ buffer : enlever ;

(buffer) ⟶ BOOL : ouvert, terminé ;

AXIOMES

soit b ∈ buffer, e et DERNIER ∈ elt ;

(1) ouvert (en) = VRAI ;

(2) ouvert (produire (b, e)) = FAUX ;

(3) ouvert (enlever (b)) = VRAI ;

(4) termine (en) = FAUX ;

(5) terminé (produire (b, e)) = SI e = DERNIER ALORS VRAI
SINON terminé (b)
FSI ;

(6) terminé (enlever (b)) = terminé (b) ;

(7) valeur (produire (b, e)) = e ;

RESTRICTIONS

(8) ouvert (b) ⟹ ATTENTE (valeur (b)) ;

(9) ¬ ouvert (b) ⟹ ATTENTE (produire (b, e)) ;

(10) terminé (b) ⟹ ERREUR (produire (b, e)) ;

(11) terminé (b) ∧ ouvert (b) ⟹ ERREUR (valeur (b)) ;

FTYPECOM

L'ensemble des axiomes et des restrictions produisent ici la loi (ensemble de règles) suivie par les opérations :

(1), (2) et (3) expriment que "buffer" est ouvert pour une nouvelle production après création ou après avoir enlevé une valeur, et non ouvert après une production.

Les axiomes suivants expriment la dualité par rapport aux précédentes pour le consommateur ((4), (5) et (6)) : (7) exprime l'élément prélévé par le consommateur.

(8) et (9) précisent que b est ouvert (resp. non ouvert) quand on est en attente de consommation (resp. attente de production).

Les restrictions (10) et (11) expriment respectivement que l'accès au type communication est une erreur pour le producteur juste après une production de valeur DERNIER et, pour le consommateur quand il a consommé cette dernière.

La synchronisation est exprimée ici par définition des opérations et par les restrictions portant sur ces opérations.

Remarque :

Le type de communication, définissant une forme de communication donnée, est précisé dès le niveau de spécification. Il peut être ensuite réalisé dans un langage de programmation donné par les primitives de communication, ou de synchronisation offerts dans ce langage.

Ceci assure une indépendance de tout langage de programmation en ce qui concerne la communication.

III - 4 CONCLUSION

Dans le cadre d'expression modulaire d'une application temps réel répartie, nous retenons les points essentiels de cette partie (chapitres I et II) :

. Dans un domaine où l'on n'a pas besoin de création dynamique de processus (c'est le cas pour notre classe d'application), la communication entre les processus de l'application doit être exprimée (à un niveau externe aux processus) de façon statique. C'est

le cas des connexions définies dans le modèle de MILNER.

. Les types de communication (ou formes de communication dont on a besoin dans le domaine choisi) doivent être tous précisés, dès l'étape de spécification (avant la réalisation dans un langage de programmation).

De ce point de vue, nous rejoignons par notre démarche (chapitre IV) l'approche exposée dans le paragraphe précédent, elle en est différente par l'aspect pragmatique que nous avons adopté.

Cette structuration de l'application en entités autonomes (processus non forcément séquentiels ; exemple : processus de MILNER) et description des voies de communication entre ces entités (éventuellement par un intermédiaire, exemple : le port de communication) rend compte du concept de communication, aussi bien au niveau spécification, qu'au niveau programmation.

PARTIE II

PROPOSITION D'UNE DEMARCHE DE CONSTRUCTION DE PROGRAMMES POUR LES APPLICATIONS TEMPS REEL REPARTIES

Dans cette partie, nous nous posons le problème de construction de programmes sur un réseau de processeurs dont le rôle est la commande/contrôle d'un ensemble de procédés industriels donnés.

Tenant compte des caractéristiques de la classe d'application choisie, notre proposition consiste à décrire à une étape intermédiaire l'application sous forme d'un réseau de modules communicants, sans pour autant tenir compte des contraintes des types de processeurs, ou de langages de programmation.

Cette démarche est menée à bien grâce, en particulier, à l'introduction du concept de communication dés l'étape de conception.

CHAPITRE IV

L'APPROCHE PRECONISEE

IV - 1 INTRODUCTION

Dans la classe d'applications temps réel réparties, nous appelons description fonctionnelle externe, les définitions de l'ensemble des échanges entre le système logique de contrôle et le système physique contrôlé, ainsi que les conditions sous lesquelles s'effectuent ces échanges. C'est-à-dire que nous supposons définies :

a - les fonctionnalités : nous entendons par fonctionnalité une définition de ce que doit faire le système logique face à un procédé physique pour garantir son fonctionnement normal décrit par le cahier des charges. Ces fonctionnalités sont associées aux entrées et aux sorties du système physique.

b - la coordination entre les entrées et les sorties du système logique : il s'agit, tenant compte de contraintes physiques imposées par le cahier des charges, de respecter des ordonnancements entre les entrées et les sorties, leur parallélisme (réel) et en particulier des contraintes temps réel.

Exemple de l'ascenseur : les entrées du système logique sont : demande d'étage d'un utilisateur, demande d'arrêt urgent (stop), l'étage où se situe la cage, etc... Les sorties sont : la commande marche/arrêt du moteur, la commande d'ouverture/fermeture de portes, etc...
- Les fonctionnalités sont par exemple :

(1) la cage de l'ascenseur doit s'arrêter (ceci se traduit par un envoi d'une commande d'arrêt par le système logique) à un étage si une demande lui est parvenue d'un utilisateur situé, soit à cet étage, soit à l'intérieur de la cage ;

(2) le système logique doit satisfaire le service suivant :
en montée (respectivement en descente), la cage doit être arrêtée finalement à un étage, sans continuer dans le même sens, lorsqu'il

ne reste plus de demandes d'étages supérieurs (respectivement d'étages inférieurs). Elle peut alors rebrousser chemin, s'il existe - en attente - des demandes d'étages inférieurs (respectivement supérieurs) à satisfaire.

b) - La coordination, entre les entrées et les sorties, est :

(1) l'ordonnancement des sorties par rapport aux entrées (suivant le service imposé) ;

(2) le parallélisme réel entre toutes les entrées du système ;

(3) la contrainte temps réel : à l'arrivée d'une demande dite arrêt d'urgence, le système doit envoyer une commande d'arrêt, au plus tard à la fin d'un intervalle de temps défini à partir de l'arrivée de cette demande (ce délai est court).

La description fonctionnelle externe peut être présentée de façon formelle [JAF 80]. Ou informelle telle que "la spécification initiale" dans [LON 83].

Nous appelons description fonctionnelle interne* une solution possible de l'énoncé du problème posé par la description fonctionnelle externe.

Dans la classe d'application choisie, nous attendons d'une telle solution les qualités suivantes :

(1) A partir d'une même description ; pouvoir passer aussi bien à la simulation qu'à la réalisation ;

(ii) Permettre des formes de communications variées (i.e ne pas imposer un seul type de communication) :

- aussi bien des communications asynchrones que synchrones ;
- aussi bien des communications entre deux interlocuteurs (dites bi-points) que des communications faisant intervenir plusieurs interlocuteurs (exemple : diffusion) ;

ces communications doivent être prédéfinies.

---

* Elle correspond dans [ABR 83] aux étapes de structuration logiques et organiques.

(iii) Disposer d'une unité de description d'application qui, à la réalisation, donne une unité "multi-tâches" (tâche dans le sens processus sequentiel) ceci est fondamental pour résoudre des problèmes tel que celui posé dans l'exemple précédent en b - (3). (Problème de préemption cf. paragraphe V.5.2.).

Nous esquissons dans ce sens notre démarche (paragraphe 2), les moyens fondamentaux utilisés pour la construction d'une solution (paragraphe 3), et finalement nos propositions concernant la communication (paragraphe 4).

V - 2 APPROCHE CHOISIE

Nous supposons que la description fonctionnelle externe est dégagée (de façon formelle ou informelle). L'énoncé du problème est alors exprimé sous forme d'une seule unité (en un tout : c'est le système logique) échangeant des messages en entrée et en sortie (cf. figure 1).

Notre approche consiste, ensuite, à développer une solution en deux grandes étapes : la première a pour résultats la description fonctionnelle interne de l'application (cf. chapitre VII et V), et la deuxième est la programmation de cette application, en tenant compte des résultats développés dans la première (cf. chapitre VI).

. Dans la première étape (appelée étape de conception), il s'agit d'atteindre deux objectifs, que l'on peut résumer sous les questions suivantes :

1 - comment décomposer l'application, à partir de la description fonctionnelle externe, en un réseau de modules communicants ? une réponse à cette question est abordée dans le chapitre VII, elle est traitée par ailleurs dans [LON 83], [FAZ 84].

2 - Comment décrire le réseau de modules obtenu, sans pour autant faire d'hypothèses sur les machines cibles et, en tenant compte le mieux possible des contraintes décrites dans la description fonctionnelle externe ? notre réponse à cette question est présentée dans le chapitre suivant.

. Dans la deuxième étape (appelée programmation), il s'agit, cette fois ci, de tenir compte de l'ensemble des contraintes :

a - logiques : celles exprimées dans la description fonctionnelle interne et celles liées à la représentation des structures et des types de communication ;

b - physiques : ce sont les contraintes liées à la répartition des modules sur l'ensemble des processeurs disponibles, et aux choix de représentation, au niveau d'un module, de divers structures de contrôle (par exemple, l'indéterminisme cf. paragraphe V.5.2) par programmation du module sur un processeur cible. (On suppose que ce langage a accés au noyau temps réel).

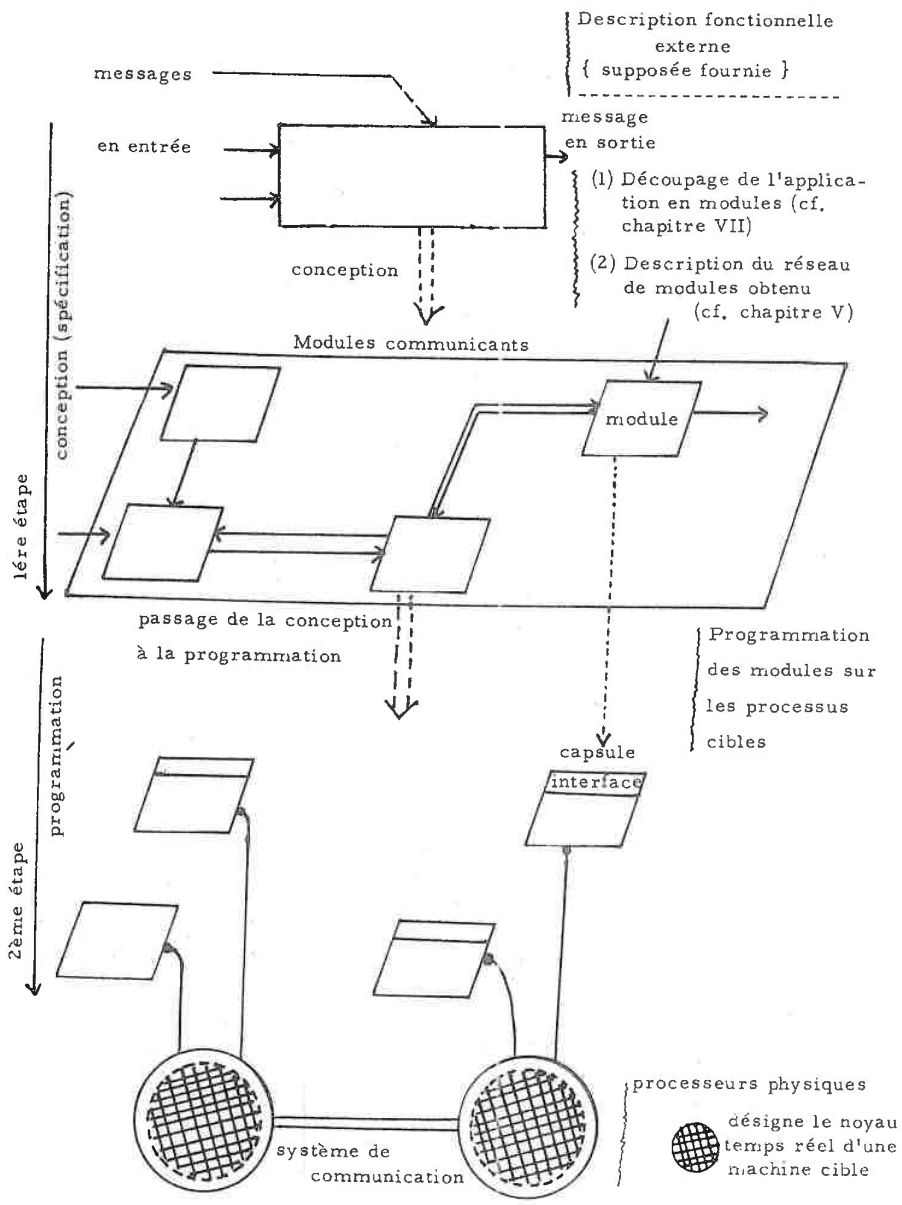
La programmation d'un module aboutit à une unité appelée "capsule". Le module est l'interface de la capsule.

REMARQUE

L'architecture de l'application au niveau conception correspond exactement à celle du niveau programmation, sans remise en cause des relations de communication, ou de découpage en modules.

Ce choix relève du caractère statique de telles applications (cf. chapitre I).

Il nous permettra d'éviter des créations dynamiques d'autres unités ; le passage entre la conception et la réalisation sera plus aisé.



IV - 3 LES MOYENS UTILISES LORS DE L'ETAPE DE CONCEPTION

3.1 Abstraction

L'abstraction est, dans notre démarche, un moyen utilisé pour ne pas prendre en compte dès l'étape de conception des contraintes de réalisation ; particulièrement celles concernant l'architecture des processeurs et du (des) langage (s) de programmation utilisé (s).

3.2 La modularité

Comme pour toutes les applications de grande taille, la modularité est employée ici essentiellement pour structurer l'application, et réduire sa complexité.

Ce concept est largement développé dans [DER 74], [SOK 80].

3.3 La communication entre modules

Toutes les relations entre les modules d'une application sont exprimées en termes de relations de communication.

Il s'agit de communications effectives entre capsules qui seront activées à l'exécution. A l'étape de programmation les relations entre capsules seront donc décrites par des relations de communication entre leurs interfaces : les modules.

Ces relations sont fondamentales pour l'étape de conception.

C'est ce concept de communication qui sert de base à la démarche de conception de l'application (cf. chapitres VII, V).

Pour préciser dès l'étape de conception les différentes relations de communication, nous analysons dans le paragraphe suivant, les "besoins en communication" dans le domaine des applications temps réel réparties, et présentons nos propositions.

IV - 4 PROPOSITIONS DE COMMUNICATIONS

4.1 Les besoins des communications dans le domaine de commande/contrôle de procédés industriels :

(1) . Dans ce domaine d'application la communication peut concerner deux entités* communicantes seulement ou plus. Exemples :
- cas d'une diffusion d'un message à plusieurs entités ;
- cas de la réception d'un message parmi n, envoyés par n entités.

(2) . L'entité émettrice, peut-être :
- bloquée à l'émission : ceci est nécessaire dans les cas où l'entité a besoin de recevoir une réponse à son envoi, avant de continuer son traitement, ou bien dans le cas de la commande d'un procédé physique [KRA 80] ;
- non bloquée à l'émission : par exemple, envoi d'une information de façon périodique de l'état d'un dispositif physique ;

(3) . L'ehtité réceptrice, peut-être :
- bloquée à la réception : par exemple, l'entité effectue un traitement à la réception du message et renvoie une réponse à l'entité émettrice ;
- non bloquée à la réception : par exemple, l'entité réceptrice effectue cycliquement l'activité suivante : si le message est arrivé, elle le traite, sinon elle fait un autre traitement en attendant cette réception (optimisation de calcul).

(4) . Dans certains cas la communication entre entités est effectuée avec anticipation ; c'est-à-dire que l'émetteur peut anticiper par plusieurs envois, avant que le récepteur prélève un message à la réception. Ceci impose en particulier que l'émetteur soit non bloqué.

---

* Par entité nous désignons un processus séquentiel

Nous présentons dans le paragraphe suivant nos propositions pour satisfaire ces besoins de communication. Elles concernent uniquement la communication entre deux entités, ces propositions sont généralisées à plusieurs interlocuteurs grâce à la séparation entre les actions de communication, et la description des voies de communication, ce qui répond au point (1). (cf. chapitre V).

Dans le cas où une anticipation est définie, notre proposition permet grâce à l'utilisation d'un outil de communication appelé port (cf. chapitre V, [OUZ 82b]) de définir une politique de prélévement de messages à la réception.

4.2. Propositions :

Nous proposons des types d'actions de communication indépendamment de tout outil de communication (par exemple le port de communication (cf. chapitre V, [OUZ 82 b]). Notre approche est d'exprimer un ensemble minimum de types de communication pouvant recouvrir les besoins énoncés précédemment.

On fait référence, dans ce paragraphe, à une communication entre deux entités, A (appelée source) et B (appelé puits). Cette communication est toujours considérée de la source au puits.

A priori on peut distinguer deux genres de communication (que l'on retrouve dans le chapitre II) :

- Celui où il n'y a qu'un seul transfert de message de la source vers le puits (on n'a pas besoin de demander une réponse à ce message) ;
- celui où l'on a besoin après un premier échange de message, de la source vers le puits, d'une réponse à ce message : un autre message s'effectue donc du puits vers la source.

Nous donnons ci-dessous, à chaque fois la définition du type de communication proposé, son interprétation éventuellement dans les langages de programmation, ainsi que des exemples.

4.2.1 Communication sans réponse

On en distingue deux types :

1 - le type T1 (ou "notification")

{ La source envoie le message sans se préoccuper, ni de sa réception, ni de son traitement par le puits.

De même dans ce type de communication, la réception d'un message s'effectue sans se préoccuper de l'émission.

Exemple : une entité source doit scruter périodiquement le niveau d'un liquide pour envoyer cette information à une entité puits ; celle-ci effectue cycliquement l'activité suivante :

- soit elle reçoit effectivement le message envoyé alors, elle le traite ;
- soit elle ne le reçoit pas, elle effectue alors un autre traitement (en évitant le temps d'attente).

L'anticipation est définie pour ce type de communication.

L'émission n'est pas bloquante pour la source même si le puits n'est pas prêt à recevoir.

La réception n'est pas bloquante s'il n'y a pas de message à prélever. Ce type de communication n'est pas présent comme une primitive dans les langages de programmation (dit de haut niveau), que nous avons analysés dans [OUZ 82-b] , (exemples : ADA, CSP, PLITS, LTR-V3, ARGUS [LIS 82]).

2 - Le type T2 (ou "commande")

La source envoie le message tout en s'assurant de sa réception du côté puits. Si le message est arrivée la communication est complète, sinon elle est incomplète.

Pour la source une fiabilité d'acheminement de message doit être assurée. Par complète, nous entendons qu'un accusé de réception est parvenu à la source, et par incomplète : l'accusé de réception n'est pas parvenu (l'accusé de réception est envoyé par le système de communication sous-jacent à l'éxécution, et non par le puits).

Exemple : dans le cadre de commande d'une machine industrielle, la source a un rôle de superviseur, le puits a pour fonction la commande effective de cette machine.

L'envoi d'une commande de la source vers le puits (par exemple, un ordre d'usinage) est une émission du type T2. La source doit être informée de la réception de son message du côté puits.

On peut utiliser, ou ne pas utiliser l'anticipation dans ce type de communication.

La réception pour le type de communication "commande" est bloquante. L'émission est bloquante ou non suivant l'anticipation (le nombre d'anticipations est borné, soit n ce nombre) ;

a) . Si l'anticipation est utilisée: la source est alors non bloquée à l'émission du message (dans la mesure du nombre d'anticipations autorisées), si au bout de n anticipations, la source n'a toujours pas reçu d'accusé de réception, elle est alors bloquée ;

b) . Si l'anticipation n'est pas utilisée: la source est alors bloquée à l'émission, jusqu'à ce qu'elle ait reçu un accusé de réception pour son message envoyé.

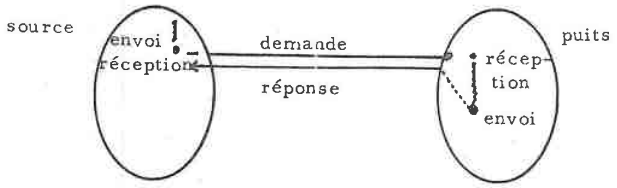
Exemples de langages de programmation :

- La représentation dans un langage de programmation du cas b) est la communication dite par rendez-vous dans CSP.
- La représentation du cas a) est celle dite par "boîte aux lettres" dans le langage LTR - V3 (version fin 1983) ; la communication est réalisée par des procédures SEND (b, M) et RECEIVE (b, M) où b et M sont respectivement la boîte et le message.

4.2.2 Communication avec rèponse

- Le type T3 (ou "service")

Le puits, à la réception d'un message, exécute le traitement associé et renvoie systèmatiquement une réponse consécutive à ce traitement. Une communication du type T3 commence à l'envoi du message et se termine à la réception par la source de la réponse du puits. On dit alors que la communication est complète, dans le cas où la réponse n'arrive pas, on dit que la communication est incomplète (cf. schéma).



Exemple : Reprenons l'exemple de commande de la machine industrielle (donné précédemment) ; une communication du type T3 est le cas où à chaque envoi d'une commande par la source, celle-ci attend immédiatement la réponse (elle n'effectue pas d'autres traitements). La source,

à la réception de la commande, effectue le traitement associé (en agissant sur la machine) et renvoie une réponse à la source.

L'anticipation n'est pas définie pour ce type de communication. La source est bloquée à l'émission jusqu'à la réception de la réponse du puits.

Le puits est bloqué à la réception du message envoyé par la source ; à la réception du message, il effectue le traitement spécifié (c' est le service) et renvoie la réponse : cette dernière émission est non bloquante pour le puits.

Ce type de communication est appelé dans la littérature "appel de procédure à distance".

Exemples de langages de programmation.

- La représentation de ce type de communication dans un langage de programmation est l'appel du "point d'entrée" (ENTRY) de ADA.
- Ce type service n'est pas celui proposé dans le langage ARGUS [LIS 82], même si la communication dans ce langage est nommée "appel de procédure à distance". En effet, cette communication est effectuée suivant le même schéma que le type T3, mais l'émission est non bloquante pour la source.

De même que dans le cas de la communication du type T2, cette comminication doit être représentée comme une communication atomique ; ceci est important du côté source pour le traitement d'erreurs si une anomalie survient dans la communication.

Remarque :

Le type service est souvent représenté dans les langages de programmation à l'aide du type T2, sous forme de deux communications : l' une est un envoi de la source vers le puits, l'autre étant dans le sens opposé.

A l'aide des types "notification", "commande" et "service", nous couvrons les besoins de communication énoncés dans le paragraphe précédent.

L'avantage, ici, est qu'ils sont prédéfinis dès le niveau de conception. Ils seront mis en œuvre à la réalisation à l'aide de primitives de communication constituant, en ce sens, une extension du langage cible.

IV - 5 CONCLUSION

Dans le cadre de notre approche, nous avons développé dans ce chapitre, essentiellement le point ii), signalé en introduction, (exprimer plusieurs formes de communication). Nous exposons dans le chapitre V notre proposition de solution, dans laquelle un module y est défini, comme pouvant prendre en compte le parallèlisme de l'environnement dans sa définition. Ceci répond au point iii).

La séparation entre la description des voies de communication, des modules et la description (interne, relativement à la précédente) de chacun des modules ; nous permet d'une part d'effectuer des contrôles statiques sur la cohérence d'expression des différents éléments de la description donnée, et d'autre part, de passer aussi bien à la simulation [GOF 84] qu'à la réalisation [MAM 82]. Ceci répond au point i) énoncé en introduction.

CHAPITRE V

L'ETAPE DE CONCEPTION ET LE LANGAGE PROPOSE

V - 1 PROPOSITION DE SOLUTION

La description de l'application à la fois syntaxique et sémantique est fournie en deux parties : le niveau application et le niveau module.

(A) Au niveau application on trouve (cf. figure 1) :

- une liste des noms de modules qui la constituent ;
- une liste des messages typés pouvant circuler entre ces modules ;
- une liste des "connexions" entre les modules ; nous appelons ainsi les chemins d'échanges de messages reliant les points d'entrée ou de sortie, des différents modules, appelés ports ;
- la description du comportement global de l'application donnée sous la forme des relations, déduites de l'énoncé du problème, devant exister entre les réceptions et les émissions des messages échangés avec l'environnement de l'application.

(B) Au niveau module, on trouve la description de chaque module (cf. figure 2).

Rappelons qu'un module est défini comme l'interface d'une capsule. Le texte d'un module décrit cet interface ; il est constitué de la liste des ports du module et des types de messages qu'ils acceptent (nous appelons cette liste partie syntaxique), et de la définition complète du comportement du module exprimé uniquement à l'aide des communications de messages par les ports, (c'est ce que nous appelons partie sémantique).

Le niveau application (partie A) est présenté au paragraphe V.4.

Le niveau module (partie B) est présenté au paragraphe V.5, dans celui-ci, nous détaillons d'une part, les expressions syntaxiques des ports (ce sont des descriptions statiques) au sous-paragraphe 5.1, et d'autre part le comportement du module au sous-paragraphe 5.2.

La définition des outils intégrés dans le langage de description proposé est donnée ci-après.

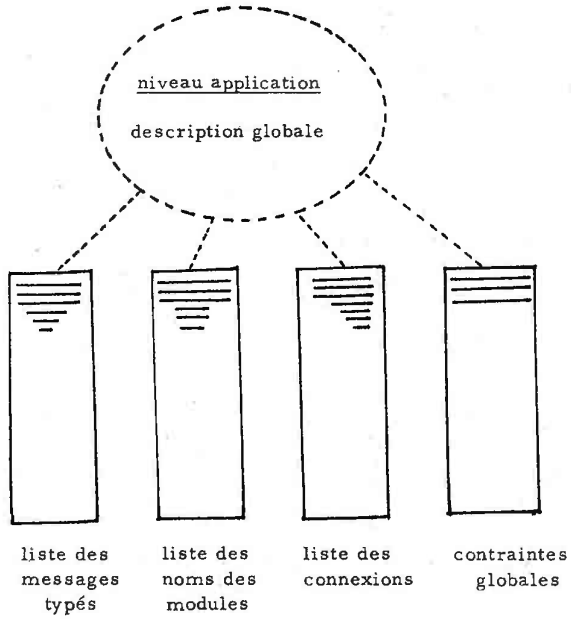
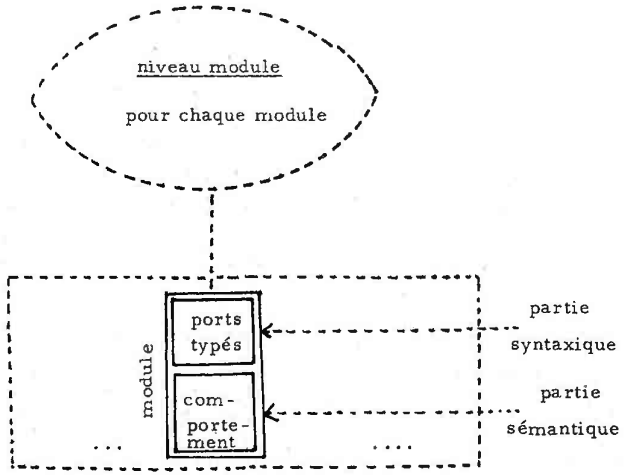


Figure 1



Spécification de chaque module

Figure 2

V - 2 LES OUTILS INTEGRES DANS LE LANGAGE DE DESCRIPTION

2.1 Les ports

Un port est considéré comme une fenêtre ("whole" dans [LIK 80]) à travers laquelle les messages sont échangés en entrée ou en sortie du module. C'est un outil de communication intermédiaire, sur lequel sont exprimés les actions de communication (typées, proposées dans le chapitre IV).

Dans le même sens que les genres de communications, définis dans le chapitre IV, nous distinguons deux sortes de ports :

a - des ports utilisés uniquement en entrée ou uniquement en sortie ; ils sont notés schématiquement " $\rightarrow$ " et " $\rightarrowtail$" ;

b - des ports de service : un port de service peut être utilisé en entrée et en sortie

. chez le module serveur, il est systèmatiquement utilisé d' abord en entrée et ensuite en sortie ; il est noté schématiquement " $\leftrightarrow$ " ;

. chez le module "client", il est systématiquement utilisé d' abord en sortie, et ensuite en entrée, il est noté schématiquement "$\triangleright\!\!\prec$" ;

Seules les actions de communication du type T3 (service) sont utilisées pour ces ports.

A l'aide des types de communication, des ports et des connexions (présentées ci-après), nous exprimons plusieurs formes de communications.

Dans le cas où les communications sont définies avec anticipation, celle-ci est alors décrite par le langage au niveau des ports de

communication. On peut alors associer une politique de prélévement de messages aux ports d'entrée.

2.2 Les connexions

Une connexion définit un ensemble de chemins de transfert (de messages) possibles entre (les ports des) modules communicants. Elle précise , suivant la structure de ces chemins, la façon dont les messages sont transférés du (des) port (s) d'entrée au (x) port (s) de sortie (exemples : diffusion d'un message à plusieurs modules, sélection d'un message, etc...).

Une connexion n'est définie (ou valide) que si les éléments suivant sont correctement énoncés :

A - les interlocuteurs : ils sont désignés, explicitement, par le couple (nom de module, nom de port) ;

B - leur structure : c'est la structure des chemins de transfert de messages reliant les ports de sortie aux ports d'entrée.

On distingue deux types de structures :

1 - structure de base : on en distingue trois ;

(a) - structure bipoint : elle est définie par un seul chemin liant un port de sortie à un port d'entrée (figure 3).

(b) - structure divergente : elle est définie par plusieurs chemins, dont l'origine, un seul port de sortie, est liée à plusieurs ports d'entrée (figure 4).

(c) - structure convergente : elle est définie par plusieurs chemins, dont le sommet, un seul port d'entrée, et lié à plusieurs ports de sortie (fig. 5).

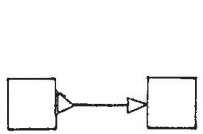


Figure 3

structure bipoint

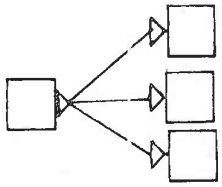


Figure 4

structure divergente

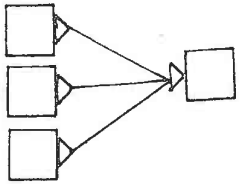


Figure 5

structure convergente

Légende : port de sortie, port d'entrée, module

2 - Structures construites : elles sont construites à partir de ces trois structures de base ;

C - La sémantique de routage, des messages communiqués, sur les chemins considérés.

C'est le role des opérateurs définis au paragraphe 4.3 (ceci permet d'exprimer sur la même structure, par exemple la divergente, soit le message est diffusé à tous les destinataires, soit le message est envoyé à un seul destinataire parmi n).

Cette approche de description des communications de façon externe aux modules communicants rejoint celle de THOMESSE [THO 80] , sur la séparation de la description des structures de contrôle de celles des traitements dans SYGARE. Ceci a pour principal avantage d'éviter toute désignation, dans un module de l'interlocuteur (par le langage proposé), et d'exprimer toute communication de façon unique sans se préoccuper, ni de la répartition des modules, ni de la nature du support physique utilisé pour la communication (mémoire commune ou réseau de transport).

Nos propositions propres concernent essentiellement le choix d'une communication parmi n à l'émission ou à la réception, et sa

formulation dans la classe d'applications temps réel réparties. (cf. opérateur "|" dans le paragraphe 4.3).

V - 3 LE LANGAGE FLEXI

Notre proposition du langage FLEXI a pour objectif de permettre la description d'une application à l'aide des outils introduits précédemment, sans faire d'hypothèse sur le langage de programmation utilisé ultérieurement. A cette étape de conception, on peut même admettre à la réalisation une coexistence de plusieurs langages de programmation, à condition que certaines structures de contrôle liées à la communication (tel que l'indéterminisme cf. 5.2) soient permises dans ces langages.

La description du langage FLEXI constitue une phase intermédiaire pour passer à l'étape de programmation. Pour ce faire, on donne :

- une description des différentes listes du niveau application (présentées dans V.1.A) ; c'est l'objet du paragraphe V - 4 ;
- une description de chaque module ; c'est l'objet du paragraphe V - 5.

La description du module est basée sur la description de ses ports de communication, et de son comportement. Ce comportement décrit comment le module communique avec les autres modules - par l'intermédiaire des ports définis - et quelles sont les valeurs émises en fonction des valeurs reçues et sous quelles conditions.

Une telle description permet de passer aussi bien à la simulation qu'à la programmation (voir les travaux de GOFFIC [GOF 84], ainsi que [BOS 82]et [GUR 82]) ; ce qui permet de tirer des conclusions sur la validation ou non du "modèle" proposé avant toute réalisation.

Des contrôles statiques, liés à la communication sont ici possibles sans pour autant utiliser ceux offerts par le langage cible.

Nous présentons ci-après quelques symboles de description du langage pouvant être utilisés aussi bien au niveau application, qu'au niveau module.

3.1 Les actions de communication

Un message est considéré ici comme une abstraction de valeurs communiquées effectivement à l'exécution. Un message est désigné par un nom ; exemple : m.Une occurence est une valeur désignée à la réception ou à l'émission dans une action de communication, elle est préfixée par le symbole " #" ;
Exemple : # m désigne une occurrence du message précédent.

Une action de communication est une émission, ou une réception décrite syntaxiquement par :

< action d'émission > ::= < identificateur_de_port > !
< identificateur_de_message >
< action_de_réception > ::= < identificateur_de_port > ?
< identificateur_de_message >.

Quand le type de communication est explicité au niveau du module, elle est notée :

$$(< \text{identificateur\_de\_port} > , \begin{Bmatrix} T1 \\ T2 \\ T3 \end{Bmatrix}) ! < \text{identificateur\_de\_message} >$$

3.2 La séquence notée " ; "

Décrit la séquence entre deux actions. (cf. le " ; " dans [MIL 78]).

3.3 La commutativité notée "com"

Deux actions liées par "com" sont effectuées séquentiellement dans un ordre non imposé à ce niveau.

3.4 Le parallèlisme noté "//" :

Deux actions liées par "//" ne sont pas effectuées séquentiellement.

Par action, nous entendons soit une action de communication, soit une description séquentielle d'actions de communication.

V - 4 DESCRIPTION DU NIVEAU APPLICATION

4.1 Liste des types de messages

Tout message circulant dans le réseau de modules, ou entre un module de l'application et son environnement est désigné par un nom.

Le type d'un message s'exprime à l'aide d'un type du langage FLEXI. (Les types sont ceux que l'on retrouve dans un langage de la famille PASCAL).

Pour effectuer des contrôles statiques liés à la communication (conjointement avec les autres listes), nous distinguons deux parties dans cette liste de messages :

- les messages "externes" : ceux envoyés (/reçus) de l'environnement externe, de l'application, au réseau de modules. Ces

messages sont donnés par l'énoncé du problème ;

- les messages "internes" ; ceux ne concernant par l'environnement. Ils sont échangés uniquement entre les modules.

Syntaxiquement cette liste est décrite sous la forme :

MESSAGES TYPES
EXTERNAL
<< message >> $^{+(;)}$
INTERNAL
<< message >> $^{+(;)}$
ENDMESSAGES

(où < a > $^{+(;)}$ désigne soit a soit a ; ... ; a ; a cf. les notations en annexe)
< message > ::= < identificateur_de_message > : < type > où
<type> désigne un type du langage FLEXI.

4.2 Liste des noms de modules

Nous distinguons, de même que dans la liste précédente, deux parties dans la liste des noms de modules :

- les modules externes : ceux pouvant communiquer avec l'environnement externe de l'application. (Leurs ports apparaissent aux expressions de communication dans la liste des contraintes globales cf. 4.4) ;
- les modules internes : ceux ne communiquant qu'avec des modules du réseau.

La liste s'exprime syntaxiquement par :

MODULES NAMES
EXTERNAL
<< identificateur_de_modules >> $^{+(;)}$

INTERNAL
<< identificateur_de_module >> + (;)
ENDMODULES

4.3 Liste des connexions

Cette liste définit toutes les connexions entre les modules de l'application. La syntaxe est la suivante :

LINKS
<< connexion >> + (;)
END LINKS

Nous distinguons deux sortes de connexions :

- connexion de base : elles sont exprimées à partir des structures de base. Dans le cas où ces structures sont convergentes, ou divergentes on définit une sémantique de routage des messages communiqués grâce aux opérateurs " &", "|", et "," présentés ci-après. (Ils définissent les règles de transfert des messages sur les chemins établis) ;
- connexions construites : elles sont exprimées à partir des structures construites (par des structures de base ayant en commun des ports d' entrée ou de sortie).

Une connexion de base dont la structure est bipoint (on dit connexion bipoint par abus de langage) s'exprime syntaxiquement par :
< identificateur_de_module > . < identificateur_de_port > ⟶
< identificateur_de_module > . < identificateur_de_port >
elle signifie que les messages envoyés par le premier module sont reçus par le second.

4.3.1 Connexion de base exprimée à l'aide de " &" (et logique)

L'opérateur " &" n'est défini que sur la structure divergente.

Il signifie que le même message issu du port de sortie (dans l'exemple : P1) est envoyé simultanément sur les ports d'entrée désignés (dans l'exemple : P2, P3 et P4).

Syntaxiquement on écrit :

< identificateur de_module> . < identificateur_de_port >
⟶ < identificateur_de_module > . < identificateur_de_port > > + (&)

Exemple : M1. P1 ⟶ M2. P2 & M3. P3 & M4. P4.

4.3.2 Connexion de base exprimée à l'aide de "|" (ou exclusif)

Cet opérateur est aussi bien défini sur une structure divergente que convergente.

A - sur une structure divergente, il signifie que le message issu du port de sortie (dans l'exemple : P1) est transmis à un et un seul des ports d'entrée désignés, (dans l'exemple : ou bien P2, ou bien P3, ou bien P4). Le choix du destinataire est déterministe. Ce choix est effectué dans la partie comportement du module ayant le port de sortie désigné (dans l'exemple : P1). Nous proposons ici, d'éviter toute désignation directe dans les communications, grâce à un étiquetage des chemins de transfert liant le port de sortie aux ports d'entrée.

syntaxe : < identificateur_de_module > . < identificateur_de_port > ⟶
<< étiquette > : < identificateur_de_module>. < identificateur_de_port> +(|)

Exemple : M1, P1 ⟶ 1 : M2, P2 | 2 : M3. P3 | 3 : M4. P4
choix d'un destinataire parmi 3 à l'émission.

B - Sur une structure convergente, il signifie que le message issu d'un et un seul des ports de sortie désignés (dans l'exemple : ou bien P1, ou bien P2 ou bien P3) est transmis au port d'entrée. Le choix de l'émetteur est déterministe.

syntaxe : << étiquette > : < identificateur_de_module > .
<< identificateur_de_port >>$^{+}$ (|)
⟶ < identificateur_de_module > . < identificateur_de_port >

Exemple : 1 : M1, P1 | 2 : M2 . P2 | 3 : M3 . P3 ⟶ M4 . P4

De même dans le cas précédent le choix de l'émetteur est effectué dans le module ayant le port d'entrée désigné (dans l'exemple ci-dessus : P4).

Remarque

Le choix de l'émetteur (ou du destinataire) à la réception (ou à l'émission) s'effectue syntaxiquement, par l'étiquette désignant le chemin reliant l'émetteur ou le destinataire. Ceci est réalisé à l'aide de la notation pointée : exemple : P4.2 ? #m signifie dans le comportement du module M4, la réception d'un message sur le port d'entrée P4, à partir du port de sortie P2.

4.3.3 Connexion exprimée à l'aide de "," (entrelacement)

L'opérateur "," est défini uniquement sur des structures convergentes.

Il signifie que les messages issus des ports de sortie (dans l'exemple : P1, P2, et P3) sont présentés au port d'entrée désigné (dans l'exemple : P4), dans l'ordre de leurs arrivées. La réception d'un message sur le port d'entrée représente un choix indéterministe de l'émetteur.

Syntaxe : <<identificateur_de_module > . < identificateur_de_port >>$^{+}$ (,)
⟶ < identificateur_de_module > . < identificateur_de_port >

Exemple : M1 . P1, M2 . P2, M3 . P3 ⟶ M4 . P4.

Remarque 1

Nous rejoignons par nos propositions d'opérateurs " & " et "," les propositions de RAYNAL dans [RAY 81], (ils sont désignés respectivement par "=" et ",").

Remarque 2

La différence essentielle entre l'opérateur "|" et l'opérateur "," (respectivement choix indéterministe et déterministe) dans une structure convergente porte obligatoirement sur l'expression du comportement du module récepteur :

- dans le premier cas (avec "|"), le récepteur doit nécessairement distinguer le chemin de transfert du message (et par là l'émetteur) ;
- dans le second, le récepteur ne peut distinguer ces émetteurs.

4.3.4 Règles de validité des connexions de base

(R1) Règle portant sur la structure d'une connexion :

- un chemin d'une structure doit lier deux ports de sens opposés et de même sorte (ceci par définition d'une connexion) ;

(R2) Règle portant sur des connexions distinctes :

- un port ne doit pas être exprimé dans deux connexions distinctes (ceci est un choix pour effectuer un type de contrôles statiques cités dans les exemples ci-dessous).

Exemples : Nous donnons ci-dessous des exemples d'erreurs détectables statiquement dans le langage. Les ports de sortie sont notés ps_i, les ports d'entrée pe_i, les ports de service utilisés en sortie pS_i et les ports de service utilisés en entrée sont notés pE_i.

(a) Exemples pour la règle (1) : . M1 . pe_1 ⟶ M2 . ps_2

. M3 . pS_3 ⟶ M4 . pe_4

(b) Exemples pour la règle (2) :

. L'utilisateur veut décrire deux connexions : la première décrit une structure convergente, et la seconde décrit une structure divergente ; Au lieu d'écrire par exemple :

X1 . ps , X2 . ps ⟶ Y1 . pe ;
X3 . ps ⟶ 1 : Y2 . pe | 2 : Y3 . pe

il décrit :

. soit X1 . ps , X2 . ps ⟶ Y1 . pe ;
X3 . ps ⟶ 1 : Y1 . pe | 2 : Y3 . pe

. soit X1 . ps , X2 . ps ⟶ Y1 . pe ;
X2 . ps ⟶ 1 : Y2 . pe | 2 : Y3 . pe

les erreurs soulignées ici sont détectées statiquement.

(R3) Règle portant sur les types de messages échangés :

. L'emploi respectivement des opérateurs " & " et "," dans des structures divergentes et convergentes impose que les types énoncés dans les ports de sortie soient les mêmes que ceux énoncés dans les ports d'entrée.

. L'emploi de l'opérateur "|" impose, seulement, que pour tout type associé à un port de sortie, il doit figurer au moins un port d'entrée ayant ce type. (les descriptions du niveau application sont alors conjointement renforcées par celles du niveau module pour effectuer les contrôles statiques nécessaires sur les types de messages).

4.3.5 Connexions construites

Ces connexions sont plus "riches" que les précédentes en possibilités de description de communications et de structures variées.

Syntaxiquement une communication construite s'énonce à partir d'une expression parenthésée d'une suite de connexions de base :

CONNEXION
< connexion_de_base >$^{+(;)}$
END CONNEXION

sémantiquement celà veut dire qu'un port d'entrée ou de sortie peut être utilisé dans les différentes connexions de base désignées. On tient compte donc conjointement de l'ensemble des règles établies par ces connexions.

Exemples :

1) CONNEXION
M1, P1 ⟶ M4 . P4 & M5 . P5 ;
1 : M1 . P1 | 2 : M2 . P2 | 3 : M3 . P3 ⟶ M4 . P4
END CONNEXION.

Ceci veut dire que le message diffusé par le module M1 peut être éventuellement choisi de façon déterministe au module M4.

2) CONNEXION
1 : M1 . P1 | 2 : M2 . P2 ⟶ M3 . P3 ;
M2 . P2 ⟶ 2 : M3 . P3 | 3 : M4 . P4
END CONNEXION.

Ceci veut dire que le module M3 choisit un message, à la réception, de façon déterministe ; ce message est envoyé par les modules M1 ou M2. Ce dernier peut à son tour choisir une émission déterministe aux modules M3 ou M4.

4.4 Description du comportement global

Cette description concerne essentiellement les contraintes globales sur la coordination des messages externes de l'application. (Coordination dans le sens du paragraphe IV.1). Pour chaque message reçu on indique quels sont les messages pouvant être émis vers l'environnement de l'application. On indique aussi des contraintes temps réel (s'il y en a) sur ces messages.

Exemple : C'est l'exemple de drainage d'eau, à l'aide d'une pompe, dans une mine de charbon. Les messages externes sont décrits ici sur le schéma représentant l'application. (Cet exemple est énoncé au chapitre VII).

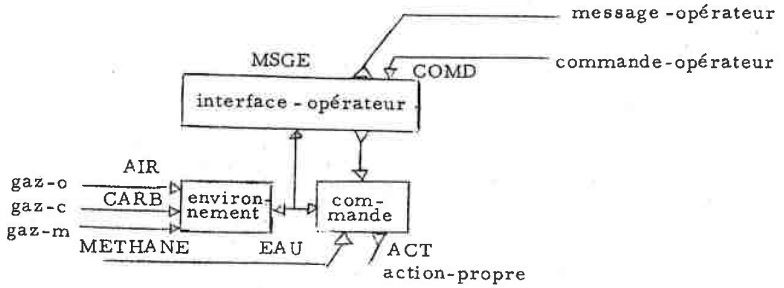


Figure 6

(cf. syntaxe donnée en annexe).

TOTAL DESCRIPTION

EAU ? # eau : (ACT ! # action-pompe ou nil)

METHANE ? # gaz-m : ((ACT ! # action-pompe //
MSGE ! # message opérateur) ou nil)

CARB ? # gaz-c : (MSGE ! # Message-opérateur ou nil)

COMD ? # commande-opérateur : ((ACT ! # action-pompe ;
MSGE ! # Message-opérateur)

ou

MSGE ! # Message-opérateur)

END DESCRIPTION.

Cette description est constituée de cinq éléments, dont chacun décrit : à l'arrivée d'une occurrence de message externe quelles sont les émissions éventuelles d'autres occurrences de messages externes.

nil désigne, ici, la non communication avec l'environnement ; ou est un "ou exclusif" exprimant, dans cette description, un indéterminisme à l'émission. (Pour lever cet indéterminisme, les détails concernant les conditions sous lequelles les communications sont effectuées, sont décrits au niveau module). Le cinquième élément de la liste par exemple, indique qu'à la réception d'une commande de l'opérateur, soit le système de contrôle répond uniquement par un message (exemple : demande de statuts), soit le système envoie une action à la pompe et transmet une réponse à l'opérateur (exemple : commande d'arrêt de la pompe).

Comme seuls les messages externes apparaissent dans les contraintes globales, c'est ici qu'interviennent les descriptions externes de la liste des types de messages, et celle des noms de modules. On vérifie alors, conjointement au niveau module qu'il n'y a pas d'erreurs d'utilisation des communications (utilisation de messages, de ports ou de modules non externes).

Pour les possibilités de communications décrites dans les contraintes globales, on peut éviter au moins syntaxiquement qu'il existe un chemin reliant deux modules externes :

- le premier possède un port d'entrée externe sur lequel est effectuée la réception ;
- le second possède un port de sortie externe sur lequel est effectuée l'émission ;

exemple : pour les possiblités de communications décrites par :

COMD ? # commande-opérateur : ((ACT ! # action-pompe ;
MSGE ! # message-opérateur)
ou
MSGE ! # message-opérateur)

les modules sont : interface-opérateur et commande ; et le chemin est ici une connexion bipoint entre les deux.

En ce qui concerne les contraintes temps réel, elles doivent être vérifiées avant toute réalisation.

V - 5 DESCRIPTION DU NIVEAU MODULE

La description de tous les modules est donnée syntaxiquement sous la forme :

MODULES
<< module >>$^{+}$ (;)
END MODULES

Le module est l'unité de spécification et de construction de l'application. Il est composé d'une partie dite syntaxique décrivant les ports d'entrée et de sortie du module, et d'une partie dite sémantique (c'est le comportement du module) décrivant les actions de communication sur ces ports, les valeurs produites en fonction des valeurs consommées sur ces ports, et les conditions sous lesquelles elles sont effectuées.

La syntaxe d'un module est : MODULE < identificateur_de_module
< ports_de_communication
< comportement >
END < identificateur_de_module >.

La partie syntaxique (ou ports de communication) est décrite au paragraphe 5.1, et la partie sémantique (comportement) est décrite au paragraphe 5.2.

5.1 Description des ports de communication

Dans la description des ports, nous distinguons deux parties :

- la première décrit uniquement les ports de communication avec l'environnement externe de l'application ;
- la seconde décrit les ports tels qu'ils peuvent être utilisés par les autres modules de l'application.

Cette distinction rejoint celles des listes des types de messages, des noms de modules et des contraintes globales pour établir des contrôles sur les communications externes (cf. paragraphe 4.4).

syntaxe : < ports_de_communication > = COMMUNICATION PORTS
EXTERNAL
< liste_de_ports >
INTERNAL
< liste_de_ports >

< liste_de_ports > ::= □ ENTRYPORTS << port-e >>$^{+}$ (;)
□ EXITPORTS << port-s >>$^{+}$ (;)
□ ENTRYEXITPORTS << port-E >>$^{+}$ (;)
□ EXITENTRYPORTS << port-S >>$^{+}$ (;)

(le métasymbole " □ " veut dire qu'il figure au moins une de ces composantes).

Les parties EXTERNAL et INTERNAL peuvent décrire chacune :

(a) - des ports de service utilisés en entrée (dans un module dit "serveur") :

< port-E > ::= < identificateur_de_port > : << identificateur_de_message > REPL
< identificateur_de_message >$^{+}$ (;)

où le premier message désigne le message reçu et le second le message réponse ;

(b) - des ports de service utilisés en sortie (dans un module dit "client") :

< port-S > ::= < identificateur_de_port > :
< < identificateur_de_message > RESPONSE
< identificateur_de_message >$^{+}$ (;)

où le premier message désigne le message émis et le second le message reçu comme réponse du premier ;

(c) - des ports de sortie :

< port-s > ::= < identificateur_de_port > : < < identificateur_de_message >
[ANTICIPATION] >$^{+}$ (;)

le message ici désigne le message émis. On précise éventuellement l'anticipation (c'est le cas des types de communication "notification" et "commande").

(d) - des ports d'entrée :

< port-e > ::= < identificateur_de_port > : < < identificateur_de_message >
[ANTICIPATION $\left\{\begin{matrix}\text{FIFO}\\ \text{LIFO}\\ \text{RAND}\end{matrix}\right\}$] >$^{+}$ (;)

le message désigne dans cette expression le message reçu ; si la communication est définie avec anticipation, on peut alors préciser une politique de prélévement des messages à la réception.

Les politiques retenues sont :

- une politique déterministe : c'est-à-dire qu'en fonction de certains critères connus statiquement "on sélectionne" un élément plutôt qu'un autre. (les autres éléments ne sont pas perdus). Les deux politiques retenues sont :

a - "Premier arrivé - premier servi" (FIFO) : l'ordre de consommation est le même que l'ordre d'arrivée ;

b - "Dernier arrivé - premier servi" (LIFO) : (ordre de consommation est l'ordre inverse de l'ordre d'arrivée.

- Une politique indéterministe (RAND) : un élément est pris au hasard parmi le reste à la réception.

Tous les messages aux ports d'entrée sont consommés (consommation dans le sens défini par THOMESSE dans [THO 80]).

5.2 Description du comportement du module

Nous distinguons trois parties dans le comportement du module :

(A) - La première décrit un ensemble de comportements élémentaires, dont chacun donne une description séquentielle d'actions de communication. (Les actions de communication sont typées dans le sens du chapitre IV) ;

(B) - La seconde donne le comportement global du module en termes de relations entre les comportements élémentaires (essentiellement le parallélisme et la préemption logique définie au paragraphe 5.2.2).

(C) - La dernière précise, pour chaque comportement élémentaire décrit précédemment, les relations devant exister entre les valeurs émises et les valeurs reçues. Cette dernière partie décrit les conditions locales, au module, entre les émissions, et les réceptions. (Une condition peut être décrite uniquement dans un comportement élémentaire, ou être le résultat de conditions exprimées dans des comportements élémentaires

différents). Cette partie est appelée "contraintes" et est décrite uniquement textuellement en français.

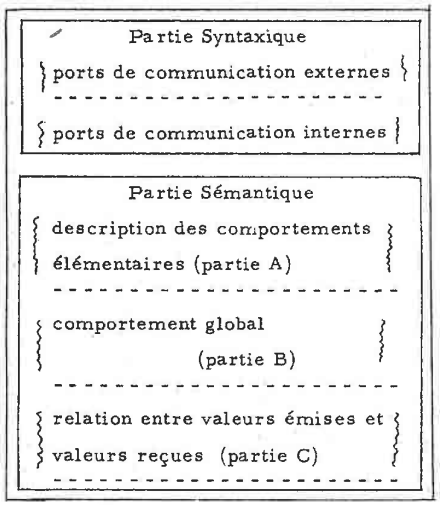


Figure 7 : schéma récapitulatif des différentes parties intervenant dans la description d'un module.

Nous présentons ci-dessus un schéma récapitulatif des différentes unités intervenant dans la description d'un module :

Nous détaillons les deux premières parties A et B dans les paragraphes suivants (respectivement 5.2.1 et 5.2.2).

5.2.1 Les comportements élémentaires

Chaque comportement élémentaire précise les ports d'entrée et les ports de service utilisés en entrée. Il décrit ensuite sa séquence d'actions de communication sur ces ports, et les ports de sortie du module. Un port d'entrée appartient à un comportement élémentaire. L'erreur d'utilisation d'un port d'entrée par plusieurs comportements est détectée statiquement.

syntaxe : < comportement_élémentaire > ::=
< identificateur_de_comportement > : < port-e >$^{+(,)}$; < port-E >$^{+(;)}$
< description >

où < description > est une description séquentielle d'actions de communication. Nous présenterons ci-après les éléments essentiels du langage pour la description de comportements élémentaires. (Les expressions cycliques sont présentées uniquement en annexe).

a - La causalité "( ) : "

Les ":" placés après une condition évaluée (exprimée entre parenthèses) expriment que cette condition constitue une cause à effet pour une action désignée.

syntaxe : (< condition-évaluée >) : (< action >
[| < action >])

Si la condition porte sur une communication, elle est alors évaluée suivant le type de communication bloquante ou non bloquante (avec attente ou sans attente à la réception ou à l'émission).

Exemple : ((pe, T2) ? # pression) : (... ;
(ps, T1) ! # alarme
... ;)

Ceci signifie que tant que l'occurrence du message pression n'est pas parvenue, on ne peut effectuer l'émission d'une occurrence du message alarme. (Ceci rejoint la relation de précedence causale entre réception et émission décrite dans [RAY 81]).

Dans le cas où une deuxième action apparaît dans une expression de causalité, elle est alors effectuée si l'évaluation de la condition a échoué (l'évaluation donne un résultat vrai ou faux comme l'évaluation d'une expression booléenne).

b - Le choix déterministe "◇"

Dans le cas où le comportement élémentaire peut avoir n alternatives possibles (avec $n \leq 2$), une alternative et une seule est choisie parmi elles. Ces alternatives s'excluent mutuellement entre elles. L'opérateur "◇" indique ce choix.

Syntaxe :

◇ (opérande)
< valeur-d'opérande > : < action >
|
◇ << valeur-d'opérande > : < action >> + (|)

Exemple : Si # pression est une occurrence de message du type (basse, normale, haute) ; le choix déterministe permet l'expression suivante à la suite d'une réception :

```
(? # pression) : (◇  # pression
                    haute : ...        % les ... décrivent
                      |
                  normale : ...        % une action associée
                      |
                    haute : ...        % à chaque cas.
                      ◇
                  )
```

Le choix déterministe peut être représenté dans ce cas dans un langage de la famille PASCAL par une structure de contrôle

CASE ... OF ... END CASE.

c - Le choix indéterministe "[]"

Une expression construite par "[]" désigne le choix d'une alternative parmi n ($n \geq 2$) ; ces alternatives ne s'excluent pas mutuellement

entre elles. Dans le cas où plusieurs alternatives peuvent être effectuées, une et une seule est choisie de façon indéterministe.

Une alternative est associée ici à une expression gardée.

Une expression gardée portant sur une communication, exprime soit une réception, soit une émission gardée.

Syntaxe :

[]
< alternative >
|
<< alternative >> + (1)
[]

une alternative est : < condition-évaluée > : < action >
où < action > est ici une description séquentielle.

Exemple : réception indéterministe des messages : pression, température et taux-oxygène ; cette réception est du type "commande" sur le même port.

```
[]
 ? # pression : ... ; ! # message-opérateur
                 |
 ? # température : ...;! # message-opérateur
                 |
 ? # taux-oxygène : ... ; ! # message-opérateur
[]
```

Si tous les messages pression, température et taux-oxygène sont arrivés, alors une réception est choisie de façon indéterministe.

La représentation de ce choix indéterministe est possible dans des langages tels que CSP (c'est le même indéterminisme noté []) ou ADA (l'expression SELECT).

d - Notion de priorité externe

Dans certains cas, le comportement élémentaire doit prendre en considération une priorité de réception d'un message parmi plusieurs pour satisfaire une contrainte externe.

Syntaxe : Priorité < réception > > Priorité (<< réception >>$^{+(;)}$)

Exemple : reprenons l'exemple précédent ; pour spécifier que la réception du # taux-oxygène doit être privilégiée par rapport aux autres réceptions ; nous écrivons (la priorité est énoncée syntaxiquement après l'identification du comportement élémentaire et les ports d'entrées ; cf. paragraphe 5.2.1) :

Priorité (? # taux-oxygène) > Priorité (? # température,
...
? # pression)

□

? # pression : ... ! # message-opérateur
|
? # température : ... ! # message-opérateur
|
? # taux-oxygène : ... ! # message-opérateur

□

5.2.2 Le comportement global

Dans le comportement global du module, on énonce le parallélisme entre comportements élémentaires et éventuellement la préemption logique que nous définissons ci-dessous.

Notion de préemption logique

L'un des grands intérêts de l'ensemble des définitions que peut procurer un module à ce niveau est la préemption que nous appelons logique, car elle n'est pas liée à un processeur particulier.

Il s'agit de préciser au niveau du module qu'à la suite d'une réception d'un message, un comportement élémentaire (dit "requérant") doit non seulement être immédiatement effectué, mais doit aussi entraîner auparavant l'arrêt des autres comportements, si c'est nécessaire.

Ces comportements peuvent être en attente de communication (bloqués), ou en train d'effectuer séquentiellement d'autres actions.

Cette description est nécessaire dans le cas de traitements d'urgence.

Syntaxe : Préemption de < identificateur_de_comportement > sur
(<< identificateur_de_comportement >>$^{+(,)}$)

Exemple : Dans un réseau de modules effectuant la commande d'un ascenseur un module contrôle le moteur agissant sur la cage de cet ascenseur, suivant le schéma simplifié ci-dessous.

Sachant que l'arrivée du message stop doit provoquer l'arrêt de la cage à n'importe quel moment, la description du comportement du module est donnée par la description de deux comportements :

- l'un appelé "monter-descendre", est attaché aux ports d'entrée DEMANDE et NIVEAU ;
- l'autre appelé "arrêt-urgent" est attaché au port URGENCE

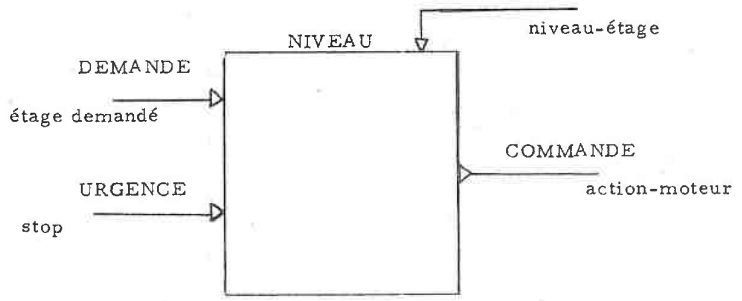


Figure 8 : Schéma d'un module avec ses ports d'entrée et de sortie

Le comportement du module s'écrit en trois parties : les comportements élémentaires, le comportement global et les contraintes sur les messages échangés :

FUNCTION
Monter-descendre : DEMANDE, NIVEAU ;
(DEMANDE ? # étage-demandé) : COMMANDE ! marche
; % envoi d'un ordre de démarrage
jusqu'à égalité (# étage-demandé, # niveau-étage)

* (NIVEAU ? # niveau-étage) ;
(égalité (# étage-demandé, # niveau-étage)) :
COMMANDE ! arrêt
% envoi d'un ordre d'arrêt

arrêt-urgent : URGENCE ;
(URGENCE ? # STOP) : COMMANDE ! arrêt
% envoi d'un ordre d'arrêt.

Comportement global

Monter-descendre // arrête-urgent.
préemption arrêt-urgent sur monter-descendre

Contraintes

étage demandé est une valeur entière qui est comparée à niveau-étage ;
égalité (# étage-demandé, # niveau-étage) délivre un résultat vrai ou faux.
END FUNCTION

action-moteur est un message du type scalaire (marche, arrêt) ;

Le comportement du module précédent, exprime qu'à la réception du message stop, le comportement élémentaire monter-descendre est arrêté, et le comportement arrêt-urgent est effectué pour l'arrêt de la cage de l'ascenseur.

V - 6 CONCLUSION

Dans le langage FLEXI on peut vérifier la cohérence (consistance), et la complétude des différents éléments de la description de l'application :

a - Cohérence

. sur les modules : un module n'utilise pas de définition non consistante concernant ses ports et ses messages ; n'utlise pas de types de communication de façon non consistante sur les ports de communication (exemples : réception sur un port de sortie, utilisation d'un type non conforme sur un port de communication) ; n'apparaît pas dans une connexion non valide ; ne peut être considéré comme externe et interne à la fois (ceci est utilisé conjointement avec les contrôles statiques établis lors des vérifications du comportement global de l'application) ;

. sur les ports : un port, ne peut appartenir à deux connexions de base différentes et est utilisé de façon consistante dans les connexions (contrôles sur les types échangés) ;

. sur les messages : on vérifie que seuls les messages externes peuvent apparaître dans le comportement global de l'application (ceci est vérifié conjointement avec la liste des modules) ;

. sur les types de communication : conformité d'utilisation des types de communication pris deux à deux sur une connexion (exemple : une émission du type "commande" n'est pas compatible avec une réception du type "notification") ;

. sur les connexions : toute connexion est valide si elle utilise des définitions consistantes de messages et de ports ;

b - Complétude :

. tous les messages définis sont utilisés ;

. toute connexion n'a pas de port ou de message qui ne soit pas utilisé dans les spécifications des modules.

. L'ensemble des modules respecte les contraintes globales décrites dans le niveau application (cf. V.4.4).

Cette spécification est abstraite dans le sens qu'on y a pas fait d'hypothèse sur les choix de réalisation des modules et des communications.

L'avantage d'une telle approche réside en trois parties essentielles :

1 - elle définit une vue globale de l'application ;

2 - elle rend possible l'évolutivité des différents éléments de l'application ; en effet, on peut changer la représentation d'un module, par une autre sans pour autant toucher au reste de l'application [DEZ - 83-b].

On peut de même (dans les cas où des modifications ne concernent que les communications), localiser une modification portant sur la communication uniquement et agir aux niveaux :

. des accès : pour changer de type de communication ou de politique de consommation de messages ;

. des connexions : pour changer de "routage" de message ;

sans pour autant modifier à chaque fois les autres éléments intervenant dans une communication.

Ceci grâce à la séparation de la description des chemins de transfert de messages des actions de communication, et à la non désignation des interlocuteurs dans toute communication.

3 - Elle permet à l'étape de programmation de ne s'intéresser qu'à la représentation d'un seul module à la fois (ceci réduit considérablement la complexité de réalisation de l'application).

CHAPITRE VI

L'ETAPE DE PROGRAMMATION

VI - 1 OBJECTIFS DE L'ETAPE DE PROGRAMMATION

Deux objectifs essentiels sont visés dans cette étape :

- le premier, est de proposer des outils adéquats qui vont dans le sens des concepts esquissés précédemment (cf. paragraphe 2, figure 1). Il s'agit essentiellement du choix d'expressions de capsule et des primitives de communications.

- Le second, est de montrer le passage entre l'étape de conception et celle de programmation (cf. figure 2, page 86) grâce à l'utilisation d'un noyau de communications (cf. exécutif réparti [SCE 80]).

Ceci n'est possible que par adjonction d'autres informations concernant le choix de répartition des capsules sur les processeurs disponibles (c'est l'objet du paragraphe 3.2). Nous utilisons le terme "configuration" pour désigner une répartition donnée.

Du point de vue programmation, nous avons choisi le terme capsule pour souligner le fait, qu'une telle unité permet la protection de ses objets internes, en interdisant tout accès à ces objets, autre que celui prévu par les ports de son interface (le module).

VI - 2 DEFINITION DE LA CAPSULE

La capsule est constituée de deux parties : un interface, et un corps. L'interface de la capsule est réalisé par la liste de ports de communication, précédemment définis par le module auxquels on adjoint des précisions concernant l'anticipation. En effet, pour chaque type de message on précise le nombre (valeur entière) d'anticipations autorisées.

Le corps de la capsule est (un "code") écrit dans un langage L (celui de la machine hôte), de façon conforme à la description donnée dans

la partie sémantique du module à l'étape précédente :

. D'une part, les types de communication sont représentés ici, par des primitives SEND et RECEIVE, constituant ainsi une extension du langage L ;

. D'autre part, les descriptions de "//", de choix déterministe ou indéterministe, de priorité externe et de préemption logique doivent trouver leurs représentations dans le langage L (supposé ici, accéder au noyau temps réel de la machine cible).

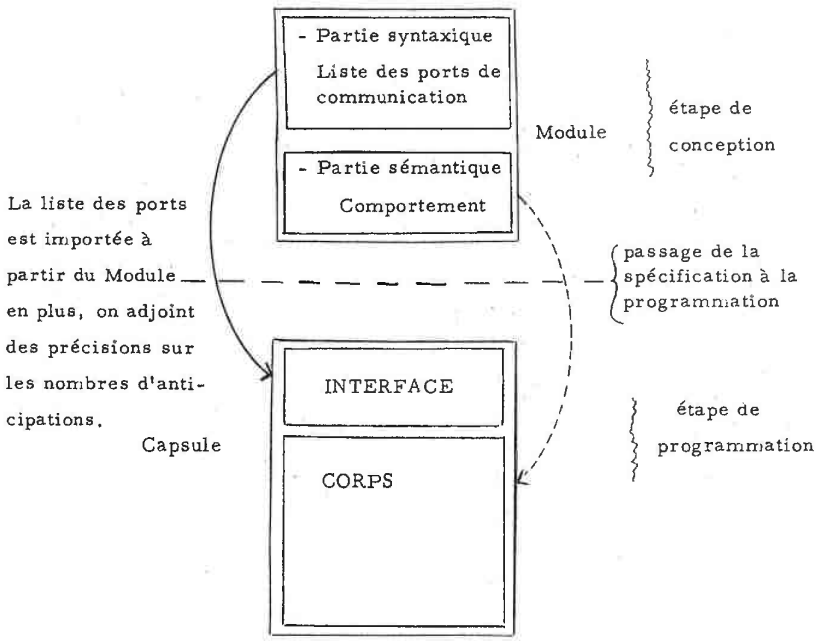


Figure 1

Le corps de la capsule peut être constitué d'une tâche (cf. paragraphe 3.3.1), ou de plusieurs tâches parallèles se partageant des objets locaux à cette capsule.

VI - 3 LE PASSAGE DE LA SPECIFICATION A LA PROGRAMMATION

Il s'agit de la représentation des différents éléments obtenus à l'étape de spécification (ou langage FLEXI), dans le langage de programmation. Le choix adopté ici est d'étendre un langage* de programmation existant pour exprimer des structures de contrôle nécessaires à la communication dans le domaine temps réel. L'hétérogénéité des langages reste donc possible, mais cela pose des problèmes d'unicité de représentation de types de messages et de certaines structures de contrôle telles qu'elles doivent apparaître dans les tâches.

Nous supposons par la suite que le langage de programmation (noté L) est unique sur l'ensemble des sites.

Le passage de l'étape de spécification à l'étape de programmation se fait comme suite (cf. figure 2) :

- la liste des messages décrite en FLEXI doit être représentée dans le langage L ; il s'agit à ce niveau d'un ensemble explicité de types. Ils serviront de base aux déclarations de variables (messages) pouvant apparaître dans les primitives de communication au niveau des tâches (cf. relation (1) figure 2) ;

- la liste des modules est représentée par la liste des capsules (cf. relation (2)), l'interface de chaque capsule est la partie syntaxique du module correspondant (plus des précisions sur les anticipations de messages). Les tâches d'une capsule sont écrites de façon conforme à la partie sémantique du module ; particulièrement les primitives de

---

* Un exemple d'extension du langage SIMONE est donné dans [GOF 83] .

communication (cf. paragraphe 6) représentent les types de communication : "notification", "commande" et "service". (cf. relation (3) figure 2).

- Une liste apparait au niveau programmation : celle de répartition de capsules sur les machines cibles (cf. paragraphe 3.2 ; (2) figure 2). Elle désigne pour chaque machine les capsules qui s'y exécutent ; pour chaque capsule, on indique le module qu'elle représente. Chaque machine est identifiée de façon unique ; elle est une unité perçue par le système de communication chargé d'acheminer les messages entre les sites.

La génération d'un ensemble de table de routage [MAM 82], (structures nécessaires au noyau de communication à l'exécution (cf. (5) et (6) figure 2) sera possible à partir de la liste de répartition des capsules et de la liste des connexions entre modules (cette dernière est interprétée à ce niveau par les connexions entre capsules).

Légende de la figure 2

(1) est la relation "réalise" ; à chaque message correspond un type écrit en L ;

(2) est la relation "représente sur la machine X" ; liant une capsule et son module ;

(3) Cette relation décrit la relation précédente : chaque partie syntaxique d'un module devient l'interface de la capsule, le corps de cette dernière doit être programmé de façon conforme à la description donnée dans la partie sémantique du module.

(4) vérification des contraintes globales sur les capsules (essentiellement les contraintes temps réel du type "la réponse doit parvenir avant la fin de Δ t").

(5) et (6) représentation des connexions entre capsules qui permettra la génération d'une table de routages à l'exécution ;

(7) et (8) chargement effectif de chaque capsule sur la machine à laquelle elle est affectée ;

Phase d'exécution :

C : capsule

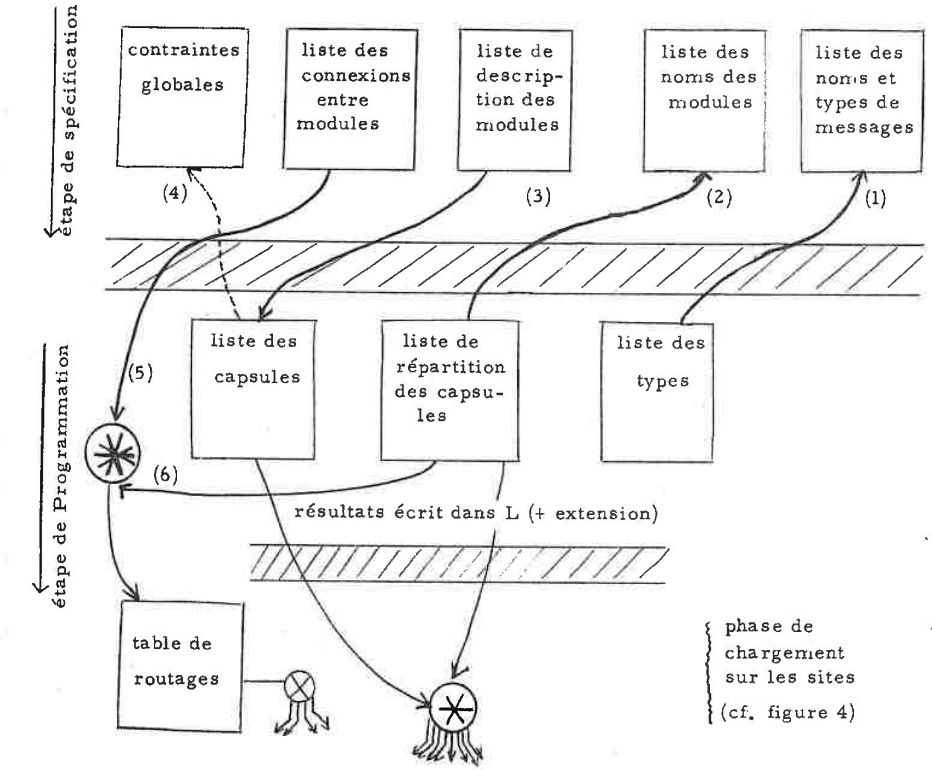
NC : noyau de communication

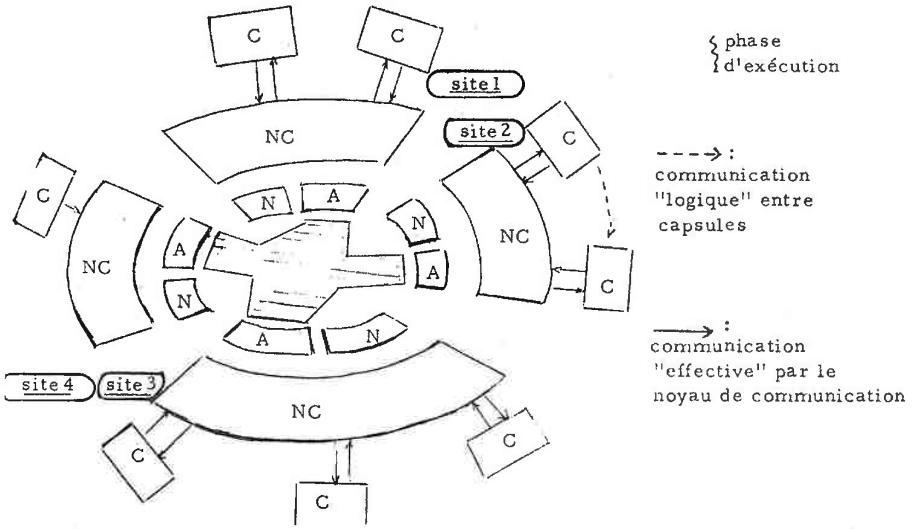
N : noyau temps réel d'une machine cible

A : agence de communication (cf. station de transport dans [MAM 82])

La partie hachurée correspond aux autres niveaux d'un système ouvert [COR 81] ;
(sauf ceux de présentation et de session [SLO 81] [LOR 79]).

Résultats de spécification écrits en FLEXI





Nous décrivons dans les sous-paragraphes suivants chacune des listes du niveau programmation. La liste des types de message (3.1), la liste des répartitions (3.2), et finalement la liste des capsules (3.3).

3.1 Explicitation de la liste des types de messages :

Les types de messages définis à l'étape de conception sont représentés à cette étape par les types du langage L.

Le texte contenant la liste des types de message constitue une unité de déclaration séparée des autres textes et sa représentation est un choix de mise en œuvre (exemple : les types peuvent être représentés en totalité ou non sur les sites). De même que dans l'étape précédente la liste des types de messages s'exprime par MESSAGES TYPES et END MESSAGES.

Remarque :

Pour des raisons de cohérence nous gardons les mêmes noms de messages pour exprimer les types.

3.2 Description de la répartition des capsules

Le but est de décrire d'un point de vue déclaratif la répartition de l'ensemble des capsules sur l'ensemble des processeurs disponibles.

L'intérêt de cette déclaration est de générer à l'exécution une table de routage de message sur chaque site.

Le regroupement et l'emplacement des capsules est fait en fonction de critères physiques pour minimiser le temps de réponse du système de commande (ce type de contraintes peut parvenir de la description globale de l'application).

Syntaxe : CONFIGURATION

< MACHINE < identificateur_de_machine > ;

<< identificateur_de_capsule > : < identificateur_de_module >>$^{+(;)}$

ENDMACHINE >$^{+(;)}$

ENDCONFIGURATION

Exemple : Répartition de trois capsules sur deux sites. (cf. annexe) :

```
CONFIGURATION  MACHINE processeur-1- : dialogue-opérateur :
                        interface-opérateur
               ENDMACHINE
               MACHINE processeur 2 : commande-moteur : commande ;
                                      veilleur : environnement ;
               ENDMACHINE
ENDCONFIGURATION.
```

3.3 Explicitation de la liste des capsules

Dans ce paragraphe, nous décrivons d'abord l'expression d'une capsule (syntaxique et sémantique - paragraphes 3.3.1 à 3.3.4), ensuite, nous présenterons la liste des capsules (paragraphe 3.3.5).

3.3.1 Expression syntaxique d'une capsule

Le texte d'une capsule se présente en deux parties (cf. figure 3) ;

- la première contient :
  - l'interface de la capsule : ensemble de ports déjà décrits par la partie syntaxique du module correspondant dont on précise pour chacun le nombre d'anticipation pour chaque type de message ;

rappelons que jusqu'alors, on avait autorisé (ou non) l'anticipation et on avait indiqué une politique donnée ;

  - l'ensemble des ressources partagées par les tâches de cette capsule : variables globales, procédures, moniteurs, etc...

- La deuxième est constituée de la suite des textes des tâches, pour chacune on indique les ports d'entrée sur lesquels elle peut attendre des messages.

Un choix est pris à ce niveau concernant l'affectation d'un port d'entrée à une seule tâche.

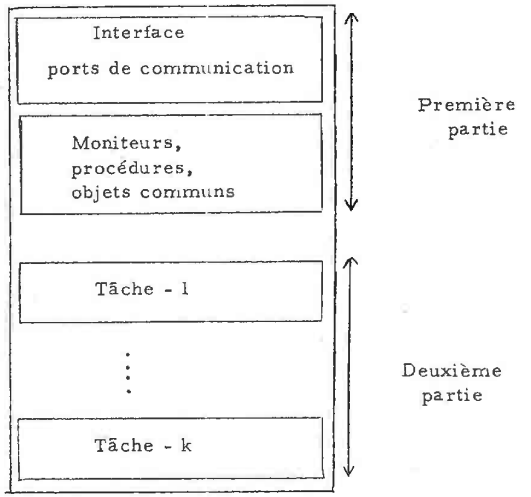


Figure 3

3.3.2 Les ports

Plusieurs choix de représentation concernant les ports sont à considérer à ce niveau.

En effet, à un port est associé plusieurs types de messages ; pour chaque type nous associerons une liste bornée sur laquelle une

primitive d'envoi aura pour conséquence d'ajouter le message, et la primitive de réception aura pour conséquence de retirer le message de cette liste (suivant la politique définie).

3.3.3 Choix de représentation et politique de communication

La taille de la liste, associée à un type de message, est une valeur entière devant être précisée pour expliciter le choix d'anticipation, ou de non anticipation, cette valeur est :

- soit strictement positive, elle indique le nombre d'anticipation possible ; la politique permet la sélection d'un élément dans la liste (communication asynchrone) ;
- soit nulle, aucune anticipation n'est possible, et on ne peut définir de politique de communication dans ce cas. (La communication est synchrone ou asynchrone suivant l'échange globalement établi entre l'émetteur et le récepteur). La précision concernant l'anticipation est faite syntaxiquement en remplaçant le mot clé ANTICIPATION (cf. chapitre V) par un nombre exprimé entre parenthèses

< identificateur_de_port > : < type_de_message > [(< nombre >)[{ FIFO / LIFO / RAND }]]

3.3.4 Les tâches

Une tâche est l'unité d'exécution perçue par le noyau temps réel de la machine hôte. Le texte d'une tâche exprime une activité séquentielle.

Les tâches définies dans une capsule coopèrent pour la réalisation d'un but ou ensemble de fonctionnalités communes. En ce sens, la capsule est à rapprocher d'autres définitions telles que : la "Tâche" dans

SYGARE [THO 80], "Agence" dans SCEPTRE [SCE 80] ou "Task-force" de Médusa [JOH 80]. Nous ne nous sommes pas intéressés, à ce niveau, aux communications et synchronisation entre tâches de la même capsule ; par contre, nous définissons ci-dessous des outils de communication entre cette dernière et son environnement (externe - physique - ou avec les autres capsules de l'application).

Ceci est justifié dans notre démarche de construction de l'application par les faits suivants :

- si nous avons pris le choix d'exprimer des activités coopérantes internes à une capsule, c'est parce qu'elles sont très fortement liées (interdépendance logique, fonctionnelle...), à l'opposé de l'expression de l'application sous forme de capsules, qui elles doivent être le moins dépendantes possible ;
- l'unité de construction de l'application étant la capsule en ce sens que si une modification s'avère nécessaire, elle s'effectue sur toute l'unité.

3.3.5 Définition de liste des capsules

De même que les listes précédentes (paragraphes 3.1 et 3.2), cette liste est décrite de façon autonôme. Chaque capsule y est décrite par son texte suivant le schéma donné dans le paragraphe 3.3.1.

Syntaxe :

CAPSULES :
< CAPSULE : < identificateur_de_capsule >
< texte_de_capsule >
END CAPSULE > $^{+(;)}$
END CAPSULES.

VI - 4 DESCRIPTION DE LA MISE EN ŒUVRE

Plusieurs choix sont possibles à ce niveau ; deux choix ont été développés en se basant sur les concepts présentés ici :

- dans le premier [GOF 84] , on a montré la faisabilité du passage de la spécification à la simulation grâce à l'utilisation d'un préprocesseur. La simulation est effectuée en SIMONE, et la communication est réalisée à l'aide de moniteurs.

- Dans le deuxième [MAM 82] , on décrit une spécification interne d'un noyau de communication supporté par un noyau SCEPTRE [SCE 80] , et une station de transport (CYCLADES).

  D'autres descriptions d'implantation de travaux voisins des nôtres sont décrites dans [GUI 82] , [KRA 83] .

  La principale démarche est l'utilisation d'un préprocesseur pour génèrer des structures nécessaires avant l'étape de compilation (en SIMONE dans [GOF 83] , et en PASCAL dans [MAM 82] , [KRA 83] et [GUI 82]).

  Les textes définis ci-dessus (liste des types de messages, de capsules et de répartition de capsules) à l'étape de programmation, ainsi qu'une représentation* de la liste des connexions, constituent les entrées du préprocesseur ; les résultats sont un ensemble de tables (représentation de la configuration avec pour chaque capsule des représentations des ports), et une génération du code pour les primitives de communications permettant ainsi, l'accès au noyau temps réel de la machine support (cf. figure 4).

---

* Dans le cas où il existe une bijection entre l'ensemble des modules, et l'ensemble des capsules, c'est un cas simple, on aura une substitution des noms.

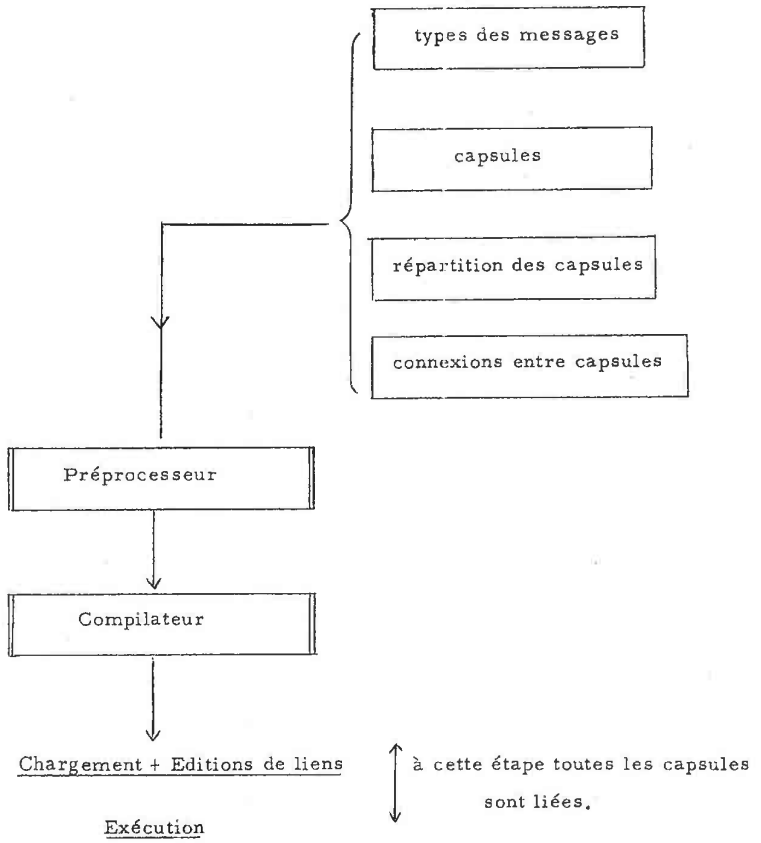


Figure 4

VI - 5 LES PRIMITIVES DE COMMUNICATION

Les primitives proposées seront donc la représentation des types T1, T2 et T3, tels qu'ils étaient définis dans le chapitre IV.

5.1 Le premier type de communication ("notification")

5.1.1 L'envoi

L'envoi est non bloquant ; après la prise en compte de la primitive d'envoi par le noyau de communication, la tâche exécutant cette primitive est débloquée immédiatement pour continuer son activité. La syntaxe proposée est la suivante, où le préfixe "N" rappelle une notification :

N-SEND < message > FROM < port-sortie > ;

< message > est le nom d'une variable dont le type est l'un des types spécifiés au port de sortie, < port-sortie >, et sur lequel agit la tâche en déposant un message pour l'émission.

5.1.2 La réception

syntaxe :

N-RECEIVE < message > ON < port-entrée >

où < message > à le même sens que précédemment, et < port-entrée > est le nom du port d'entrée dans lequel est défini le type de message. Cette expression signifie que si le message est dans la liste (représentant le port), la valeur est alors prélevée et est affectée à l'identificateur < message >, sinon la tâche continue son activité (non bloquée).

5.2 Le deuxième type de communication ("commande")

C'est ce qui correspond dans le domaine des applications industrielles à des "commandes" et permet d'exprimer aussi d'autres besoins, tels que le traitement d'erreurs dans le cas où l'acheminement du message n'a pas eu lieu. Nous avons pu constater (chapitre IV) que

dans ce type de communication, nous avons le choix entre échanges synchrones et asynchrones.

Ici, un choix de représentation syntaxique s'offre pour les deux types d'échanges ; nous avons préféré adopter la même expression linguistique (pour le même type de communication), et exprimer la différence de synchronisation au niveau de la représentation du port.

Comme la déclaration d'un port est reprise à ce niveau de programmation en y rajoutant une précision concernant le nombre d'anticipations de messages à l'envoi :

- un nombre strictement positif correspond à une communication asynchrone ;
- un nombre nul correspond à une communication synchrone.

Ceci est purement un choix de représentation.

5.2.1 L'envoi

L'envoi est bloquant dès que l'on ne peut anticiper par une nouvelle émission (ceci est explicite dans le deuxième cas ci-dessus).

L'expression syntaxique est la suivante :

SEND < Message > FROM < port-sortie > [TIMEOUT (p)]

où < message > et < port-sortie > sont respectivement les identificateurs du message envoyé et du port sur lequel porte la primitive de communication. p exprime la durée d'attente maximale d'un acquittement. (Ceci constitue la définition de TIMEOUT).

5.2.2 La réception

La réception associée est bloquante, mais une temporisation éventuelle est offerte pour éviter une attente infinie.

Elle s'exprime syntaxiquement sous la forme :

RECEIVE < Message > ON < port-entrée > [TIMEOUT (p)]

< message > et < port-entrée > sont des identificateurs comme précédemment. Contrairement à l'envoi, la temporisation ici est relative à la réception de message, et non à un acquittement.

Sémantiquement cela signifie :

- si l'option TIMEOUT n'est pas présente, cette primitive est bloquante jusqu'à réception du message (cela peut entraîner une attente infinie).
- si l'option est employée :

a. si le message est dans la liste représentant le port d'entrée il est prélevé, et la tâche réceptrice continue son activité ;

b. si le message n'est pas encore reçu, la tâche est bloquée et une temporisation de p unité de temps est déclenchée :
   - si le message parvient, par la suite (pendant l'attente), il est alors prélevé et la tâche est débloquée ;
   - si la durée d'attente p est expirée ; plusieurs choix de mise en œuvre sont possibles, nous avons pris le choix suivant : si l'utilisateur met l'option TIMEOUT, il doit envisager un traitement "d'erreur" par la structure d'une garde :

```
RECEIVE...TIMEOUT ( ) : (
                          S1  % traitement normal %
                        |
                          S2  % traitement d'erreur
                                de communication %
                        )
```


On peut statiquement vérifier de telles structures de contrôle, c'est-à-dire, détecter une erreur à la compilation si le TIMEOUT est présent et la garde non exprimée.

5.3 Le troisième type de communication

Ce type représente bien un appel de procédure à distance [LIS 82] (cf. chapitre V). Le but essentiel ici est d'exprimer cette communication comme une action atomique, qui peut se rapprocher linguistiquement du côté source d'un appel de procédure et sémantiquement du côté puits d'une structure parenthésée (structure de bloc).

5.3.1 L'envoi

L'envoi, que nous appellerons dans ce type de communication une demande de service est donc bloquant jusqu'à réception de la réponse.

L'expression syntaxique est dans ce cas :

S - SEND < Message-1 > FROM < port-service > RESPONSE < message-2

où < message-1 > et < message-2 > sont respectivement les messages envoyés et reçus dont les types étaient déclarés au niveau du port < port-service > ; ce dernier figurait dans l'interface sous la clause EXITENTRYPORT.

Pour éviter une attente infinie une temporisation est ajoutée, ceci a pour effet de permettre son emploi dans une garde :

S - SEND < message-1 > FROM < port-service > W-RESPONSE < message-2

```
TIMEOUT (p) ... :   (
                    % traitement normal %
                    S1
                    |
                    % traitement dans le cas où la
                      primitive a échoué %
                    S2
                    )
```

Dans ce cas, si le délai p est expiré avant que la réponse arrive S2 est exécuté. On peut détecter de même statiquement l'erreur consistant à employer un timeout sans mettre une structure de garde.

5.3.2 La réception

Nous exprimons la réception et le traitement du service comme un seul bloc monolithique (cf. expression atomique dans [LIS 82]). De même que dans le cas précédent, ou bien le serveur attend indéfiniment la demande de service, ou bien il exprime une temporisation dans une structure gardée pour un traitement d'erreur de communication.

- La primitive s'écrit donc dans le premier cas :

```
S-RECEIVE < message-1 > ON < port-service >
DO
 % traitement du service %
 :
 :
END
REPLY < message-2 >
```


Le port < port-service > a une expression duale, si elle est exprimée dans le port de la primitive de demande de service ; il doit être déclaré en ENTRY EXIT PORT ; le type de message < message-1 > y est explicité en entrée et le type de message < message-2 > y est explicité en sortie après la clause REPLY.

- Dans le deuxième cas, une attente infinie peut être évitée grâce à l'expression :

```
S-RECEIVE < message-1 > ON < port-service > TIMEOUT (p) :
    ( DO
      % traitement du service %
      :
      :
      END
      REPLY < message-2 >
    |
      S2
      % traitement dans le cas où la demande < message-1 >
        n'est pas parvenue %
    )
```

Le TIMEOUT ici porte sur la réception de la demande.

VI - 6 REGLES D'UTILISATION DE PORTS ET DES PRIMITIVES PAR LES TACHES

Il s'agit de certaines règles d'utilisation des primitives de communication par les tâches, ainsi que des ports d'entrée, de sortie et de service. Nous présentons ci-dessous nos choix :

- Un port d'entrée correspond à un traitement particulier et doit être assuré par une seule tâche. L'accès donc à un port d'entrée doit être pour le compte d'une tâche unique ayant déclaré ce port dans l'ensemble des ports accédés. Un conflit d'accès à un port d'entrée est détecté statiquement grâce à l'expression syntaxique de la tâche. (La tâche doit déclarer ses ports d'entrée).

- Un port de sortie peut être partagé par plusieurs tâches de la même capsule, il sera alors utilisé en exclusion mutuelle.

- Un port de service n'appartient qu'à une seule tâche.

VI - 7 CHOIX D'IMPLANTATION DES PRIMITIVES

Au niveau de la mise en œuvre du noyau de communication, assurant ainsi, l'implantation des primitives de communication, telles qu'elles sont précisées ci-dessus (voir les travaux de GOFFIC [GOF 84] sur une implantation orientée simulation, ainsi que les travaux de MAMMERI [MAM 82], sur une implantation des mêmes concepts pour la machine MUGUET utilisant les primitives du noyau SCEPTRE [SCE 80]), plusieurs choix d'implantation sont possibles. Les mécanismes d'adressage au niveau de l'envoi d'un message, peuvent être par le contenu du message (par recopie dans un buffer) ou seulement par le passage de l'adresse du message : le buffer lui-même peut être d'une structure simple (cf. chapitre III.3), ne contenant qu'un seul message ou de structure multiple pouvant permettre ainsi d'anticiper par plusieurs envois à l'avance. Le mécanisme par recopie de message est lent et peut demander une gestion non négligeable du buffer, au détriment d'un délai d'échange de message, par contre le mécanisme par échange d'adresse et plus rapide, mais à priori, bloquant pour un envoi.

VI - 8 PRIMITIVES ET CONNEXIONS : ETUDE PAR CAS

Les connexions correspondent à des "schémas" de communication bien définis, comme une primitive de communication à un sens précis, on ne peut utiliser n'importe quelle primitive sur n'importe quelle connexion ; l'exemple d'une demande de service sur une connexion en diffusion (ex. opérateur " &") justifie l'utilisation restrictive des connexions par les types de communication établis. Cependant, la forme ou la syntaxe de la primitive ne doit pas être transformée pour cette raison (on doit garder la même expression linguistique).

8.1 Cas d'une diffusion (" & ")

C'est le cas où l'opérateur " & " est appliqué à une structure divergente le message est diffusé a tous les destinataires.

1) Les primitives de notification (T1) : la source et les puits s'expriment sémantiquement et syntaxiquement de la même façon que dans un cas de liaison bipoint.

2) Les primitives de commande (T2) : du point de vue des attentes (cf. Chap. IV), au niveau de la source, il est clair que cette primitive n'est pas équivalente à celle d'une suite d'envoi (répété un à un) pour les destinataires. Dans le premier cas, un acquittement n'est considéré que par la présence de tous les acquittements des destinataires.(On bénéficie d'un problème d'exécutions des destinataires). Dans la deuxième par contre, une attente d'un acquittement est nécessaire avant de recommencer un nouvel envoi : le délai de communication total ici peut être très grand comparé au premier :

Exemple : cas d'une commande simultanée sur plusieurs procédés physiques se déroulant en parallèle.

Du point de vue syntaxique l'ensemble des interlocuteurs s'expriment de la même façon que dans un cas simple, du point de vue sémantique seul l'émetteur doit attendre un acquittement global des destinataires. Dans tous les cas, l'utilisateur n'a pas à se soucier des détails impliqués.

3) La demande de service (T3) : cette communication ne correspond pas à cette structure de connexion et est donc interdite pour la diffusion ; l'erreur est detectée au moment de la prise en compte de la connexion, (l'opérateur & ne s'applique pas à des ports de services).

8.2 Cas d'un entrelacement (" , ")

C'est le cas de l'opérateur "," appliqué à une structure convergente. Cela permet un traitement équitable pour l'ensemble des émetteurs.

1) Les primitives de notification (T1) : les sources et puits s'expriment syntaxiquement et sémantiquement de même que dans le cas d'une liaison bipoint.

2) Les primitives de commande (T2) : ce cas permet au puits de traiter les messages reçus sans se soucier de leurs émetteurs ; sémantiquement et syntaxiquement les primitives sont employées de façon identique à un cas de liaison bipoint.

Remarque : ce schéma correspond à plusieurs utilisateurs d'un même port dans les propositions de SILBERCHATZ [SIL 81].

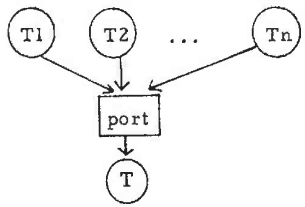


Figure 5

Il faut souligner que dans le cas où la politique est indéterministe, l'intérêt réside dans le fait que l'indéterminisme est alors assuré au niveau du port et non au niveau du processus T. (Contrairement au schéma suivant, T n'utilise sont port qu'en entrée).

3) Demande de service (T3) : De même que dans le cas précédent cela permet au puits de prendre l'une quelconque des demandes de service effectuées par un client. Syntaxiquement, et sémantiquement les primitives s'expriment de la même façon que dans un cas de liaison simple.

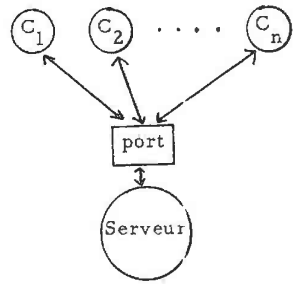


Figure 6

8.3 Cas d'une sélection ("|")

C'est l'application de l'opérateur "|" soit à une structure de divergence. La source alors a un choix d'un, parmi n à l'émission, soit à la convergence. Le puits alors a le choix d'un parmi n à la réception.

1) Les primitives de notification (T1) :
   - dans le cas d'une sélection à l'émission, seul l'émetteur indique en paramètre, l'index de la liaison en postfixant le port de sortie :

N - SEND n FROM S - port-1

où le type de n est déclaré en port de sortie S-port

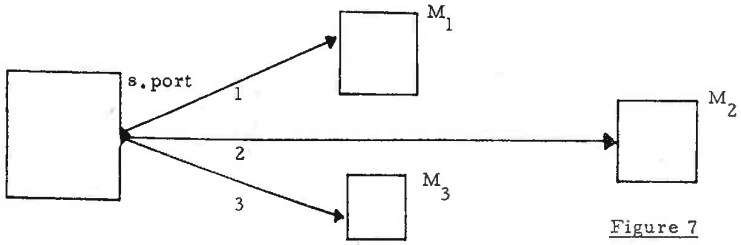


Figure 7

Cette primitive aura donc pour effet l'envoi du message n au seul module M_1. Du côté de ce dernier module la primitive habituelle pour la réception d'une notification est employée (i.e. N-RECEIVE).

- le cas d'une réception est dual à celui-ci :

N-RECEIVE n ON e-port.2

Cette opération a pour effet de recevoir le message n de la part du module M_2 (cf. figure ci-dessous) :

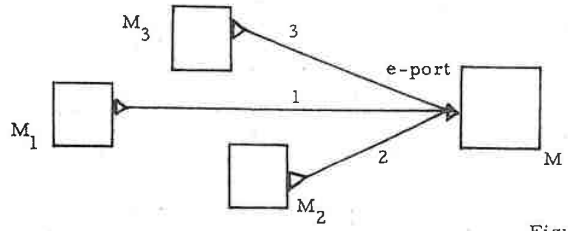


Figure 8

2) Les primitives de commande (T2) :

Celles-ci ont pour effet d'envoyer une commande sur une liaison donnée, ou de recevoir une commande à partir d'une liaison indiquée.

- Dans le cas d'une sélection à l'émission, si nous utilisons le premier schéma ci-dessus, l'envoi d'une commande est :

SEND m FROM S-port.1

- Dans le cas d'une sélection à la réception, nous reprenons le deuxième schéma ci-dessus, la réception d'une commande est la suivante :

RECEIVE m ON e-port-2.

3) Les primitives de demande et de traitement de service (T3) : Ces primitives, dans ces cas de liaisons, permettent soit un choix d'un serveur parmi n à l'envoi ou le choix d'un client parmi n à la réception.

- A l'émission, la sélection d'une demande de service sur une liaison précise, s'effectue sous la syntaxe :

S-SEND n FROM s-port.1 W-RESPONSE m TIMEOUT (p)

s-port ici est un port de service ayant le type de n en envoi, et le type de m en réponse. Du côté puits, tout se passe comme dans le cas d'une liaison simple.

- La réception sélective, permet le choix d'une demande parmi n possibles ; suivant la syntaxe :

```
S-RECEIVE m ON es-port . 2
DO
% service %
:
:
END
REPLY q ;
```

Pour le client tout se passe comme si c'était une liaison simple.

VI - 9 CONCLUSION

1 - Le but poursuivi ici est de cacher à l'utilisateur au niveau d'une capsule la structure des liaisons de l'environnement de cette capsule, de sorte que l'utilisateur ne perçoit cet environnement que par l'intermédiaire des ports et des primitives autorisés.

La sémantique ou la syntaxe d'une primitive ne doit pas changer (ou changer le moins possible), si la structure est amenée à changer. De cette façon les détails décrits au niveau externe n'ont pas à être repris au niveau interne de la capsule.

2 - A l'aide des structures de connexions simples (bipoint, divergence, convergence) et construites, les formes possibles de communication établies à l'aide des opérateurs " &", "|" et "," et les types de communication T1, T2, T3 on peut couvrir une majorité des possibilités de communication dans le domaine des traitements de procédés industriels et des applications temps réel réparties et permettre aussi un maximum de contrôles statiques.

CHAPITRE VII

LA PHASE DE DECOMPOSITION D'UNE APPLICATION

UN EXEMPLE

VII - 1 BUT

Dans cette phase le but est d'exhiber une décomposition de l'application en un réseau de modules communicants. (Ceci est traité par ailleurs dans [LON 83] , [FAZ 84]). Ce réseau est ensuite décrit en FLEXI.

Nous esquissons, ici, des éléments d'aide à la décomposition de l'application pouvant guider l'utilisateur pour prendre des choix de découpage parmi un ensemble de choix possibles.

La démarche est descendante ; l'application est décrite à une première étape comme un seul module, ensuite par raffinements successifs l' on arrive à une dernière étape décrivant une décomposition.

Le concept de communication intervient dans la démarche de décomposition, dans ce sens, des relations de coordination sont introduites entre les actions de communication (les symboles utilisés ici peuvent être considérés comme faisant partie d'un langage de décomposition ou "surlangage" du langage FLEXI).

VII - 2 ELEMENTS D'AIDE A LA DECOMPOSITION

1 - On peut attribuer un module pour la commande d'un procédé physique particulier, à ce niveau le module abstrait un système de commande d'une machine industrielle (ceci est propre à la classe d'applications de commande de procédés) ;

2 - Un module est affecté à la gestion d'une ressource informatique particulière contre tout accès non autorisé (autre que par la communication) ; le module agit ici, comme un gardien dans le sens de LISKOV [LIS 79] dans un milieu réparti ;

3 - Une sortie externe ne peut être attribuée qu'à un seul module (le module peut avoir éventuellement plusieurs sorties).

4 - L'analyse des relations de coordination entre messages contribue dans la démarche de décomposition ; il s'avère donc nécessaire d'analyser toutes les relations entre les actions de communication (de parallélisme, de précédences causales) pour préciser les interfaces de communication (pour préciser sur quel module arrive le message analysé), et par là préciser une décomposition donnée.

C'est dans ce sens que nous introduisons la relation d'indépendance (notée Ind) ; liant deux actions de communication. Elle signifie qu'il n'y a pas de relation de causalité entre elles. Cette relation exprime le parallélisme de l'environnement.

Pour désigner qu'une action de communication peut être éventuelle, elle est notée entre crochets. (Ceci pour désigner que l'action est soumise à des conditions détaillées par la suite).

Le parallélisme et la séquence sont notés de même que dans le chapitre V.

- On note les communications en entrée et en sortie effectuées ; c'est une description statique, elle est exprimée entre parenthèses. Exemple : (? m1, ?m2, ?m3) ;
- On donne les relations entre les actions de communication, on cherche alors à décrire les parallélismes évidents (ceux imposés par l'environnement ; relation Ind), ensuite, on décrit les autres relations. Une description d'actions de communication est appelé ici comportement.

VII - 3 POINTS DE VUE UTILISES POUR PROCEDER A UNE DECOMPOSITION

Deux points de vue peuvent être adoptés pour décrire les différentes actions de communication d'un module et par là même opter pour une décomposition donnée.

3.1 Procéder par les entrées

Ce point de vue est celui où l'observation externe est dirigée par les entrées. Pour chaque message d'entrée m_e, on cherche les messages de sortie m_s, tel que chaque message m_s est lié par une relation de précédence causale avec m_e.

On peut isoler alors dans des modules différents, les entrées du module initial et déterminer par la suite les autres communications.

3.2 Procéder par les sorties

L'observation externe est dirigée ici par les sorties ; pour chaque message en sortie, il s'agit de préciser des relations logiques et fonctionnelles avec les messages en entrées qui sont à l'origine de son émission.

On isole donc dans des modules différents les sorties (indépendantes) du module initial et on détermine par la suite les autres communications.

Les choix de décompositions doivent s'effectuer de façon cohérente avec les éléments 1, 2, 3 et 4 du paragraphe 2.

Lorsqu'on procéde à une décomposition il faut déterminer les communications intermédiaires. Ces communications doivent se déduire logiquement des communications existantes du module initial.

Ces communications intermédiaires sont précisées dans nos propositions à l'aide d'une table où chaque colonne représente le comportement d'un module en terme de communications.

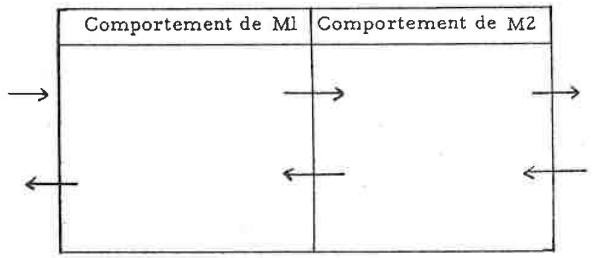


Figure 1 : table de communications intermédiaires

Légende : les arcs externes à la table représentent les communications du module initial décomposé, les arcs internes représentent les communications intermédiaires.

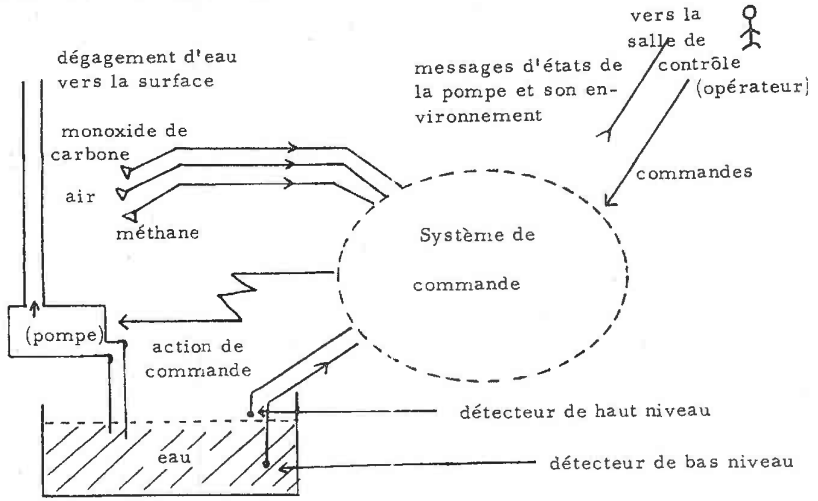
VII - 4 TRAITEMENT D'UN EXEMPLE

Nous traitons ci-après, l'exemple d'une application consistant à effectuer la commande et le contrôle d'une pompe de drainage d'eau.

Note : on fait référence aux quatre éléments d'aide à la description dans le développement de l'exemple.

Exemple de traitement d'une pompe souterraine

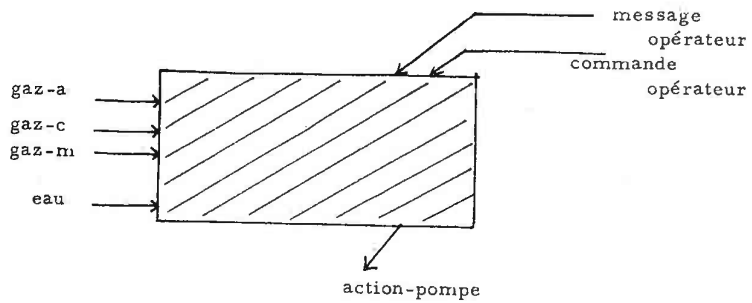
Cet exemple est dû à Kramer [KRA 80]. Il s'agit de décrire le fonctionnement d'une pompe de drainage dans les mines de charbons, présenté dans la figure suivante sous forme simplifiée (figure 1)



De la surface, le système reçoit la commande d'initialisation, il doit réguler le fonctionnement de la pompe en tenant compte des niveaux d'eau et de méthane. Le détecteur de haut niveau provoque la mise en marche de la pompe, tandis que celui de bas niveau provoque son arrêt : la pompe ne doit pas fonctionner si le niveau de méthane atteint un seuil dangereux.

Le système reçoit aussi (de la salle de contrôle) une commande d'arrêt de la pompe ou de demande d'état de cette dernière. En plus, des réponses aux commandes reçues, le système envoie le taux de chacun des trois gaz s'ils dépassent des seuils fixés.

Le système de commande peut être schématisé comme suit :



En termes d'échanges externes, ce système reçoit et émet sept messages. Les communications sont notées :

(? gaz-a ?gaz-c ?gaz-m ? eau ? commande-opérateur)
(! action-pompe ! message-opérateur).

Le module M_o (système logique de commande/contrôle) s'écrit de la façon suivante en prenant compte des contraintes de spécification fonctionnelle externe :

M_o ::* ((?# gaz-a Ind ? # gaz-c Ind ? # gaz-m Ind ? # eau
Ind ? # commande-opérateur) : C_o
[! # action-pompe]
//
[! message - opérateur]
)

Ceci résume les faits suivants :

- les messages en entrée arrivent, au système, de procédés physiques indépendants ; ils sont donc en situation de parallélisme réel (imposé par l'environnement). Les relations décrites en entrée peuvent s'écrire deux à deux : ? # gaz-a Ind ? gaz-c et ! # gaz-m Ind ? # gaz-n...

- les messages en sortie peuvent être envoyés en paralléle (ce n'est pas du parallélisme imposé) ;
- les envois éventuels sont dûs au fait que l'on ne peut déterminer toutes les relations entre les messages d'entrée et ceux de sortie. Celles-ci sont explicitées par le schéma suivant (elles sont essentiellement des relations de précédence causales) :

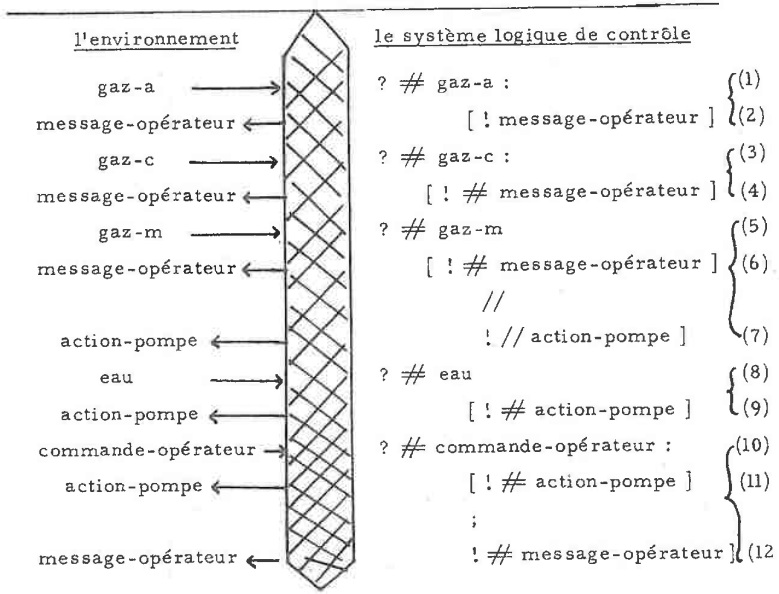


schéma descriptif des communications externes.

Ces relations sont définies par la spécification fonctionnelle externe :

Une expression de M_o, en privilégiant les entrées, est le contenu de la partie droite du schéma ci-dessus.

M_o :: (! # gaz-a : C_{01}

[! # message-opérateur]

//

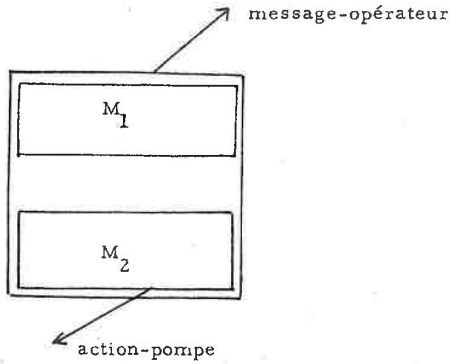
? # gaz-c : C_{02}

[! message-opérateur]

:

)

Si l'on procède à une décomposition par les entrées en isolant les comportements (ou groupe de comportements) C_{01}, C_{02} ... alors plusieurs modules auront la même sortie (! message-opérateur et/ou ! action-pompe). Ceci va à l'encontre du 3ème élément (cf. VII.2). On procède alors par les sorties.



En considérant le 1er élément (cf. VII.2 : associer un module à un processus physique) : on peut associer le module M_1 à la gestion de dialogue avec l'opérateur, l'entrée ? commande-opérateur se trouve aussi affectée à M_1 ; les autres entrées seront donc affectées à M_1 : ceci revient à considérer que le module M_2 est associé à la gestion des constituants atmosphériques et au contrôle de la pompe.

Pour préciser les interfaces de communication des modules, il faut préciser les communications intermédiaires. Nous analyserons pour celà le schéma précédent (page 113).

a - . un message que nous nommons "cd" est nécessaire ; il sera émis par M_1 et reçu par M_2 pour satisfaire la relation décrite dans les lignes (10) et (11) ;

. un autre message "réponse" est de même nécessaire, de M_2 vers M_1 pour satisfaire la relation décrite dans les lignes (10) et (12) ;

b - . un message intermédiaire "mg-a", de M_2 vers M_1 pour satisfaire la contrainte des lignes (1) et (2) ;

c - . un message intermédiaire "mg-1", de M_2 vers M_1 pour satisfaire la contrainte des lignes (3) et (4) ;

d - . un message intermédiaire "mg-m", de M_2 vers M_1 pour satisfaire la contrainte des lignes (5) et (6).

Ces trois derniers messages, sont des messages d'alarme. Les coordinations de ces messages sont décrites dans la table suivante :

Comportement de M_1	Comportement de M_2	
? # commande-opérateur :		
! # cd	? # cd	(1)
	[! # action-pompe]	(2)
	;	
? # réponse :	! # réponse	(3)
! # message-opérateur		
? # mg-a :	? # gaz-a :	(4)
! # message-opérateur	[! # mg-a]	(5)
? # mg-c	? # gaz-c	(6)
! # message-opérateur	[! # mg-c]	(7)
? # mg-m :	? # gaz-M :	(8)
! # message-opérateur	[! # mg-m	(9)
	//	
	! action-pompe	(10)
	]	

Table 1

Les communications des deux modules sont donc précisées :

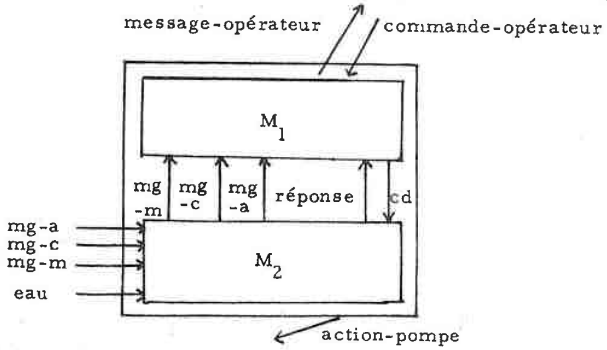
- pour M_1 on a :

  (? commande-opérateur ? mg-a ? mg-c ? mg-m ? réponse)

  (! cd ! message-opérateur)

- pour M_2 on a : (? gaz-a ? gaz-c ? gaz-m ? cd)

  (! réponse !mg-a !mg-c !mg-m ! action-pompe)



. Les relations de coordination au niveau des communications de M_1 sont :

? # commande-opérateur Ind ? # mg-a Ind ? # mg-c Ind ? mg-m

? # commande-opérateur ; ! # cd ; ? # réponse ;
! # message-opérateur

? # mg-a ; ! # message-opérateur

? # mg-c ; ! # message-opérateur

? # mg-m ; ! # message-opérateur.

. Les relations de coordination au niveau des communications de M_2 sont :

. ? # cd Ind ? # gaz-a Ind ? # gaz-c Ind ? # gaz-m Ind ? # eau

. ? # cd ; ! # réponse.

Les comportements respectifs de ces modules s'écrivent :

$M_1 :: ^*$(? # Commande-opérateur ; C_{11}
! # cd
;
? # réponse
;
! # message-opérateur

//
? # mg-a : C_{12}
! # message-opérateur

//
? # mg-c : C_{13}
! # message-opérateur

//
? # mg-m : C_{14}
! # message-opérateur

)

$M_2 :: ^*$ (? # cd : C_{21}
condition-1 (# cd) : ! # action-pompe
;
! # réponse

//
? # gaz-a : C_{22}
condition-2 (# gaz-a) : ! # mg-a

//
? # gaz-c : C_{23}
condition-3 (# gaz-c) : ! # mg-c

//

? # gaz-m : C_{24}
condition-4 (# gaz-m) : ! # mg-m
//
! # action-pompe
//
? # eau : C_{25}
condition-5 (# gaz-m, # eau) : ! # action-pompe
)

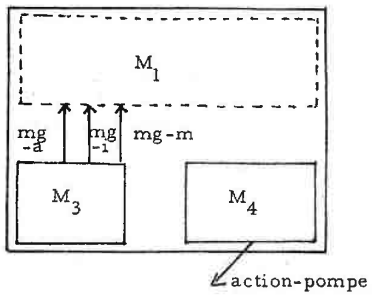
M_0 s'écrit donc : $M_0 :: M_1 // M_2$

Le module M_1 exprime bien un interface de dialogue avec l'opérateur, la sortie représentant cette fonctionnalité est ! message-opérateur ; l'autre sortie ! cd n'est utilisée que dans une communication du type demande-réponse. M_1 ne sera pas donc décomposé. Quant au module M_2, il exprime deux fonctionnalités.

Le contrôle de la pompe (! action-pompe) et la gestion des constituants atmosphériques (génération des alarmes ! gm-a ! gm-c ! mg-m). Ce module peut donc être décomposé (cf. VII.2, 1er élément proposé).

Si l'on procède par les entrées, le 3ème élément ne sera pas respecté, à cause de la sortie ! action-pompe.

On procède donc par les sorties en isolant d'une part ! action-pompe, et d'autre part le groupe de sortie ! mg-a ! mg-c ! mg-m

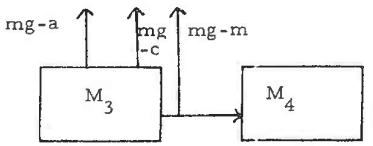


Pour préciser les communications des modules M_3 et M_4, on se trouve dans cette situation confronté à un ensemble de choix d'affectation des entrées :

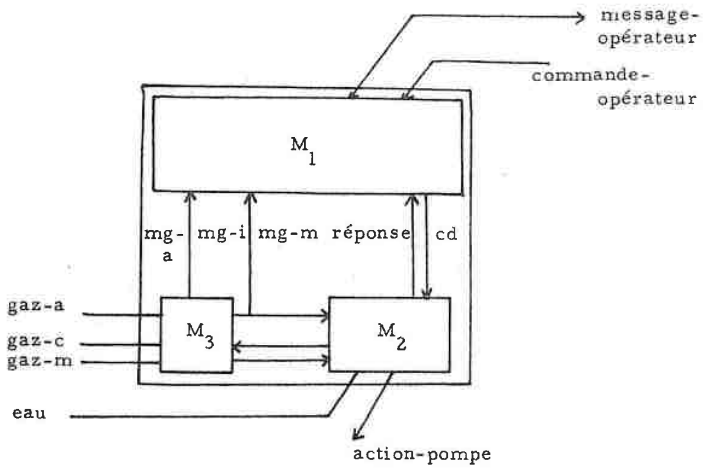
- Affecter gaz-a, gaz-c et gaz-m à M_3 et eau à M_4 est justifié par la nature des fonctionnalités ;

- Affecter (aussi) gaz-m à M_4 ou définir une communication intermédiaire - pour satisfaire la relation de précédence causale décrite dans la table par les lignes (8) et (10) - sont deux choix distincts. Nous optons pour ce dernier : En effet, le premier considère toutes les valeurs produites de mg-m, alors que l'on ne s'intèresse qu'à des valeurs au-dessus d'un seuil donné.

Le message intermédiaire est le message mg-m. Ce qui donne le schéma



Cette communication intermédiaire n'est pas suffisante. Le comportement C_{25} montre que l'on a besoin de connaître la valeur de gaz-m à la réception du message eau à M_4 ; d'où une communication "demande-réponse" entre M_3 et M_4 : un message "demande-gaz-m" de M_4 à M_3 et un message "état-gaz-m" de M_3 à M_4.



- Enfin, l'affectation de cd et réponse à M_4 sont justifiés par le comportement C_{21} du Module M_2.

Seules les communications intermédiaires sont précisées dans la table suivante :

Comportement de M_3	Comportement de M_4
... ! # mg-m →	? # mg-m ; ! # action-pompe
	? # eau condition b (# eau) : ! # demande gaz-m ;
? # demande-gaz-m : ←	
? # état- gaz-m : →	? # état-gaz-m ;
	condition (# état-gaz-m) : ! # action-pompe →

- Les communications de M_3 s'écrivent :

(? # gaz-a ? # gaz-c ? # gaz-m ? # demande-gaz-m)
(! # mg-a ! # mg-c ! # mg-m ! # état-gaz-m)

- Celles de M_4 :

(? # cd ? # eau ? # demande-gaz-m ! # action-pompe)

Les comportements respectifs de ces modules s'écrivent :

M_3 :: * (? # gaz-a : C_{31}
Condition-2 (# gaz-a) : ! # mg-a
//
? # gaz-c : C_{32}
Condition-3 (# gaz-c) : ! # mg-c
//
? # gaz-m : C_{33}
Condition-4 (# gaz-m) : ! # mg-m
//
? # demande-gaz-m : C_{34}
! # état-gaz-m
)

M_4 :: * (? # cd : C_{41}
Condition-1 -(# cd) : ! # action-pompe
;
! # message-opérateur
? # mg-m : C_{42}
! # action-pompe

? # eau : C_{43}

(Condition-5 (# eau) : ! # demande-gaz-m ;
? # état-gaz-m ,
condition (# état-gaz-m) : ! # action-pompe

|

Condition-6 (# eau) : ! action-pompe

)

)

Le système logique de commande M_0 s'écrit donc :

$$M_0 :: M_1 // M_3 // M_4$$

L'ensemble des conditions ci-dessus $condition_i$ (# ...) est détaillé lors de l'expression dans le langage FLEXI. Ces conditions portent sur les valeurs des messages.

Nous donnons en annexe, la description de cet exemple en FLEXI.

CHAPITRE VIII

CONCLUSION

VIII - 1 En conclusion.

VIII - 2 Perspectives

2.1 Relations avec les travaux de MILNER.

2.2 Réalisation.

VIII - 1 EN CONCLUSION

Dans la classe d'applications temps réel réparties, le concept de communication a constitué le concept noyau de notre démarche. Il intervient aussi bien à la phase de décomposition qu'à la phase de description de l'application en réseau de modules communicants.

Nous avons pu montrer que les langages de programmation - exprimant de façon prédéfinie la communication - ne disposent généralement que d'une forme de communication. D'où notre approche de spécifier les formes (ou type) de communication, dont on a besoin, dans cette classe d'applications, de façon à pouvoir construire automatiquement ces communications dans un langage cible à la réalisation. Ceci permet pour nos propositions de communications d'être indépendantes d'un langage de programmation donné.

Par rapport à d'autres propositions similaires [RAY 81], [KRA 83], nos propositions, concernant la description statique des communications (connexions) et la description du module, se distinguent par la définition d'un choix déterministe sur les structures convergentes et divergentes (l'opérateur "|" défini sur ces structures) et par la définition de la préemption logique. Ces définitions sont essentielles dans la classe d'applications choisie.

Nous avons aussi pu établir des contrôles statiques et particulièrement sur les communications. La démarche adoptée, comme ceci a été précisée en introduction, nous permet de passer aussi bien à la simulation qu'à la réalisation. Ceci est important pour valider une application avant son implantation sur des processeurs cibles.

VIII - 2 PERSPECTIVES

Le langage FLEXI proposé n'est pas "figé". On peut éventuellement exhiber d'autres politiques de communication concernant les ports d'entrée. La partie dite contrainte dans le comportement du module peut être formalisée.

Il subsiste cependant d'autres points non traités (ou traités partiellement), nous en mentionnons deux qui nous paraissent très importants.

2.1 Relations avec les travaux de Milner

Dans [OUZ 82-a] il y a eu un essai de rapprochement des travaux entre la vue théorique adoptée par Milner, et celle pragmatique adoptée dans nos travaux. Nous nous sommes inspirés du principe d'observation externe de la communication d'un comportement que nous avons pu appliquer au niveau conception (exemples : éléments d'aide à la décomposition ; spécification d'un module).

Il s'agit alors pour établir le lien entre les deux travaux d'utiliser le modèle de Milner pour valider un comportement de façon formelle ou de prouver l'équivalent d'un comportement par un autre (nous pensons particulièrement aux contraintes globales cf. chapitre V).

2.2 Réalisation

Cette thèse a débuté dans le cadre d'une collaboration université-industrie avec la Régie RENAULT. Elle s'est ensuite développée dans un cadre universitaire (particulièrement au sein de l'ATP parallélisme de Nancy). La majorité des concepts sont réalisés actuellement et testés dans le cadre de simulation d'applications industrielles, et les

premières conclusions en sont satisfaisantes (Travaux de GOFFIC [GOF 83] validation de modèles par simulation. Des exemples sont donnés dans [GUR 82], [BOS 82]). Un autre essai de réalisation s'est manifesté dans les travaux de MAMMERI [MAM 82], et dont le but était de montrer la possibilité d'implantation des concepts développés sur une machine cible MUGUET ([MUG 81] architecturée autour du processeur MOTOROLA 68000 en utilisant un interface SCEPTRE [SCE 80]). Enfin, une validation de l'ensemble des concepts définis ne serait vraiment possible que dans le cadre d'une application industrielle à grandeurs réelles (type atelier flexible).

ANNEXES

ANNEXE 1 - LA SYNTAXE DU LANGAGE FLEXI

I - AVERTISSEMENT

Pour faciliter la description syntaxique du langage, nous utilisons un metasymbole dont les notations sont les suivantes :

. le symbole * veut dire que l'élément spécifié est répété zéro fois ou plus ;

< a >* ::= vide || a < sa >

< sa > ::= a || a < sa >

. Le symbole + veut dire que l'élément spécifié est répété au moins une fois.

< a >+ ::= a || a < sa >

< sa > ::= a || a < sa >

nous utilisons la notation < a >+(s) pour désigner le séparateur "S" :

< a >+(S) ::= a || a S < sa > avec < sa > ::= a || a S < sa >

. Le symbole □. placé entre deux éléments veut dire que l'un au plus est optionnel. S'ils figurent ensemble ; ils le sont dans l'ordre de rencontre.

a □ b ::= a || b || a b

II - LES ELEMENTS DU LANGAGE

A - Liste des Messages

< liste_des_messages > ::= MESSAGES
EXTERNAL < liste_des_messages_externes > ;
INTERNAL < liste_des_messages_internes > ;
END MESSAGES

< liste_des_messages_externes > ::= < message_type >+ (;)

< liste_des_messages_internes > ::= < message_type >+ (;)

< Message_type > ::= < identificateur_de_message > : < type >

B - Listes des noms des Modules

< liste_des_noms_des_modules > ::= MODULES NAME
EXTERNAL
< liste_de_Modules_externes >
INTERNAL
< liste_de_modules_internes >
END MODULES

< liste_de_Modules_externes > ::= < identificateur_Modules >$^{+}$ (;)
< liste_de_modules_internes > ::= < identificateur_module >$^{+}$ (;).

C - Liste des connexions

< liste_des_connexions > ::= LINKS
< connexions >$^{+}$ (;)
END LINKS

< connexion > ::= < connexion_de_base > || < connexion_construite >
< connexion_de_base > ::= < structure_bipoint > || < structure_divergente > ||
< structure-convergente >

< structure_bipoint > ::= < identificateur_module > . < identificateur_port - s > ⟶
< identificateur_module > . < identificateur-port_e >
|| < identificateur_module > . < identificateur_port_s > ⟶
< identificateur_module > . < identificateur_port_E >

< structure_divergente > ::= < strucutre_et > || < structure_ou >

< structure_et > ::= < identificateur_module > . < identificateur_port_s > ⟶
<< identificateur_module > . < identificateur_port_e >>$^{+}$ (&)

< structure_ou > ::= < identificateur_module > . < identificateur_port_s >
<< identificateur_module > . < identificateur_port_e >>$^{+}$ (|)
|| < identificateur_module > . < identificateur_port_s > ⟶
<< identificateur_module > . < identificateur_port_E >>$^{+}$ (|)

< structure_convergente > ::= < structure_ou_c > || < structure_entrelacement >

< structure_ou_c > ::= < identificateur_module > . < identificateur_port_s > |
<< identificateur_module > . < identificateur_port_s >>$^{+}$ (|)
⟶ < identificateur_module > . < identificateur_port_e > ||
< identificateur_module > . < identificateur_port_S > |
<< identificateur_module > . < identificateur_port_S >>$^{+}$ (|)
⟶ < identificateur_module > . < identificateur_port_E >

< structure_entrelacement > :: < identificateur_module > . < identificateur_port_s >,
<< identificateur_module > . < identificateur_port_s >>$^{+}$ (,)
⟶ < identificateur_Module > . < identificateur_port_e > ||
< identificateur_module > . < identificateur_port_S > ;
<< identificateur_module > . < identificateur_port_S >>$^{+}$ (,)
⟶ < identificateur_module > . < identificateur_port_E >.

< connexion_construite > ::= CONNEXION
< connexion_de_base >$^{+}$ (;)
END CONNEXION

D - Contraintes globales

< description_globale > ::= TOTAL DESCRIPTION
< description_entrée_sortie >$^{+}$
END DESCRIPTION

< description_entrée_sortie > ::= < communication > ;
< contrainte_temps_réel > ;

< communication > ::= < réception > : (< expression_parenthésée >$^{+}$ (ou))

< expression_parenthésée > ::= (< expression_parenthésée > ; < expression_parenthésée >)
|| (< expression_parenthésée > // < expression_parenthésée >)
|| émission || nil

< réception > ::= { < identificateur_port_e > / < identificateur_port_E > } ? < identificateur_de_message >

< émission > ::= { < identificateur_port_S > / < identificateur_port_S > } ! <identificateur_de_message >

< contrainte_temps_réel > ::= < WITH | #=<identificateur_message >
MAXIME < delta-t >
FROM ?#= < identificateur > $^{+}$ (;)

< delta-t > exprime une constante entière.

E - Liste des Modules

< liste_des_modules > ::= MODULES
< définition_d'un_module >$^{+}$ (;)
END MODULES

< définition_d'un_module > ::= MODULE < identificateur_module >
COMMUNICATION PORTS
[EXTERNAL
< ports_externes >]
INTERNAL
< ports_internes >
FUNCTIONS
< comportement >
ENDFUNCTIONS
END < identificateur_module >

< ports_externes > ::= < liste_de_ports >

< ports_internes > ::= < liste_de_ports >

< liste_de_ports > ::= ENTRYPORTS < port_e >$^{+}$ (;)
□ EXITPORTS < port_s >$^{+}$ (;)
□ EXIT ENTRY PORTS < port_E >$^{+}$ (;)
□ ENTRY EXIT PORTS < port_S >$^{+}$ (;)

< port-e > ::= < identificateur_port-e > : << identificateur_de_message >
[ANTICIPATION { LIFO / FIFO / RAND }]$^{+}$ (;) >

< port_s > ::= < identificateur_port_s > : << identificateur_de_message >
[ANTICIPATION]$^{+}$ (;)

< port_E > ::= < identificateur_port_E > : << identificateur_de_message REPLY
< identificateur_de_message >>$^{+}$ (;)

< port_S > ::= < identificateur_port_S > : << identificateur_de_message > RESPONSE
< identificateur_de_message >>$^{+}$ (;)

< comportement > ::= << identificateur_de_comportement > : < port_e >$^{+(,)}$;
< port_E >$^{+}$ (,) ; < description >>$^{+}$ (;)
comportement global < description_globale >
contraintes < texte >

< description > ::= < action_s > || < action_c >

< action_c > ::= < action_c > ; < action_c > || (< action_c >) com (< action_c >)
||* (< action_c >) || tantque < condition >* (< action_c >)
|| jusqu'a < condition >* (< action_c >) || < action_évaluée >
|| < action_s >

< action_évaluée > ::= <la_condition_action > || < la_déterministe > ||
< l'indéterministe >

< la_condition_action > ::= (< condition_évaluée >) : (< action_c >)
[|
< action_c >)

< condition_évaluée > ::= < réception_gardée > || < émission_gardée >
|| < condition_gardée > || < envoi_réponse_gardée >

< la_déterministe > ::= ◇ < opérande >
< sélection_de_valeur >
◇

< sélection_de_valeur > ::= < valeur > : < action_c >
|
<< valeur > : < action_c >>$^{+}$ (|)

< L'indéterministe > ::= □
< sélection_de_garde >
□

< sélection_de_garde > ::= < alternative >
|
<< alternative >>$^{+}$ (|)

< alternative > ::= { < condition_évaluée > : < action_c > / < réception_T3 > : < traitement_service > ; < émission_T3 > }

< réception_gardée > ::= ((<identificateur_port_e >, T2) ? #identificateur_de_message)

< émission_gardée > ::= ((< identificateur_port-s >, T2) ! #identificateur_de_message)

< envoi_réponse_gardée > ::= ((< identificateur_port_S >,T3) ! #identificateur_de_message)
(< identificateur_port_S >, T3) ? #identificateur_de_message)

< réception_T3 > ::= ((< identificateur_port_E >, T3) ? #< identificateur_de_message >)

< emission_T3 > ::= (< identificateur_port_E >, T3) ! #identificateur_de_message)

< action_S > ::= < communication_typée > || < attente > || nil || < priorité >

< communication_typée > ::= < notification > || < commande > || < service >

< notification > ::= (< identificateur_port_e >, T1) ? #< identificateur_de_message ||
(< identificateur_port_s >, T1)! #identificateur_de_message

< commande > ::= (< identificateur_port_e >, T2) ? #< identificateur_de_message >
|| (< identificateur_port_s >, T2 ! #< identificateur_de_message >

< service > ::= (< identificateur_port_E >, T3) ? #identificateur_de_message
< traitement_service > ;
(< identificateur_port_E >, T3 >) ! #identificateur_de_message
||
(< identificateur_port_S >, T3) ! #identificateur_de_message
(< identificateur_port_S >, T3) ? #identificateur_de_message

< traitement de service > ::= < action >

< action > ::= << action >>$^{+(;)}$ || (< action >) com (< action >) || (< action évaluée_s >)
|| jusqu'a < condition >* (< action >) || tantque < condition >* (< action >)
|| < notification > || < commande > ||
(< identificateur_port_S >, T3) !#identificateur_de_message
(< identificateur_port_S >, T3) ? #identificateur_de_message

< attente > ::= attente < delta_t >

< condition > est une expression booléenne à résultat vrai ou faux ;

< condition_gardée > est une expression à résultat vrai ou faux utilisée dans une garde.

< priorité > ::= priorité < réception > > priorité (<< réception >>$^{+(,)}$)

< réception > ::= ? < identificateur_de_message >

< action_évaluée _s > ::= < condition_action_s > || < déterministe_s > || < indéterministe_s >

< action_condition_s > ::= < condition-évaluée_s > : (< action >
[|
< action >])

< déterministe_s > ::= ◇ < opérande >
< valeur > : < action >
|
<< valeur > : < action >>$^{+(|)}$
◇

< indéterministe_s > ::= □
< condition_évaluée_s > : < action >
|
<< condition_évaluée_s > : < action >>$^{+(|)}$
□

< condition_évaluée_s > ::= < réception_gardée > || < émission_gardée > ||
< envoie_réponse_gardée > || < condition_gardée >

< description_globale > ::= < parallélisme > ,
[< préemption_logique > ;]
[< contraintes_temps_réel >]

< parallélisme > ::= < identificateur_de_comportement >
// << identificateur_de_comportement >>$^{+(//)}$

< préemption_logique > ::= préemption de < identificateur_de_comportement >
sur (<< identificateur_de_comportement >>$^{+(,)}$)

III - SYNTAXE DE L'ETAPE DE PROGRAMMATION

A - Liste des types de Messages

< liste_des_types_de_messages > ::= MESSAGES TYPES
<< identificateur_de_messages = type >>$^{+(;)}$
END MESSAGES TYPES

Nous prenons comme types ici, ceux de la famille PASCAL :

< type > ::= INTEGER || REAL || BOOLEAN || < scalaire > || < tableau >
|| < enregistrement >

< scalaire > ::= (< identificateur >$^{+(,)}$)

< tableau > ::= ARRAY (<b_inférieure > : < b_supérieure >)$^{+(,)}$ OF < type >

< enregistrement > ::= RECORD
<< identificateur >$^{+(,)}$: < type >>$^{+(;)}$
END

B - Liste de répartition

< liste_de_répartition > ::= CONFIGURATION
< MACHINE : < identificateur_de_processeur > ;
<< identificateur_capsules > : < identificateur_
module >>$^{+(;)}$
END MACHINE
END CONFIGURATION

C - Liste des Capsules

< liste des capsules > ::= CAPSULES
< déclaration_d'une_capsule >$^{+(;)}$
END CAPSULES
< déclaration_d'une_capsule > ::= CAPSULE < identificateur_capsule > :
< interface >
< objets_communs_aux_tâches >
< tâches >$^{+(;)}$
END CAPSULE

< tâche > ::= TASK < identificateur_tâche > : < port_e >$^{+(,)}$; < port_E >$^{+(;)}$
< corps_de_la_tâche >
END TASK

< interface > est décrit précédemment par le module ("COMMUNICATION PORTS") ;
< objets_communs_aux_tâches > sont les objets locaux à la capsule et partagés par les tâches.

ANNEXE 2 - EXEMPLE D'APPLICATION

Nous traitons ci-dessous l'exemple de la pompe de drainage d'eau dans une mine de charbon. L'application a été décomposée en modules communicants dans le chapitre VII.

Nous expliciterons donc cinq listes dans le langage proposé :

- la liste des messages ;
- la liste des noms des modules,
- la liste des connexions ;
- la liste des contraintes globales ;
- la liste de descriptions de modules.

1) Liste des messages

```
MESSAGES TYPES

EXTERNAL
Commande_opérateur : (marche, arrêt, status) ;
message_opérateur : (mise_en_marche, arrêtée, en_marche, alarme_air,
                     alarme_carbone, alarme_méthane) ;
gaz-a : REAL ;
gaz-c : REAL ;
gaz-m : REAL ;
eau   : (bas, normal, haut)
action-pompe : (marche, arrêt)

INTERNAL
cd : (marche, arrêt, status) ;
réponse : (mise_en_marche, arrêtée, en_marche) ;
demande_gaz-M : (demande) ;
état_gaz-m : REAL ;
mg - a : (alarme_air) ;
mg - c : (alarme_carbone) ;
mg - m : (alarme_méthane) ;
END MESSAGES
```

2) Listes des noms des modules

```
MODULES NAMES
EXTERNAL    interface_opérateur ;
            environnement ;
            commande
END MODULES
```

3) Listes des connexions

```
CONNEXIONS
interface_opérateur . CDE - REP ⟶ commande . CDE- REP ;
commande . DEM - MET ⟶ environnement . DEM - MET ;
environnement . ALM ⟶ commande.ALARME
                    &
                    interface opérateur.ALM ;

environnement . ALC - ALA          interface_opérateur . ALC_ALA
END CONNEXIONS
```

4) Contraintes globales

```
TOTAL DESCRIPTION
  EAU ? #= eau : (ACT ! #= action_pompe
                  ou
                  nil
  METHANE ? #= gaz_m : (ACT ! #= action_pompe // MSGE ! message_opérateur
                  ou
                  nil)
COMD ? #= commande_opérateur : (MSGE ! #= Message_opérateur
                  ou
                  ALT ! #= action_pompe ; MSGE ! #= message_opérateur
                  )
  AIR ? #= gaz_a : ( MSGE ! #= message_opérateur
                  ou
                  nil
                  )
```



```
  CARB ? #= gaz_c :   (MSGE ! #= message_opérateur
                      ou
                      nil)
END DESCRIPTION
```

5) Liste des modules

```
MODULES
MODULE Commande ;
COMMUNICATION PORTS
    EXTERNAL
        ENTRY PORTS   EAU : eau ; EXITPORTS ACT : action_pompe ;
    INTERNAL
        ENTRY PORTS   ALARME : mg - m ;
        ENTRYEXITPORTS   CDE-REP : cd REPLY reponse ;
        EXITENTRYPORTS   DEM-MET : demande_gaz_m
                                   RESPONSE état_gaz_m ;
    FUNCTIONS
    dialogue_opérateur : CDG - REP
(CDE_REP,T3) ? #= cd :
                ◇ (#= cd)
                  arrêt : (ACT, T2) ! arrêt     % arrêt de la pompe %
                      |
                  marche : nil
                      |
                  status : nil
                ◇
                  ;
    (CDE - REP, T3) !  réponse     % une réponse est envoyée suivant les cas %
    veilleur_méthane : ALARME ;
(T2) ?  mg - m : (ACT,T2) ! arrêt          % arrêt de la pompe car le seuil de
                                             méthane est atteint %
```

contrôle_d'eau : EAU, DEM-MET ;

(T2) ? #= eau : ◇ (#eau)

bas : (ACT,T2) ! arrêt % arrêt de la pompe %

|

normal : nil

|

haut : (DEM - MET, T3) ! demande_gaz_m ;

(DGA - MET, T3) ! ? état_gaz_m ;

(état_gaz_m < seuil - m) : (ACT, T2) ! marche

% mise en marche de la pompe %

◇

comportement global dialogue_opérateur // veilleur_méthane // contrôle d'eau

contraintes

Dans toutes les communications en sortie (!#action_pompe) la valeur arrêt n'est pas envoyée si la pompe est arrêtée.

- Dans dialogue opérateur !#réponse doit contenir l'une des valeurs :
  - . mise_en_marche si #Cd reçue est marche
  - . arrêtée si # Cd reçue est arrêt
  - . en_marche si la dernière valeur envoyée par ! action_pompe est marche.
- Dans le comportement veilleur_méthane la valeur arrêt est envoyée dès réception de #mg - m.
- Dans Contrôle_d'eau : . la réception d'une valeur bas entraîne (envoi de la valeur arrêt.

  . La réception de la valeur haut entraîne une demande au module environnement pour s'assurer que l'état du méthane est en dessous du seuil, dans ce dernier cas seulement la valeur marche est envoyée.

END FUNCTIONS

END Commande ;

MODULE environnement

COMMUNICATION PORTS

EXTERNAL

ENTRYPORTS AIR : gaz-a ; CARB : gaz-c ; METHANE : gaz-m

INTERNAL

ENTRYEXITPORTS DGM-MET : demande-gaz-m REPLY état-gaz-M ;

EXITPORTS ALM : mg-m ; ALC-ALA : mg-a, mg-c ;

FUNCTIONS

Contrôle_air_carbone : AIR ; CARB ;

T1 ? #= gaz-a : ((#= gaz-a < seuil-a) : T1 ! #= mg-a)

;

T1 ? #= gaz-c : ((#= gaz-c > seuil-c) : T1 ! #= mg-c)

Contrôle_méthane : METHANE

T1 ? #= gaz-m : ((#= gaz-m > seuil-m) : T1 ! #= mg - m)

service : DEM - MET ;

T3 ? #= demande_gaz - m ; T3 ! #= état_gaz - m.

Comportement global : contrôle_air_carbone // contrôle_méthane // service

contraintes :

- . Les occurrences #= mg-a, #= mg-c et #= mg-m représentent à l'envoi des alarmes, dès que les valeurs reçues sont en dehors des seuils fixés ;
- . dans le comportement service : #= état_gaz-m doit contenir la dernière valeur de gaz-m prise en compte par contrôle_méthane.

END FUNCTIONS

END environnement

MODULE interface_opérateur

COMMUNICATION PORTS

EXTERNAL

ENTRYPORTS COMD : Commande_opérateur ; EXITPORTS MSGE ! message_opérateur

INTERNAL

ENTRYPORTS ALM : Mg - m ;

ALA-ALC : mg-c ; mg-a ;

EXITENTRYPORTS CDE - REP : Cd RESPONSE reponse ;

FUNCTIONS

Opérateur : COMD ;

(T2) ?#commande_opérateur : (T3) !#Cd

(T3) ?# réponse ;

;

(T2) !#message_opérateur

alarme_méthane : ALM ;

(T2) ?#mg-m : (T2) ! # message_opérateur

alarmes_air_c : ALC - ALA

[]

(T2) ?#mg-c : (T2) !# message_opérateur

|

(T2) ?#mg - a ; (T2) !#message_opérateur

[]

Comportement global : opérateur // alarme_méthane // alarme_air_c

contraintes

- Dans alarme_méthane et alarme_air_c les valeurs envoyées ! message_opérateur correspondent aux cas respectifs d'alarmes.

- Dans le comportement opérateur les valeurs envoyées dans #cd est la même que celle reçue dans #ommande_opérateur ; la valeur envoyée dans message_opérateur est celle reçue dans réponse.

END FUNCTIONS

END interface_opérateur

END MODULES.

Bibliographie

[AND 80] ANDRE F., HERMAN D., VERJUS J.P.
Contrôle du parallelisme et de la répartition.
Cours de l'Ecole d'Eté de l'AFCET - AIX - Juillet 1980. (96 p).

[AND 81] ANDRE F., HERMAN D., LE GUERNIC P., RAYNAL M., VERJUS J.P.
Synchronisation et communication dans un ensemble de processus coopérants.
Cours sur le parallèlisme - organisé à RENNES du 12-14 Octobre 1981.

[AND 79] ANDREWS G.R.
Synchronizing Ressources.
Technical Report 79-20 - University of Arizona (1979)
(figure également dans Toplas).

[ABR 83] ABRECHVILLER
Nom collectif pour : CREHANGE M., DERNIAME J.C., FINANCE J.P., JARAY J., LONCHAMP J., PERRIN G.R., THOMESSE J.P., ZAKARI A.
Du cahier des charges à une solution -
- Entités communicantes - Liaisons et communications.
Rapport de travail d'ATP parallélisme - CRIN - NANCY I.

- B -

[BRI 75] BRINCH HANSEN P.
Distributed Processes : A concurrent Programming concept.
Com. of ACM Vol. 21, 11 - Nov. 1978.

[BOS 82] BONFILS P., SANCHEZ F.
Modélisation d'un atelier de mastiquage de carrosseries automobiles par la méthode des réseaux de modules.
Rapport de stage ENSAM effectué à la Régie RENAULT (DAST - REGIENOV) 1982.

[BRI 75] BRINCH HENSEN P.
The programming Language Concurrent Pascal.
IEEE Trans. on soft. Eng. SE-1 (June 1975) pp. 195-207.

- C -

[COR 81] CORNAFION
Nom collectif pour : ANDRE F., BANINO J.S., BETOURNE C., FERRIE J., HERMAN D., KRAKOVIAK S., MAZARE G., MOSSIERE J., ROUSSET DE PINAT X., SEGUIN J., VERJUS J.P.
Systèmes informatiques répartis - Concepts et techniques. Editions DUNOD - 1981.

[CRO 76] CROCUS
Nom collectif pour : BELLINO J. BETOURNE C., BRIAT J.J., CANET B., CLEEMANN E., DERNIAME J.C., FERRIE J., KAISER C., KRAKOVIAK S, MOSSIERE J., VERJUS J.P.
Systèmes d'exploitation des ordinateurs. Editions DUNOD (BORDAS PARIS 75).

- D -

[DAH 70] DAH O.J., MYHRAUG B., NYAARD K.
Simula 67 Common Base Language. Pub. 22, Norvegian Computing Center, OSLO, Octobre 70.

[DER 74] DERNIAME J.C.
Le projet CIVA. Thèse de Doctorat ès-Sciences - Université de Nancy I - Janvier 1974.

[DEF 79] DERNIAME J.C., FINANCE J.P.
Spécification, utilisation et réalisation. Ecole d'Eté de l'AFCET - Monastir 1979.

[DER 80] DERNIAME J.C.
"Cours de D.E.A. - Option Génie Logiciel" Université de NANCY I - Année 1980/1981.

[DIJ 75] DIJKSTRA E.W.
"Guarded commands, non determinary and a calculus for the derivation of programs". Com. ACM 18,8 - Aug. 1975 - p. 453 - 457.

[DPT 82] DERNIAME J.C., PERRIN G.R., THOMESSE J.P.
Une classification des concepts relatifs à la communication en vue de définir une démarche de conception de systèmes. Communication interne - ATP Parallélisme - NANCY I - 1982.

[DEZ 83-a] DERNIAME J.C., ZAKARI A.
Spécification d'application de conduite d'un atelier flexible. Convention informatique Latine (CIL) Barcelone - 6 - 9 Juin 1983.

[DEZ 83-b] DERNIAME J.C., ZAKARI A.
Langage de conception pour des applications réparties de conduites de procédés. Journées BIGRE 83, LE CAP D'AGDE.

- F -

[FAZ 84] EL FAZZIKI AZIZ
Thèse de 3ème cycle - Université de Nancy I (à paraître).

[FPA 81] FINANCE J.P., PAIR C.
"Cours de DEA d'Informatique" - Année 81 - 82. "Spécification et construction de programmes". Université de NANCY I - 1980/1981.

[FEL 79] FELDMAN J.A.
High Level Programming for distributed Computing. Com. ACM Juin 1979. Vol. 22.

- G -

[GUI 82] GUILLEMONT M.
Intégration du système réparti CHORUS dans le langage de haut niveau PASCAL. Thèse de 3ème cycle - GRENOBLE - Mars 1982.

[GUT 77] GUTTAG J.
Abstract data types and the developpement of data structures. Com. ACM 20 - 1977 - p. 396 - 404.

[GOF 83] GOFFIC P.

Thèse de Docteur Ingénieur (à paraître 1984).

[GUI 82] GUILBERT D., ROULY G.

Représentation d'un atelier flexible en réseau de modules.
Rapport de stage ENSAM 1982 effectué à la Régie RENAULT (DAST - Régienov).

- H -

[HER 81] HERMAN D.

Synchronisatrion des processus de l'expression abstraite à la mise en œuvre.
Doctorat ès-Sciences - Univ. de RENNES I - Avril 1981.

[HOA 74] HOARE I.A.R.

Monitors : an operating Systems Structuring Concept.
Com. ACM Vol. 17, n° 10, Oct. 1974, p. 41 - 49.

[HOA 78] HOARE I.A.R.

Communicating Sequential Processes.
Acta Informatica 11 - 1978 - p. 271 - 281.

[HEL 80] HEHNER, LENGAUER

A methodology for programming with concurrency.
University of Toronto, Computer Systems Research Group 121 st. Joseph st., Toronto, Ontario, Canada, M5S 1A1 (1980).

- I -

[ICH 79] ICHBIAH J. et al.

Rational for the Designe of the ADA Programming Language
Sigplan Notices - Vol. 14, n° 6 (June 1979).

- J -

[JAF 80] JARAY J., FINANCE J.P.

Towards a methodology to specify and construct concurrent programs"
Rapport interne CRIN n° 09-P-80 - Univ. de NANCY I.

[JAC 76] JACKSON L.

Language Design for Modular Software Construction.
Technical Comittee an Operating Systems. Institution : R5RE Malvern. 3 Dec. 76 - OP/SYS III.12.1.

[JEN 80] JENSEN E.D.,

The Honeywell Experimental Distributed Processor.
An Overview.
Paru dans les sytèmes informatiques répartis - Journée de synthèse AFCET Informatique 24 - 27 Novembre 80. NANCY.

[JOH 80] JOHN K.O., DONALD A.S., PRADEEP S.S.

MEDUSA : An experiment in Distributed Operating System Structure.
Com ACM - Vol. 23 n° 2 Feb. 1980.

[JUL 81] JULLIAND J.

Expression des communications entre processus d'un programme parallèle par des types abstraits.
Thèse de 3ème cycle - Université de Franche-Comté, Mai 1981.

- K -

[KAH 81] KAHN K.C., CORNIC W.M., DON DENNIS T., D'HOOGE H., HUBKA D.E., HUTCHINS L.A., MONTAGUE J.T., POLLACK F.J.

¡MAX : a multiprocessor operating system for on object - based computer.
Com. ACM 1981 p. 127 - 136.

[KRA 80] KRAMER J., MGEE J., SLOMAN M.

Intertask Communication primitives for distributed computer control systems.
Research Report N. DOC 80/17.Dept. of Computing Imperial College - LONDON - 1980.

[KRA 83] KRAMER J. et al.

CONIC : an Integrated to Distributed Computer control Systems.
IEEE process E., 130 E, (Jan. 1983).

- L -

[LEV 78] LEVEN A.A.

Development of specification and design tools in Great Britain. At Seminar on methods and Tools for specification and design of process control systems.
I.D.T. Kernforschungszartrum, KARLSRUHE - 7th - Jun 1978.

[LEG 80] LEGUERNIC P., RAYNAL M.

Eléments d'un langage adaptés à la communication entre processus.
Actes du congrès AFCET INFORMATIQUE - NANCY 1980, p. 667 - 676.

[LIK 80] LISTER A., MAGEE J., SLOMAN M., KRAMER J.

Distributed Processes Control Systems : Programming and configuration.
Imperial College Research Report n° 80/12 - May 1980.

[LIS 74] LISKOV B., ZILLES S.

Programming with abstract Data types.
Proc. ACM SIGPLAN Notices 9 - 1974, p. 50 - 59.

[LIS 79] LISKOV B.

Primitives for distributed Computing
Proc. of 7th Symp. on Operating Systems principles.

[LIS 82] LISKOV B., SCHEIFFER R.

Guardians and Actions : Linguistic Support for Robust, Distributed Programs.
POPL Janv. 1982.

[LIS 75] LISKOV B.

An Introduction to CLU in New Direction in Algorithmic languages.
Schuman ed. 1975.

[LOR 79] LORRAIN
Nom collectif pour : DERNIAME J.C., MEYER D., SCHAFF A., THOMESSE J.P., VIAULT D.

Réseaux Téléinformatiques
Ed. HACHETTE (1979)

[LON 83] LONCHAMP J.

Structuration logique en termes d'agents communicants des applications réparties en conduite de commande/contrôle de processus.
Journées BIGRE 1983, LE CAP D'AGDE.

- M -

[MIL 78] MILNER R.

Synthesis of Communicating Bahavior.
University d'Edinburgh April 1978.

[MIN 79] MINOT R.

ATM : un système de fabrication de programmes basés sur les concepts de modularité et de type abstrait.
Thèse de 3ème cycle Université de NANCY I 1979.

[MUG 81] GROS C.

MUGUET : Multiprocesseur généralement utilisable efficacement en Temps Réel.
Note technique CNET (Lannion) : NT/LAA/DIR/16 documentation remise lors des journées d'informations techniques destinées aux industriels et universitaires ((Architecture des Systèmes et Microprocesseurs)).

[MAM 82] MAMMERI Z.

Etude de la spécification interne et de l'implantation d'applications réparties en conduite de procédés et étude de la réalisation d'un noyau de communications.
Rapport de DEA - Université de NANCY I - 1982.

- O -

[OWI 76] OWIKI S., GRIES D.

Verifying properties of paralleles programs - an axiomatic approach.
Com. ACM Vol. 19 n° 5 - 1976.

[OUZ 82-a] OUERGUI M.S., ZAKARI A.

Un exemple de systèmes communicants.
Rapport interne CRIN n° 82 - R - 032.

[OUZ 82-b] OUERGUI M.S., ZAKARI A.

Vers une bonne expression de la communication dans un milieu parallèle.
Rapport interne CRIN n° 82 - R - 061.

- P -

[PAR 84] PARAYRE M.

LTRV3 - Manuel officiel de référence.
CRIN Janvier 1984.

[PAI 78] PAIR C.

La programmation de l'énoncé au programme.
Publication interne - CRIN 78 - P - 061 - Université de NANCY I.

[PER 80] PERRIN G.R.

Le codencement par les données : une méthode de conception de programmes.
Actes du congrès de l'AFCET 24-27 Nov. 1980 - NANCY.

[PAR 72] PARNAS D.L.,

A Technique for software modules specification with examples.
Com. ACM - May 1972.

[PAR 77] PARNAS D.L.

Use of Abstract interfaces in the development of software for Embeded Computer Systems.
Dept of Computer Science - Univ. of North Cordina at diapol Hil. Naval Research Laboratory WASHINGTON June 1977.

[PTH 82] PERRIN G.R., THOMESSE J.P.

Cours de DEA - Option parallélisme - Université de NANCY I - 1982.

[PER 82] PERRIN G.R.

Specification and verification of communications of processes.
Rapport interne CRIN n° 82 - R - 020.

[PER 83] PERRIN G.R.

Sémantique d'un système parallèle.
Rapport interne CRIN 83 - R - 024.

- R -

[RAY 81] RAYNAL M.

Contribution à l'étude de la coopération entre processus dans les langages et les systèmes informatiques.
Doctorat ès sciences - Université de Rennes - Avril 1981.

[ROB 77] ROBERT P., VERJUS J.P.

Towards autonomus - Description of Synchronisation Modules.
Proc. IFIP Congress - North Holland - (1977) p. 981-986.

- S -

[SCE 80] FALLOUR H., FAULLE C., FEBVRE J., KRONENTAL M., MEMMI G., MINEL F., SIMON J.J., VOJNOVIC D.

Projet SCEPTRE : Proposition de standard de noyau d' exécutif temps réel.
B.N.I. convention n° 79.2.36.055 - Mai 1980.

[SCO 71] SCOTT D.S.

"The Lattice of flow diagrams"
Symposium on the semantics of Algorithmic Languages.
Ed. E- Engeler Lect. Notes in comp. Sciences (1971).

[SIL 79] SILBERCHATZ A.

Communication and Synchronisation in distributed Systems.
I.E.E.E. tran. on soft. Eng. Vol. SE 5,6 (Nov. 1979)
p. 242 - 264.

[SIL 81] SILBERCHATZ A.

Port-Direct communication.
The Computer Journal Vol. 24 n° 1 - 1981.

[SLO 81] SLOMAN M., MAGEE J, KRAMER J.

Communication system architecture for Monitoring and Control in Cool Mines.
Jan. 81 - Research Report DCC 81/1. Department of computing. Imperial College - LONDON SW7.

[SOK 80] SOK S.

Evolutivité du logiciel.
Thèse de 3ème Cycle Université de NANCY I - 1980.

- V -

[VAN 81] VAN DEN BOS J.

Comments on ADA process communication.
Department of informatics - University of Nymegen - NYMEGEN - Netherlands.

[VIT 81] VITINS M.

Requirements specifications for Industrial Real-Time Automation Systems.
KLR 81 - 59 C - 75 pages. April 1981.

- W -

[WIR 77] WIRTH N.

Systematic Programming, an Introduction.
Prentice Hall (1973).

[WAL 72] WALDEN D.C.

A systems for interprocess communication in a ressource staring computer Network.
Com. ACM. Vol. 15, n° 4, p. 221 - 230. Ap. 72.

NOM DE L'ETUDIANT : ZAKARI Abdelouahed

NATURE DE LA THESE : Doctorat 3ème cycle en Informatique

VU, APPROUVE ET PERMIS D'IMPRIMER

NANCY, le - [illegible] 1984

LE PRESIDENT DE L'UNIVERSITE DE NANCY I

UNIVERSITE DE NANCY I

R. MAINARD

RESUME

Dans la classe d'applications temps réel réparties, le concept de communication a constitué le concept noyau de notre démarche. Il intervient aussi bien à une phase de décomposition de l'application en modules communicants, qu'à une phase de description de cette application, à l'aide d'un langage de conception, sans tenir compte des choix de réalisation.

Nous avons appelé ce langage **FLEXI.** Les outils de communication prédéfinis dans ce langage sont variés (synchrones ou asychrones, la communication fait intervenir deux interlocuteurs ou plus), et permettent de satisfaire les besoins de la classe d'application choisie. Nous proposons à l'aide de FLEXI d'établir des contrôles statiques de cohérence liés à la communication, ainsi que de permettre un passage aussi bien à la simulation qu'à la réalisation à partir de la même description.

Nous faisons dans nos travaux l'hypothèse de l'existence d'un noyau temps réel sur chaque machine cible, et d'un système de communication reliant ces différentes machines.

MOTS CLES

Temps réel, traitement de procédés industriels, parallélisme, abstraction, modularité, langage de conception, répartiti module, communication, port, message, connexion.

-=-=-=-=-