CONKER : un modèle de répartition pour processus communicants. Application à OCCAM

Michel Riveill

▶ To cite this version:

Michel Riveill. CONKER : un modèle de répartition pour processus communicants. Application à OCCAM. Réseaux et télécommunications [cs.NI]. Institut National Polytechnique de Grenoble - INPG, 1987. Français. NNT : . tel-00010614

HAL Id: tel-00010614
https://theses.hal.science/tel-00010614v1

Submitted on 13 Oct 2005

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THESE

présentée par

RIVEILL Michel

pour obtenir le titre de

DOCTEUR
de l'INSTITUT NATIONAL POLYTECHNIQUE
DE GRENOBLE

(arrêté ministériel du 5 juillet 1984)

Spécialité : INFORMATIQUE

CONKER : un modèle de répartition pour processus communicants. Application à OCCAM

Soutenue le 13 février 1987 devant la Commission d'Examen composée de :

MM.	Jacques	MOSSIERE	Président
	Jean	FERRIE	Rapporteurs
	André	SCHIPER	
	Guy	MAZARE	
	Traian	MUNTEAN	
	Jean-Pierre	VERJUS	

Thèse préparée au sein du Laboratoire de Génie Informatique.

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Président : Daniel BLOCH

Année 1987

Vice - Présidents : René CARRE
Jean-Marie PIERRARD

Professeurs des Universités

BARIBAUD Michel	ENSERG
BARRAUD Alain	ENSIEG
BAUDELET Bernard	ENSPG
BEAUFILS Jean-Pierre	ENSEEG
BESSON Jean	ENSEEG
BLIMAN Samuel	ENSERG
BLOCH Daniel	ENSPG
BOIS Philippe	ENSHMG
BONNETAIN Lucien	ENSEEG
BOUVARD Maurice	ENSHMG
BRISSONNEAU Pierre	ENSIEG
BRUNET Yves	IUFA
BUYLE-BODIN Maurice	ENSERG
CAILLERIE Denis	ENSHMG
CAVAIGNAC Jean-François	ENSPG
CHARTIER Germain	ENSPG
CHENEVIER Pierre	ENSERG
CHERADAME Hervé	UFR PGP
CHERUY Arlette	ENSIEG
CHIAVERINA Jean	UFR PGP
CHOVET Alain	ENSERG
COHEN Joseph	ENSERG
COUMES André	ENSERG
DARVE Félix	ENSHMG
DELLA-DORA Jean	ENSIMAG
DEPORTES Jacques	ENSPG
DOLMAZON Jean-Mar	ENSERG
DURAND Francis	ENSEEG
DURAND Jean-Louis	ENSIEG
FONLUPTJean	ENSIMAG
FOULARD Claude	ENSIEG
GANDINI Alessandro	UFR PGP
GAUBERT Claude	ENSPG
GENTIL Pierre	ENSERG
GREVEN Hélène	IUFA
GUERIN Bernard	ENSERG
GUYOT Pierre	ENSEEG
IVANES Marcel	ENSIEG
JAUSSAUD Pierre	ENSIEG
JOUBERT Pierre	ENSIEG
JOURDAIN Geneviève	ENSIEG
LACOUME Jean-Louis	ENSIEG
LESIEUR Marcel	ENSHMG
LESPINARD Georges	ENSHMG
LONGEQUEUE Jean-Pierre	ENSPG
LOUCHET François	ENSEEG
MASSE Philippe	ENSIEG
MASSELOT Christian	ENSIEG
MAZARE Guy	ENSIMAG
MOREAU René	ENSHMG
MORET Roger	ENSIEG
MOSSIERE Jacques	ENSIMAG
OBLED Charles	ENSHMG
OZIL Patrick	ENSEEG
PARIAUD Jean-Charles	ENSEEG
PAUTHENET René	ENSIEG
PERRET René	ENSIEG
PERRET Robert	ENSIEG
PIAU Jean-Michel	ENSHMG
POUPOT Christian	ENSERG
SAUCIER Gabrielle	ENSIMAG
SCHLENKER Claire	ENSPG
SCHLENKER Michel	ENSPG
SERMET PIERRE	ENSERG
SILVY Jacques	UFR PGP
SIRIEYS Pierre	ENSHMG
SOHM Jean-Claude	ENSEEG
SOLER Jean-Louis	ENSIMAG
SOUQUET Jean-Louis	ENSEEG
TROMPETTE Philippe	ENSHMG
VEILLON Gérard	ENSIMAG
ZADWORNY François	ENSERG

Professeur Université des Sciences Sociales (Grenoble II)

BOLLIET Louis

Personnes ayant obtenu le diplôme d'HABILITATION A DIRIGER DES RECHERCHES

BECKER Monique
BINDER Zdenek
CHASSERY Jean-Marc
COEY John
COLINET Catherine
COMMAULT Christian
CORNUEJOLS Gérard
DALARD Francis
DANES Florin
DEROO Daniel
DIARD Jean Paul
DION Jean Michel
DUGARD Luc
DURAND Robert
GALERIE Alain
GAUTHIER Jean Paul
GENTIL Sylviane
PLA Fernand
GHIBAUDO Gérard
HAMAR Sylvaine
LADET Pierre
LATOMBE Claudine
LE GORREC Bernard
MADAR Roland
MULLER Jean
NGUYEN TRONG Bernadette
TCHUENTE Maurice
VINCENT Henri

Chercheurs du C.N.R.S

Directeurs de recherche 1ère Classe

CAILLET Marcel
CARRE René
FRUCHART Robert
JORRAND Philippe
LANDAU Ioan
MARTIN

Directeurs de recherche 2ème Classe

ALEMANY Antoine
ALLIBERT Colette
ALLIBERT Michel
ANSARA Ibrahim
ARMAND Michel
BINDER Gilbert
BONNET Roland
BORNARD Guy
CALMET Jacques
DAVID René
DRIOLE Jean
ESCUDIER Pierre
EUSTATHOPOULOS Nicolas
JOUD Jean Charles
KAMARINOS Georges
KLEITZ Michel
KOFMAN Walter
LEJEUNE Gérard
MERMET Jean
MUNIER Jacques
SENATEUR Jean-Pierre
SUERY Michel
TEDOSIU
WACK Bernard

Personnalités agrées à titre permanent à diriger des travaux de recherche (décision du conseil scientifique)

E.N.S.E.E.G

BERNARD Claude
CHATILLON Catherine
CHATILLON Christian
COULON Michel
DIARD Jean-Paul
FOSTER Panayotis
HAMMOU Abdelkader
MALMEJAC Yves
MARTIN GARIN Régina
SAINTFORT Paul
SARRAZIN Pierre
SIMON Jean-Paul
TOUZAIN Philippe
URBAIN Georges

E.N.S.E.R.G

BOREL Joseph
CHOVET Alain
DOLMAZON Jean-Marc
HERAULT Jeanny

E.N.S.I.E.G

DESCHIZEAUX Pierre
GLANGEAUD François
PERARD Jacques
REINISCH Raymond

E.N.S.H.G

BOIS Daniel
DARVE Félix
MICHEL Jean-Marie
ROWE Alain
VAUCLIN Michel

E.N.S.I.M.A.G

BERT Didier
COURTIN Jacques
COURTOIS Bernard
DELLA DORA Jean
FONLUPT Jean
SIFAKIS Joseph

E.F.P.G

CHARUEL Robert

C.E.N.G

CADET Jean
COEURE Philippe
DELHAYE Jean-Marc
DUPUY Michel
JOUVE Hubert
NICOLAU Yvan
NIFENECKER Hervé
PERROUD Paul
PEUZIN Jean-Claude
TAIB Maurice
VINCENDON Marc

Laboratoires extérieurs

C.N.E.T

DEMOULIN Eric
DEVINE
GERBER Roland
MERCKEL Gérard
PAULEAU Yves

ECOLE NATIONALE SUPERIEURE DES MINES DE SAINT-ETIENNE

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

PROFESSEURS DE 1ère CATEGORIE

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

PROFESSEURS DE 2ème CATEGORIE

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

DIRECTEUR DE RECHERCHE

LESBATS Pierre	Métallurgie

MAITRE DE RECHERCHE

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

Personalités habilitées à diriger des travaux de recherche

DRIVER Julian	Métallurgie
GUILHOT Bernard	Chimie
THOMAS Gérard	Chimie

Professeurs à l'UER de Sciences de Saint-Etienne

VERGNAUD Jean-Maurice	Chimie des Matériaux et Chimie Industrielle

CONKER : un modèle de répartition pour processus communicants. Application à OCCAM.

Résumé

CONKER est un noyau de communication construit à l'aide du modèle CSP. Une application décrite à l'aide de CONKER sera composée de connecteurs qui sont les objets chargés de la communication-synchronisation et de processus qui se chargent du traitement.

Chaque connecteur du noyau réalise un protocole de communication particulier entre les processus "applications". Ceux-ci n'utilisent que des primitives d'envoi et de réception de messages, de manière homogène et transparente à la réalisation des connecteurs qui les relient.

Cette caractéristique de transparence, ainsi que la possibilité d'implémenter CONKER sur un ensemble de systèmes "hôtes hétérogènes", assurent une transportabilité aisée des applications en environnement multi-processeurs, ou sur un réseau hétérogène.

De part sa conception, CONKER permet de mettre à la disposition des processus "applications", des types de connecteurs, construits de façon incrémentale à la demande, et utilisant pour cela, les calculs de processus issus du modèle CSP.

Mots clés : modèle de communication, système réparti, application "temps critique", CSP, OCCAM, Transputer.

CONKER : a model for distributed communicating processes. Application to OCCAM.

Summary

This theisis proposes a model CONKER for distributed systems of communicating sequential processes (Hoare 's CSP). An application program will be structured in CONKER as a set of processes (processing objects) commmunicating throught a set of connectors (communication-synchronization objects).

Each connector implement a specific and possibly complex communication protocol between application processes which still use the CSP basic communication protocol (rendez-vous for input-output primitives). Therefore this basic protocol is used transparently to the connectors complex protocol. More over the connectors can realise a specific synchronisation control scheme between communicating processes (synchronous or asynchronous diffusion, time constrained communication, total asynchrony, etc).

This transparency in CONKER allows to easy implement a distributed application on a multiprocessor or a heterogeneous network of different host machines. Our model also facilitates transportability and improve reliability. The connectors are strongly typed objects and for which an incremental construction is proposed.

An experimental implementation presented in the framework of a distributed robotic systems.

Je tiens à remercier :

Mr Jacques MOSSIERE, Professeur à l'Institut National Polytechnique de Grenoble et Directeur du Laboratoire de Génie Informatique, qui m'a fait l'honneur de présider le jury de cette thèse.

Mr Jean FERRIE, Professeur à l'Université de Montpellier,
Mr André SCHIPER, Professeur à l'Ecole Polytechnique Fédérale de Lausanne, qui ont bien voulu être rapporteur de ce travail. Les échanges que nous avons eus, m'ont certainement permis d'améliorer la présentation de ce manuscrit.

Mr Guy MAZARE, Professeur à l'Institut National Polytechnique de Grenoble, pour la confiance qu'il m'a manifestée en m'accueillant dans son équipe et pour le soutien amical qu'il m'a accordé tout au long de ce travail.

Mr Traian MUNTEAN, Directeur de Recherche au Laboratoire de Génie Informatique, pour les idées et les encouragements qu'il m'a prodigués lors de la réalisation de ce travail. Sans les nombreuses discussions que nous avons eues, cette thèse n'aurait sans doute jamais vu le jour.

Mr Jean-Pierre VERJUS, Professeur à l'Université de Rennes, qui a bien voulu accepter de participer à ce jury.

Je remercie également tous ceux qui ont collaboré de près ou de loin à cette réalisation : Melle Cécile ROISIN, MM. Christian TRICOT, Dominique DECOUCHANT, Victor SANCHEZ, les personnes de l'équipe robotique d'ITMI.

Pour terminer je ne voudrais pas oublier tous ceux qui ont accepté de relire ce manuscrit afin d'en corriger les "fôtes d'ortografes", qu'ils trouvent ici mes remerciements pour ce travail peu gratifiant mais nécessaire.

Introduction et plan de l'étude

La démarche suivie dans ce travail est basée sur une sémantique précise du parallélisme dans les systèmes de processus communicants. Parmi les différents modèles proposés dans la littérature, nous avons choisi le modèle CSP car il inclut un schéma primitif de communication et de synchronisation : le rendez-vous synchrone. Celui-ci permet la construction incrémentale d'autres types de communication-synchronisation plus complexes, nécessaires au développement des applications.

CONKER est un système de communication, intégrant un modèle sémantique précis basé sur le calcul développé par HOARE [HOARE81]. Il permet l'écriture d'une application distribuée. Celle-ci sera composée de connecteurs (objets chargés de la communication-synchronisation) et de processus (objets chargés du traitement). Le noyau du système CONKER permet à ces différentes entités de communiquer par un rendez-vous temporel, et par la-même de se synchroniser.

Chaque type de connecteur de CONKER est construit comme un ensemble de processus coopérants. Ces processus (internes au connecteur) réalisent un protocole de communication particulier entre les processus "applications" qu'ils relient.

Pour communiquer ou se synchroniser, les processus "applications" utiliseront des primitives d'envoi et de réception de messages, indépendantes de la réalisation des connecteurs et indépendantes de leur répartition. De cette manière, le changement du type de communication et de synchronisation entre deux processus de l'application pourra se faire sans aucune modification de leur programme.

Cette caractéristique de transparence, ainsi que la possibilité d'implémenter CONKER sur un ensemble de systèmes de traitements "hôtes hétérogènes", assurent la transportabilité des applications en environnement multi-processeurs ou sur un réseau hétérogène.

CONKER a été développé initialement pour permettre l'écriture aisée d'applications "temps-critique". Nous les définirons brièvement comme étant :

- **non terminantes :** elles délivrent des résultats en fonction des entrées qu'elles reçoivent.

- **dépendantes :** ces applications inter-agissent en permanence avec leurs environnements.

- **parallèles :** elles sont décrites par des tâches concurrentes coopérant par communication.

- **temporelles :** le comportement de chaque application doit obéir à certaines règles de synchronisation dans le temps, fixées par la structure de l'application

ou par l'environnement avec lequel chaque application inter-agit.

- **sûres** : elles exigent une grande sûreté de fonctionnement.

Pour programmer cette classe d'application, communément appelée "temps réel", il existe de nombreuses propositions de langage (ADA [CII80], Fortran_TR, LTR, ...) et de nombreuses propositions de système (iRMX86 [INTEL82], RSR-11M [Digit79], SCEPTRE [BROWA84], ...). Tous offrent une possibilité de manipuler une notion de temps, mais aucun ne permet de contrôler la durée de chaque action. Ce projet vise à proposer simultanément un nouveau langage pour la programmation des applications "temps-critique" : **OCCAM-TO** (langage parallèle permettant d'exprimer des contraintes de synchronisation et des contraintes de délai) et une couche système : **CONKER** permettant la réalisation de différents modes de communication/synchronisation nécessaires à une programmation correcte.

Néanmoins CONKER peut être utilisé indépendamment d'OCCAM-TO. Il suffit d'introduire dans le langage de programmation choisi, l'ensemble des commandes nécessaires à l'utilisation des services de communication/synchronisation de CONKER, et de réaliser le noyau CONKER à l'aide du système de traitement hôte.

Actuellement, CONKER a été développé sur différents noyaux de système (iRMX86, UNIX V7 et 4.2 BSD, OCCAM) et deux implémentations sont en cours de réalisation. La première de ces implémentations sera réalisée sur une architecture multi-processeurs (architecture CESAR et système CLEOPATRE [CARTO84]) et la seconde sur une machine multi-Transputers (en réutilisant le système développé en OCCAM).

Le chapitre 1 est une étude, centrée uniquement sur les mécanismes de communication/synchronisation nécessaire à l'exécution d'une application répartie. Dans ce type d'application, la communication permet à la fois d'échanger de l'information et de synchroniser des processus. Nous verrons dans ce chapitre, comment les différentes propositions de système assurent ces deux services.

Aprés le rapide tour d'horizon du chapitre 1 et la définition du langage OCCAM étendu au chapitre 2, nous ferons au chapitre 3 la proposition du modèle de système CONKER. Cette description du modèle de système se fera en utilisant le langage OCCAM, qui nous a permis de spécifier le modèle et d'en réaliser une implémentation. CONKER offre un seul mode de communication/synchronisation : le rendez-vous temporel, mais il permet de construire (et de prouver) les modes de communication nécessaires au déroulement de l'application.

Dans le chapitre 4, nous définirons les fonctionnalités des différentes entités présentes dans le système CONKER, afin d'en permettre une implantation à l'aide d'un système de traitement hôte possédant un minimum de propriétés (surtout si l'implantation doit respecter certaines contraintes temporelles).

Pour terminer la présentation de ce travail, nous décrirons dans le chapitre 5 une implantation particulière de ce noyau pour une application de robotique (étude réalisée avec la collaboration de la société ITMI). Elle est une illustration de différents types de communication/synchronisation construits.

Dans le chapitre 6, nous proposerons quelques types de connecteurs nécessaires à la programmation de différentes classes d'applications à l'aide de CONKER (implémentation répartie des constructeurs OCCAM, implémentation d'un système de gestion d'événements multiples).

I. Communication et synchronisation dans des systèmes distribués

Résumé

Dans ce premier chapitre nous allons tenter de faire un rapide tour d'horizon de certains systèmes distribués, existants ou en cours de définition, afin de pouvoir présenter quelques concepts, et, de pouvoir par la suite, motiver les choix qui ont conduit à la définition de CONKER.

Pour chaque système présenté, nous essayerons de dégager en quelques lignes une vue d'ensemble du système, puis nous en regarderons un aspect spécifique afin d'introduire un concept nouveau. En fin de chapitre, nous tenterons de faire une étude comparée des différentes propositions en matière de synchronisation et de communication faites par les différents systèmes.

1. Les systèmes distribués

1.1 Brève description

Les **systèmes distribués** [SHATZ84] sont constitués par des processeurs autonomes interconnectés soit par un réseau, soit par un bus. Chaque processeur constitue un noeud de l'architecture maillée, il possède un minimum de mémoire qui lui permet d'exécuter des programmes et d'accéder aux périphériques locaux. Le système ne possède généralement pas de contrôleur central.

Un **programme distribué** est constitué de différents modules, qui s'exécutent de façon asynchrone sur différents noeuds, et qui coopèrent pour participer à la réalisation d'un même but. Les modules coopèrent en s'échangeant de l'information par envoi de messages.

Dans cette partie nous allons essayer de présenter **quels sont les mécanismes de communication et de synchronisation que doit offrir un noyau de système de traitement distribué.**

Pour nous permettre de mieux visualiser les différents problèmes, nous allons découper le système en trois couches :

1) La couche de base, appelée aussi **la couche réseau,** assure le transport des messages d'un noeud à un autre. Cette couche assure les niveaux 1 à 4 (niveau transport) du modèle ISO/OSI. Elle est nécessaire uniquement pour envoyer ou recevoir des messages d'un site à un autre (selon les réseaux, un niveau session peut être nécessaire ou au moins agréable). Le protocole d'échange implémenté entre les sites est étroitement dépendant des services offerts par le réseau. La réalisation de cette partie du système ne nous intéressera pas directement mais nous utiliserons ses services pour communiquer entre différents sites. Sa réalisation peut être fort différente selon le type de machine reliée et leur mode de connexion (réseau à diffusion, architecture matricielle ou hypercube, ...).

2) A l'aide de la couche réseau et du système de traitement présent sur le site, on peut construire un **noyau de communication** spécifique à chaque noeud. Il supporte l'exécution répartie de programmes parallèles. Ce noyau de communication, présent sur chaque noeud, réalise des mécanismes de communication et de synchronisation spécifiques au type d'application dont il supporte l'exécution. Chaque noyau gère localement les ressources partageables du site, et assure en outre une protection des données encapsulées par les processus et des messages échangés entre les processus. C'est ce noyau qui permet d'accéder aux périphériques présents sur le noeud, par l'intermédiaire de primitives ou de commandes de haut niveau.

3) La troisième couche est constituée par les programmes distribués qui s'exécutent de façon concurrente. Ils utilisent toutes les facilités offertes par le noyau de communication pour coopérer.

Dans cette étude, nous essayerons de définir le noyau de communication qui doit être réalisé sur chaque site, afin de pouvoir supporter l'exécution de programmes parallèles.

1.2 Quelle communication/synchronisation ?

Une communication entre processus induit nécessairement une synchronisation particulière. En règle générale, un système implémente plusieurs modes de communication/synchronisation : pour chaque envoi de message, le processus émetteur est bloqué pendant un temps plus ou moins long. Quatre modèles de base sont habituellement utilisés [LISKO79] : asynchrone, synchrone, rendez-vous et invocation.

i) Communication asynchrone

Le processus émetteur est bloqué juste le temps nécessaire au système pour prendre en compte sa demande : c'est l'envoi asynchrone utilisé dans la plupart des systèmes répartis (par exemple : CONIC [KRAME85] et DEMOS [BASKE77]).

Dans ce cas, le processus émetteur prend seul l'initiative de la communication, sans se préoccuper de l'état du récepteur. Pour le processus émetteur, la communication a toujours lieu : après avoir émis le message, il continue son traitement local, comme si le message avait été correctement acheminé et traité par le processus destinataire. Il ne se préoccupe pas du bon acheminement du message.

Néanmoins, il peut exister différents types de contrôle permettant au processus émetteur d'être prévenu ultérieurement d'une "erreur"[1] dans la communication. Un premier contrôle peut être assuré par le noyau de communication, qui précise si le message a été correctement mis dans la "porte"[2] associée au processus destinataire du message (après ce contrôle l'émetteur sait simplement que le message a été correctement acheminé et que la porte destinataire existe). Un deuxième contrôle peut être à l'initiative du processus destinataire du message, qui prévient l'émetteur que le message a été pris en compte.

Dans ce type de synchronisation, le noyau doit nécessairement pouvoir stocker les messages émis et non encore reçus (l'asynchronisme infini théorique devient borné lors d'une implémentation, ce qui peut entraîner un risque d'interblocage).

ii) Communication synchrone

Le processus émetteur est bloqué jusqu'au moment où son message est déposé sur la "porte" destinataire : c'est l'envoi synchrone utilisé par exemple dans le système CHORUS [ZIMME84] ou SR [ANDRE81].

On retrouve l'ensemble des problèmes liés à la communication asynchrone, hormis le fait qu'un processus émetteur est prévenu de la non existence d'une "porte"

1. *Nous appellerons erreur dans la communication toutes erreurs, pannes, ... qui n'ont pas permis d'acheminer correctement le message vers le processus destinataire.*

2. *Une porte est ici uniquement réceptrice et unidirectionelle : elle permet la réception des messages par le processus selon un protocole de synchronisation particulier (généralement First-In/First-Out). Chaque porte possède un nom global et est associée, à un instant donné, qu'à un seul processus.*

destinataire. Mais dans ces deux cas, le processus émetteur n'a aucun contrôle sur la durée de la communication, et ne peut pas savoir à quel moment le message a été pris en compte par le processus destinataire.

iii) Rendez-vous

Le processus émetteur est bloqué jusqu'à ce que le processus destinataire reçoive son message : c'est le rendez-vous utilisé dans le langage CSP et implanté directement dans le Transputer. Nous l'avons choisi comme mode de communication de base dans CONKER [MUNTE85].

Dans ce cas, un message émis par un processus est toujours reçu par le processus récepteur. L'initiative de la communication est partagée : pour que la communication (et donc le rendez-vous) puisse avoir lieu, il est nécessaire que les deux processus qui désirent communiquer soient prêts ; si l'un deux ne l'est pas, alors l'autre l'attend.

Dans ce mode de communication, le noyau n'a pas besoin de stocker les messages en cours de transfert ; ils passent directement du processus émetteur au processus destinataire. Par contre, la communication par rendez-vous introduit un risque d'interblocage, qui peut être détecté par le noyau, mais qui doit être résolu par l'utilisateur. Il s'agit généralement d'une mauvaise programmation de l'application.

Le processus émetteur peut connaître la durée de la communication, et peut imposer à celle-ci d'être réalisée avant une date précise.

iv) Appel de procédure à distance ou invocation

L'émetteur est bloqué jusqu'à ce que le processus destinataire du message en ait effectué le traitement et ait envoyé un message de réponse : c'est l'appel de procédure (éventuellement à distance) ou **invocation.** Ce mode de communication est utilisé par exemple dans DP (Distributed Process) [BRINC78], ARGUS [ALLEN85] ou MASCOT [SIMPS79].

L'invocation (ou appel de procédure) est une extension de l'appel de procédure connu dans les langages séquentiels. L'émetteur a l'initiative de l'échange, il envoie les paramètres de l'appel au processus distant, et se met en attente de la fin d'exécution de la procédure distante. A sa terminaison, (si elle est correcte), il reçoit les paramètres de retour. Comme pour la communication synchrone, une mauvaise programmation peut entraîner des risques d'interblocage.

v) Synthèse

Pour le processus récepteur, la réception d'un message n'est effective que lorsqu'il demande explicitement cette réception : les arrivées de message ne perturbent généralement pas son déroulement (sauf dans le cas de messages spécifiques dont l'arrivée interrompt son exécution pour déclencher un traitement d'exception dans son propre contexte interrompu). Du point de vue de la synchronisation, la réception d'un message signifie pour le récepteur, que le traitement qui précède l'émission du message chez l'émetteur a été effectué.

Les quatre modes de communications présentés représentent, pour l'émetteur, un couplage de plus en plus fort entre lui et le récepteur.

Pour les modèles et les preuves de programmes, les modes de communication par rendez-vous et par appel de procédure sont plus simples à prendre en compte et, de fait, de nombreux modèles de ce type existent. CCS de Milner et CSP de Hoare sont deux modèles basés sur le rendez-vous.

1.3 Un complément nécessaire : le temps

De nombreuses variantes aux schémas de communication précédents ont été introduites par différents auteurs. Nous proposons d'inclure systématiquement dans les primitives de communication un contrôle du temps [KRAME81].

Ce contrôle du temps, permet :

- **à l'émetteur** de préciser l'attente maximale qu'il est prêt à supporter pour réaliser une communication.[3] Après ce délai, le message n'a plus aucun sens pour l'application et la communication n'aura pas lieu.

- **au récepteur**, de préciser l'attente maximale qu'il est prêt à supporter pour recevoir un message.

Cet ajout devient nécessaire pour la programmation d'application "temps critique", dans laquelle les processus réagissent en fonction de leur environnement. Il permet d'introduire la notion de rendez-vous temporel dans le cas où les deux processus communiquent par rendez-vous. Dans ce cas, en effet, pour qu'une communication puisse avoir lieu il est nécessaire que les deux processus soient prêts dans le même intervalle de temps.

Le temps choisi pour contrôler la durée d'une communication pourra être le temps de l'environnement (c'est le temps communément utilisé dans les langages "temps réel") mais il pourra être aussi un temps propre à chaque processus (temps isomorphe ou non au temps de l'application) ou un temps logique.

1.4 Comment désigner le correspondant

La désignation d'une entité est étroitement dépendante d'un référentiel de base. Par exemple plusieurs entités peuvent avoir un même nom par rapport à des référentiels différents.

---

3. . *Selon le mode de synchronisation, pour qu'une communication soit réalisée, il faut que :*

- *le système ait pris en compte la demande (communication asynchrone).*

- *le message ait été déposé dans la porte destinataire (communication synchrone).*

- *le récepteur ait reçu le message (rendez-vous).*

- *le récepteur ait donné sa réponse (invocation).*

Quelques définitions habituelles :

- un identificateur est dit **global,** lorsqu'il a la même signification quel que soit l'environnement où il est utilisé. Au contraire, un nom **local** n'a de sens que dans un environnement donné (éventuellement l'objet désigné peut apparaître dans un autre référentiel avec un autre nom).

- un **nom spécifique** désignera une entité bien définie qui ne pourra être remplacée par aucune autre dans le système. Ce nom est unique : il désigne explicitement cette entité et elle seule.

- un **nom fonctionnel** représente une classe d'entité jouant un rôle analogue ou assurant une fonction identique. Ce nom désigne l'une des quelconques entités de la classe (toutes les entités de la classe ont le même nom fonctionnel).

- si l'on désire accéder à un ensemble quelconque d'entités, on peut utiliser un **nom générique.** Un tel nom désigne un groupe d'entités qui peuvent également posséder chacune un nom spécifique ou fonctionnel.

- la désignation est **explicite** si elle comporte de façon formelle la localisation de l'entité. A l'opposé, un nom sera dit **implicite** s'il ne fait aucune référence à la localisation de l'entité (ceci permet de déplacer une entité d'un site à un autre sans qu'elle change de nom).

2. ARGUS

ARGUS intègre un langage et un système pour la programmation distribuée. Chaque noeud physique d'un système réparti ARGUS [LISKO83, ALLEN85] est composé d'au moins un processeur, de mémoire locale et de périphériques externes. Chaque noeud est relié à son (ou ses) voisins par un réseau de communication.

i) Buts

Les applications visées sont essentiellement les applications transactionnelles : système bancaire, système de réservation de places d'avion, système réparti de base de données. Dans ces applications, l'aspect sécurité de l'information est plus important que l'aspect temporel.

ARGUS définit un modèle pour la programmation sur les systèmes distribués : définition d'un langage issu du langage orienté objet CLU et d'un noyau. Les objectifs du système sont : la continuité de service, la reconfiguration, l'autonomie, la distribution, le maintien de la cohérence, la concurrence et la consistance.

Ces objectifs, définis pour les applications transactionnelles, nous semble primordiaux pour les applications "temps réels". En effet, les caractéristiques essentielles du système ARGUS se retrouvent bien souvent dans les systèmes "temps réel", même si les raisons qui les ont amenées ne sont pas les mêmes.

ii) Action atomique répartie

Afin de répondre aux objectifs visés, ARGUS a utilisé la notion d'action atomique répartie qui généralise le concept d'action atomique des systèmes transactionnels. Ce concept, déjà utilisé par ailleurs [GRAY76] est fondamental pour ce système.

L'état d'un système distribué consiste en une collection d'objets qui résident en différents points du réseau. Une activité transforme certains objets du système d'un état initial à un état final, en passant par un certain nombre d'états intermédiaires. Deux propriétés définissent une action atomique : l'indivisibilité et le recouvrement.

Indivisibilité : pendant une exécution, une action atomique ne fait apparaître aucune autre action (l'action atomique est le plus petit grain observable), seul l'état initial et l'état final de cette action sont observables. On peut traduire cette propriété d'une autre façon : lorsque deux actions atomiques sont exécutées en parallèle, tout se passe comme si elles étaient exécutées en série (sérialisation des actions).

Recouvrement : l'action atomique est exécutée complètement si elle est correcte et à sa terminaison seul l'état final est observable. En cas d'erreur, l'action n'a aucun effet sur le système et seul l'état initial peut être observé.

Dans ARGUS, une action atomique peut être répartie de façon transparente sur différents processeurs. Dans ce schéma, une implantation correcte d'une action atomique dépend avant tout de la correction des primitives utilisées.

iii) Les objets actifs et la communication

Un programme ARGUS est composé d'un ensemble de **"guardians"** ; chacun possède des données locales. Ils s'exécutent de façon indépendante sans pouvoir migrer d'un site à un autre. Afin de pouvoir implémenter les propriétés précisées ci-dessus (continuité de service, reconfiguration, autonomie, distribution, ...) un guardian encapsule des données rémanentes aux pannes (les données peuvent être dupliquées et enregistrées sur disque). L'ensemble de ces données constitue l'état observable du système.

Chaque "guardian" encapsule des ressources et en contrôle l'accès par des "moniteurs". Un modèle de "guardian" implémente un type particulier de données sur lesquelles les opérations possibles sont implémentées par les "moniteurs".

Dans ARGUS, pour communiquer, un "guardian" invoque un "moniteur" auquel est associé un processus qui réalise le service demandé. ARGUS ne possède que ce mode de communication/synchronisation qui ne permet pas à un "moniteur" de faire un choix non-déterministe sur un ensemble de messages. En effet mission de message entraîne systématiquement la création d'un processus.

Chaque "guardian" possède des objets (instance de types de données abstraits), des processus dynamiques qui exécutent les "moniteurs" (durée de vie égale à celle nécessaire pour exécuter le "moniteur") et un processus particulier : "background" (ce processus est créé lors de la création du "guardian").

En cas de panne du site sur lequel s'exécute le guardian, le système régénère le "guardian" à l'aide de ses variables rémanentes et d'une section de reprise nommée "recover", définie par le programmeur et qui permet de réinitialiser les variables volatiles du "guardian".

3. CONIC

CONIC [KRAME85] a été développé à l'Impérial Collège de Londres afin de permettre l'évolution dynamique d'un système. En effet, il peut être nécessaire d'adapter le système à son environnement en perpétuelle évolution. Ce peut être l'environnement matériel (changement des connexions physiques), l'environnement technologique (nouveautés techniques) ou l'environnement des applications dont il supporte l'exécution (selon le type d'application il peut être nécessaire de construire de nouvelles primitives système).

En général, faire évoluer de façon incrémentale un système est un problème complexe car il est difficile de savoir, à chaque instant, à quelle version du système l'application s'adresse.

3.1 Configuration statique

La configuration statique est le modèle utilisé habituellement en programmation. On peut l'apparenter au cycle d'écriture de programme :

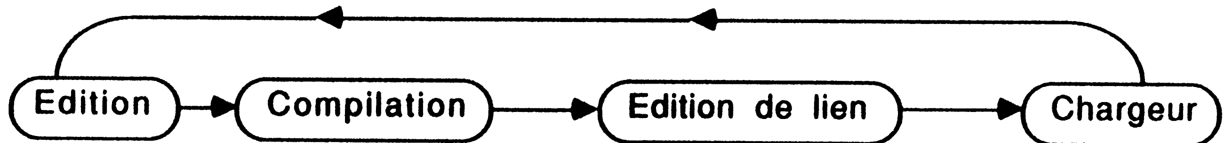


Figure 1. — Cycle de programmation statique.

Si on désire ajouter une fonctionnalité au programme, il est nécessaire de réécrire le code source et de recommencer le cycle. Pour un système de traitement, cela revient à prendre des spécifications et à en produire une image exécutable. Chaque partie du système doit être spécifiée au même instant, et si l'on désire modifier une partie, il est nécessaire d'arrêter le système, d'écrire de nouvelles spécifications, puis de charger une nouvelle image du système.

C'est cette approche qui a été prise dans la plupart des systèmes actuels : on procède par versions successives.

3.2 Configuration dynamique

La configuration dynamique permet de faire évoluer de façon incrémentale le système, en effectuant des modifications selon un calendrier non programmé et sans arrêter le système. En effet, s'il s'agit d'un système de traitement ou de communication qui est chargé de la surveillance d'un procédé physique, il peut être difficile d'arrêter le procédé, soit pour des raisons économiques, soit pour des raisons liées à la nature même du procédé.

Les spécifications de configuration décrivent la structure logique et physique du système. Changer de système revient à changer de spécifications en leur introduisant de nouveaux composants et en modifiant les composants existants. Les nouvelles spécifications sont obtenues après validation (changement compatible avec le système actuel). Après ces étapes, le "manager" de configuration charge le code nécessaire, crée ou détruit des composants, rompt ou attache des connexions. Le résultat est un nouveau système décrit par de nouvelles

spécifications. Le cycle de programmation s'apparente au schéma suivant :

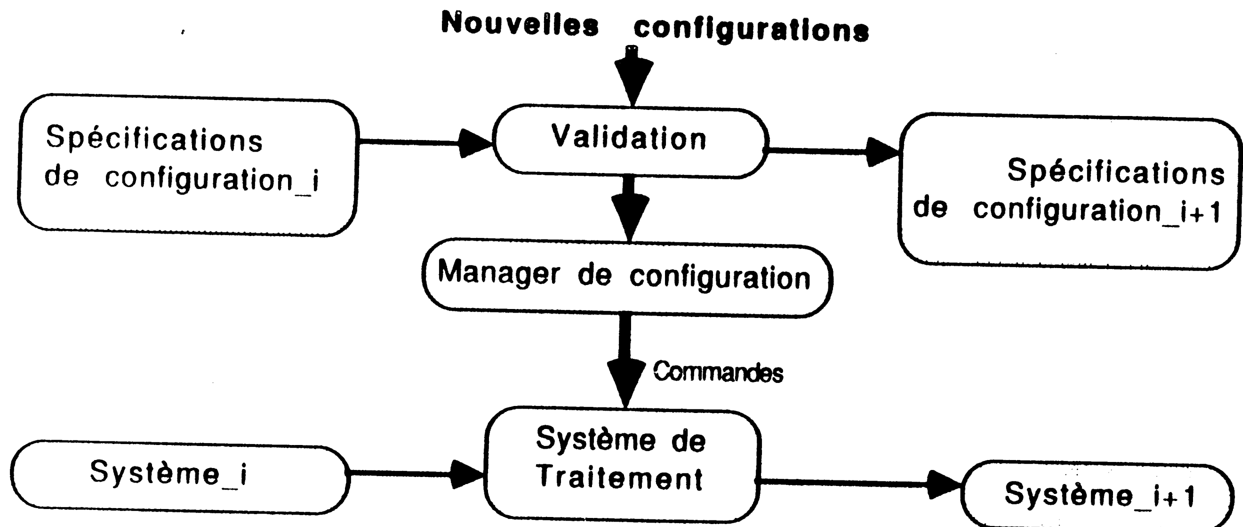


Figure 2. — Cycle de programmation dynamique.

Pour pouvoir assurer une évolution dynamique du système, il est nécessaire :

- que le langage permette la description du système en modules indépendants, chaque module ne référence alors que des objets locaux.
- que le système permette l'interconnexion des modules indépendamment de leur localisation : présence sur le système d'un noyau d'interconnexion.
- que tout appel à des objets ou modules externes soit géré par un processus d'interface afin de rendre homogène toute importation de modules.
- que la communication soit transparente, c'est à dire que la syntaxe et la sémantique d'une communication locale ou distante soient identiques quel que soit le mode d'échange réalisé.

3.3 Le système CONIC

Une application CONIC est composée d'un ensemble d'instances de modules interconnectés dynamiquement et s'exécutant sur différents sites.

i) Langage de programmation de CONIC

CONIC a été développé en PASCAL auquel ont été adjointes des primitives de communication par messages. Deux types de communications sont implémentés : l'envoi asynchrone et la communication par invocation. Chaque module possède une interface de communication définie par la liste des ports d'entrée et des ports de sortie. Chaque port peut être typé : type de la variable et mode d'échange (échange asynchrone ou invocation).

Afin de permettre la compilation indépendante de chaque module, CONIC ne possède pas d'objets (ou ressources) globaux.

Exemple de définition de module : tampon.

```
MODULE tampon ;
  ENTRYPORT entrée : message ;
  EXITPORT sortie : message ;
  { corps du programme }
  END.
```

ii) Système de configuration de CONIC

Chaque module précédemment défini est instancié puis interconnecté avec les autres instances de modules à l'aide de liens unidirectionnels (appel asynchrone) ou bidirectionnels (invocation). Pour cela le programmeur procède en trois étapes : définition des modules utilisés (USE) ; création des instances de modules (CREATE) ; interconnexion des instances de modules (LINK).

Exemple de configuration : producteur/consommateur.

```
MODULE producteur ;
  EXITPORT sortie : message ;
  { corps du programme }
  END.

MODULE consommateur ;
  ENTRYPORT entrée : message ;
  { corps du programme }
  END.

SYSTEM prod_cons ;
  USE tampon, producteur, consommateur ;
  CREATE file : tampon ;
  CREATE prod : producteur ;
  CREATE cons : consommateur ;
  LINK
    prod.sortie TO file.entrée ;
    file.sortie TO cons.entrée ;
  END.
```

Afin de permettre une programmation aisée, CONIC introduit la notion de groupe de modules, qui définit un nouveau module composé d'un ensemble de modules.

iii) Changement des spécifications

Pour pouvoir changer de système sans arrêter l'application, il suffit de modifier les spécifications. Elles sont ensuites comparées aux anciennes afin de savoir quelles sont les modifications à effectuer.

Le programmeur utilise le même ensemble de commandes (USE, CREATE, LINK), enrichi de trois autres permettant de défaire : les liens (UNLINK), les instances de module (DELETE), les modules utilisés (REMOVE). Puis par le premier ensemble de commandes, le programmeur remplace une instance d'un modèle par une

autre instance d'un autre modèle, ou étend un système par un nouveau modèle connu ou par une nouvelle instance.

Exemple : changement des spécifications du processus consommateur.

```
MODULE new_consommateur ;
 ENTRYPORT entrée : message ;
 { corps du programme }
 END.

CHANGE prod_cons ;
 UNLINK
  file.sortie FROM cons.entrée ;
 DELETE cons ;
 REMOVE consommateur ;
 USE new_consommateur ;
 CREATE cons : new_consommateur ;
 LINK
  file.sortie TO cons.entrée ;
 END.
```

Exemple : connexion directe sans tampon.

```
CHANGE prod_cons ;
 UNLINK
  prod.sortie FROM file.entrée ;
  file.sortie FROM cons.entrée ;
 DELETE file ;
 REMOVE tampon ;
 LINK
  prod.sortie TO cons.entrée ;
 END.
```

iv) Allocation dans CONIC

Cette étape permet d'allouer les modules aux processeurs physiques. Les liens d'interconnexion entre instances de module permettent de décrire les connexions physiques.

Exemple d'allocation du système sur deux noeuds :

```
SYSTEM prod_cons ;
 USE producteur, consommateur ;
 CREATE prod : producteur AT NODE( 0 ) ;
 CREATE cons : consommateur AT NODE( 1 ) ;
 LINK
  prod.sortie TO cons.entrée ;
 END.
```

4. DEMOS, EMBOS, OMPHALE

Ces trois systèmes ont des caractéristiques proches. DEMOS [BASKE77] qui est un système d'exploitation pour le CRAY-1, est le travail original qui a inspiré les autres études. EMBOS [OLSON85], réalisé pour une machine multi-processeurs avec mémoire partagée, a introduit la migration des modules et OMPHALE [DELAT84, GIEB84], la répartition du système sur une architecture réseau. La particularité de ce type d'architecture réside dans les contrôles effectués lors des échanges de messages et des protections associées.

Dans ces propositions, toutes les entités encapsulent des données propres et coopèrent par échanges de messages. La synchronisation est vue comme une conséquence de la communication.

Dans la suite, la terminologie utilisée sera celle d'OMPHALE.

4.1 Architecture

L'architecture OMPHALE pour application répartie voit l'ensemble des sites interconnectés comme une seule machine virtuelle, les problèmes de localisation et d'échanges d'information entre les sites sont rendus invisibles au niveau de l'application. .

Toute application se décompose en modules qui possèdent un seul port de communication. Un port est une structure de donnée adaptée à la communication. A un port est associé un ensemble de quais qui peuvent être ouverts ou fermés. A chaque quai est associée une notion de service. En effet l'arrivée d'un message sur un quai déclenchera par la suite une étape de traitement. Le mécanisme de fermeture des quais permet au module propriétaire du port de refuser momentanément d'assurer tel ou tel service, mais à aucun moment ce choix ne peut être fait en fonction de l'appelant éventuel (la réception d'un message est toujours anonyme).

4.2 Communication/synchronisation

Les modules communiquent entre eux uniquement par échanges asynchrones de messages grâce à des liens créés dynamiquement entre les modules. Le nom de l'émetteur du message n'est pas donné au récepteur. La réalisation d'une invocation est néanmoins possible. Elle nécessite deux envois de messages, un dans chaque sens, mais il est nécessaire de mettre dans le message d'appel le nom du quai sur lequel doit être déposée la réponse.

Chaque module est une instance d'un modèle connu du système et est attaché à un module père. La façon dont sont créés les modèles de module n'est généralement pas décrite dans les articles. Chaque module (père) doit avoir une durée de vie supérieure à celle des processus (fils) qu'il a créés.

La communication entre les sites est assurée par un module qui joue le rôle de station de transport. Il permet le lien entre les sites et assure la gestion des noms globaux internes au noyau.

4.3 Protection

Dans ces systèmes, la protection est assurée par une décomposition en modules qui encapsulent des données. Un module est une entité autonome qui n'a de contact avec le

reste du système que par les communications. On part de deux constatations :

- L'envoi d'un message donne la possibilité à un module de perturber le reste du système.

- La réception d'un message rend un module "fragile" aux fonctionnements défectueux (ou malhonnêtes) des autres modules.

Il y a donc nécessité de contrôler l'émission et la réception des messages.

Contrôle en émission

Classiquement, un mécanisme de contrôle définit à chaque instant **un état de protection**, qui précise quels sont les ports accessibles par chaque module présent dans le système. Parallèlement, un mécanisme de protection assure qu'à chaque instant un module n'envoie des messages qu'aux ports indiqués dans l'état de protection.

Un module doit utiliser les services d'un certain nombre d'autres modules pour faire son travail. Chaque service est accessible par le quai qui lui est associé. L'état de protection doit donc définir pour chaque module la liste des services qu'il peut appeler (liste des quais accessibles par chaque module).

Dans OMPHALE, à chaque module est associé un **environnement**. L'environnement est formé d'une table de LIENS. Un lien est une connexion logique entre deux modules.

L'état de protection est constitué par l'ensemble des environnements des modules présents dans le système.

Pour pouvoir communiquer un module doit désigner un lien qui lui permet d'envoyer son message. De fait, le nom local utilisé par le module appelant correspond à l'index du lien dans l'environnement. A chaque envoi de message, le noyau teste la validité du nom local utilisé par le module appelant : ce nom doit correspondre à un lien utilisable dans l'environnement du module. Si ce test échoue, le message est détruit.

Contrôle en réception

Pour chaque module, le noyau du site utilise un port de communication pour stocker les messages en attente d'être traités. Le contrôle en réception consiste en l'exécution par le noyau d'une procédure de contrôle associée au port de communication. Cette procédure est indépendante du contrôle en émission, elle est exécutée comme test préalable à la mise en file d'attente d'un message, elle n'a pas accès aux données internes du module.

Une telle procédure de contrôle en réception est à la charge du programmeur du module. Des procédures standards existent, la plus simple étant de contrôler si le service demandé est bien accessible par le module appelant.

Des procédures de contrôle en réception plus sophistiquées peuvent faire des tests de contrôle de spécification à partir des arguments du message.

Si la procédure de contrôle de réception donne un résultat négatif, le message est détruit et l'émetteur du message prévenu.

4.4 Contrôle de la dispersion des liens

L'état de protection, à un instant donné, peut être représenté par un graphe orienté dont les noeuds sont les modules et les arcs, les liens (un lien est une capacité contenant entre autre : le nom du module destinataire, la liste des services accessibles et un indicateur d'utilisation). Les modules et le noyau peuvent faire évoluer ce graphe par copie d'un lien (le propriétaire du lien change) ou par duplication d'un lien (création d'un nouveau lien).

Le contrôle de l'évolution du graphe est à la charge du GESTIONNAIRE D'ENVIRONNEMENT qui offre plusieurs commandes :

- EXTRAIRE : ce service permet à un module d'extraire un lien de son environnement. Le module reçoit en réponse un CONTENEUR, contenant toutes les informations du lien demandé.

- AJOUTER : ce service permet à un module d'ajouter un lien à son environnement en fournissant un CONTENEUR. Le gestionnaire renvoit un nom local permettant d'utiliser ce lien.

- RETOURNER : ce service permet à un module de supprimer un lien de son environnement. Pour cela le module envoie le nom local du lien.

- TESTER : Ce service permet à un module de savoir si un nom local est utilisable comme lien.

5. EDEN

EDEN [LAZOW81, ALMES83] est un système distribué intégrant un langage de programmation spécifique : EPL (Eden Programming Language) dérivé de Concurrent EUCLID. L'implémentation actuellement réalisée, connecte un ensemble de sites homogènes : machines VAX et SUN toutes deux possédant le système UNIX BSD 4.2, reliées par un réseau Ethernet. EDEN est un système distribué possédant un seul espace d'adressage : tout objet du système possède un nom indépendant du site sur lequel il s'exécute, cet espace de noms est géré par le noyau du système.

i) Les objets d'EDEN

Une application d'EDEN se décompose en un ensemble d'objets, physiquement répartis sur l'ensemble des sites interconnectés et coopérant par échange de messages. Les objets d'EDEN ont les caractéristiques suivantes :

- chaque objet est référencé par des capacités.
- un objet communique avec un autre objet par invocation.
- un objet est mobile : la localisation physique d'un objet peut changer. Puisqu'un objet peut migrer, chaque objet ne connaît à aucun moment la localisation des autres objets, le noyau gérant cette localisation.
- un objet est une instance d'un type abstrait, il possède un segment de code définissant l'ensemble des procédures d'invocation supportées par cet objet. Ce morceau de code est écrit en EPL.
- un objet possède des données propres qui se décomposent en données rémanentes (qui survivent à un arrêt même d'urgence du système) et en données locales initialisées à chaque invocation des processus et utilisées en cours d'exécution (ces données ne subsistent pas après chaque invocation).
- un objet est actif : il est composé d'au moins un processus qui peut communiquer avec les autres par un moniteur. Quand un processus d'un objet invoque une opération sur un autre objet, ce processus est bloqué jusqu'à la termisaison invocation. Mais les autres processus de l'objet peuvent continuer à s'exécuter.
- un objet peut possèder des points de reprise. Ces points permettent à l'objet de recopier son état (c'est à dire ses données rémanentes) dans une mémoire stable.

ii) Le noyau d'EDEN

Dans les implémentations actuelles réalisées sur le système UNIX 4.2 BSD, le noyau d'EDEN est écrit en C. Dans chaque noeud, il est constitué par un processus UNIX qui s'exécute en parallèle avec les processus implémentant les objets. Le noyau exécute les opérations suivantes :

- il active les processus UNIX associés aux objets actifs d'EDEN.
- il maintient à jour l'état de chaque objet.
- il rend transparent la localisation d'un objet et l'état des machines distantes lors d'une invocation.
- il implémente un ensemble d'opérations utilisées par les objets (invocation, point de reprise, ...)

En plus du noyau, chaque site possède une librairie des primitives du noyau. Elle sera reliée aux objets, et elle permettra l'accès aux primitives du noyau.

La communication entre les machines est réalisée par un protocole datagramme implémenté directement par le réseau Ethernet. Le protocole TCP/IP [POSTE80] d'Ethernet est utilisé seulement lors de la migration d'un objet (code exécutable et données rémanentes).

iii) Communication/synchronisation

Chaque objet d'EDEN peut invoquer un autre processus associé à un objet par l'intermédiaire du noyau d'EDEN. Celui-ci assure que les règles de synchronisation sont respectées (nombre maxima d'invocations en parallèle pour un même processus, possibilité ou non d'activer en parallèle deux processus d'un même objet).

Lors d'une invocation, le noyau d'EDEN transfère le contrôle de l'exécution du processus émetteur au processus destinataire. Le processus émetteur est réactivable quand l'invocation est terminée. Par contre les autres processus du même objet ne sont pas bloqués.

6. MASCOT

MASCOT [SIMPS79, JACKS83] est l'acronyme de "Modular Approach to Software Construction, Operation and Test". Ce noyau permet la description et la structuration d'application "temps réel" sous la forme de sous-systèmes autonomes coopérant par communication.

MASCOT propose un formalisme pour la description d'une application "temps réel" qui :

- peut être représentée sous la forme d'un graphisme permettant de repérer les activités et les modes de communication.
- est indépendante de la configuration des processeurs et de l'architecture du système réparti.
- est indépendante du ou des langages de programmation utilisés.

MASCOT propose aussi une méthodologie pour la structuration de cette application (et en particulier des communications entre les activités). Il est implémenté par un noyau réduit, directement écrit en langage machine, et totalement intégré au système de traitement hôte.

Le noyau de MASCOT supporte l'exécution des différentes activités (qui sont implémentées chacune par un seul processus) et il assure la réception des interruptions et l'interprétation des primitives système.

6.1 Structure d'une application MASCOT

Pour décrire une application, MASCOT offre trois types de processus :

- **les activités.** Elles s'exécutent en parallèle et assurent le traitement des messages. Sur la figure numéro 3, elles sont représentées par un cercle.
- **les canaux.** Ils représentent la communication selon le schéma producteur/consom- mateurs, où les données échangées entre les deux types processus transitent par le canal ; ils sont représentés par la lettre "I". Avec ce mode de communication, une activité ne peut être à la fois productrice et consommatrice. Mais il peut y avoir, sur le même canal, plusieurs activités productrices et plusieurs activités consommatrices.
- **les réservoirs.** Ils représentent un autre mode de communication où les données ne transitent plus et où il peut y avoir accumulation. Ils sont représentées par la lettre "U". Une activité peut, à la fois, écrire et lire dans un réservoir.

La description par diagramme, dont un exemple est donné ci-après permet de structurer l'application. Mais ce mode de description, même s'il permet de séparer les communications en "mode réservoir" et "mode canal", ne permet pas de décrire le comportement particulier de chaque réservoir ou canal.

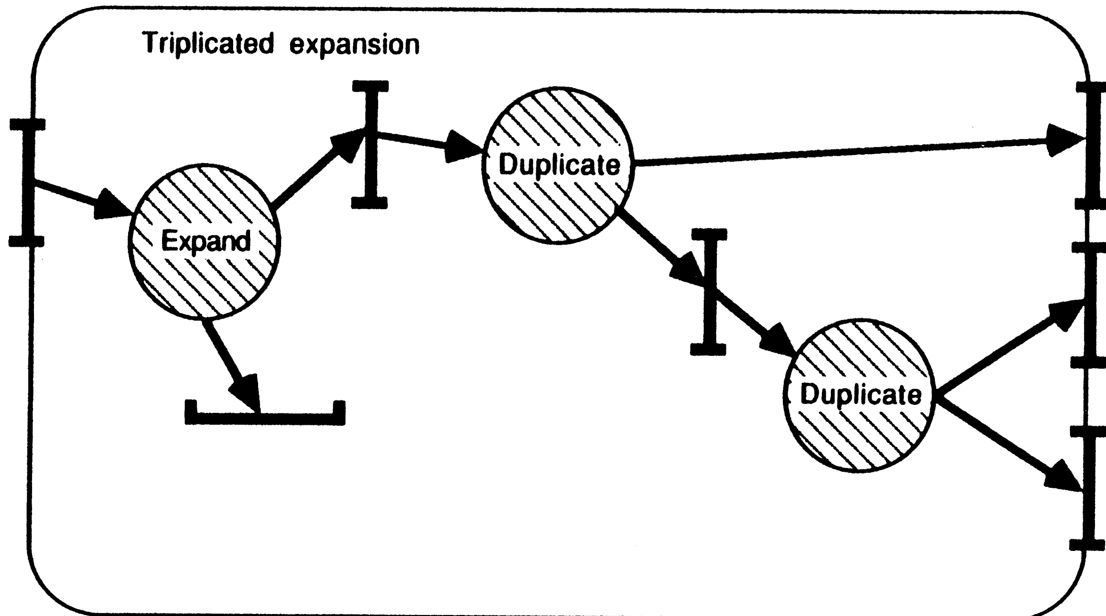


Figure 3. — Schéma d'une application.

Pour répartir une application, MASCOT propose de séparer l'application en "sous-systèmes" qui s'exécutent sous le contrôle d'un noyau présent sur chaque site. Sur un même site peut coopérer un ensemble de sous-systèmes interconnectés.

Un sous-système consiste en un ensemble d'activités interconnectées par les données qui s'exécutent sur le même site (on peut, suite à diverses reconfigurations, exécuter deux sous-systèmes sur un même site). Le schéma précédent montre comment un programmeur peut spécifier les interfaces de communication entre chaque module et aux frontières du sous-système, et le programme suivant en donne la description.

```
ROOT PROC Expand =
   (REF OneCHAR CHAN indata, outdata,
    REF DICT POOL lookuptable )

ROOT PROC Duplicate =
   (REF OneCHAR CHAN indata, outdata1, outdata2 )

FORM Triplicated expansion =
   ( expand( in, trans1, dictionary ),
    duplicate( trans1, out1, trans2 ),
    duplicate( trans2, out2, out3 ) ) ;
```

"Expand" et "Duplicate" sont des activités propres au sous-système "Triplicated expansion". Elles sont interconnectées entre elles par les canaux trans1 et trans2 de type caractère, et l'activité "Expand" possède un réservoir de type dictionnaire.

6.2 Communication

Dans l'exemple ci-dessous nous allons écrire un canal "tampon" permettant à deux processus, l'un rédacteur de déposer des messages, et à l'autre de prendre ces messages.

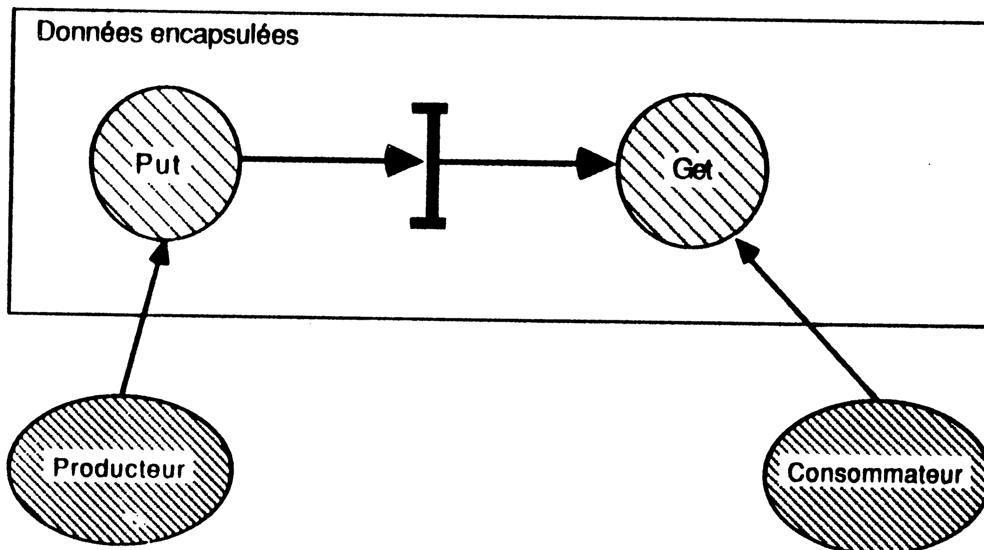


Figure 4. — Structure d'un canal.

```
MODE tampon =
  STRUCT (CONTROLQ entrée, sortie, BOOL vide, INT donnée ) ;

  ACCESS PROC Put = ( INT x, REF tampon CHAN ) :
   BEGIN
     JOIN (entrée OF CHAN ) ;
     WHILE NOT (vide OF CHAN ) DO WAIT (entrée OF CHAN ) ;
     donnée OF CHAN := x ;
     vide OF CHAN := FALSE ;
     STIM (sortie OF CHAN) ;
     LEAVE (entrée OF CHAN) ;
   END ;

  ACCESS PROC get = ( REF INT x, REF tampon CHAN ) :
   BEGIN
     JOIN (sortie OF CHAN ) ;
     WHILE (vide OF CHAN ) DO WAIT (sortie OF CHAN ) ;
     x := donnée OF CHAN ;
     vide OF CHAN := TRUE ;
     STIM (entrée OF CHAN) ;
     LEAVE (out OF CHAN) ;
   END ;
```

Cette interface d'accès encapsule les données propres aux canaux (et aux réservoirs). Dans cet exemple, les données communes aux procédures d'accès sont les deux files de

contrôle "entrée" et "sortie", le booléen "vide", la donnée "donnée". Elle permet un appel des primitives de communication indépendamment du protocole de communication réalisé par le canal (ou du réservoir), de leur répartition ou non, du mode de synchronisation réalisé et du système de traitement implémentant le noyau MASCOT.

Les canaux et les réservoirs permettent de synchroniser les activités d'une application décrite à l'aide de MASCOT. Pour cela le programmeur utilise un outil spécifique à MASCOT : les files de contrôle ("control-queue") partagées entre les différentes activités. Dans l'exemple ci-dessus, "entrée" et "sortie" sont les files de contrôle associées au tampon.

Chaque file de contrôle de MASCOT possède une priorité et peut être dans cinq états (libre ou occupé selon différents modes). Pour la manipuler, le programmeur dispose de quatre primitives qui implémentent des sémaphores étendus. Dans l'exemple nous avons utilisé : JOIN qui permet de réserver une file (proche de la primitive P() sur un sémaphore), LEAVE qui libère une file (proche du V()), WAIT qui permet d'attendre un événement sur une file et STIM qui permet d'en envoyer un.

6.3 Répartition

Pour répartir un canal entre différents sites, il suffit d'en faire une copie sur chacun d'eux, puis d'introduire :

- **sur l'ensemble des sites de production** un "Exportateur" qui prend les messages présents dans le canal et les met sur le réseau d'interconnexion

- **sur chaque site de consommation** un "Importateur" qui prend les messages en circulation sur le réseau d'interconnexion et les écrit dans le canal.

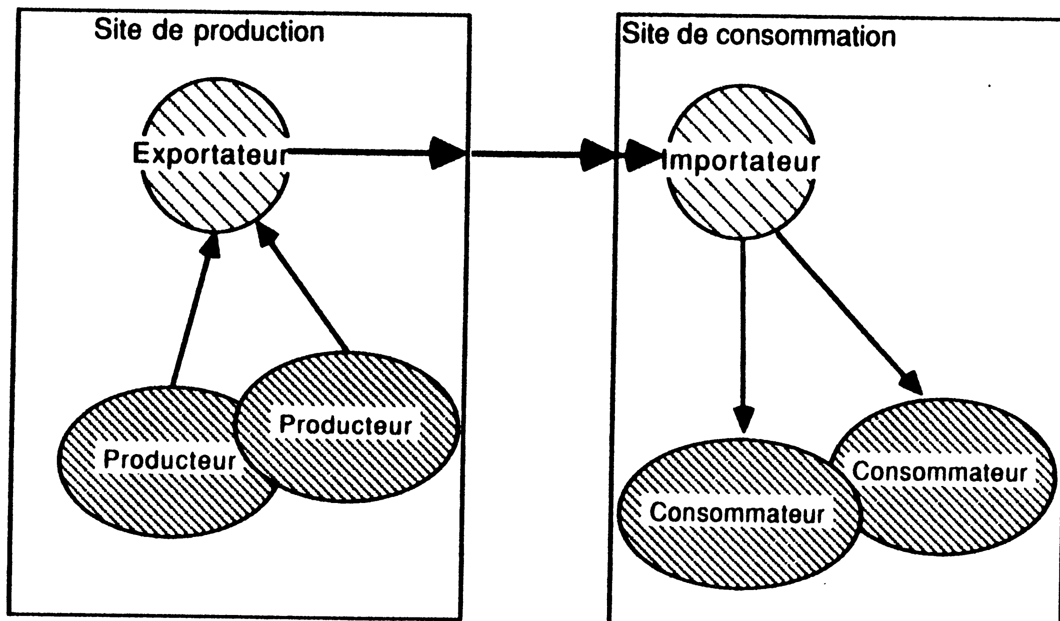


Figure 5. — Répartition d'un canal.

Ce schéma permet de répartir aisément un canal, partagé entre deux sites, qui gère une file de messages, à condition que les activités productrices soient sur le même site et que toutes les activités consommatrices soient sur l'autre site.

Pour répartir un canal qui gère un protocole de communication plus complexe, il est nécessaire :

- soit d'écrire des processus importateur et exportateur qui ont une connaissance du protocole de communication réalisé

- soit de réécrire les processus d'accès au canal.

7. Proposition de [Natar85]

"L'université" de BOMBAY (Inde) dans [NATAR85] propose un ensemble de primitives pour programmes distribués.

7.1 Primitives de base

Un programme distribué est constitué d'agents, qui s'exécutent de façon autonome et asynchrone, tout en respectant les autres agents. Chaque agent peut communiquer par messages avec d'autres agents par l'intermédiaire de port.

7.1.1 Spécification d'un agent

Chaque agent est spécifié par deux modules :

- **le module de spécification** : il est constitué de deux morceaux. La définition de l'interface qui précise la nature (port d'entrée, port de sortie, ...) et le type de chaque port (type du message véhiculé par le port : entier, booléen structuré, ...) et la description de configuration qui décrit les connexions entre les ports.

- **le module d'implémentation** : ce module constitue, pour chaque agent, une implémentation réalisant les spécifications du module précédent.

Il existe six natures de ports : port d'entrée, port de sortie, port d'appel d'entrée, port d'appel de sortie, le port de réception et le port de diffusion. Un port d'entrée ne peut être connecté qu'avec un port de sortie, un port d'appel d'entrée qu'avec un port d'appel de sortie et un port de réception qu'avec un port de diffusion. Ces natures de ports permettent d'implémenter différents modes de communication.

7.1.2 Mode de connexion

Il existe quatre modes de connexion :

1) **Entrée/Sortie** : une sortie sur un port P se fait vers le port correspondant connu[4] de P de manière synchrone. Si un agent fait une entrée sur un port qui n'est pas restreint, il peut recevoir un message d'un processus quelconque.

4.

i) **le communicant** *d'un port P appartenant à un agent est un autre agent ayant un port Q connecté à P. Le port Q est appelé correspondant du port P. Il n'est pas nécessaire de spécifier dans la description de configuration les communicants et les correspondants de chaque port.*

ii) **Port restreint ou non restreint.** *Un port est restreint quand la description de configuration a spécifié les communicants et les correspondants, (il est non restreint dans le cas contraire). Les ports de sortie et d'appel de sortie sont toujours restreints, les autres sortes de ports sont libres (ils peuvent être soit restreints, soit non restreints).*

iii) On appelle **correspondants connus ou communicants connus** *les ports ou les agents connectés lors de la description de configuration. Un port restreint ne peut communiquer qu'avec le correspondant connu de ce port.*

2) **Invocation** : dans ce mode, l'appelant envoie le message d'un port d'appel et attend la réponse de l'agent communicant.

3) **Délégation** : un agent qui reçoit une invocation peut demander à ce quelle soit traitée par un autre agent. Pour cela il envoie le message par un port d'appel vers un autre agent, en précisant le port qui est à l'initiative de l'appel. L'agent qui effectue la délégation attend que son communicant ait commencé de traiter l'invocation pour continuer son exécution.

4) **Diffusion** : un agent diffuse un message par l'un de ses ports de diffusion. L'envoi en diffusion s'effectue de façon asynchrone vers tous les agents qui ont ce port comme communicant. L'émetteur ne connaît pas l'ensemble des ports qui ont reçu le message, ni l'ensemble des ports qui ne l'ont pas reçu.

7.1.3 Implémentation d'un agent

Conformément à ce que nous venons de voir, l'écriture d'un programme se fera en deux étapes complémentaires : la première permet de spécifier les communications et la seconde le corps de chaque agent.

i) Etape de spécification

Exemple : 1_tampon

```
AGENT tampon SPECIFICATION ;
 PROVIDES
  -- description des ports de l'agent tampon
  insert : INPUT PORT (INTEGER) ;
  remove : OUTPUT PORT (INTEGER) ;
  close : INPUT PORT ;
  finish : OUTPUT PORT ;
 SPECIFICATION
  -- description des restrictions éventuelles
  remove = consumer.remove ;
  finish = consumer.finish ;
 END ;

AGENT producer SPECIFICATION ;
 PROVIDES
  insert : OUTPUT PORT (INTEGER) ;
  close : OUTPUT PORT ;
 SPECIFICATION
  insert = tampon.insert ;
  close = tampon.close ;
 END ;

AGENT consumer SPECIFICATION ;
 PROVIDES
  remove : INPUT PORT (INTEGER) ;
  finish : INPUT PORT ;
 END ;
```

Pour chaque agent est spécifiée : la liste de ses ports (nom, sorte et type) puis les restrictions éventuelles (par exemple : le correspondant du port "remove" de l'agent

"tampon" est le port "remove" de l'agent "consumer").

ii) Etape d'implantation

Un agent est un programme séquentiel constitué de déclarations et d'une séquence de commandes. Le langage Pascal a été enrichi de nouvelles commandes (communication, alternative, cycle) et de primitives de traitement des erreurs.

```
AGENT tampon IMPLEMENTATION ;
 CONST
  N = 10 ;
 VAR
  in, out : INTEGER ;
  buff : ARRAY[ 0..N-1 ] OF INTEGER ;
  closed, finished : BOOLEAN ;
 BEGIN
  in, out := 0 ;
  closed, finished := FALSE ;
  CYCLE
   WHEN NOT closed ; in-out < N ; insert(buf[ in MOD N ])
     DO in := in+1
```

Ces deux dernières lignes pourraient s'écrire en OCCAM :

```
ALT
  ((NOT closed) AND (in-out < N)) & insert ? buf[ in \ N ]
   in : := in+1
```

```
   OR
  WHEN NOT finished ; out < in ; remove(buf[ out MOD N ])
    DO out := out+1
   OR
  WHEN NOT closed ; close()
    DO closed := TRUE
   OR
  WHEN NOT finished ; closed ; out=in ; finish()
    DO finish := TRUE
  END
 END ;
```

Le processus ci-dessus est équivalent à la structure répétitive de CSP : il se termine quand toutes les gardes s'évaluent à faux.

```
AGENT producer IMPLEMENTATION ;
 VAR
  item : INTEGER ;
 BEGIN
  REPEAT
   "produce an item" ;
   insert( item ) ;
  UNTIL "all items have been produced"
  close() ;
 END ;

AGENT consumer IMPLEMENTATION ;
 VAR
  item : INTEGER ;
  finished : BOOLEAN ;
 BEGIN
  finished := FALSE ;
  CYCLE
   WHEN NOT finished ; remove( item )
      DO "consume the item"
    OR
   WHEN NOT finished ; finish()
      DO finished := TRUE
   END
 END ;
```

7.1.4 Création dynamique d'agent

Si l'on désire que l'application soit dynamique, il est nécessaire d'introduire la notion de type d'agent dont on pourra créer une instance. Dans le cas précédent les agents tampon, producer et consumer, sont statiques.

```
TYPE
 AGENT tampon ( N : NODE ; T : BOOLEAN ) IMPLEMENTATION ;
 ...
 END ;
```

Si A est un agent de type tampon, alors une instance de tampon sera créée par :

CREATE A(N, TRUE).

7.2 Primitives pour la gestion des erreurs de communication

Pour résoudre les problèmes liés aux erreurs de communication (interblocage, panne d'un site, panne d'un lien, ...), NATARAJAN introduit la possibilité de réincarner un agent défaillant ou fou, puis de se reconnecter à la nouvelle incarnation de l'agent.

Pour chaque communication, il existe la possibilité de connaître la cause de la non-communication : lien physique rompu entre deux processeurs, communication impossible (l'agent avec lequel le port doit communiquer a terminé), interblocage. En fonction du type de panne il est possible de recréer un nouvel agent et de continuer l'exécution.

8. SCEPTRE

SCEPTRE est une proposition de standard pour un noyau d'exécutif "temps réel" [BROWA84] qui spécifie un ensemble d'opérations élémentaires. Une application décrite à l'aide du noyau SCEPTRE se décompose en un ensemble de tâches coopérantes.

Cet exécutif fournit des services de :

- communication.
- synchronisation.
- gestion et ordonnancement des tâches.
- gestion de la mémoire.
- gestion des interruptions et des entrées/sorties physiques.
- entrées/sorties logiques et gestion des périphériques.
- gestion des programmes.
- gestion du temps.

La communication entre machines s'effectue au moyen de canaux qui fournissent le support matériel à la transmission d'information entre tâches.

Dans SCEPTRE, les notions de communication et de synchronisation sont distinctes.

i) Communication

Pour communiquer entre elles, les tâches de SCEPTRE utilisent des files FIFO bornées (Premier Entré - Premier Sorti). La communication s'effectue selon un mode synchrone, elle est bloquante pour celui qui effectue la commande, aussi bien en lecture (file vide) qu'en écriture (file pleine).

Afin de permettre l'écriture d'un message (ou la lecture) uniquement si la file est non pleine (non vide), le noyau offre deux prédicats qui sont vrais si la file est pleine pour le prédicat "PLEINE(file)" ou vide pour le prédicat "VIDE (file)".

Dans le noyau SCEPTRE, les communications sont non typées : la même file gérée selon le protocole FIFO (premier entré - premier sorti) peut servir à transporter des messages de formats différents (un message est considéré comme une suite d'octets).

ii) Synchronisation

Pour la synchronisation, SCEPTRE offre deux mécanismes. Le premier permet d'entrer (et de sortir) d'une section critique et permet par conséquent de contrôler l'accès à une variable partagée. Le second permet la gestion des événements mémorisés.

La signalisation entre deux processus nécessite une opération dans chacune des deux tâches : attente d'un signal dans l'une et envoi du signal dans l'autre. La signalisation est asymétrique : elle exprime une relation de cause à effet.

Dans le noyau SCEPTRE, tout événement est associé à une tâche. Chaque tâche dispose par conséquent d'un ensemble fini d'événements qu'elle peut attendre sélectivement ou purement effacer. De plus il est possible qu'une tâche manipule

globalement un ensemble d'événements : attente de la première occurence d'un événement parmi plusieurs, effacer les événements d'une liste.

Un événement peut être dans deux états : ARRIVE ou NON ARRIVE. SCEPTRE autorise les opérations :

- **ARRIVE(E1, ..., En)** : ce prédicat est vrai si tous les événements sont arrivés.
- **SIGNALER(E, T)** : l'événement E est mis dans l'état arrivé et si le processus T est en attente de l'événement E, il est activé. Si aucun processus est en attente de l'événement celui-ci est mémorisé.
- **EFFACER(E1, ..., En)** : les événements sont mis dans l'état NON ARRIVE.
- **ATTENDRE(E1, ..., En)** : la tâche est mise en attente jusqu'à ce que tous les événements soient dans l'état ARRIVE.

Pour se synchroniser, les tâches du noyau SCEPTRE utilisent des événements qui sont produits par d'autres tâches

9. Comparaison et synthèse

Après avoir décrit un certain nombre de systèmes répartis (ou répartissables) et pour mieux faire ressortir les parallèles et les différences avec CONKER nous allons faire une synthèse de ces différents systèmes sous forme de quatre tableaux successifs présentant : la situation générale, la communication, la synchronisation, la protection.

Même si nous avons essayer d'employer des termes neutres (**voie** pour désigner tous les objets pouvant réaliser une communication, **objet** pour désigner les objets actifs du systèmes : processus, module, activité, ...) il est évident que ces objets n'ont pas la même caractéristique sur les différents systèmes. Pour affiner cette analyse nous donnerons, quand ce sera possible, quelques précisions sur ces objets.

Les descriptions, succintes, s'attachent essentiellement aux principales caractéristiques de communication et de synchronisation réalisées dans chacun des systèmes.

9.1 Présentation générale

Ce premier tableau comparatif présente les caractéristiques générales de chaque système.

La plupart de ces systèmes sont relativement récents : les études ayant permises ces réalisations ont débutées vers les années 1980. Ces systèmes sont actuellement à l'état de maquettes implémentant tout ou une partie seulement des spécifications initiales.

Le système de référence devient peu à peu Unix. Cette généralisation est certainement due à l'environnement de programmation offert par ce système.

Chaque proposition développe son langage, mais toutes offrent des constructions algorithmiques de haut niveau rappelant par exemple celles offertes par le langage PASCAL.

De nombreux systèmes citent des références communes. Parmi les systèmes les plus référencés, nous pouvons citer : DEMOS (c'est le plus ancien), MEDUSA, SR.

9.2 La communication

La première comparaison de ce tableau porte sur le vocabulaire : objet actif et objet de communication. Il est nécessaire de prendre quelques précautions : deux objets ayant le même nom dans deux systèmes n'ont généralement pas la même fonction.

Les objets actifs : qu'ils s'appellent processus, module, activité, les objets actifs sont toujours implémentés par au moins un processus. Dans le cas où ils sont implémentés par plusieurs processus, le système offre généralement des mécanismes d'accès en exclusion mutuelle aux variables globales des objets qui sont partagées.

La voie de communication : c'est généralement un objet passif sur lequel au moins un processus peut venir lire et un autre écrire. Parmi les choix extrêmes, nous pouvons citer ARGUS qui n'autorise qu'un seul lecteur (le "moniteur") et un seul rédacteur (même si plusieurs guardians peuvent appeler le même "moniteur") et [Natar85] qui permet N émetteurs et M récepteurs sur le même port. Rares sont les systèmes où il existe des prédicats sur l'état de la voie : SCEPTRE permet de savoir si la voie est pleine ou vide ; dans les autres, c'est à la fin de l'exécution de la primitive (si elle n'est pas asynchrone) que

le processus peut déduire l'état de la voie.

En ce qui concerne la communication, nous pouvons définir au moins quatre façons de désigner émetteur et récepteur :

- la communication ne fait intervenir qu'un seul objet intermédiaire : porte, boîte à lettres. Nous utiliserons le terme de **boîte à lettres.**
- la communication fait intervenir un objet chez l'émetteur, et un autre chez le récepteur sans que ces deux objets soient liés. Nous utiliserons le terme de **porte.** Ce modèle est utilisé dans CHORUS.
- la communication fait intervenir un couple d'objets définis en tant que tels, ou un objet liant un émetteur et un récepteur. Dans ce cas, nous utiliserons le terme de **liaison.** Fréquemment la liaison est une association de deux portes.
- la communication active directement un objet processus chez le destinataire. Dans ce cas, nous utiliserons le terme d'**opération** ou de communication directe.

Afin de pouvoir comparer les choix effectués par les différentes équipes, nous avons essayé de regrouper tous les types de communication implémentés selon un de ces quatre modèles.

Par contre la communication avec le noyau est encore fréquemment réalisée par primitives (héritage des systèmes mono-processeurs).

Tous les systèmes font l'hypothèse que la communication distante (par messages) est aussi fiable que la communication locale (par messages ou par variables partagées).

La plupart des systèmes considèrent que les messages échangés possèdent un type (mais rare sont les systèmes qui offrent une possibilité de multiplexer sur une même "voie" plusieurs types de messages).

Dans chaque système étudié, une "voie" possède généralement **plusieurs émetteurs** potentiels. Par contre, presque tous ont choisi d'associer à chaque "voie" un seul récepteur. En effet si la "voie" possède plusieurs récepteurs et qu'elle est répartie entre plusieurs sites, il est nécessaire de synchroniser l'ensemble des récepteurs. A cause de cela, presque tous les systèmes qui offrent la possibilité d'avoir plusieurs récepteurs sur la même "voie" ont choisi de faire la restriction suivante : *les récepteurs doivent résider sur le même site.*

La communication entre les objets du système et le noyau est rarement explicitée. Les quelques éléments donnés sur les réalisations effectuées, montrent que presque tous ont fait le choix de communiquer par appel de procédure (c.a.d que le schéma de communication être les objets du système et le noyau, habituellement utilisé dans les systèmes centralisés, a été conservé).

Le temps n'étant pas une caractéristique importante, peu de système ont fait le choix de temporiser chaque communication. Pourtant, nous restons persuadés que, même dans une application qui ne nécessite pas une manipulation du temps physique, une notion de durée peut être non seulement agréable mais aussi utile.

9.3 La synchronisation

Nous présentons ici, la ou les synchronisations primaires réalisées dans chaque système.

La plupart des systèmes ont fait le choix d'implémentation le plus simple : l'émission vers la voie se fait de façon asynchrone (i. e. l'émetteur est suspendu, le temps que le noyau prenne en compte sa demande).

En réception, si un objet actif (processus, tâche, ...) peut lire sur plusieurs voies, il peut généralement faire un choix : non-déterministe ou déterministe, bloquant ou non, ...

Tous les systèmes permettent de construire d'autres modes de synchronisation que ceux offerts par le système. Il est généralement nécessaire d'insérer entre les deux objets, un objet réalisant ce nouveau mode de synchronisation.

La plupart des "voies" de communication étant des objets passifs, il ne peuvent pas (sauf si c'est le noyau qui s'en charge) produire des événements. Comme pour la réalisation d'autres modes de communications, il est nécessaire d'introduire un objet intermédiaire capable de produire les événements désirés.

9.4 Désignation et protection

La plupart du temps, l'objet désigné pour communiquer est la "voie".

Les concepteurs de systèmes ont choisi la communication anonyme : un récepteur lit sur une "voie" sans connaître l'objet émetteur, sauf si la communication est une invocation (dans ce cas la connaissance de l'émetteur est implicite).

Quelques systèmes se sont penchés sur les problèmes de protection : un objet encapsule généralement des données locales, mais il reste encore à définir des méthodes de protection pour les messages, bien qu'il existe quelques propositions (EDEN, DEMOS).

Il nous semble nécessaire de pouvoir effectuer sur chaque message un contrôle sémantique : *est-ce que ce message est structuré correctement ?* Question qui est plus large que celle généralement résolue : *est-ce que ce processus peut accepter ce service ?* ou *est-ce que ce service est présent sur ce site ?* (ce problème est lié à la définition d'un langage d'expression des messages).

9.5 Conclusion

L'étude comparative de tout ces systèmes fait apparaître de nombreuses similitudes dues aux influences réciproques des équipes de recherche. Parmi cet éventail de systèmes, CONKER a une place originale et nous verrons dans les chapitres suivants :

- qu'il introduit systématiquement un objet actif programmable pour réaliser la communication : le connecteur.
- que le temps est un objet manipulable, que chaque communication peut être temporisée et que chaque message a une durée de vie.

- que le mode de communication de base entre objet du système est le rendez-vous.

9.1. Présentation générale

	ARGUS	CONIC	D DEMOS E EMBOS O OMPHALE	EDEN	MASCOT
Université	M.I.T	Londres	Los Alamos (D) Lille (O)	Washington	
Années	1980->	1983->	1976->(D) 1983-> (E) 1983-> (O)	1980->	1972->1978
Réalisation	Vax/Unix	Vax/Unix 68000 Cambridge Ring	Cray1 (D) ELXSI (E) Z8001 (O)	Vax/Unix Ethernet	8080 Ti990
Langages	CLU	?	?	EPL	Pascal
Applications	Transactionnel	Programmation dynamique d'un système réparti	Application répartie (D) Temps réel (E)	Programmation Orienté Objet répartie	Temps réel
Références	StarOS CLU	Mascot Argus SR Mesa	StarOS (E) Chorus (O) Accent (O) Mesa (E)	Multics Hydra Medusa CLU Smalltalk	

MEDUSA	NATARAJAN	SR	CHORUS	SCEPTRE	CONKER
Carnegie-Mellon	Bombay	Arizona	INRIA	France	Grenoble
1980->	1983->	1979->	1980->	1981->	1983->
Cm*	?	PDP11/Unix	SM90 Ethernet	Multiples	8086/iRMX86 Vax/Unix Transputer
?	Pascal	?	Pascal	Assembleur	Plm86, C, OCCAM
Application répartie	Programmation distribuée	Système réparti	Système réparti	Temps réel	Temps critique
Unix Demos StarOS	Chorus SR Ada Clouds *Mod DP CSP	CSP DP Mesa Plits	Accent Medusa	Mascot LTR	Chorus CSP

9.2. La communication

	ARGUS	CONIC	D DEMOS E EMBOS O OMPHALE	EDEN	MASCOT
Objet actif	Processus	Module	Objet	Processus	Activité
Objet de com- -munication (voie)	Handler	Port->Port	Lien	Opération	Queue
Modèle	Opération	Liaison	Liaison	Opération	Boite aux lettres
Entre site	Messages	Messages	Messages	Messages	Messages
Dans un site	Messages	?	Mémoire partagée	Messages	Mémoire partagée
Types	Oui	Structure	Structure	Oui	Oui
Nombres émétteurs	1	N	N	N	N (sur même site)
Nombres récepteurs	1	1	1	1	M (sur même site)
Objet->noyau	Primitive	?	Messages	Primitives	Primitives
Noyau->objet		?	Messages	Messages	Primitives
Temporisation		Non	?	Oui	Oui

MEDUSA	NATARAJAN	SR	CHORUS	SCEPTRE	CONKER
Activité	Agent	Processus	Acteur	Tâche	Processus
	Port (entrée et sortie)	Opération	Port	Canal	Connecteur
Liaison	Liaison	Opération	Porte	Boite aux lettres	Liaison
Messages	Messages	Messages	Messages	Messages	Messages
?	?	Mémoire partagée	Messages	Mémoire partagée	Messages
?	Structure	Oui	Oui	Non	Structure
1	N 1 (si restreint)	N	N	N (sur même site)	N
1	M 1 (si restreint)	1	1	M (sur même site)	1
Primitives		Primitives	Messages	Primitives	Messages
Oui		Primitives	Messages	Non	Messages
?		Non	Oui	Non	Oui

9.3. La synchronisation

	ARGUS	CONIC	D E O DEMOS EMBOS OMPHALE	EDEN	MASCOT
Emission vers "voie"	Synchrone	Asynchrone	Asynchrone		Invocation
Réception	Activation d'un processus	?	Choix alternatif	Diverses possibilités	Invocation
Synchronisation entre objet actif	Invocation	Asynchrone Invocation	Asynchrone (Fifo •)	Invocation	Divers
Evénement généré par la voie	Non	Non	Non	?	Non

9.4. Désignation et protection

Désignation	ARGUS	CONIC	D E O DEMOS EMBOS OMPHALE	EDEN	MASCOT
Direct/Indirect	Direct	Indirect	Indirect	Direct	Indirect
Objet désigner	Handler	Port local	Lien	Opération	Procédure
Emetteur connu	Non	?	Non	Non	Non
Protection					
des données	Encapsulation	Encapsulation	Encapsulation	Encapsulation	Encapsulation
des messages	Oui	?	Oui	Oui	Oui
detection deadlock	Oui	?	?	?	?
gestion erreur (site, réseau)	Oui	?	?	Réincarnation	?

MEDUSA	NATARAJAN	SR	CHORUS	SCEPTRE	CONKER
Asynchrone Synchrone	Synchrone	Synchrone	Asynchrone	Invocation	Rendez-vous
	Avec ou sans suspension	Choix	Choix	Invocation	Choix
Asynchrone Synchrone Invocation	Synchrone Invocation Délégation Diffusion	Synchrone Invocation Invocation multiple	Synchrone	Asynchrone (Fifo bornée)	Divers
?	Oui (gestion de panne)	?	Non	Oui (vide, plein)	Oui (multiples)

MEDUSA	NATARAJAN	SR	CHORUS	SCEPTRE	CONKER
Indirect	Indirect	Direct	Direct	Direct	Indirect
Tube	Port local	Opération	Port	Nom de canal	Nom de canal
Oui	Non	Non	Oui	Non	Non
?	Encapsulation	Encapsulation	Encapsulation	Non	Encapsulation
Oui	?	Oui	Oui	Non	Oui
?	Oui	?	Non	Non	Non
/	Réincarnation reconnexion	?	Non	Non	Non

II. Présentation du langage OCCAM et de ses extensions

Résumé

Le noyau de CONKER a été écrit dans différents langages selon les implémentations réalisées, mais celui qui nous a permis d'arriver à une meilleure structuration est le langage OCCAM. Afin d'avoir pour tous les exemples donnés une présentation homogène (quelle que soit leur origine) nous avons choisi de les décrire à l'aide du langage OCCAM. La sémantique formelle d'OCCAM, permet la lecture d'un programme non seulement comme un ensemble d'instructions exécutables, mais également comme un prédicat formé d'un ensemble d'assertions.

Ce chapitre présentera successivement, le langage OCCAM et les extensions que nous y avons apportées, l'architecture du Transputer (machine construite pour exécuter efficacement le langage OCCAM) et la démarche ayant abouti à la proposition du modèle CONKER.

1. Langage et modèle de départ

La sémantique formelle de CSP [HOARE78, HOARE81, HEHNE83], que nous ne présenterons pas ici, permet la lecture d'un programme OCCAM non seulement comme un ensemble d'instructions exécutables, mais également comme un prédicat formé d'un ensemble d'assertions dans une extension du calcul des prédicats où les règles de déduction peuvent être appliquées. Ainsi, un ensemble de règles de transformation [HOARE84] peut être systématiquement utilisé pour la construction correcte et l'optimisation des programmes.

Note : OCCAM est un langage qui respecte le modèle CSP, une sémantique dénotationnelle du langage OCCAM est décrite dans : [ROSCO85].

Dans les paragraphes suivants, nous allons présenter le langage OCCAM qui a été réalisé par la société INMOS en respectant la sémantique de CSP. OCCAM nous servira de langage pour spécifier et exécuter le noyau de communication CONKER.

Pour une définition plus complète du langage OCCAM, et en particulier des déclarations de variables, de canaux ou de processus, nous renvoyons le lecteur au manuel de référence [INMOS84].

2. Description du langage OCCAM

Le langage OCCAM permet de structurer une application parallèle. La synchronisation entre les processus s'effectue par communication synchrone à l'aide de canaux élémentaires, réalisant une communication uni-directionnelle par entrée-sortie synchrone, entre un processus émetteur et un processus récepteur utilisant le même canal statiquement nommé.

Le langage OCCAM comprend **six processus élémentaires** que l'on peut composer entre eux par différents constructeurs pour construire des programmes (qui sont eux même des processus).

2.1 Processus élémentaires

Le nombre des processus élémentaires d'OCCAM est volontairement réduit, de façon à avoir un noyau de langage minimal, suffisant pour la programmation parallèle. Ces processus de base doivent être pris comme des briques qui permettent de construire des processus plus complexes.

i) l'affectation (:=)

variable := expression

Le processus affectation transfère la valeur d'une expression après évaluation correcte à une variable locale d'un processus (le langage ne possède pas la notion de variable globale ou de variable appartenant simultanément à plusieurs processus). Ce processus commence par évaluer l'expression, et si cette évaluation est possible, l'affecte à la variable au bout d'un temps fini dépendant de l'implantation ; le processus termine ensuite. Dans le cas contraire, ce processus ne termine pas.

ii) L'émission (ou sortie) d'un message (!)

canal ! expression

Après avoir évalué correctement l'expression spécifiée, une sortie attend jusqu'à ce qu'une entrée utilisant le même canal soit prête à s'exécuter. Alors la valeur de l'expression est écrite sur le canal et le processus termine.

ANY peut être utilisé à la place de l'expression et alors une valeur arbitraire est écrite sur le canal. ANY sert de signal de synchronisation entre deux processus.

iii) Réception (ou entrée) d'un message (?)

canal ? variable

Une entrée attend jusqu'à ce qu'une sortie utilisant le même canal soit prête à s'exécuter. Elle affecte alors la valeur courante présente sur le canal à la variable locale et le processus termine.

ANY peut remplacer l'utilisation de la variable, la valeur présente sur le canal est alors ignorée. Ce mécanisme permet de traiter la synchronisation à l'aide de la primitive de communication.

Remarque : un canal OCCAM est un objet dont le nom est partagé entre deux processus, l'un ne pouvant qu'écrire sur ce canal et l'autre ne pouvant qu'y lire.

iv) Le processus d'attente (AFTER)

```
TIME? AFTER expression
```

Ce processus commence son exécution par l'évaluation de l'expression et termine quand la valeur lue sur le canal TIME (canal prédéfini par l'implémentation d'OCCAM) est supérieure à l'expression.

v) Le processus d'arrêt (STOP)

```
STOP
```

Ce processus commence et ne termine jamais. Il permet de modéliser une mauvaise terminaison du programme (une bonne terminaison étant l'arrêt de tous les processus actifs sur la fin de leur code).

vi) Le processus nul (SKIP)

```
SKIP
```

Dès que ce processus a commencé son exécution il termine. Il permet de modéliser une terminaison correcte (à l'opposé du processus STOP qui modélise une terminaison incorrecte) et de décrire des comportements vides.

2.2 Les constructeurs d'OCCAM

OCCAM permet de construire de nouveaux processus, grâce à l'utilisation des constructeurs du langage et de processus déjà construits.

i) Le constructeur séquentiel (SEQ)

```
SEQ
  proc1
  proc2
  ...
```

Note : La hiérarchie de construction du langage OCCAM est matérialisée dans la syntaxe du langage par des niveaux d'indentation (de deux caractères) qui déterminent l'ensemble des processus à composer avec le constructeur de niveau immédiatement supérieur.

Les processus composants sont exécutés l'un après l'autre et le processus séquentiel s'assure que chaque composant termine avant que le suivant ne s'exécute. La construction termine son exécution quand le dernier processus a terminé.

S'il n'a pas de composant, le processus séquentiel est équivalent au processus élémentaire SKIP.

ii) Le constructeur parallèle (PAR)

```
PAR
  proc1
  proc2
  ...
```

Le constructeur parallèle impose à ses processus composants d'être exécutés de façon concurrente. Tous les processus composants commencent leur exécution en même temps et le processus parallèle termine quand tous les processus composants ont terminé. Si le processus parallèle n'a pas de composant il est équivalent au processus SKIP.

Chacun des processus composants opère sur ses variables locales, qui ne peuvent être partagées avec aucun autre processus composant. Néanmoins, dans une implémentation particulière il peut s'avérer utile qu'une variable puisse être utilisée par deux ou plusieurs processus, à condition qu'aucun des processus ne change sa valeur par une entrée ou une affectation (ce mécanisme d'héritage de variables permet d'implémenter les constantes globales et a été choisi par INMOS pour la réalisation de ses compilateurs).

Il est important de souligner l'utilisation du PAR lors de l'encapsulation d'un ensemble de processus par un processus de niveau supérieur. Dans ce cas chacun des processus de l'ensemble peut être vu comme une continuation partielle du processus dont ils sont composants par le mécanisme d'héritage de variables précédemment défini.

Remarque : il est possible en OCCAM d'imposer l'implantation du constructeur parallèle, en fixant des priorités relatives entre les processus qui doivent s'exécuter de façon concurrente. Ce nouveau constructeur (qui ne change pas la sémantique du PAR d'OCCAM, mais qui impose simplement une stratégie d'implémentation) se note : **PRI PAR.**

```
PRI PAR
  proc1
  proc2
  ...
  procN
```

Le processus 1 est plus prioritaire que le processus 2, le processus 2 est plus prioritaire que le processus 3, et ainsi de suite. Le "scheduler" d'OCCAM fait évoluer le processus prêt qui est le plus prioritaire.

iii) Le constructeur alternatif (ALT)

```
ALT
  garde1
    proc1
  garde2
    proc2
  ...
```

Le processus alternatif attend jusqu'à ce qu'au moins un de ses processus gardés composants soit prêt à être exécuté (la garde correspondante devant s'évaluer à vrai). Si un ou plusieurs processus sont prêts, un seul est choisi (non-déterminisme interne résolu par l'implémentation du langage) et exécuté. Ensuite le processus alternatif se termine. Si celui-ci n'a pas de composant ou si aucune de ses gardes ne peut s'évaluer à vrai, il est équivalent au processus STOP.

Une garde peut être une condition locale (qui est facultative) suivie d'une entrée, ou d'un processus d'attente ou du processus SKIP. Une entrée s'évalue à vrai si le processus d'écriture sur le canal est prêt ; le processus d'attente s'évalue à vrai si le temps lu sur le canal est supérieur à la valeur de l'expression et le processus SKIP est toujours prêt.

Remarque : comme pour le constructeur PAR, il est possible en OCCAM de spécifier le choix de la garde vraie à exécuter. Ce nouveau constructeur qui respecte la sémantique du constructeur alternatif se note : **PRI ALT.**

```
PRI ALT
  garde1
    proc1
  garde2
    proc2
  ...
```

La garde 1 est plus prioritaire que la garde 2, la garde 2 est plus prioritaire que la garde 3, et ainsi de suite. Le constructeur PRI ALT spécifie l'exécution de la garde la plus prioritaire parmi celles qui sont évaluées à vrai.

iv) Le constructeur répétitif (WHILE)

```
WHILE expression
  proc
```

Le constructeur "WHILE" répète l'exécution du processus composant, tant que l'évaluation de l'expression donne la valeur vraie, sinon le constructeur termine.

2.3 Les extensions au langage OCCAM

Ces extensions au langage OCCAM sont transformées en une séquence d'intruction OCCAM, permettant d'utiliser les compilateurs du langage existant sur les Transputers, et de bénificier de la sémantique opérationnelle d'OCCAM et des travaux faits sur le modèle CSP.

Les constructeurs environnementaux (ENV)

Ces constructeurs n'existent pas dans le langage OCCAM, nous les avons introduits de façon à pouvoir décrire aisément des comportements liés à un environnement non-programmable [DIX83] (pouvoir spécifier qu'une application est prête à accepter à tout moment un événement externe). Une présentation formelle de l'extension temps critique d'OCCAM a été définie par ailleurs [MUNTE86].

Le constructeur ENV séquentiel

```
ENV SEQ
  proc_alt
  proc
```

Si le processus "proc" est actif et que le processus alternatif "proc_alt" est prêt, alors le constructeur ENV **suspend au plus tôt** le processus "proc" pour exécuter le processus "proc-alt". Quand le processus "proc_alt" termine, le

constructeur ENV reprend l'exécution du processus "proc".

Le constructeur ENV termine quand le processus "proc" termine.

Si le processus "proc" ne termine pas ou si le processus "proc_alt" ne termine pas alors qu'il a été prêt pendant l'exécution du processus "proc", alors le constructeur ENV est équivalent au processus STOP.

Si le processus "proc" est équivalent au processus SKIP, le constructeur ENV est lui aussi équivalent au processus SKIP.

Le constructeur ENV alternatif

```
ENV ALT
  proc_alt
  proc
```

Si le processus "proc" est actif et que le processus alternatif "proc_alt" est prêt, alors le constructeur ENV **arrête au plus tôt** le processus "proc" pour exécuter le processus "proc-alt". Le constructeur ENV termine lorsque le processus "proc_alt" termine, si celui-ci a été prêt au cours de l'exécution du processus "proc" ou lorsque le processus "proc" termine dans le cas contraire.

Remarque : ce constructeur permet d'implémenter, en OCCAM, les communications temporisées (communication qui doit être réalisée avant la date : "heure_limite") de la façon suivante :

```
ENV ALT
  TIME ? AFTER heure_limite
    -- processus exécuté si la communication n'a pas
    -- eu lieu avant l'heure limite
    processus
  canal ! message
```

Note : les autres constructeurs du langage OCCAM, moins spécifique à CONKER, seront présentés, si nécessaire lors de leurs utilisations.

3. Le Transputer

En annexe de cette partie, nous donnons une description sommaire du Transputer. Ce micro-processeur a été développé par INMOS [BRAIN84, BRETE85, INMOS85], et utilise OCCAM comme "langage d'assemblage".

Le Transputer est un circuit réalisé en technologie CMOS 1.5 microns qui comporte plus de 200.000 transistors. C'est une machine RISC (Reduced Instruction Set Computer) qui comporte 64 instructions de base. Elles permettent une implantation efficace des langages de haut niveau (et en particulier d'OCCAM, car il existe des instructions machines de communication). La structure interne du Transputer est présentée dans le schéma suivant.

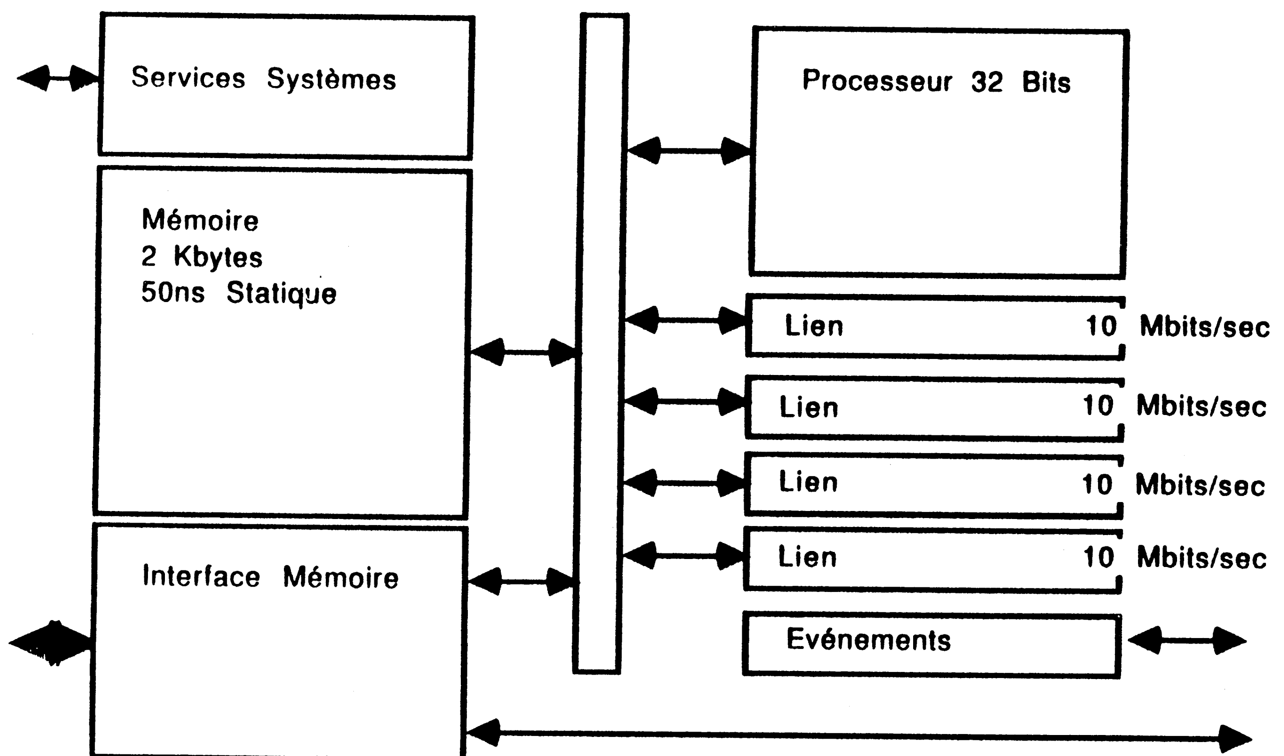


Figure 1. — Architecture du Transputer.

Chaque Transputer est composé :

1) d'un microprocesseur 32 bits ne reconnaissant que 64 instructions.

2) d'une mémoire locale de 2 Kbytes.

3) d'une interface lui permettant d'accéder à une mémoire externe (32 bits d'adresses).

4) de 4 liens permettant les communications soit avec d'autres Transputers, soit avec des interfaces spécifiques (Voie RS232, Contrôleur de disque, Bus VME ou PC).

5) L'ensemble de ses composants est construit autour d'un bus interne de 32 bits.

Le Transputer est déjà une architecture parallèle : il exécute simultanément plusieurs processus. Chaque processus possède un contexte réduit qui réside dans les registres du Transputer ce qui lui permet de commuter rapidement les contextes. Le Transputer permet l'exécution simultanée de tâches élémentaires qui peuvent être imbriquées et dont l'ensemble constitue des sous-tâches, elles-même imbricables : il supporte (presque) directement le langage OCCAM.

Le Transputer implante directement la communication : les processus résidant sur le même Transputer communiquent en utilisant des instructions de transfert de messages permettant de passer le contrôle d'un bloc mémoire d'un processus à un autre.

Le Transputer implante des facilités de communication pour l'accès aux ports externes : une communication entre deux processus s'exécutant sur deux Transputers (même si la fréquence de leur horloge n'est pas idendique) s'exprime de la même façon qu'une communication entre deux processus s'exécutant sur le même Transputer. Cette facilité permet de construire aisément des machines comportant plusieurs transputers (comme dans la figure suivante). Chaque flèche du schéma représente un lien d'un Transputer, et chaque carré un Transputer (les demi-carrés de chaque extrémité représentent le même Transputer). Ce mode de connexion permet de modéliser une surface plane sur un nombre fini de Transputers.

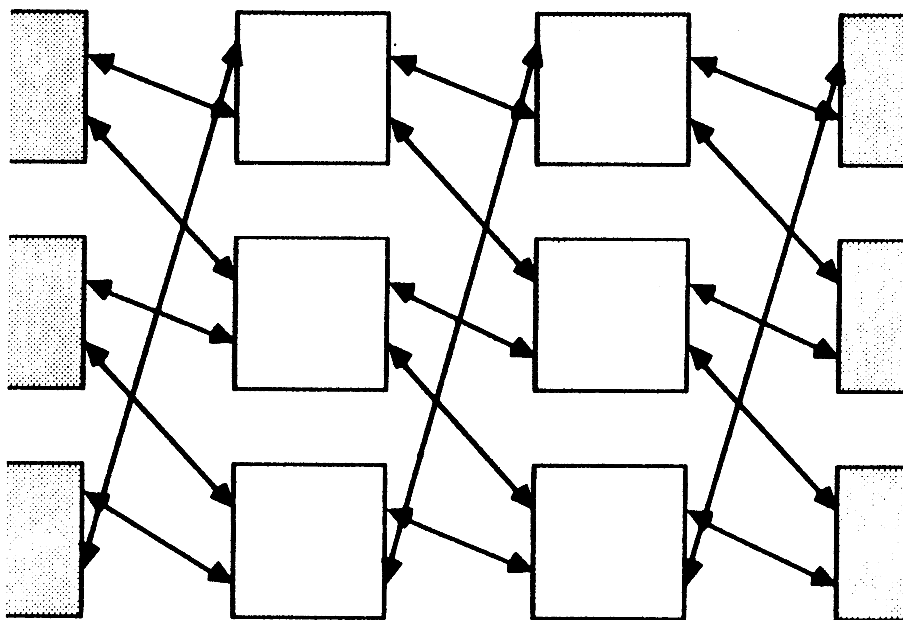


Figure 2. — Réseaux de Transputers.

Note : un Transputer "virgule flottante" est en cours de réalisation. Le circuit intègrera les algorithmes de calcul habituellement placés dans un coprocesseur.

4. Exemples

Les deux exemples qui suivent sont issus de la programmation "temps critique" et permettent d'illustrer l'utilisation du langage OCCAM et de ses extensions.

4.1 Exemple de programme OCCAM : un compteur d'essieux

Dans cet exemple, on désire compter le nombre d'essieux circulant dans chaque sens sur une voie ferrée. Pour cela, la voie ferrée est divisée en cantons, et entre chaque canton sont disposées deux pédales qui se recouvrent (voir figure 3 ci-dessous).

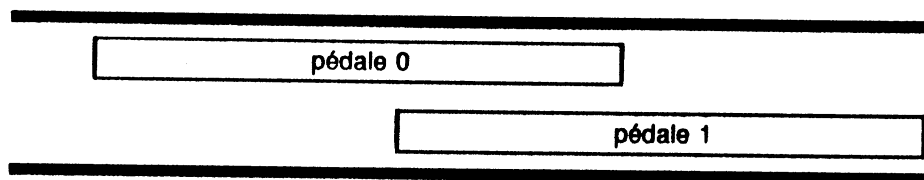
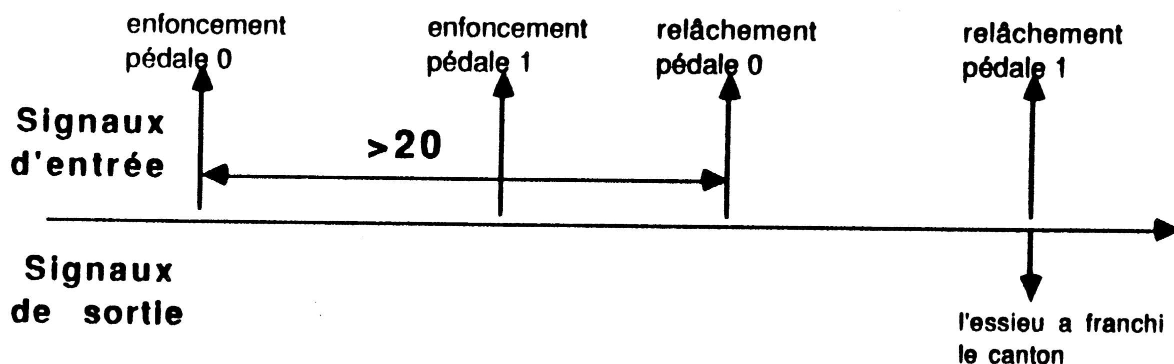
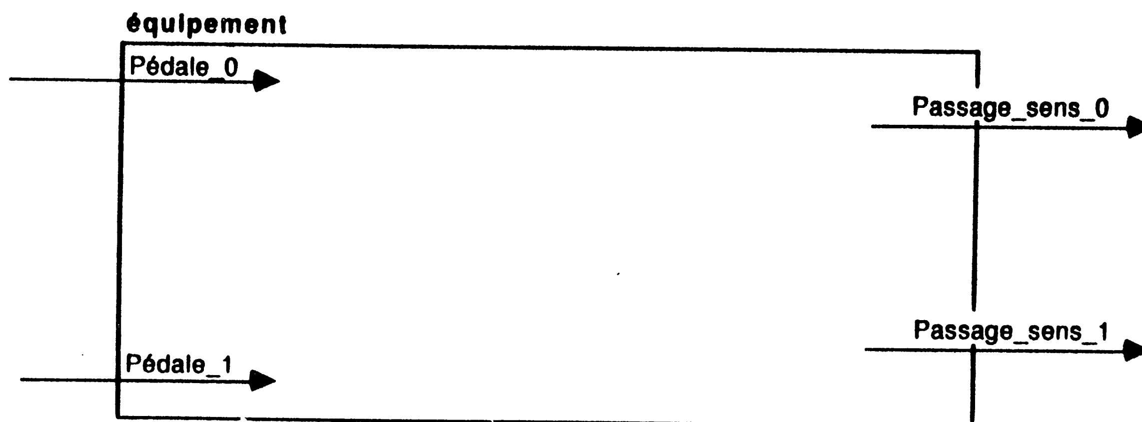


Figure 3. — Schéma de la voie.

Le système de contrôle, disposé entre deux cantons, envoie un signal propre à chaque pédale, chaque fois qu'elle est manoeuvrée (enfoncement ou relâchement). L'équipement de comptage envoie un signal sur un de ses deux canaux de sortie (un pour chaque sens) chaque fois qu'un essieu est passé.





Nous donnons ci-après un programme illustratif qui résoud ce problème, sans tenir compte d'éventuel demi-tour effectué lors du franchissement du canton. Le processus OCCAM est construit à l'aide de quatre processus qui s'exécutent en parallèle. Deux processus permettent de dupliquer les signaux et les deux autres "sorties" contrôlent le franchissement de la voie (un processus pour chaque sens de franchissement).

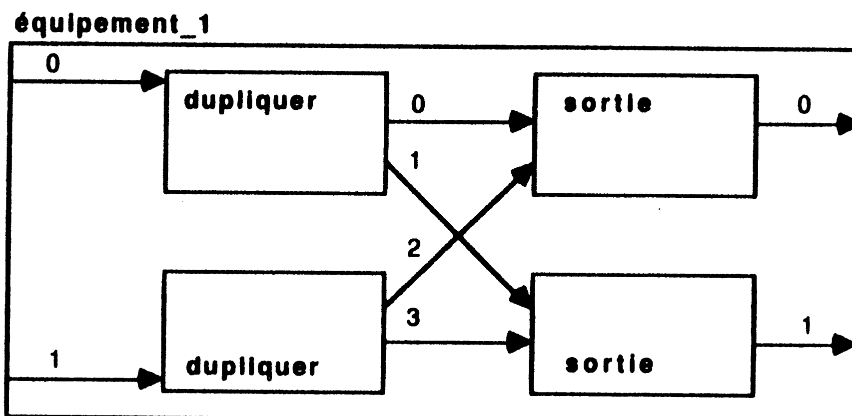


Figure 5. — Schéma de l'équipement 1.

```
PROC équipement_1( CHAN entrée[], sortie[] ) =

 PROC duppliquer(CHAN in, out1, out2) =
  WHILE TRUE
   SEQ
    in ? ANY
    PAR
     out1 ! ANY
     out2 ! ANY :

 PROC sortie( CHAN in1, in2, out ) =
  WHILE TRUE
   SEQ
    in1 ? ANY
    in2 ? ANY
    in1 ? ANY
    PAR
     out ! ANY
     in2 ? ANY :
```

Note_1 : le constructeur PAR utilisé dans le processus "sortie", permet de réaliser de façon indéterministe l'une des deux séquences suivantes (out ! ANY ; in2 ? ANY) ou (in2 ? ANY ; out ! ANY).

Note_2 : les seconds processus "in1 ? ANY" ou "in2 ? ANY" appartenant au processus "sortie" sont nécessaires pour "absorber" le signal de relâchement de chacune des deux pédales.

```
CHAN signal[ 4 ] :

PAR
  dupliquer( entrée[ 0 ], signal[ 0 ], signal[ 1 ] )
  dupliquer( entrée[ 1 ], signal[ 2 ], signal[ 3 ] )
  sortie( signal[ 0 ], signal[ 2 ], sortie[ 0 ] )
  sortie( signal[ 1 ], signal[ 3 ], sortie[ 1 ] ) :
```

Des parasites peuvent en outre se produire sur les capteurs qui détectent l'enfoncement des pédales. Ces parasites provoquent l'émission d'un signal sur le canal associé au capteur, mais la durée de maintien de la valeur sera inférieure à 10, alors que la présence effective d'un essieu sur une pédale maintient la pédale enfoncée durant au moins 20.

Le processus OCCAM, qui résoud l'équipement du nouveau comportement, est construit à l'aide de trois processus qui s'exécutent en parallèle. Les deux premiers "filtres" se chargent d'éliminer les parasites et le troisième, "équipement_1" est constitué par l'équipement précédemment défini.

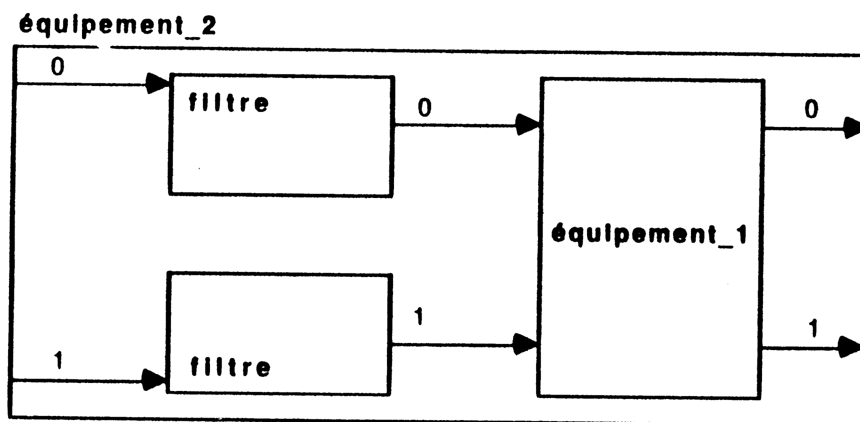


Figure 6. — Schéma de l'équipement de contrôle complet.

```
PROC équipement_2( CHAN entrée[], sortie[] ) =

 PROC filtre( CHAN in, out ) =
  WHILE TRUE
   SEQ
    in ? ANY
    TIME ? date
    ALT
     -- si le signal dure moins de 10 c'est un parasite
     in ? ANY
      SKIP
     TIME ? AFTER date + 10
      SEQ
       out ! ANY
       in ? ANY
       out ! ANY :
```

```
PROC équipement_1( CHAN in[], out[] ) =
  <comportement défini précédemment>

CHAN signal[ 2 ] :

PAR
  filtre( entrée[ 0 ], signal[ 0 ] )
  filtre( entrée[ 1 ], signal[ 1 ] )
  équipement_1( signal, sortie ) :
```

Ces processus résolvent le problème posé sous deux conditions d'implémentation :

- que le temps d'attente de 10 exécuté par le processus filtre corresponde à un temps d'attente de 10 pour l'environnement (correspondance entre l'horloge locale du processus et le temps de l'application).

- que l'équipement qui récupère la sortie soit prêt dès que le signal est disponible.

Dans le rapport nous discutons d'une autre condition d'implémentation, liée à l'implémentation du temps et au respect des règles temporelles indiquées dans le programme source. Dans le cas présent, quand le concepteur écrit **TIME ? AFTER date + 10**, il désire que l'implémentation réalise effectivement un temps d'attente de 10 quelle que soit la charge du processeur.

4.2 Un autre exemple de programme OCCAM : un diviseur de fréquence

Cette exemple montre la nécessité d'étendre le langage OCCAM afin de permettre une programmation "temporelle".

Dans le programme ci-dessous, le processus horloge doit envoyer sur le canal top un message à chaque intervalle de temps qui a une durée équivalente à "attente". Le respect de cette contrainte dépend directement de l'implémentation réalisée : est-ce que cette implémentation garantit qu'à chaque intervalle de temps défini, le processus connecté au canal top pourra lire le message ? Et est-ce que le temps d'exécution de la boucle "WHILE" est inférieur à la longueur de l'intervalle ?

```
PROC horloge ( CHAN top, VALUE attente ) =
  VAR t :
  SEQ
    TIME ? t
    t := t + attente
    WHILE TRUE
      SEQ
        TIME ? AFTER t
        top ! ANY
        t := t + attente :
```

Afin de pouvoir programmer ce genre de contrainte, nécessaire dans les applications "temps critique", nous avons introduit dans un paragraphe précédent le constructeur ENV. Nous présentons ci-dessous, le même processus "horloge" qui implémente correctement la

contrainte temporelle : si l'implémentation ne permet pas de tenir cette contrainte, le processus horloge exécute le processus d'erreur "proc_erreur".

```
PROC horloge ( CHAN top, VALUE attente ) =
 VAR t :
 SEQ
  TIME ? t
  t := t + attente
  WHILE TRUE
   ENV
    TIME ? AFTER (t + attente)
     proc_erreur
    SEQ
     TIME ? AFTER t
     top ! ANY
     t := t + attente :
```

III. CONKER : définition du noyau

Résumé

La fonction essentielle d'un noyau de communication pour systèmes répartis est de permettre la réalisation des différents protocoles d'échanges (communication et synchronisation) nécessaires à l'exécution correcte de l'application.

CONKER est un noyau de communication pour application "temps critique" qui repose sur la séparation des fonctions de traitement assurées par les processus de l'application et de la fonction de communication-synchronisation assurée par les connecteurs. CONKER permet d'exprimer de façon homogène la communication et la synchronisation dans le temps entre différents processus coopérants, que ceux-ci s'exécutent sur le même processeur ou sur plusieurs (réseau, multi-processeurs).

Pour offrir cela, CONKER met à la disposition des applications un ensemble adéquat de types de **"connecteurs de communication"** qui permettent la construction des différents modes de communication intégrant la synchronisation nécessaire à l'exécution d'une application "temps réel".

Un connecteur est un objet typé. L'ensemble des connecteurs présents sur le système permettent de réaliser les différents protocoles d'échange de messages nécessaires à l'application, ainsi que l'intégration dans la communication de la synchronisation souhaitée entre les processus communicants.

Ce chapitre nous permettra de spécifier le système CONKER à l'aide du langage OCCAM.

1. Démarche

Défini pour un type particulier de systèmes : les systèmes "environnementaux" (c'est à dire ceux qui évoluent en fonction et en interaction permanente avec l'environnement), CONKER repose sur l'utilisation d'un modèle de langage parallèle particulier (CSP de Hoare).

Cette démarche nous a permis :

- en visant un type d'applications particulier, de mieux "cibler" ce que devait être le noyau CONKER.
- en s'appuyant sur un modèle, de pouvoir ultérieurement valider la construction correcte des connecteurs et des processus (étape qui n'est pas décrite dans ce travail, mais pour plus de précisions nous renvoyons le lecteur à [MUNTE86a]). Actuellement, CONKER propose une méthodologie pour la description des communications et synchronisations.
- en utilisant un langage parallèle issu de CSP : OCCAM [MAY83, INMOS84] d'avoir des spécifications directement exécutables sur différentes machines (VAX780 VMS ou UNIX, LSI11, ...).

Comme pour de nombreux systèmes (CHORUS, OMPHALE par exemple), **les seuls événements manipulables** par le système CONKER sont l'envoi et la réception d'un message, la notion de message allant d'un simple signal (interruption, top d'horloge, signal de capteur, ...etc) à des types construits comme des structures complexes de données.

Pour CONKER, **l'envoi et la réception d'un message sont deux événements simultanés** : à chaque envoi d'un message correspond au même instant la réception du même message puisque la communication entre deux entités se fait par rendez-vous.

Le langage OCCAM a été présenté dans le chapitre précédent et nous allons examiner les caractéristiques des applications "environnementales" en essayant de mettre en avant l'influence d'OCCAM sur la structuration du modèle CONKER.

2. Caractéristiques des applications visées

CONKER a été défini pour permettre l'implantation aisée des applications communément appelées "temps réel". Ce type d'application se caractérise par la nécessité d'exprimer à la fois : des **contraintes temporelles de délai** pour les communications entre les différents processus parallèles qui la composent, et des **contraintes particulières de synchronisation** entre ces processus.

Prenons, par exemple, le contrôleur de robot schématisé à la figure suivante. Il se décompose en trois modules :

- **le contrôleur des capteurs** qui envoie un message au contrôleur du déplacement chaque fois qu'un des capteurs qu'il surveille dépasse un certain seuil.
- **le contrôleur de vision** qui envoie un message au contrôleur du déplacement chaque fois qu'il détecte la présence d'un objet sur la trajectoire du robot.
- **le contrôleur du déplacement** qui envoie un message à intervalle régulier au système d'asservissement du robot en suivant les consignes qu'il peut recevoir des autres modules. La fréquence d'envoi des messages dépend directement du matériel (moteur d'asservissement des bras du robot).

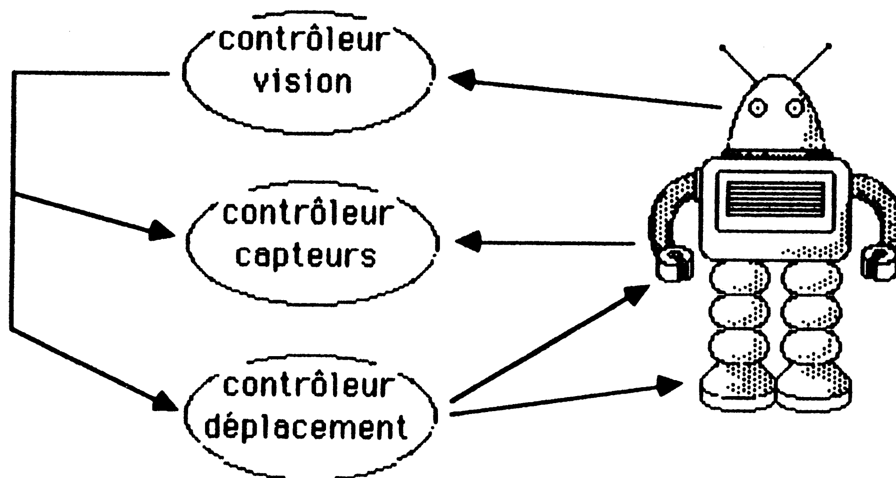


Figure 1. — Contrôleur de robot.

Dans un système de commande de robot, les capteurs permettent de contrôler les déplacements d'un bras (rencontre d'un obstacle, suivi de surface) ou les mouvements d'un outil terminal (prise d'un objet, passage d'un objet d'un bras manipulateur à un autre). Pour que les capteurs utilisés assurent leurs rôles de contrôle (mécanisme équivalent aux gardes du langage CSP), il est nécessaire que les informations qu'ils donnent puissent être connues suffisamment tôt par le contrôleur du déplacement (Pour chaque information, il existe une date après laquelle elle est obsolète). **Le système de robot impose une contrainte de délai**

pour la communication entre le contrôleur des capteurs et le contrôleur du déplacement.

De même les informations données par le contrôleur de vision (présence d'objet, analyse de scène) n'ont un sens que pendant un délai temporel dépendant de l'application. Si le robot ne se déplace qu'après l'analyse d'une scène et que les objets sont statiques ce délai peut être infini. **Le système impose alors uniquement une contrainte de synchronisation entre le processus de prise d'image et le contrôleur du déplacement.** Mais si le déplacement s'effectue pendant l'analyse, il devient nécessaire de pouvoir exprimer en plus une **contrainte de délai entre le processus de prise d'image et le contrôleur du déplacement qui utilise cette image** : utilisation de la dernière image prise, qui peut être considérée comme exploitable seulement durant un délai donné.

Si l'application nécessite de faire passer un objet d'un bras manipulateur à un autre, **le système de robot impose alors une contrainte de synchronisation dans le temps, mais aussi dans l'espace** : les deux bras manipulateurs doivent pouvoir saisir un objet situé au même endroit (sans se gêner), et un bras doit prendre l'objet avant que l'autre ne le lâche.

Pour pouvoir s'exécuter correctement, ce type d'application nécessite, en outre, la prise en compte de contraintes dictées par l'environnement (qui est généralement un processus physique non programmable ayant sa propre dynamique) :

- nécessité de garantir la **prise en compte des communications avec l'environnement** dans lequel les équipements ne respectent pas toujours des protocoles de communication de type question-réponse (comme c'est le cas par exemple pour les capteurs où il faut effectuer une prise de valeur pour en connaître la valeur courante). Ces événements sont souvent non prévisibles et par conséquent non programmables (à chaque instant l'application doit être prête à accepter tous les événements significatifs pouvant survenir de l'environnement). En effet une **perte d'information** en provenance de l'environnement est très **pénalisante** parce que l'on n'a aucun moyen de la récupérer. Par contre, il peut être intéressant de mettre en place une stratégie pour traiter le flot d'informations dans le cas où l'application ne va pas suffisament vite. Par exemple dans le cas d'un gestionnaire de "souris" [JOLOB84], si entre deux commandes on n'a pas le temps de mettre à jour la mémoire d'écran on peut choisir soit de perdre l'information (charge à l'environnement de répéter la commande), soit de mémoriser la suite des opérations à effectuer.

- pour certaines applications, on a besoin de **manipuler une notion de temps.** Ce temps propre à l'application, ne prend un sens que s'il est défini par rapport à celui de l'environnement qui fixe les points de référence. En effet, **les points de synchronisation**[5] entre les différentes entités concurrentes participant à la

5. *Un point de synchronisation est un événement observable simultanément par toutes les entités devant se synchroniser sur cet événement.*

réalisation de l'application sont définis par **la structure de l'environnement.** C'est aussi celui-ci qui impose, par sa dynamique, les **délais de réponse** face à des stimulis externes et au système de processus, une certaine "vitesse" d'exécution afin de respecter certaines règles de synchronisation particulières.

- ces applications ont dans leur ensemble **besoin d'outils pour exprimer la coopération** entre des activités qui participent à la réalisation d'une tâche commune. Ces outils langage et système doivent permettre de décrire et de réaliser de nombreux modes de communication et de synchronisation temporels. Pour offrir toutes les possibilités de communication-synchronisation nécessaires, deux démarches peuvent être prises :

  - soit définir un mode de connexion[6] de base et permettre ainsi aux concepteurs d'applications de construire les autres modes à l'aide de celui-ci (c'est ce qui se fait en OCCAM, le mode primitif étant le rendez-vous).

  - soit définir des objets et un modèle algébrique qui permettent de construire, au fur et à mesure des besoins de l'application, les différents types de communication-synchronisation nécessaires (c'est la démarche que nous avons adoptés pour la conception de CONKER).

Un intermédiaire entre ces deux démarches a parfois été choisi : le système offre différents modes de communication simple (asynchrone, synchrone, invocation) et permet la construction de modes plus élaborés par insertion de processus intermédiaires. Ce choix a été fait par exemple pour STARMOD [COOK80].

Notre démarche s'inscrit dans un cadre de recherche plus général. En effet, contrairement aux systèmes de traitement actuels, nous préconisons une nouvelle méthode pour la construction des services que les systèmes de traitement doivent offrir à une application future. A partir d'un modèle de service de base bien défini sémantiquement et d'un ensemble de constructeurs algébriques, le noyau du système devra construire correctement, et à la demande, des services nouveaux par inférence sur les services connus.

Bien que les applications visées en premier lieu par CONKER soient issues de la programmation environnementale, ce modèle pourra être utilisé (et si c'est nécessaire étendu) pour la programmation d'autres types d'applications concurrentes. Par exemple :

- dans un langage de description de réseau systolique [KUNG82], les connecteurs de CONKER permettent d'exprimer la connexion entre les cellules (processus). Chaque connecteur assure la synchronisation des flots de données entre les

---

6. *Nous appellerons connexion, un type de communication-synchronisation particulier assuré par une entité spécifique (canaux d'OCCAM, connecteurs de CONKER, pipe d'UNIX, ...). Nous dirons par exemple que les canaux d'OCCAM réalisent une connexion par rendez-vous uni-directionnel entre deux processus.*

cellules ce qui permet d'assurer le fonctionnement de l'ensemble des cellules en mode systolique.

- dans la simulation logique, les connecteurs permettent la gestion répartie d'un système d'horloge et le cadencement des processus sur ces horloges.
- dans les applications transactionnelles, les protocoles de validation-reprise intégrés dans les connecteurs libèreront le programmeur d'application de leurs constructions explicites.

3. Le modèle de communication CONKER

Nous allons introduire à l'aide du langage OCCAM, la notion de communication par canal. Cette partie nous permettra de mettre en évidence les limites de cette approche à l'aide du langage OCCAM et de proposer le modèle CONKER.

3.1 La communication par canal en OCCAM

La communication en OCCAM s'exprime de la même façon que les processus s'exécutent sur le même Transputer ou sur deux Transputeurs voisins (exemple de programme donné à la page suivante).

Soient les programmes des deux processus consommateur et producteur suivants :

```
PROC consommateur( CHAN in ) =
 VAR msg :
 WHILE TRUE
  SEQ
   in ? msg
   traiter( msg ) :

PROC producteur( CHAN out ) =
 VAR msg :
 WHILE TRUE
  SEQ
   produire( msg )
   out ! msg :
```

3.1.1 Connexion directe

Si on connecte ces deux processus par un canal OCCAM, on réalise une synchronisation forte par rendez-vous entre le processus consommateur et le processus producteur. Ceci peut être fait de la façon suivante :

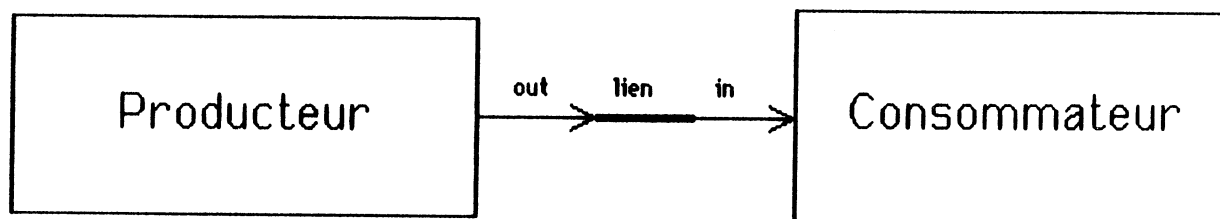


Figure 2. — Connexion directe.

```
CHAN lien :
PAR
 consommateur( lien )
 producteur( lien )
```

Si on veut exécuter ces deux processus sur deux Transputers [BARRO83, WALKE85] distincts on écrira le programme suivant (en reprenant les mêmes processus consommateur et producteur) :

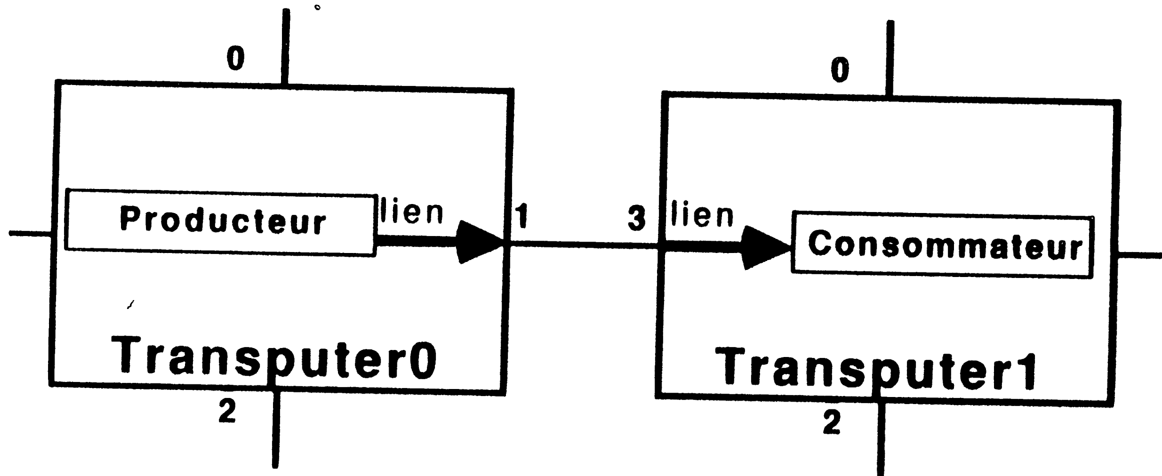


Figure 3. — Architecture multi-Transputers.

```
CHAN lien :
PLACED PAR
  ALLOCATE 0
    PORT 1 :- lien
    LOAD 0 :
    producteur( lien )
  ALLOCATE 1
    PORT 3 :- lien
    LOAD 1 :
    consommateur( lien )
```

Note : Le port 1 du Transputer 0 est relié physiquement au port 3 du Transputer 1.

Dans ce court programme sont introduits quatre nouveaux mots clés du langage OCCAM permettant de "placer" les processus sur les différents Transputers :

- **PLACED PAR** : cette commande permet de prévenir le compilateur que le programme qui suit, sera composé de plusieurs processus et que chaque processus devra être chargé sur un Transputer différent.

- **ALLOCATE number** : le processus qui suit cette commande sera chargé sur le Transputer "number". Ce numéro dépend uniquement de la réalisation matérielle. Dans l'exemple ci-dessus, ce programme s'exécutera sur deux Transputers.

- **PORT number** : cette commande permet d'allouer les canaux externes du processus (au plus quatre dans chaque sens) aux ports externes du Transputer. Sur chaque port du Transputer, il est possible d'affecter au plus un canal dans chaque sens. Dans l'exemple ci-dessus, seul le canal "lien" est alloué à un port de chaque Transputer.

- **LOAD number :** cette commande permet de spécifier par quel port physique du Transputer sera chargé le code du processus à exécuter.

- *Note : il n'est pas possible, avec les instructions du langage OCCAM de spécifier le réseau d'interconnexion entre les Transputers. Celui-ci est réalisé statiquement par la carte supportant les Transputers et le concepteur connait l'interconnexion des Transputers.*

Le processus "Producteur" sera exécuté sur le Transputer 0, le canal lien sera associé au port 1 et le port 0 permettra de charger le code du processus. Le processus "Consommateur" sera, exécuté sur le Tranputer 1, son canal lien sera connecté sur le port 3 et son code sera chargé par le port 1.

3.1.2 Réalisation d'autres protocoles

Pour réaliser d'autres types de synchronisation entre les processus il est nécessaire d'insérer entre le processus consommateur et le processus producteur un autre processus que l'on nommera **connecteur.** Celui-ci gèrera ce nouveau protocole d'échanges, et on aura par exemple :

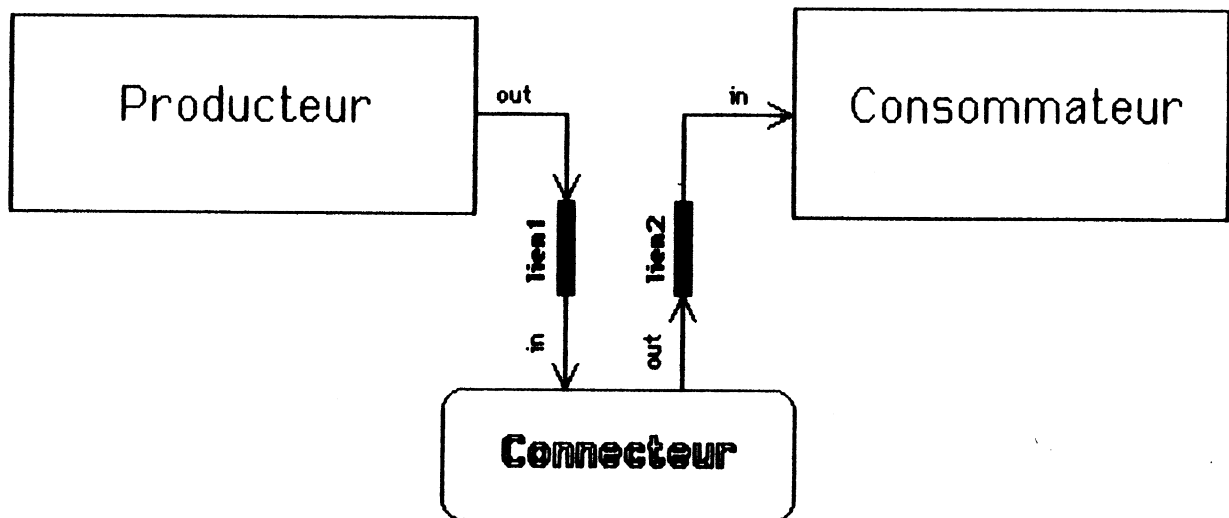


Figure 4. — Processus intermédiaire.

```
PROC connecteur( CHAN in, out ) =
  comportement :

CHAN lien1, lien2 :
PAR
  producteur( lien1 )
  connecteur( lien1, lien2 )
  consommateur( lien2 )
```

Exemple de comportements :

Exemple de processus simple : connecteur 1_tampon (ce connecteur gère un tampon de longueur un).

```
PROC comportement( CHAN in, out ) =
 WHILE TRUE
 VAR msg :
  SEQ
   in ? msg
   out ! msg :
```

Exemple de processus composé : connecteur N_tampon (ce connecteur gère un tampon longueur N, N étant supérieur ou égal à un).

```
PROC comportement( CHAN in, out, VALUE N ) =

 PROC tampon( CHAN in, out ) =
  VAR msg :
  WHILE TRUE
   SEQ
    in ? msg
    out ! msg :

 IF
  N=1
   tampon( in, out )
  N>1
   CHAN lien[ N-1 ] :
   PAR
    tampon( in, lien[ 0 ] )
    PAR i = [ 0 FOR N-2 ]
     tampon( lien[ i ], lien[ i+1 ] )
    tampon( lien[ N-2 ], out ) :
```

Note : le multiplexeur "i = [base FOR cardinal]" est utilisé avec un constructeur pour créer une liste de processus, identiques à celui défini par le constructeur. Le multiplexeur introduit un identificateur qui sera utilisé par le constructeur comme index à partir de la valeur entière "base" et autant de fois que l'indique la valeur entière "cardinal".

Cette construction réalise un N_tampons par un pipeline de longueur N ou chaque cellule réalise un 1_tampon. Les cellules extrêmes du pipeline constituent les bouts du N_tampons.

3.1.3 Limite de ce mode de construction

Cette approche permet de réaliser agréablement un certain nombre de types de communication, à partir d'un protocole de base (rendez-vous). Mais il n'est pas possible de garder la même interface d'accès, pour réaliser tous les protocoles d'échanges désirés. Ceci est dû aux limites du langage OCCAM et au nombre restreint de paramètres :

- le langage OCCAM n'autorisant pas les sorties dans les gardes, il n'est pas possible d'écrire **des protocoles d'échange qui nécessitent de pouvoir choisir alternativement** entre une entrée d'un message sur le connecteur ou une sortie du message courant à moins de changer l'interface de connexion des processus

au connecteur.

Par exemple, la description d'un processus connecteur qui enverrait à un processus (et sur sa demande) le dernier message émis par un autre processus n'est pas réalisable sans une modification soit du langage OCCAM, soit du mode de connexion (ajout de canaux). Ce type de connecteur est utilisé pour réaliser par exemple une horloge. En effet dans ce type de connexion, il est nécessaire de connaître uniquement la valeur courante véhiculée par le connecteur et non pas l'ensemble des valeurs que celui-ci a véhiculées.

```
PROC comportement =
-- solution avec garde de sortie
-- qu'on ne peut réaliser
 VAR produit, msg :
 SEQ
  produit := FALSE
  WHILE TRUE
   ALT
    in ? msg
     produit := TRUE
    produit & out ! msg
     SKIP :
```

```
-- solution qui peut être adoptée, si le processus consommateur
-- ne met pas le canal "out" dans une alternative
   ALT
    in ? msg
     produit := TRUE
    -- demande de lecture qui rompt
    -- l'homogénéitée recherchée
    produit & out1 ? ANY
     out2 ! msg :
```

- *on ne peut pas écrire un* **type de connecteur paramétrable.** Par exemple, le connecteur N_tampon, écrit précédemment, nécessite l'initialisation de la taille du tampon qui peut être faite par l'envoi d'un message (qu'il faut distinguer des messages de communication) ou par une instanciation de la taille lors de la création du connecteur. Ce type de connecteur écrit directement avec le langage OCCAM nécessite un paramètre supplémentaire.

- les communications en OCCAM se font par des canaux qui sont unidirectionnels et qui relient deux processus. Ceci ne permet pas de créer (avec l'interface précédente) des **protocoles d'échange entre plusieurs processus émetteurs et plusieurs processus récepteurs** (pour assurer, par exemple, la diffusion d'un message).

Vouloir écrire l'ensemble des protocoles d'échanges nécessaires à une application donnée en OCCAM, nécessite l'écriture d'une interface de connexion, spécifique à chaque protocole, et rend peu modulaire et peu évolutif un programme.

3.2 Les connecteurs

Le modèle CONKER reprend cette démarche de construction de nouveau mode de communication par insertion systématique d'un objet spécifique chargé de la communication-synchronisation : **le connecteur.** CONKER devra permettre de réaliser des programmes modulaires où les modes de communication-synchronisation pourront évoluer.

- les connecteurs de CONKER permettent de changer le type de communication (et par conséquent le type de synchronisation) sans modifier l'écriture des processus.

- l'utilisation d'une bibliothèque de connecteurs prouvés permet d'assurer une communication qui protège les données transmises. Les connecteurs peuvent être spécifiés (et programmés) en OCCAM (par exemple), ce qui permet d'utiliser les méthodes de preuve issues du modèle CSP.

- CONKER devra offrir, dans un proche avenir, la possibilité de construire de nouveaux connecteurs, assurant des protocoles d'échanges spécifiques. Ces connecteurs seront construits de manière sûre à l'aide des connecteurs déjà connus. Cette démarche permettra au programmeur d'avoir à sa disposition un ensemble de connecteurs prouvés.

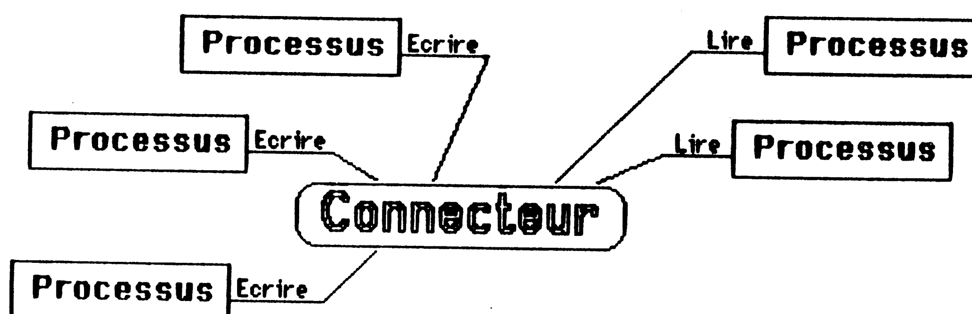


Figure 5. — Connecteurs.

Dans CONKER, nous proposons de construire un programme parallèle à l'aide de deux types d'entités (voir figure précédente) :

- **les processus de traitement** qui effectuent les calculs. Ce sont des programmes séquentiels. A l'intérieur de ces programmes apparaîtront des opérations de communication portant sur des variables locales aux processus : écrire(canal, message) et lire(canal, message). Ces processus séquentiels peuvent être écrits dans différents langages "séquentiels" pourvus de primitives d'écriture de "message" sur un canal.

- **les connecteurs** qui réalisent les échanges de messages selon la stratégie de communication à implémenter. Le connecteur devient, à la différence des canaux d'OCCAM, un objet dynamique pouvant assurer différents services (transport des messages mais aussi gestion des événements propres au canal).

Pour différentes raisons, dont la principale est une implantation du noyau CONKER sur une architecture multi-Transputers réalisée en grande partie à l'aide du langage OCCAM, nous construirons dans ce mémoire ces objets en OCCAM. Certains ont été réalisés en PLM86 pour l'implémentation de CONKER sur le système iRMX86 (nous renvoyons le lecteur au chapitre 3 de ce même mémoire).

L'utilisation du rendez-vous comme primitive de communication de base entre entités dans CONKER, permet d'assurer à l'entité émettrice du message qu'après chaque communication, le message émis a été correctement reçu par l'entité réceptrice : cette correction est assurée par l'implémention du rendez-vous réalisée par le noyau de CONKER. La gestion des communications entre les processus se faisant par des connecteurs, il est évident que pour pouvoir utiliser pleinement CONKER, il est souhaitable de posséder un langage de spécification permettant de typer les connecteurs.

Les caractéristiques d'un tel langage doivent permettre :

- **la spécification de l'architecture.** Elle doit permettre de décrire les structures de communication existantes entre les différents processeurs du système et les ressources associées à chacun d'eux [COOK80]. En outre un tel langage doit permettre la description des caractéristiques propres à chaque machine : vitesse d'exécution, processus spécifique (architecture systolique, GAPP [BRETE85a]) afin que le système puisse ultérieurement gèrer au mieux la répartition des processus.

- **la spécification externe de chaque entité.** Elle doit permettre de définir pour chaque entité l'interface de communication avec son environnement : type de signal ou de message, mode de synchonisation à effectuer sur chaque connexion (sens de l'échange, protocole, ...), équipement de connexions spécifiques, ...

- **l'implémentation des processus.** Cette étape doit permettre la description du comportement des processus et l'allocation des différents processus aux processeurs du système : soit de manière statique et c'est ce que permet l'instruction du langage OCCAM, PLACED PAR ; soit de manière dynamique et le système doit alors permettre de mettre en correspondance une description logique avec une description physique. Mais dans un langage de haut niveau, le programmeur doit non seulement pouvoir imposer le site de création du processus, mais il doit aussi pouvoir laisser le choix de création au système CONKER afin de réaliser des optimisations selon différents critères (réduction de l'encombrement du réseau, temps d'exécution minimal, parallélisme maximal).

- **d'offrir une méthodologie de construction incrémentale des connecteurs.** Outre la possibilité de pouvoir programmer de façon agréable les connecteurs, il s'agit d'offrir une méthodologie permettant de construire de façon sûre les connecteurs devant réaliser de nouveaux protocoles d'échange. Cette méthologie devra être supportée par un modèle algébrique offrant un ensemble d'opérations et devant permettre la description :

- des données échangées. Ces données devront être typées, et chaque connecteur pourra en accepter un ou plusieurs types. Ceux-ci pourront être un ensemble de mots mémoire devant être lus dans leur intégralité (et le système devra garantir que le message le soit) ou avoir une structure plus complexe (et chaque entité participant à la communication devra savoir manipuler ce type de message).

- de l'ensemble des entités devant participer à la communication : un connecteur pourra être implémenté par un ou plusieurs processus.

- de la synchronisation à réaliser entre les différentes entités : description des relations entre l'ensemble des valeurs émises, l'ensemble des valeurs reçues et l'ensemble des valeurs consommables. Ces relations temporelles devraient permettre la validation de nouveaux protocoles d'échange.

4. Définition du noyau de CONKER

Dans ce chapitre, nous allons définir les fonctionnalités réalisées par le noyau de CONKER, les mécanismes de protection et de désignation mis en place.

Le noyau de CONKER permet de réaliser entre deux entités du système (un processus et un connecteur) une communication par rendez-vous temporel et synchrone d'un type proche de celui que réalise CSP. Pour réaliser cette communication chaque entité de CONKER, utilise la primitive "SEND". Cette primitive, la seule du noyau CONKER, sera implémentée dans chaque noyau CONKER, et sera décrite dans le paragraphe 4.1.1.

Le noyau de CONKER est **homogène** dans le sens où, quel que soit le système de traitement hôte sur lequel il repose, il offre aux processus de l'application la même interface pour la communication et la synchronisation. En effet un événement externe, une interruption du système ou une synchronisation particulière entre deux processus est toujours perçu par le système sous la forme d'un message.

Cette homogénéïté entre la synchronisation et la communication vise à assurer au système :

- un maximum de **portabilité :** ce qui permet de pouvoir implémenter CONKER sur des systèmes hôtes ayant des primitives de base différentes.
- une certaine **adaptabilité** dans la réalisation des connecteurs : ce qui permet d'utiliser avec le maximum d'efficacité les primitives de chaque système de traitement hôte.
- une **transparence** sur la localisation d'un processus : un processus communique toujours avec un connecteur local, sans se préoccuper de savoir si le processus destinataire du message est local ou distant.
- une **modularité dans les connexions entre les processus :** le changement du protocole d'échange réalisé par un connecteur entre deux processus de l'application ne nécessite pas la réécriture du code exécuté par les processus.

Chaque entité de CONKER est l'unité de découpage fonctionnel, l'unité de protection (par encapsulation des données) et l'unité d'exécution (chaque entité est autonome) de l'application.

La notion de mémoire commune n'existe pas au niveau de l'architecture de CONKER. Les entités communiquent uniquement par envoi de messages, en utilisant la primitive "SEND", qu'elles soient ou non sur le même site.

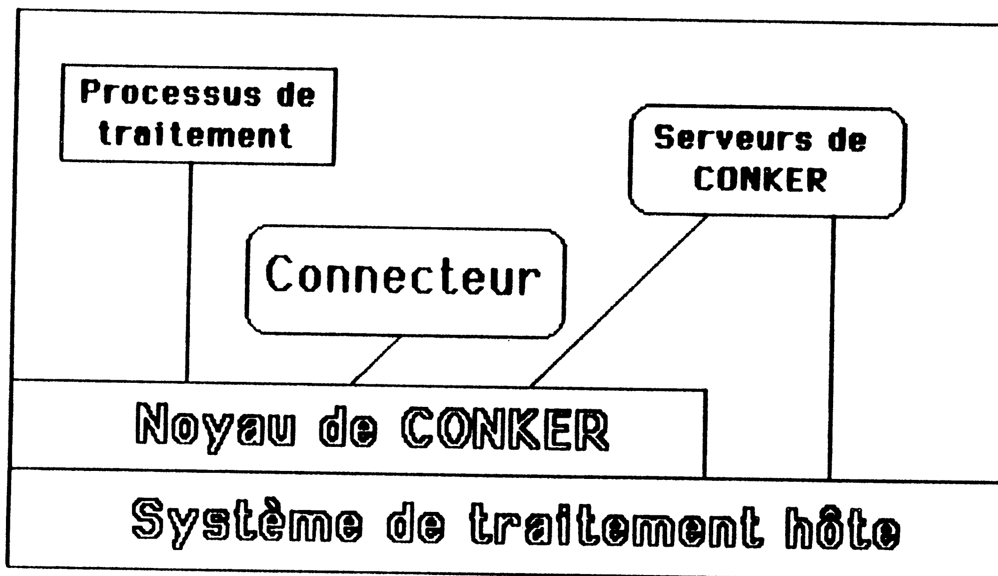


Figure 6. — Architecture de CONKER.

L'architecture du système (voir figure ci-dessus) repose sur trois types d'entités qui sont programmées sur le même modèle, mais qui diffèrent par leurs fonctionnalités. Pour pouvoir s'exécuter, ces entités utilisent les services de communication et de synchronisation réalisés par le noyau.

- les **serveurs de CONKER** offrent tous les services que l'on est en mesure d'attendre d'un système de traitement et qui ne sont pas réalisés par le noyau. Par exemple : création des entités, gestion des fichiers, gestion des noms, ... Ces services ne sont pas réalisés par le noyau :
  - parce qu'ils ne sont pas indispensables dans toutes les réalisations.
  - parce que leur complexité dépend étroitement du système de traitement hôte.
  - afin de réaliser un noyau minimal qui ne gère que les communications et les synchronisations.
- les **processus de l'applications** sont séquentiels et s'exécutent localement. Ils assurent le traitement des messages reçus. Ils interagissent avec leur environnement uniquement par envoi de messages qui sont convoyés par les connecteurs.

- les **connecteurs** assurent le transport des messages entre les processus de l'application qui leurs sont connectés. Chaque connecteur réalise un protocole d'échange spécifique (file FIFO, file à priorité de longueur bornée ou d'autres modes d'échange plus spécifiques à l'application). Si les processus qu'ils relient sont sur des sites différents, ils assurent alors le transport des messages d'un site à l'autre de manière homogène et transparente (une communication distante a la même interface qu'une communication locale).

Le noyau est composé d'un ensemble de processus s'exécutant en parallèle et coopérant par échange de messages. Ce choix nous permet d'utiliser au mieux les facilités d'expression du langage OCCAM (tout en étant portable sur des systèmes ne possédant pas le langage OCCAM : UNIX BSD 4.2, iRMX86, César et Cléopatre pour ne citer que les systèmes sur lesquels CONKER a été porté).

Ce choix d'implémentation permet :

- de rendre la réalisation de chaque fonction indépendante des autres (il suffit de fixer la sémantique des messages échangés lors des coopérations)
- permet de rendre modulaire l'implémentation (une fonction ne sera réalisée que si elle est nécessaire)
- permet de rendre le noyau extensible (une nouvelle fonction pourra être insérée dès qu'elle deviendra nécessaire)

4.1 Communication entre entités

Le noyau de CONKER réalise, comme dans le modèle CSP, un **rendez-vous entre deux ou plusieurs entités** qui désirent communiquer. Si le rendez-vous n'est pas possible (i. e. une des entités n'est pas prête), le noyau suspend les activités participant au rendez-vous qui sont prêtes.

Dans le cas d'une implémentation répartie du noyau de CONKER, les seules entités réparties sont les connecteurs. Si un connecteur relie deux processus situés sur deux sites différents, un morceau du connecteur s'exécute sur chacun des deux sites.

Ces connecteurs répartis utilisent les services d'un serveur d'accès au réseau d'interconnexion et doivent résoudre les problèmes liés aux pannes de sites (pertes de messages, protocoles de reprise). Ils seront étudiés en détail dans le chapitre 4 suivant au paragraphe 1.4.

Communication non-symétrique

Dans CONKER, la communication est unidirectionnelle et de type N-1. Elle s'effectue d'un processus gardé émetteur[7] vers un processus gardé récepteur. Mais

7. *Chaque entité de CONKER est composée d'un ensemble de processus gardé, partageant des données. Un processus gardé de CONKER a la même structure qu'un processus gardé du langage OCCAM. Il est activé uniquement par la réception d'un message reçu sur son canal associé. Ce canal peut être dans deux états : ouvert et le processus gardé associé peut recevoir des messages ; fermé et le processus gardé ne peut pas recevoir de messages et l'émetteur est suspendu jusqu'à l'ouverture du*

celui-ci peut recevoir sur son canal associé des messages émis par plusieurs processus.

Pour qu'une communication puisse avoir lieu, il est nécessaire qu'une entité, par l'intermédiaire d'un de ses processus gardé, prenne l'initiative de l'échange en désirant émettre un message. Le processus gardé destinataire du message doit nécessairement être en attente sur la résolution de sa garde d'activation. Dès que cette garde est choisie par le noyau, le processus gardé destinataire devient éligible et la communication a lieu.

La communication entre deux processus gardés peut se faire selon deux modes qui imposent deux modes de synchronisation (ils sont schématisés dans la figure 7 suivante) :

- **envoi simple.**
- **appel procédural** (ou invocation, ou mode question/réponse).

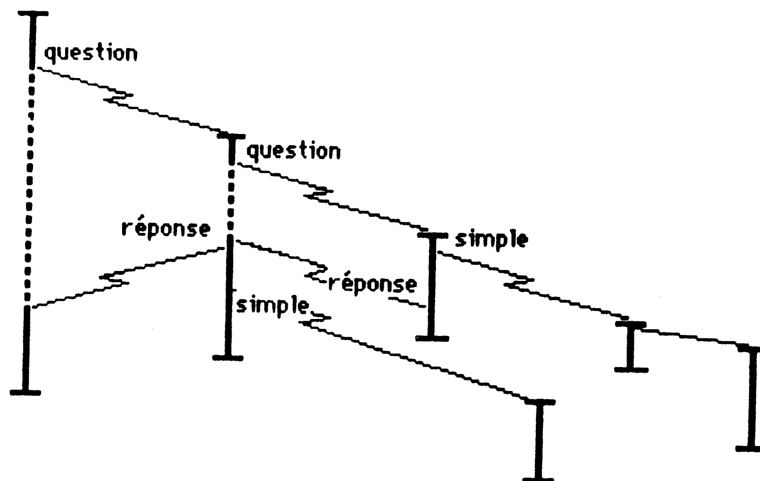


Figure 7. — Schéma d'exécution d'une application.

CONKER permet de suspendre l'exécution d'un processus gardé jusqu'à la terminaison d'un autre : ce mode de synchronisation est réalisé par une communication qui ne véhicule qu'un signal. Lors de sa terminaison, un processus gardé diffuse à l'ensemble des processus gardés qui attendent cet événement (diffusion réalisée par le noyau suite à la réception du message de terminaison).

Remarque : dans CONKER, les communications n'étant pas symétriques, il est possible qu'il y ait plusieurs messages en provenance de différents processus gardés et à destination du même canal (par exemple : plusieurs processus veulent écrire sur le

---

canal.

même connecteur). Le rendez-vous devant se réaliser entre un processus gardé émetteur et un processus gardé récepteur, le noyau choisit alors le couple de processus gardés entre lesquels s'effectue le rendez-vous (choix réalisé par le processus de séquencement implémenté).

Pour éviter qu'une entité puisse être bloquée indéfiniment sur une communication, **l'émission d'un message est temporisée.** Ceci permet de spécifier qu'une entité désire émettre un message avant telle date. Si le message n'a pu être émis, alors l'émetteur est réactivé et il a connaissance de la non-communication.

Le noyau CONKER réalise des **communications anonymes** : le nom de l'émetteur du message n'est pas automatiquement fourni à l'entité réceptrice du message. En particulier cela signifie qu'une entité ne peut **en aucun cas accepter ou refuser de recevoir** un message en fonction des appelants éventuels. Les communications par question-réponse, qui nécessitent la connaissance par l'appelé du nom de l'appelant sont néanmoins possibles (il suffit d'insérer le nom de l'appelant dans le message transmis ou d'activer le processus gardé destinataire en mode question).

Envoi d'un message : SEND

Pour communiquer, soit avec le noyau (ouverture ou fermeture de canal), soit avec un autre processus gardé, une entité utilise une primitive du noyau : SEND, qui réalise l'émission d'un message du canal du processus gardé émetteur vers le canal destinataire.

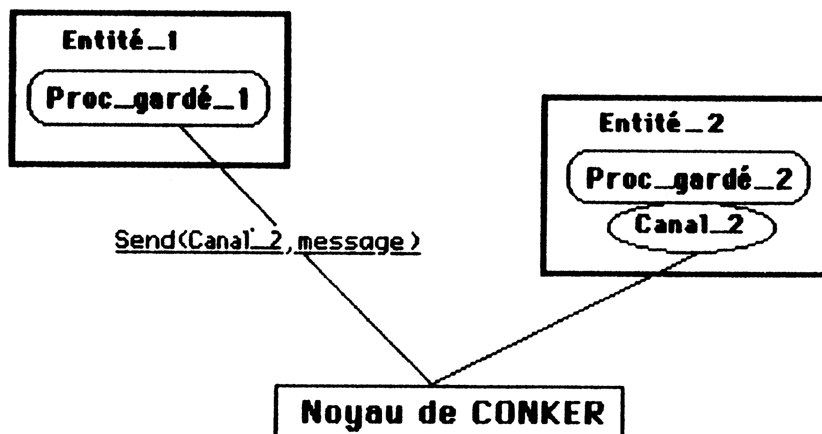


Figure 8. — Primitive de communication.

Lors de l'exécution de cette primitive, le noyau suspend l'entité émettrice et assure le passage du contrôle de l'exécution à une autre entité. Si le processus gardé destinataire est prêt (i. e. il est en attente sur la résolution de sa garde d'activation), le noyau choisira de l'activer afin de réaliser le rendez-vous entre les deux processus gardés. Nous donnons ci-dessous à titre d'exemple le code en OCCAM de la primitive SEND (celui-ci est introduit de façon systématique dans la phase de transformation du

langage OCCAM étendu en langage OCCAM).

```
PROC SEND ( CHAN in, out,
  VALUE canal, type, temporisation,
  VAR message, code_retour ) =
  SEQ
    out ! canal ; type ; temporisation ; message
    IF
      type = question
        in ? message ; code_retour
      type = terminaison
        SKIP
      TRUE
        in ? code_retour :
```

Les différents paramètres de cette commande sont :

- **in** : canal d'entrée associé au processus gardé qui exécute la commande.
- **out** : canal de sortie associé au processus gardé qui exécute la commande.
- **canal** : identification du processus gardé destinataire du message. Dans la réalisation à l'aide du langage OCCAM, chaque processus gardé est connu par l'intermédiaire de son canal associé.
- **type** : ce paramètre permet de spécifier le type de synchronisation qu'effectue le message.
    - **simple** : le processus gardé appelant désire activer un autre processus gardé.
    - **question** : le processus gardé appelant désire effectuer un appel procédural.
    - **réponse** : le processus gardé appelant précise que ce message est destiné au processus gardé qui a effectué l'appel procédural.
    - **synchronisation** : le processus gardé appelant désire se synchroniser sur la terminaison d'un autre processus gardé appartenant à la même entité.
    - **terminaison** : le processus gardé appelant signifie au noyau qu'il a terminé son exécution.
- **temporisation** : temps que le processus gardé désire attendre avant que la communication soit établie ou que le service demandé soit résolu (selon le type du message).
- **message** : texte du message à transmettre au processus gardé destinataire (ce texte peut être nul et le message est équivalent à un signal : analogie avec le ANY d'OCCAM).

4.2 Protection et confidentialité

Dans CONKER, chaque entité du système assure (ou n'assure pas) un minimum de protection. En effet, celle-ci est à la charge du programmeur. Cependant, le noyau, assure de façon systématique quelques contrôles.

4.2.1 Contrôles effectués par le noyau

Le domaine de protection minimal étant l'entité, le noyau effectue des contrôles uniquement lors des échanges de messages.

Contrôle de la légalité des activations

Afin qu'une entité n'active pas un processus gardé alors qu'elle n'en possède pas le droit, le noyau gère l'ensemble des connexions et assure une vérification pour toutes les demandes. Pour cela le noyau utilise pour chaque entité une table des liens. Lors de la création de l'entité, cette table est initialisée à l'ensemble des canaux auxquels l'entité peut accéder. Puis, pour les processus de traitement, cette table est mise à jour lors de chaque liaison avec un connecteur. Pour les autres entités cette table est statique.

Contrôle des modes d'activation

Un autre type de contrôle concerne le mode d'activation des processus gardés. Un processus gardé peut être activé soit sur le mode question (et son exécution devra envoyer une réponse au processus gardé émetteur), soit sur le mode simple. Mais en aucun cas, le mode d'activation du processus gardé évolue.

Lors de la phase de programmation de l'application, le concepteur spécifie le mode d'activation de chaque processus gardé. Il a le choix entre le mode simple (qui permet de lancer une autre activité en parallèle), le mode question (qui permet de réaliser l'appel de procédure) et le mode indifférent. Lors de chaque activation, le noyau contrôle que le mode choisi correspond à celui défini par le concepteur.

Le mode indifférent laisse le choix au processus qui envoie le message, de préciser au dernier moment s'il choisi le mode simple ou le mode question. Par exemple, une entité pourra choisir de fermer (ou ouvrir) un canal associé à un processus gardé soit sur le mode simple, soit sur le mode question.

Information mise à jour par le noyau

Le noyau possède une table des canaux qui lui permet de connaître :

- l'état de chaque canal (ouvert ou fermé).
- le processus gardé associé à chaque canal et son mode d'activation.
- l'identification du canal vers lequel doit être envoyée la réponse dans le cas d'un processus gardé s'exécutant sous le mode question. Si un processus termine sans envoyer le message de réponse, le noyau produit un message d'erreur à destination du processus gardé appelant.

Les règles de connexions des processus gardés sont :

- un **processus** ne peut demander l'exécution des processus gardés propres à un connecteur que s'il s'est auparavant lié à ce connecteur pour ces processus gardés.

- un **connecteur** ne peut demander l'exécution d'un processus gardé appartenant à un processus que si celui-ci en a auparavant demandé le traitement. Il peut s'agir : soit d'une demande de traitement asynchrone faite par un processus avec le mode de communication simple (mécanisme d'interpellation par message), soit de la poursuite du processus gardé qui a fait une demande avec le mode de communication réponse (réponse à une demande de lecture par exemple).

- Un **serveur système** peut demander la poursuite du processus gardé qui a fait la demande avec le mode de communication réponse, mais il ne peut pas être à l'initiative d'un échange.

4.2.2 Contrôle effectué par chaque type d'entité

Les **serveurs système** assurent un contrôle dynamique sur la validité d'une requête. Le contrôle effectué dépend étroitement de la réalisation du serveur. Il est possible, par exemple, qu'en phase de mise au point d'une application, les serveurs assurent plus de contrôle qu'en phase d'utilisation. L'émetteur reçoit en cas d'erreur un descriptif qui lui permet de savoir les causes du refus.

Par contre **pour les connecteurs ou les processus,** rien n'est prévu de façon automatique et l'utilisateur peut définir un système de contrôle spécifique à l'application. Ce système de contrôle peut être plus ou moins renforcé selon que l'on est en phase de test ou en phase d'utilisation (vérification de la structure et de la sémantique des messages, contrôle effectué lors de la réception ou de l'émission des messages).

4.3 Utilisation de CONKER dans un environnement réparti

La désignation des objets en environnement réparti est un problème qui a déjà été abordé de nombreuses fois. Nous n'allons pas ici faire l'inventaire des différentes options qui ont pu être prises pour réaliser ces mécanismes, de nombreuses thèses (par exemple celle de C. Senay [SENAY83]) ou études ayant déjà été publiées. Nous prendrons seulement quelques exemples types et nous renverrons le lecteur aux études les définissant.

4.3.1 Serveurs systèmes

Sur chaque site où le service peut être rendu un serveur système est présent. Par exemple, si l'on regarde le service d'impression, il est tout à fait vraisemblable que le système ne possède qu'une seule imprimante. Par contre, il peut être utile (ou nécessaire) de pouvoir imprimer des fichiers à partir de n'importe quel site. Pour réaliser cela, CONKER propose la démarche suivante :

- sur chaque site où le service peut être réalisé un serveur système réalisera directement le service quel que soit le site demandeur du service.

- sur chaque site où le service peut être rendu mais pas réalisé, un serveur système renverra la demande sur un site qui pourra réaliser ce service. Le choix de ce site (s'il est multiple) est à la charge du concepteur du système CONKER.

Pour illustrer cette démarche, reprenons l'exemple du service d'impression. Soit un système réparti qui possède une seule imprimante. Ce service sera effectivement réaliser sur un seul des sites du système : celui qui a comme périphérique l'imprimante. Sur tous les autres sites, un serveur système de CONKER renverra au serveur du site d'impression les requêtes qui lui sont adressées.

Si ce système possède au moins deux imprimantes de qualité équivalente, le service d'impression sera effectivement rendu sur chacun des sites qui possèdent une imprimante comme périphérique. Pour les autres sites, le serveur d'impression choisira le site d'impression en fonction de différents critères : moindre charge, disponibilité, etc... Une fois ce choix effectué, il enverra la requête au site élu et retournera à l'entité l'identification du site élu.

Pour réaliser ce mécanisme de délégation, CONKER offre un mode de désignation par des noms fonctionnels et chaque processus de l'application ou chaque connecteur, ayant besoin de ce service, s'adressera au serveur local par l'intermédiaire de ce nom.

Chaque serveur système appartenant à la même classe de service connaît tous les serveurs de sa classe afin de pouvoir coopérer ou de pouvoir rendre ce service par délégation.

Pour CONKER, un serveur système possède deux noms : un nom fonctionnel connu par les processus de l'application, les connecteurs et les serveurs systèmes appartenant à une autre classe de service et un nom global spécifique connu par les serveurs systèmes appartenant à la même classe de service et par le noyau. CONKER gère la correspondance entre le nom fonctionnel d'un serveur système et le nom global de ce serveur utilisé par les autres serveurs appartenant à la même classe de service.

4.3.2 Modèle

Les modèles de connecteurs ou de processus sont connus par CONKER sous la forme d'un nom global. CONKER maintient dans une table le lien entre le nom global et le site où est stocké le modèle.

Les noms globaux sont des chaines de caractères et le noyau vérifie, à chaque création modèle, l'unicité de ces noms dans le système.

4.3.3 Connecteurs

Un connecteur de CONKER est constitué d'un ensemble de morceaux de connecteurs représentant le connecteur. Chaque morceau possède un nom local sur le site sur lequel il se trouve (et un nom global formé de la concaténation du nom de site et du nom local sur ce site) et le connecteur un nom fonctionnel. CONKER gère la correspondance entre le nom fonctionnel et l'ensemble des noms globaux.

Pour créer une connexion avec un connecteur, un processus s'adresse au gestionnaire de connecteur local et désigne le connecteur par un nom fonctionnel (chaîne de caractères).

Ce gestionnaire de connecteur crée sur le site (si c'est nécessaire) du processus de l'application demandeur un morceau de ce connecteur. Puis il retourne au processus demandeur le nom local de ce connecteur et au connecteur la liste des noms globaux de chaque morceau de ce connecteur. Chaque gestionnaire de connecteur où est présent ce connecteur envoie à ce morceau de connecteur le nom global du morceau nouvellement créé.

Par la suite le processus de l'application s'adresse directement au morceau local du connecteur en utilisant son nom local.

Note : dans le cas ou le connecteur n'est pas réparti, il est constitué d'un seul morceau et ne possède, par conséquent, qu'un nom fonctionnel et un nom local.

Les connecteurs répartis

Dans l'architecture CONKER, un connecteur réparti est constitué d'un ensemble de connecteurs présents sur chacun des sites que le connecteur réparti relie. Chacun des morceaux du connecteur peut offrir les mêmes services que le connecteur réparti, pour cela l'ensemble des morceaux du connecteur coopèrent par échange de messages.

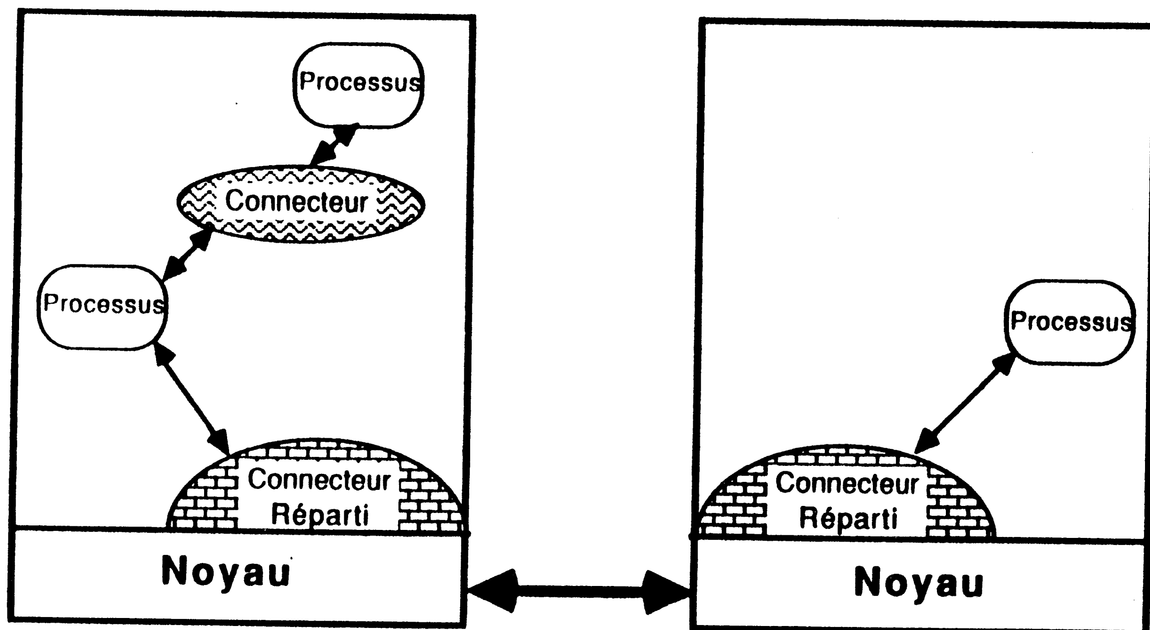


Figure 9. — Connecteurs répartis.

Pour pouvoir s'exécuter, un connecteur réparti utilise les services offerts par une entité système de CONKER qui permet l'échange de messages entre différents sites indépendamment de l'architecture du système d'interconnexion.

Hypothèse : chaque morceau d'un connecteur réparti connaît les autres morceaux. Cette hypothèse nous semble réaliste si les connecteurs sont relativement statiques (i. e. ils s'exécutent presque toujours sur le même ensemble de sites)

4.3.4 Processus de l'application

Chaque processus de l'application possède un nom local qui est le nom de l'instance.

4.3.5 Gestion des noms

Il n'a pas été nécessaire de développer pour CONKER des modes de désignation sophistiqués. En effet, chaque processus n'effectue que des communications locales et ne peut pas migrer.

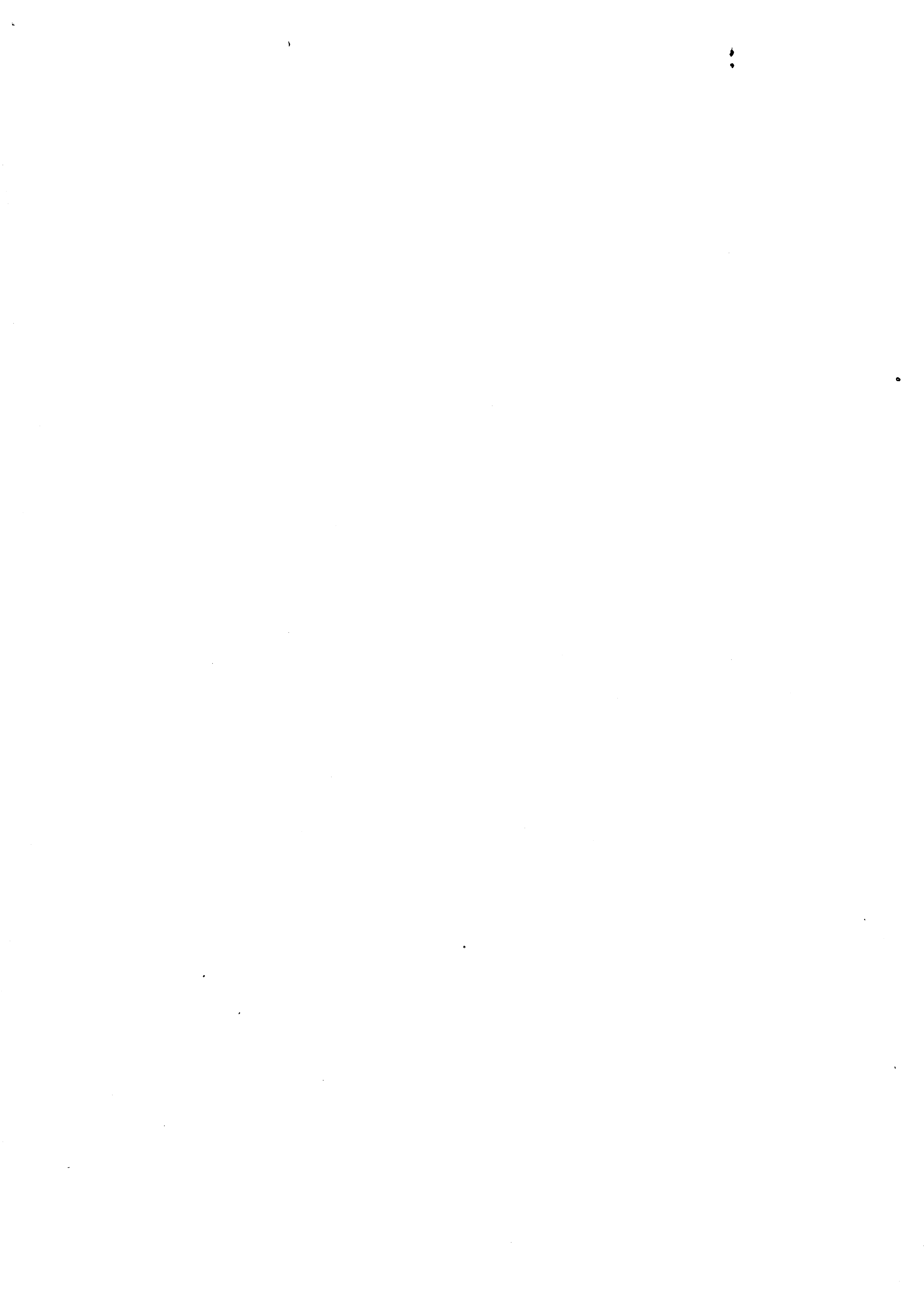
Les seules communications entre sites sont le fait de communications entre des parties d'un connecteur réparti ou entre serveurs systèmes de même type et dans tous les cas chacune des entités connaît toutes les autres.

Le format du nom local dépendra du site (et en particulier du système de traitement hôte) sur lequel s'exécutera le serveur.

Les noms locaux sont uniques. Ceci est réalisé à l'aide des systèmes de traitement hôtes qui offrent généralement une possibilité de désignation pour les objets du système. Par exemple : iRMX86 offre des "TOKENs" (estampilles délivrés par un séquenceur d'iRMX86), UNIX donne un numéro à chaque processus (pid).

CONKER donne pour nom local, le nom donné par le système de traitement hôte pour les objets présents sur le système de traitement. Quand le système n'offre pas un système de numérotation, par exemple OCCAM, CONKER peut utiliser une fonction de génération de nom (estampille de Lamport par exemple).

Une numérotation unique des noms de sites garantit l'unicité des **noms globaux** qui sont constitués de la concaténation du numéro de site et du nom local.



IV. CONKER : description du système

Résumé

Après avoir présenté dans les pages précédentes le modèle CONKER, nous allons dans ce chapitre définir le système CONKER.

Ce système sera généralement implémenté à l'aide d'un système de traitement hôte. Il permettra l'exécution d'une application décrite selon le modèle CONKER.

La première partie de ce chapitre permettra de définir l'architecture du système CONKER et la façon dont sont structurées les différents types d'entité : serveur système, connecteur, processus.

La seconde partie définit l'ensemble des primitives à implanter, si le système est réalisé sur un système de traitement hôte et si le langage de programmation de l'application ne permet pas la programmation parallèle : communication par envoi de message, gestion de processus, ...

1. Architecture du système CONKER

Le système CONKER comporte un noyau dont les grandes lignes ont été tracées dans le chapitre précédent, et un ensemble d'entités qui réalisent les traitements désirés par l'application.

Dans ce paragraphe, nous présenterons une structure d'implémentation pour le noyau. Puis nous définiront les éléments qui composent une entité de CONKER (structure, contexte, ...). Et nous terminerons par la structure :

- d'un serveur du système CONKER.

- d'un processus de l'application.

- d'un connecteur.

1.1 Structure d'implémentation du noyau

Rappel : le noyau de communication CONKER s'appuie dans l'implantation actuelle sur les services d'un système de traitement hôte, qui peut être centralisé sur un mono-processeur (UNIX [RITCH74], iRMX86 [INTEL82], CHORUS [GUILL82]) ou réparti (CHORUS sur SM90 multi-processeurs [ZIMME84]), de manière à offrir aux processus de CONKER des mécanismes de communication appropriés.

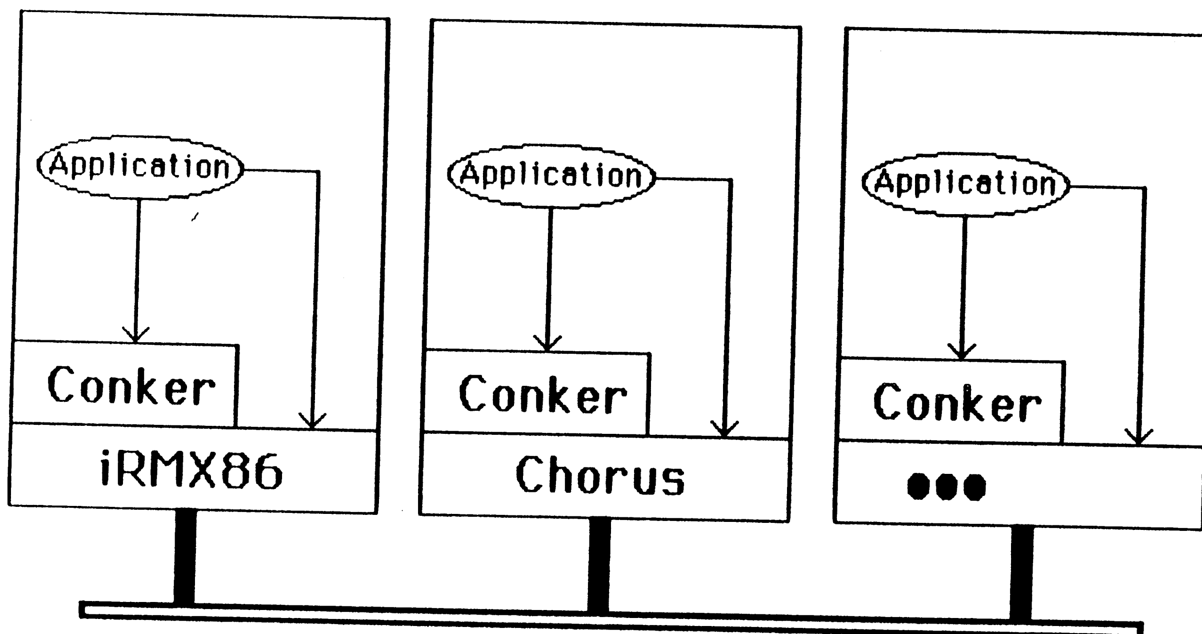


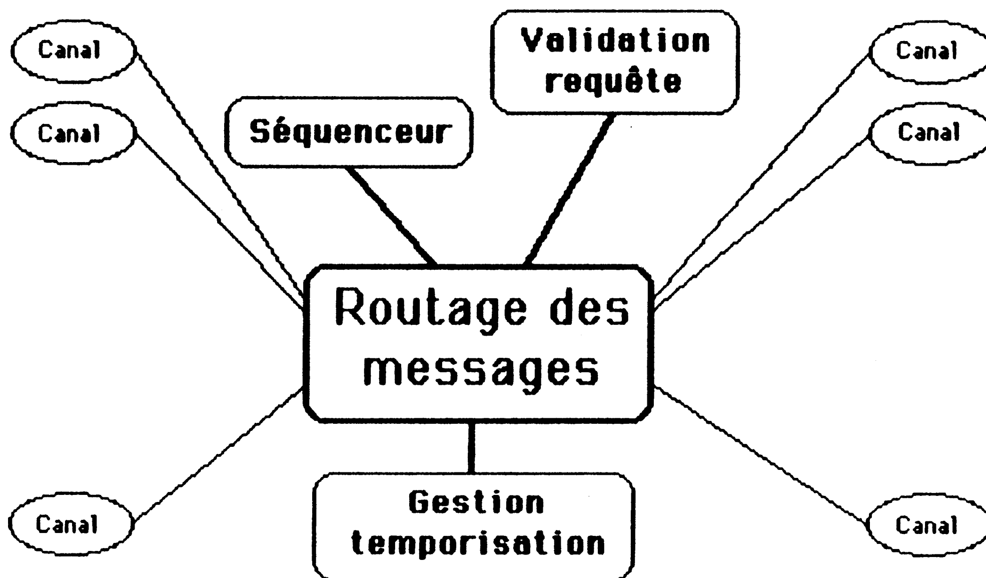
Figure 1. — CONKER : couche de communication homogène.

Remarque : CONKER pourrait aussi être développé directement sur une machine nue. Mais la partie qui nous intéresse dans ce travail étant la construction d'un noyau de communication homogène pour systèmes répartis hétérogènes et la définition de primitives

de communication agréables à utiliser dans un langage parallèle pour applications réparties, nous n'avons pas adopté cette approche.

L'implémentation du noyau de CONKER peut être réalisée par un ensemble de processus, (voir figure ci-après) qui s'exécutent de façon parallèle et qui communiquent entre eux pour réaliser les différents services offerts par le noyau :

- le **séquencement** est réalisé par un processus qui est programmable par l'utilisateur, afin de lui laisser le soin de choisir le mode d'allocation du processeur le mieux approprié à l'application choisie (avec ou sans préemption, avec ou sans priorité, ...). CONKER connait un séquencement par défaut : sans préemption, sans priorité.
- une autre fonction du noyau est le **routage local des messages** entre un canal émetteur (canal de sortie associé à un processus gardé) et un canal destinataire (canal d'entrée associé à un processus gardé) afin de pouvoir multiplexer les communications (sur chaque canal d'entrée, il y a N émetteurs potentiels et un seul récepteur).
- la **vérification de la validité des requêtes** est assurée par un autre processus à partir de la **table des connexions**[8] qui permet la connaissance des liens entre les processus gardés et leurs modes d'activation (envoi simple ou question/réponse).
- la **gestion des temporisations.**



8. *Cette table permet de connaitre l'ensemble des connexions réalisées entre les entités. La description de la table des connexions est donnée dans le chapitre suivant.*

1.2 Structure d'implémentation d'une entité CONKER

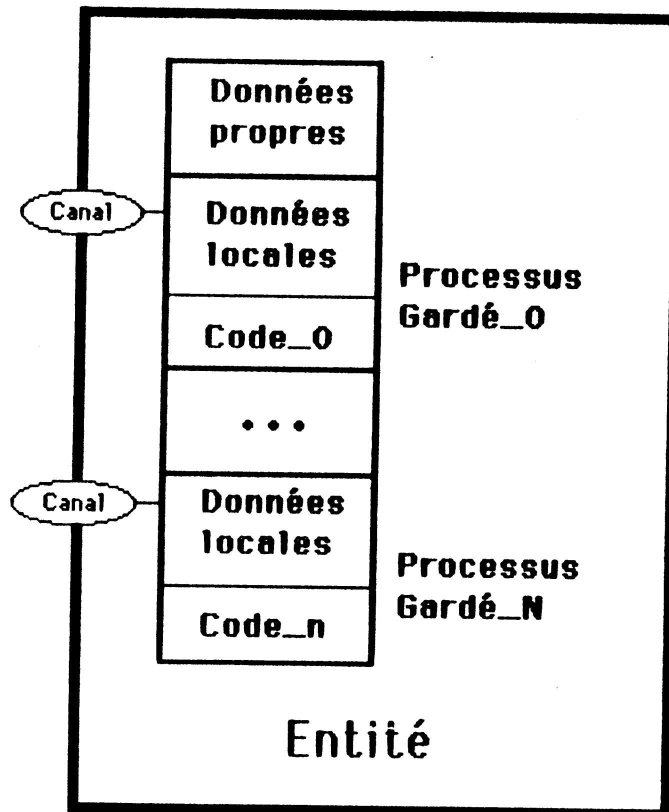
Chaque entité du système CONKER constitue une unité de découpage fonctionnel, une unité de protection (par encapsulation des données) et une unité d'exécution (chaque unité est autonome) de l'application.

Toute entité de CONKER est composée d'un seul processus (au sens OCCAM), formé lui-même d'un ensemble de processus gardés qui s'exécutent de façon alternative.

Une entité peut s'implémenter en OCCAM de la façon suivante :

```
VAR données propres :
WHILE condition d'arrêt
  ALT
    canal_0 ? message
      VAR données locales :
      code_0
    ...
    canal_n ? message
      VAR données locales :
      code_n
```

L'ensemble **canal_i ; données locales et code_i** forme le processus gardé_i (gardé par l'arrivée d'un message sur le canal_i associé).



1.2.1 Variables globales

Chaque entité encapsule, pour des raisons d'implémentation, un certain nombre de données propres, inaccessibles de l'extérieur, mais qui peuvent être partagées par les processus gardés qui la composent. Nous suivons en cela les règles de partage de variables utilisées dans le langage OCCAM entre processus gardés, ou le mécanisme de partage des variables par les différents processus parallèles implémentant une ressource dans le langage SR [ANDRE81] dans le but d'alléger l'écriture des programmes.

En effet, puisqu'une entité de CONKER ne s'exécute que sur un seul site, son exécution peut être considérée comme l'exécution séquentielle des différents processus gardés qui la composent. Il n'était pas utile d'imposer la réalisation des variables globales par des "processus secrétaires" capables de donner leurs valeurs aux différentes variables à implémenter.

Par exemple, nous utilisons cette possibilité de mémoire partagée pour permettre à deux processus gardés complémentaires, d'un connecteur "tampon" (lecture et écriture), d'utiliser la même file pour mémoriser les messages.

Les processus gardés d'une même entité sont en concurrence pour l'accès aux variables globales de l'entité. Toutefois l'accès en mutuelle exclusion est contrôlé par le mécanisme d'activation des processus gardés.

Outre ses variables globales, chaque entité de CONKER possède un ensemble de **variables d'état** qui la caractérise :

- chaque entité de CONKER possède une **priorité d'exécution.** Le noyau étant particulièrement dédié, dans sa définition actuelle, à des applications dans lesquelles des contraintes de temps peuvent apparaître, il est essentiel de pouvoir choisir les entités à exécuter de façon prioritaire à celles dont l'exécution peut être différée (nous rejoignons en cela le constructeur PRI PAR d'OCCAM qui permet, pour une implémentation parallèle de différents processus, de fixer des priorités relatives entre eux).
- l'état de chacune des gardes associées aux processus gardés de l'entité.

1.2.2 Processus gardés

Les processus gardés associés à une entité s'exécutent de façon alternative. Ils peuvent être spécifiques à l'entité ou communs à plusieurs entités locales (chaque entité utilise alors une instance de ce processus gardé) mais leur **activation est toujours conditionnée,** comme dans le constructeur ALTernatif d'OCCAM, par **l'arrivée d'un message.**

L'enchaînement des activités exécutées par le système CONKER dépend étroitement des messages échangés entre les activités, et du type de synchronisation réalisé par ces messages. Actuellement CONKER reconnait les types de connexion suivants :

i) Activation simple : avec ce type d'échange, un processus gardé, émetteur du message, active un processus gardé appartenant à une autre entité. Une fois la communication réalisée, les deux processus gardés poursuivent leur exécution en parallèle.

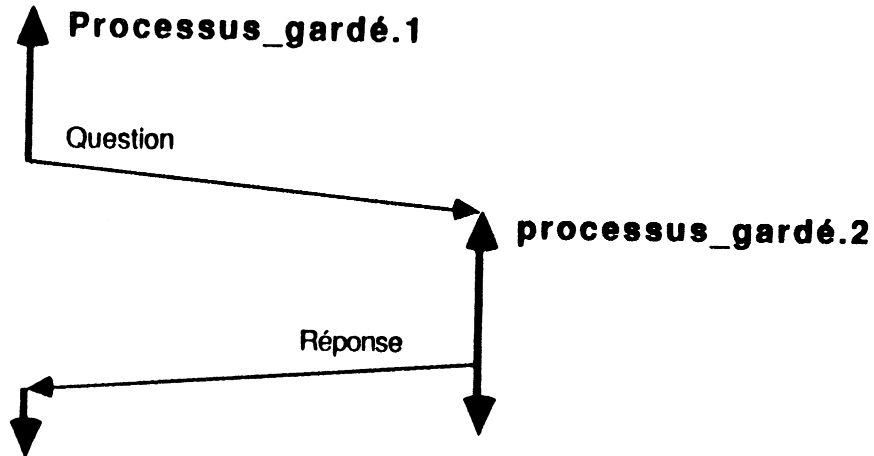


Figure 4. — Activation simple.

ii) Appel procédural : le processus gardé appelant demande l'exécution d'un service à un processus gardé appartenant à une entité et attend de sa part un message de réponse.

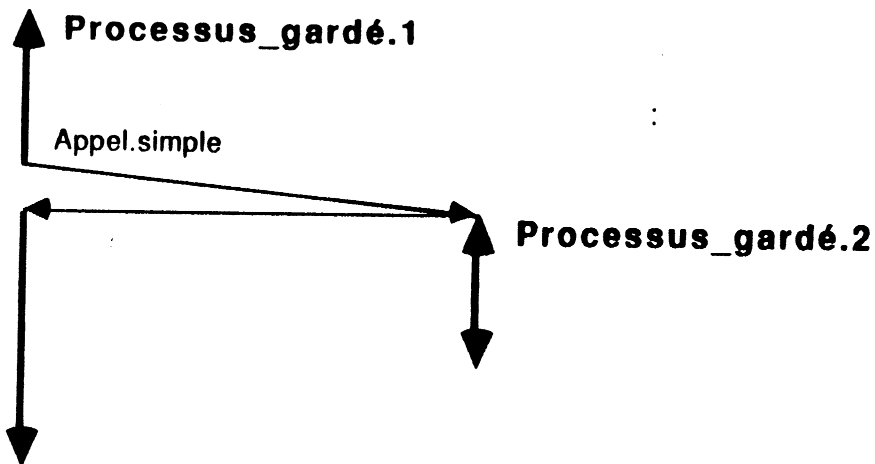


Figure 5. — Appel procédural.

iii) Synchronisation sur terminaison : contrairement aux deux précédents types de communication, le message émis par le processus gardé appelant est destiné au noyau. Par ce message, le processus gardé émetteur désire reprendre son exécution quand le processus gardé spécifié termine. Si le processus gardé spécifié ne termine pas, le processus gardé

émetteur reste bloqué indéfiniment.

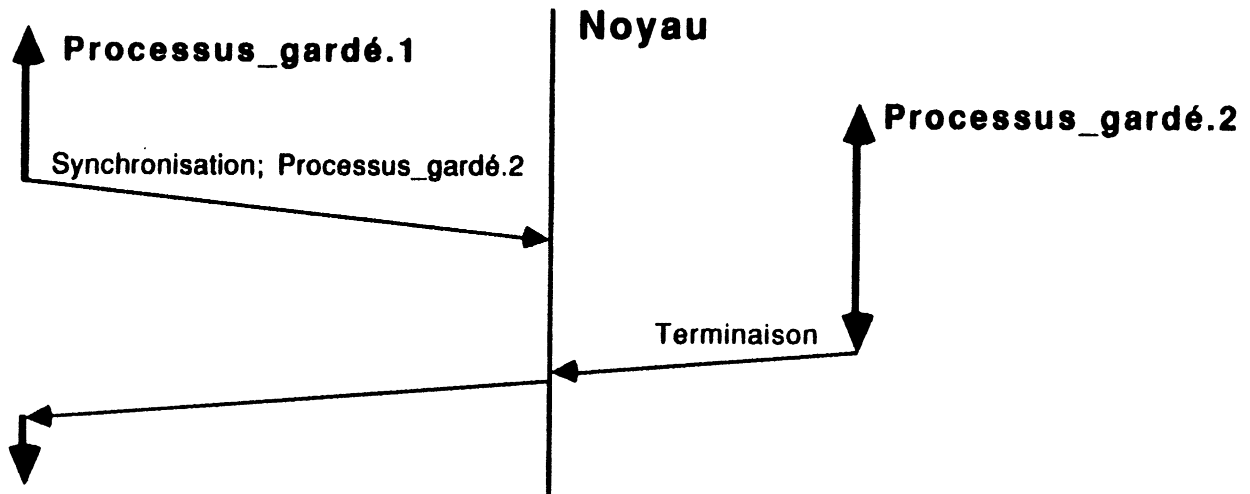


Figure 6. — Synchronisation sur terminaison.

Chaque terminaison est implémentée dans CONKER comme une communication entre le processus gardé qui termine et le noyau. Sur l'arrivée de ce message, le noyau doit activer les processus gardés en attente de l'événement : *"terminaison de ce processus gardé"*.

1.2.3 Canal de communication

Pour communiquer avec le noyau, **chaque processus gardé appartenant à une entité possède deux canaux de communication** : un pour recevoir des messages et un autre pour pouvoir émettre (chaque canal de CONKER peut être réalisé par un canal du langage OCCAM à condition que celui-ci accepte les messages structurés). Seule la réception d'un message sur ce canal permet d'activer le processus gardé par ce canal (voir pour cela le constructeur ALTernatif du langage OCCAM).

CONKER associe à chaque canal une expression booléenne permettant de forcer l'évaluation de la garde à Faux. Nous dirons que le canal est :

- **ouvert** si l'expression booléenne associée s'évalue à Vrai. L'entité peut alors recevoir des messages sur ce canal de communication.

- **fermé** si l'expression booléenne associée s'évalue à Faux. L'entité refuse alors de recevoir des messages sur ce canal et le processus gardé correspondant ne peut pas être activé (la garde associée n'est jamais évaluée à vrai).

Pour respecter le modèle initial, où tout événement est perçu par le système comme une communication, le changement d'état d'un canal d'une entité sera réalisé uniquement par un échange de messages entre un processus gardé appartenant à la même entité et le noyau.

Un processus gardé ne peut pas changer l'état d'un canal associé à un processus gardé appartenant à une autre entité. En effet, les états des canaux seront considérés comme des

variables d'état de l'entité et ne pourront donc pas être partagés entre différentes entités.

1.2.4 Etat d'un processus gardé

Chaque processus gardé du système peut être dans différents états au cours de son exécution, le passage d'un état à un autre se fait uniquement en fonction des communications réalisées.

- **suspendu** : le processus gardé est bloqué sur la réalisation de sa garde d'activation.

- **éligible:** le processus gardé est prêt à s'exécuter et attend le processeur (sa garde associée s'évalue à vrai).

- **actif** : le processus gardé s'exécute. Il occupe le processeur et le libérera :

  - soit pour exécuter une communication. Il se retrouvera alors en attente (cf ci-dessous).

  - soit parce qu'il aura terminé son exécution (envoi du message de terminaison au noyau), il se retrouvera alors dans l'état suspendu.

  Note : un processus actif occupe le processeur tant qu'il ne désire pas communiquer. Si le processus boucle, l'application ne terminera pas.

- **attente** : le processus gardé désire réaliser un rendez-vous avec un autre processus gardé. Selon le type de synchronisation réalisé par le message émis, le processus gardé se retrouve dans l'un des trois états suivants :

  - **en attente sur synchronisation (synchronisation sur terminaison) :** synchonisation sur la terminaison d'un autre processus gardé appartenant à la même entité.
  - **en attente de communication simple (activation simple) :** synchronisation pour effectuer un rendez-vous avec un autre processus gardé.
  - **en attente de communication question/réponse (appel procédural) :** le premier état représente la prise en compte par l'autre processus gardé du message d'appel, et le second état, la réalisation du service demandé.

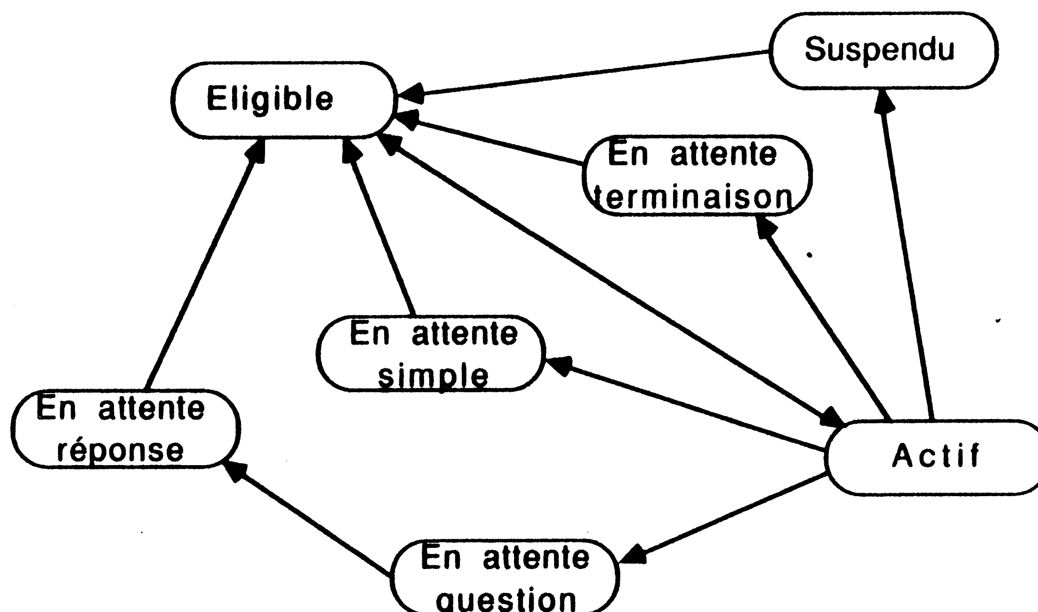


Figure 7. — Etats d'un processus gardé.

Note : un processus peut sortir de l'état actif pour se retrouver dans l'état eligible uniquement si l'environnement communique avec l'application. Cette entorse au modèle est rendue nécessaire par le fait que l'environnement ne peut ni attendre, ni répéter un événemenent.

1.3 Le noyau et les serveurs du système CONKER

Une des premières questions qui se pose au concepteur de systèmes (que ce soit de systèmes de traitement ou de systèmes de communication) est le choix des services de base (ou primitifs) que le noyau doit offrir, sachant qu'il faut faire un compromis entre des critères contradictoires : transportabilité, puissance, modularité, protection, ...

Sur chaque site, le noyau fournit aux entités présentes, en coopération avec le système de traitement hôte et si nécessaire avec ceux des autres sites, des services leur permettant de s'exécuter en parallèle :

- **contrôle de l'exécution des entités.** Comme on l'a vu précédemment, le changement d'état d'un processus gardé coïncide toujours avec une communication entre deux processus gardés ou avec l'arrivée d'un message externe rendant activable un processus gardé suspendu à l'arrivée de cet événement. Le noyau choisit alors le processus gardé activable à exécuter selon des critères de priorité qui peuvent être définis dans CONKER en fonction des besoins spécifiques des applications.

- **commutation des messages au niveau local.** Le transport des messages d'un processus gardé vers un autre est assuré par le noyau à l'aide des services offerts par le système de traitement hôte. Ce transfert de message s'effectuera sans perte, sans duplication et sans erreur, de manière à réaliser un rendez-vous entre les deux entités communicantes.

- **réception des interruptions** et activation du connecteur lié à ces interruptions. Une interruption ne déclenche pas directement un traitement, en effet cela serait contradictoire avec le modèle choisi dans lequel le seul événement observable par le système est un échange de messages entre deux entités. A chaque arrivée d'interruption le noyau active le connecteur associé. Celui-ci transforme le signal d'interruption en un message manipulable par les processus de l'application.

- **contrôle des canaux d'entrée/sortie** : une demande d'entrée/sortie se fait par l'intermédiaire d'un connecteur relié à un canal d'entrée/sortie, qui offre une interface de haut niveau (reprenant en cela les concepts du langage OCCAM où l'accès à un fichier se fait par un canal spécifique). Le noyau assure l'activation de ces connecteurs en fonction des interruptions survenant aux ports du système et des messages donnés par les processus.

Afin de réaliser un système de communication "sur mesure", adapté aussi bien au système de traitement hôte qu'à l'application, nous avons fait le choix d'implémenter CONKER, en essayant d'utiliser au mieux les possibilités offertes par le système de traitement hôte, et de n'offrir que les services nécessaires à l'application.

La définition du noyau, de même que son interface d'accès, reste cependant indépendante d'une architecture particulière (mono-processeur, multi-processeurs ou réseau) ou d'un système de traitement spécifique.

Ce noyau est extensible : l'ajout de nouvelles fonctionnalités se fait sous forme de serveurs systèmes spécifiques aux système de traitement hôte et présents sur le site si les services qu'ils réalisent sont nécessaires. Ces serveurs offrent une interface identique quel que soit le site sur lequel ils s'exécutent et permettent d'accéder aux systèmes de traitement hôtes.

Nous avons fait le choix d'un noyau minimal de façon à ce qu'il soit aisément transportable (peu de code à réécrire), facile à mettre au point (car petit) et par là même plus sûr.

1.3.1 Serveurs du système CONKER

Les serveurs du système CONKER, toujours locaux à un site, réalisent sur celui-ci des services habituels :

- création et destruction d'entités.
- gestion des connexions entre processus et canaux de communication.
- gestion de la mémoire (si nécessaire).

- gestion des noms.

Ces services ne sont généralement pas tous nécessaires sur l'ensemble des sites du système. Leur présence dépend du système de traitement du site hôte (peut-il offrir ce service?) et de l'application (est-il utile d'offrir ce service sur ce site?) ; seule la gestion dynamique des entités et la gestion dynamique des connexions sont nécessaires sur chaque site.

i) Serveur pour la gestion des processus

Ce serveur du système CONKER est chargé de la gestion dynamique des processus. Il crée, sur une demande d'un processus, une nouvelle instance d'un modèle de processus sur un site particulier.

Ce serveur, présent sur chaque site, ne peut créer des instances de processus que pour les modèles de processus connus sur ce site. En effet, le système CONKER pouvant s'exécuter sur un ensemble de sites hétérogènes, les modèles de processus ne peuvent s'exécuter que sur des sites ayant les mêmes caractéristiques et pour lesquels ils ont été prévus.

ii) Serveur pour la gestion des connecteurs

Ce serveur du système CONKER se charge de la gestion dynamique des connecteurs et de leurs connexions.

Sur une requête d'un processus, il crée, en liaison avec les gestionnaires des connecteurs des autres sites, une instance d'un modèle de connecteur. Ce connecteur sera réparti si le processus le désire et si le modèle le permet.

Puis, tant que ce connecteur est présent, le gestionnaire des connecteurs gère pour lui les liaisons ouvertes par les processus sur ce connecteur. Ceci permet d'assurer un contrôle sur les requêtes adressées par les processus et de contrôler éventuellement les liaisons.

iii) Serveur pour la construction de modèles

Ce serveur du système CONKER, appelé "constructeur", peut construire un nouveau modèle de processus ou de canal. Il peut, si c'est nécessaire, construire le modèle désiré avec l'aide des serveurs de même type des autres sites.

Ce serveur est spécifique à un site et ne peut construire ou supprimer des modèles que pour le site sur lequel il s'exécute. Si la demande concerne un autre site, le serveur local enverra la requête au site concerné.

iv) Serveur pour la gestion mémoire

Si c'est nécessaire, un serveur spécialisé est chargé d'allouer une partie de la mémoire aux autres entités. Ce serveur système est, bien entendu, local à un site, et ne gère que la mémoire locale, mais il peut effectuer cette gestion en lien avec les serveurs des autres sites de façon à équilibrer le taux d'occupation mémoire entre chaque site : un site pouvant servir de "trop plein" pour un autre site. Ce serveur, qui permet d'implémenter une notion de mémoire virtuelle globale au système, n'a pas

été réalisé actuellement.

Dans le cas d'une réalisation sur une machine multi-processeurs possédant de la mémoire commune : SM90 [FINGE82] ou CESAR [CARTO84a][9] ; un serveur de ce type gère dynamiquement l'attribution de la mémoire commune.

v) Serveur d'accès réseau

Dans le cas où le système est réparti (réseau local ou machine multi-processeurs), un serveur système joue le rôle d'interface avec la station de transport (accès à un réseau) ou assure le routage des messages (sur une architecture multi-processeurs) vers les autres processeurs.

Il permet de connecter les bouts d'un même connecteur présent sur plusieurs sites en implémentant entre ces morceaux un protocole d'échange spécifique : diffusion des messages vers tous les "morceaux" du connecteur ou émission du message vers un seul "morceau".

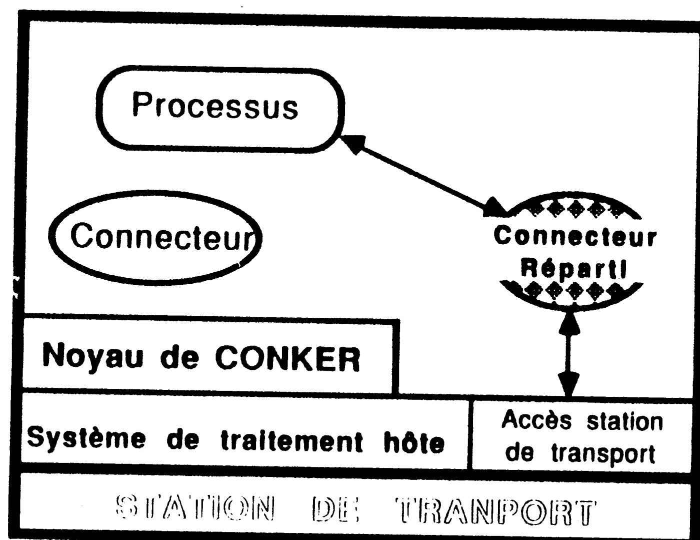


Figure 8. — Accès à un réseau.

1.4 Communication entre processus de l'application

Un connecteur pourra être relié, dans l'exemple de la figure 10 ci-dessous, à plusieurs processus de l'application. Ceux-ci utiliseront, pour communiquer, des primitives qui

9. *Le logiciel CESAR (Commande et Exploitation des Systèmes pour l'Automatique et la Robotique) mis au point par le CERT/DERA est un système d'exploitation multi-processeurs, multi-tâches. Il est conçu pour assurer des échanges très rapides d'informations et de signaux de synchronisation entre tâches. Ce logiciel est développé autour d'un ensemble de micro-processeurs 68000 connectés sur un bus VME.*

réaliseront une succession d'échanges de messages entre le processus appelant, le noyau et le connecteur désiré. Ces primitives permettent d'offrir une interface "agréable" au programmeur.

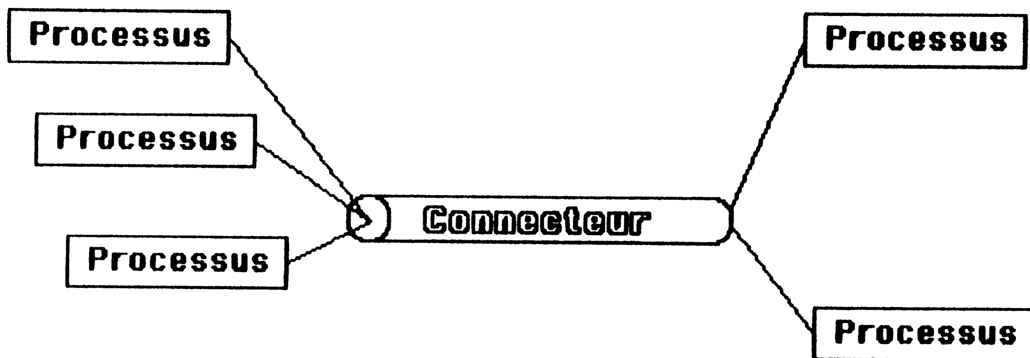


Figure 9. — Communication par connecteur.

Par la suite, nous noterons **lire (connecteur, message)** la primitive qui permet de recevoir le message courant disponible sur le connecteur, le texte du message est recopié dans la variable locale "message" du processus exécutant cette primitive et **écrire (connecteur, message)** la primitive qui permet d'émettre le message spécifié sur le connecteur. Ces deux primitives réalisent les processus d'entrée/sortie du langage OCCAM (**connecteur ? message et connecteur ! message**).

Mais outre la possiblité d'envoyer ou de recevoir des messages, les processus auront la possiblité de se synchroniser :

- sur la terminaison d'un ensemble de processus (réalisation par exemple du constructeur PARallèle d'OCCAM).
- sur un ensemble d'événements qui peuvent survenir sur les connecteurs (réalisation par exemple du constructeur ALTernatif d'OCCAM : l'événement attendu est alors vu comme une écriture).

Les processus peuvent aussi demander le traitement asynchrone des événements propres aux connecteurs. Lors de l'arrivée de l'événement sur le canal, celui-ci génère un message qui suspend l'exécution du processus demandeur et active le traitement désiré, celui-ci reprend son exécution dès que le traitement est terminé.

1.5 Connecteurs

Le connecteur de CONKER facilite la programmation d'applications où l'activité de communication est importante (voire primordiale) et où les synchronisations entre les processus peuvent être complexes. Le connecteur est chargé du transport des messages entre les processus selon le protocole d'échange spécifique qu'il réalise. Il assure par conséquent la synchronisation des différents processus qui lui sont connectés.

Ces connecteurs peuvent être répartis si l'architecture du système est répartie (réseau local ou machine multi-processeurs) et, si les processus qu'ils relient sont sur des sites distincts, ils permettent alors l'échange de messages entre les processus de sites différents.

La communication par connecteur supprime le problème de la désignation du processus destinataire. Lors d'une communication, un processus de l'application émet un message à destination d'un connecteur qui représente une classe de service : tous les processus qui lui sont connectés en lecture sont à même de comprendre et de traiter correctement ce message.

Dans CONKER, un processus de l'application communique toujours avec un morceau de connecteur local (qui s'exécute sur le même site que le processus). C'est celui-ci, par sa structure, qui véhicule les messages jusqu'aux processus destinataires.

i) Caractéristiques d'un connecteur

Les connecteurs seront créés à partir de modèles de connecteur. Chaque modèle pourra offrir plusieurs types de synchronisation dérivés car ils sont paramétrisables.

Par exemple : si on possède un modèle de connecteur qui représente la gestion d'une file FIFO de longueur bornée, avec une synchronisation entre le processus lecteur et le processus rédacteur sur des événements de type "connecteur plein" ou "connecteur vide", on peut réaliser des communications asynchrones si la longueur de la file est supérieure à zéro et une communication par rendez-vous si la longueur de la file est nulle.

Les paramètres de programmation du modèle de connecteur pourront être :

- **le nombre maximal de processus lecteurs et de processus écrivains** qui définissent les connexions possibles du connecteur.
- **la longueur de la file d'attente** qui définit le type de synchronisation entre les processus (asynchrone ou rendez-vous).
- la possibilité de **répartition ou non du modèle.**
- **les événements** que ce connecteur sera susceptible de signaler.

ii) Connecteurs prédéfinis

Il existe un certain nombre de connecteurs prédéfinis, déjà créés lors du lancement du système et ayant une fonctionnalité bien précise :

- les connecteurs chargés de récupérer les interruptions, et de les transformer en messages manipulables par le système.
- les connecteurs chargés de gérer les périphériques présents sur le site et d'offrir aux processus utilisateurs des primitives de communication homogènes avec celles utilisées pour communiquer entre deux processus.

Remarque : certains de ces connecteurs peuvent avoir des contraintes d'accès : par exemple, les connecteurs qui récupèrent les interruptions ne sont accessibles qu'en lecture par les processus de traitement.

iii) Connecteurs répartis

Un connecteur réparti sera présent sur chacun des sites où un processus lui est connecté. Sur chaque site réside un morceau du connecteur, connecté à un serveur système d'accès au réseau, lui permettant de coopérer avec les autres morceaux du connecteur afin de réaliser le mode d'échange souhaité.

1.6 Programmation des processus gardés

Dans ce chapitre, nous présenterons successivement les différents processus gardés présents dans chaque entité, puis ceux spécifiques aux connecteurs et ceux spécifiques aux processus.

1.6.1 Processus gardés communs

Toutes les entités de CONKER possèdent deux processus gardés qui permettent l'initialisation des variables globales et la gestion des erreurs de communication.

i) Initialisation

Ce processus gardé est spécifique à chaque modèle d'entité. Il permet la configuration du modèle et l'initialisation des variables globales de l'entité.

Il peut aussi être réactivé chaque fois qu'il est nécessaire d'initialiser les variables globales.

ii) Traitement des erreurs issues de la communication

Chaque entité possède un processus gardé activé directement par le noyau chaque fois qu'une erreur de communication le concernant est détectée. Ces erreurs peuvent être :

- **permanentes** : une erreur de ce type est consécutive à un mauvais fonctionnement ou à une action extérieure qui **interdisent la continuation de l'application**. Nous pouvons citer entre autres : une erreur dans le code, une erreur d'un équipement périphérique, un arrêt programmé, une violation des droits d'accès...

- **temporaires** : au contraire du type d'erreur précédent, une erreur de ce type est susceptible de ne pas se reproduire si l'application est réexécutée. Nous pouvons citer le cas du délai de communication dépassé parce que le système a une surcharge temporaire (si l'architecture du système a été correctement choisie cet événement critique doit être peu fréquent mais il peut être nécessaire de pouvoir le traiter de façon prioritaire).

Ce processus gardé peut aussi traiter tout autre type d'erreur, liée directement à la programmation de l'application : processus destinataire inexistant, syntaxe de message incorrecte, ...

Ces processus gardés de reprise d'erreur permettent au programmeur de traiter, de la façon qu'il juge la plus efficace pour l'application, certains types d'erreurs liés à la communication.

Le système CONKER permet la construction d'un ensemble de processus gardés de traitement d'erreur permettant de résoudre différents types d'erreur. Cela permet, lors de l'écriture d'une application de pouvoir choisir le processus gardé de traitement d'erreur le plus approprié à chaque entité.

Note : Le lien entre l'entité et le processus de traitement associé est effectué lors de l'étape de création du modèle d'entité ou lors de la création de l'entité (si le modèle permet les connexions dynamiques de processus gardé).

1.6.2 Connecteurs

Les entités de ce type possèdent un ensemble de processus gardés qui s'exécutent de façon alternative. A chaque processus gardé correspond un service offert par le connecteur : écriture, lecture, traitement d'événement (mécanisme d'interpellation par messages), éventuellement accès à un réseau.

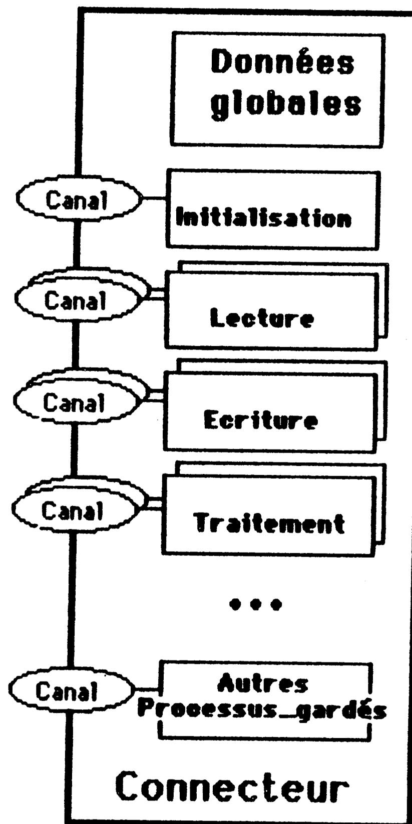


Figure 10. — Architecture d'un connecteur.

```
PROC connecteur ( CHAN in[], out[], suspend ) =
 VAR données_globales :
 ENV
  suspend ? ANY
   suspend ? ANY
  WHILE TRUE
   ALT canal = [ 0 FOR nbre_proc_gardé ]
    ouvert[ canal ] & in[ canal ] ? message
      processus_gardé ( message ) :
```

1.6.2.1 Programmation d'un processus gardé pour un connecteur

Pour une entité de type connecteur, un processus gardé sera toujours une étape de traitement activée à la suite d'un envoi d'un message. Au cours de son exécution, un processus gardé peut seulement communiquer par envoi de message, ce qui lui permet de réaliser des protocoles de type émission simple, question/réponse ou synchronisation sur la terminaison d'un autre processus gardé.

Exemple :

```
SEQ
  début_traitement
  SEND( proc_dest_0, simple, 10, message )
  suite_traitement
  SEND( ANY, réponse, 40, message )
  suite_traitement
  SEND( proc_dest_0, question, 20, message )
  suite_traitement
  SEND( noyau, synchronisation, ANY, proc_gardé )
  fin_traitement :
```

Note : les entiers "10", "40", "20" signifie que le processus qui exécute cette commande est prêt à attendre "10", "40", "20" unités de temps pour que le processus destinataire réalise le service demandé. Si ce paramètre est omis (ou vaut "ANY"), le processus attend jusqu'à ce que le service demandé soit terminé.

Un processus gardé d'un connecteur peut être activé selon deux modes (les autres modes sont utilisés pour la programmation d'un processus gardé) :

- **simple** : la seule contrainte est que le processus gardé termine par l'envoi du message de terminaison.
- **question** : il est nécessaire que le processus gardé envoie un message de réponse au processus gardé appelant, avant l'envoi du message de terminaison (le noyau connait l'identification du canal attendant la réponse). Le noyau contrôle, lors de la terminaison du processus gardé, si le message de réponse a été envoyé. Dans le cas contraire, afin de pas générer un interblocage, le noyau produit un message d'erreur à destination du processus gardé appelant.

La possibilité de fermer ou d'ouvrir un canal permet de réaliser aisément des synchronisations sur des événements associés aux processus gardés. Par exemple, pour un connecteur :

- à l'événement "connecteur plein", le processus gardé d'écriture fermera le canal correspondant à ce processus gardé, il sera réouvert lors d'une lecture par le processus gardé de lecture.
- d'une façon symétrique, à l'événement "connecteur vide" le processus gardé de lecture fermera le canal associé à ce processus gardé qui sera réouvert lors d'une écriture.

1.6.2.2 Exemple de programmation de connecteurs

i) Gestion d'une file

Un processus gardé permet d'exécuter la lecture d'un message présent sur le connecteur (et la procédure "sortir" effectue le choix du message courant). Un autre processus gardé permet d'exécuter l'écriture d'un message dans le connecteur (et la procédure "entrer" met à jour le message courant du canal).

Dès que le connecteur possède un message, une lecture est possible et le canal du processus gardé correspondant est ouvert.

Processus gardé d'initialisation :

```
PROC_GARDE initialisation( VALUE message ) =
 SEQ
  initialiser( message )
  SEND( noyau, simple, (fermer,lecture) )
  SEND( noyau, simple, (ouvrir, écriture) ) :
```

Processus gardé de lecture :

```
PROC_GARDE lecture( VALUE message ) =
 SEQ
  sortir( texte, vide, plein )
  SEND( ANY, réponse, texte )
  IF
   NOT (ouvert( écriture ) OR plein)
    SEND( noyau, simple, (ouvrir, ecriture) )
  IF
   vide
    SEND( noyau, simple, (fermer, lecture) ) :
```

Note 1 : SEND(noyau, simple, (fermer, lecture)) signifie que le processus gardé qui exécute cette commande désire fermer le canal associé à son processus gardé lecture. Ce message est envoyé au noyau.

Note 2 : SEND(ANY, reponse, texte) signifie que le processus gardé qui exécute cette commande désire envoyer au processus qui l'a activé le message texte.

Processus gardé d'écriture :

```
PROC_GARDE écriture( VALUE message ) =
 SEQ
  entrer( texte, plein )
  IF
   NOT ouvert( lecture )
    SEND( noyau, simple, (ouvrir, lecture) )
  IF
   plein
    SEND( noyau, simple, (fermer, écriture) ) :
```

Selon la façon dont on programme les processus "entrer" et "sortir", le connecteur peut réaliser différents protocoles s'appuyant tous sur l'utilisation d'une file, avec N processus produisant des messages, M processus les consommant et avec une gestion de file particulière (FIFO, pile, lecture du dernier message).

ii) Rendez-vous entre deux processus

Outre le processus gardé d'initialisation, ce connecteur possède deux processus gardés qui permettent de réaliser soit une lecture d'un message, soit une écriture. Une variable globale permet l'échange du message "msg" entre les deux processus gardés lecture et écriture. L'accès à cette variable commune est contrôlé par le noyau, qui ne permet pas l'activation simultanée de deux processus gardés appartenent à la même entité.

Processus gardé d'initialisation :

```
PROC_GARDE initialisation ( VALUE message ) =
 SEQ
  initialiser( message )
  SEND( noyau, simple, (ouvrir, lecture) )
  SEND( noyau, simple, (fermer, écriture) ) :
```

Processus gardé de lecture :

```
PROC_GARDE lecture ( VALUE message ) =
 SEQ
  SEND( noyau, simple, (ouvrir, écriture) )
  SEND( noyau, synchronisation, écriture )
  SEND( ANY, retour, msg ) :
```

Note : SEND(noyau, synchronisation, écriture) signifie que le processus gardé qui exécute cette commande désire se synchroniser sur la terminaison du processus gardé écriture.

Processus gardé d'écriture :

```
PROC_GARDE écriture( VALUE message ) =
 SEQ
  msg := message
  SEND( noyau, simple, (fermer, écriture) ) :
```

Cette description est donnée à titre d'exemple et permet de réaliser un rendez-vous entre un processus producteur et un processus consommateur, alors qu'il y a N processus consommateurs et M processus producteurs possibles (le rendez-vous s'effectue entre le premier processus qui lit sur le connecteur et le premier qui écrit). Il est bien évident que la réalisation du rendez-vous tel qu'il est réalisé dans le langage OCCAM est immédiate (le noyau offrant directement ce mode de connexion).

1.6.3 Processus de l'application

Un processus de l'application possède un processus gardé spécifique qui est le corps séquentiel du processus. Les autres processus gardés permettent le traitement asynchrone des événements et s'exécutent selon le mode environnemental (ils sont activés chaque fois que l'environnement envoie un message sur le canal associé).

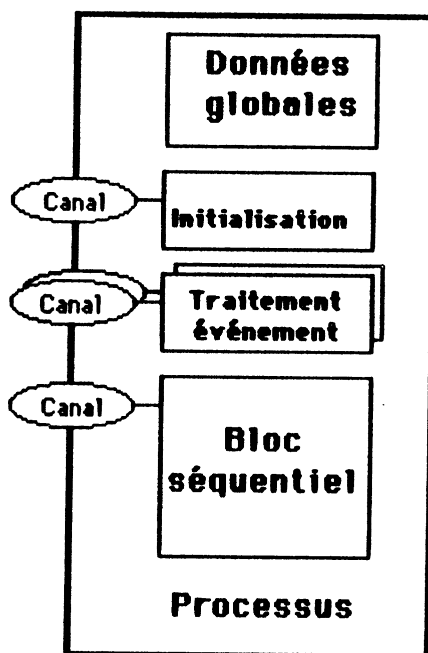


Figure 11. — Architecture d'un processus.

```
PROC processus ( CHAN in[], out[], suspend ) =
 VAR données_globales :
 ENV
  ALT
   suspend ? ANY
    suspend ? ANY
   ALT canal = [ 1 FOR nbre_proc_gardé ]
    ouvert[ canal ] & in[ canal ] ? message
      processus_gardé( message )
  ouvert[ séquentiel ] & in[ séquentiel ] ? message
   bloc_séquentiel( message ) :
```

Note1 : "in[séquentiel]" est le nom du canal d'entrée associé au processus gardé qui compose le corps séquentiel du processus.

Note2 : le canal "suspend" permet au noyau de suspendre l'exécution du processus si l'environnement désire communiquer avec l'application.

1.6.3.1 Les processus gardés d'un processus de l'application

Les trois premiers types de processus gardés (initialisation, traitement des erreurs et prise en compte d'événement) compose la partie ALTernative de la description précédente :

```
ALT canal = [ 1 FOR nbre_proc_gardé ]
  ouvert[ canal ] & in[ canal ] ? message
    processus_gardé( message )
```

i) Initialisation

Ce processus gardé, bien que spécifique à chaque entité, est toujours présent. Il permet d'initialiser les variables d'état de l'entité et de traiter le message d'initialisation.

Comme pour les autres entités, ce processus gardé est activé par un message émis par le serveur gestionnaire des processus, dès que le processus est créé.

ii) Traitement des erreurs issues de la communication

Ce processus gardé permet un traitement efficace des erreurs dues aux communications : temporisation écoulée, destinataire inconnu ou processus gardés inexistants. Il est propre à chaque entité et doit être écrit par le concepteur de l'application (qui peut, bien entendu, utiliser le même traitement d'erreur dans des entités différentes).

iii) Prise en compte d'événements

Ces processus gardés, attachés à un processus, permettent le traitement asynchrone des événements survenus sur un connecteur. L'arrivée d'un message pour ce processus gardé suspend momentanément le traitement séquentiel. Dans le cas où le processus gardé en cours d'exécution est déjà un traitement d'événements, celui-ci n'est pas interrompu.

Ces processus gardés peuvent être activés en mode simple ou question comme tous processus gardés. Ils sont programmés de la même manière que tout autre processus.

iv) Partie séquentielle

Ce processus gardé particulier constitue le corps du processus séquentiel. Programmé à l'aide d'un langage séquentiel, il utilise pour communiquer des primitives de haut niveau qui réalisent une succession d'envois de messages, ceci afin de rendre homogène chaque service demandé par un processus.

Ce processus gardé débute son exécution par la réception du message de configuration émis par le processus créateur.

1.6.3.2 Exemple de programmation des processus

Programmation du système de producteur/consommateur :

```
PROC_GARDE init_prod =
  -- description des liaisons entre le processus de l'application
  -- qui exécutera ce processus gardé
  -- et les différents connecteurs du système

  -- les primitives permettant de lier les processus gardés d'un
  -- processus aux processus gardés des connecteurs sont décrites
  -- au chapitre 5

PROC_GARDE init_cons =
  -- description des liaisons entre le processus de l'application
  -- qui exécutera ce processus gardé
  -- et les différents connecteurs du système

  -- les primitives permettant de lier les processus gardé d'un
  -- processus aux processus gardés des connecteurs sont décrites
  -- au chapitre 5

PROC_GARDE production =
  VAR msg :
  SEQ
    produire( msg )
    SEND( écriture, simple, msg ) :

PROC_GARDE consommation =
  VAR msg :
  SEQ
    SEND( lecture, question, msg )
    consommer( msg ) :

CONNECTEUR tampon =
  { -- description précédente du chapitre 1.5.2.2 }
PROCESSUS  producteur =
  { DESCRIPTION : init_prod, production }
PROCESSUS  consommateur =
  { DESCRIPTION : init_cons, consommation }

PAR
  producteur
  consommateur
  tampon
```

Remarque : la synchronisation entre les différents processus étant assurée par un connecteur, le nombre de processus "producteur" et le nombre de processus "consommateur" n'est pas limité.

2. Ensemble des primitives

Nous avons défini sous forme de primitives, une interface homogène utilisable par tous les processus. Chaque primitive réalise la succession d'envois de messages désirée, à l'aide de la primitive SEND décrite précédement.

Cet ensemble de primitives pourra être utilisé quel que soit le langage de programmation des processus de l'application. Ce sont elles qui garantissent la facilité (et la faisabilité) du transport d'une application d'un système à un autre.

2.1 Construire un modèle d'entité

Tout processus peut demander au constructeur du site où il s'exécute, d'assembler un nouveau modèle de processus ou de connecteurs pour un site particulier. Le serveur du site coopère avec les constructeurs des autres sites. Lorsque le modèle est construit, le serveur système prévient le gestionnaire correspondant (processus ou connecteur) afin que celui-ci le connaisse.

Un modèle est défini par ses règles de construction et on lui associe un nom global.

Primitive : CONSTRUIRE_MODELE(site, modèle, nom, réponse)

Paramètres d'entrée :

- **site** : nom du site sur lequel doit être créé le modèle.
- **modèle** : description du modèle à créer.
- **nom** : nom à associer au modèle nouvellement construit. Ce nom sera un nom global (chaine de caractères).

Paramètre de sortie :

- **réponse** : modèle construit ou code d'erreur.

Actuellement la description du modèle comprend :

- le type du modèle (processus ou connecteur).
- la liste des processus gardés qui le composent.

Le serveur système se charge de construire le modèle, à partir des processus gardés qu'il connaît et de ceux qu'il peut acquérir par dialogue avec le programmeur. Si ce n'est pas possible, le serveur prévient le processus qui a fait la demande.

Remarque : certains processus gardés ne peuvent être utilisés que pour construire des processus, d'autres des connecteurs, ou être associés à d'autres processus gardés particuliers.

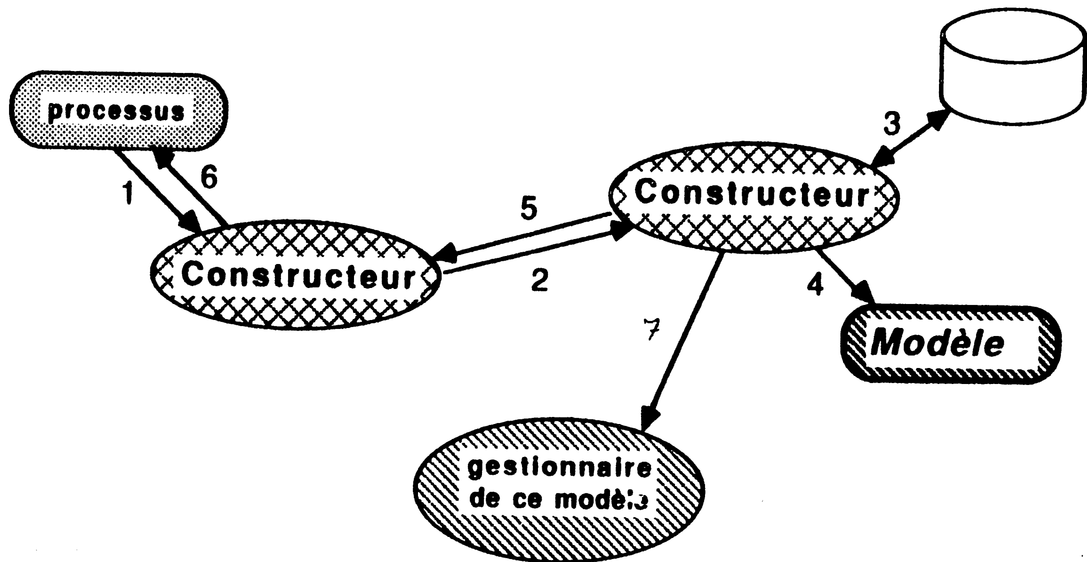


Figure 12. — Création d'un modèle.

Ce schéma permet de visualiser les différentes communications nécessaires pour créer un modèle :

1) demande de création de modèle : le processus transmet au serveur de construction la description du modèle : site de création, type et liste des processus gardés nécessaires pour composer le modèle, nom à associer au modèle.

2) si le site de création du modèle n'est pas le site du processus demandeur, le constructeur local transmet la demande au constructeur du site concerné.

3) le constructeur du site où doit résider le modèle, construit le modèle par consultation d'une base de connecteurs ou par coopération avec d'autres constructeurs qui possèdent les processus gardés demandés.

4) si le constructeur est arrivé à construire le modèle, il lui donne le nom spécifié.

5) si le modèle a été construit sur un autre site que celui où réside le processus demandeur, le contructeur du site de création donne au constructeur du site de demande un compte rendu : modèle créé ou non (dans ce cas, le constructeur donne les raisons de l'échec).

6) le constructeur du site de demande transmet au processus demandeur le compte rendu qu'il a reçu.

7) le constructeur du site où réside le modèle prévient le gestionnaire de ce modèles qu'un nouveau modèle est créé.

2.2 Supprimer un modèle d'entité

Le processus qui a demandé la construction d'un modèle, peut en demander sa suppression. Pour cela il s'adresse au constructeur du site sur lequel il s'exécute. Le serveur de construction prévient le serveur de gestion correspondant (processus ou connecteur) que ce modèle n'est plus disponible sur le site spécifié. Il n'est alors plus possible de créer une instance de ce modèle sur ce site, à moins d'effectuer une nouvelle construction.

Primitive : SUPPRIMER_MODELE(site, nom, réponse)

Paramètres d'entrée :

- **site** : nom du site sur lequel doit être supprimé le modèle.
- **nom** : nom du modèle à supprimer (ce nom est celui qui a été associé au modèle lors de sa création).

Paramètre de sortie :

- **réponse** : modèle supprimé ou code d'erreur.

2.3 Créer un processus

Tout processus peut créer un autre processus, à partir d'un modèle de processus connu par le système. Lors de cette création le gestionnaire de processus crée une nouvelle instance du modèle et un ensemble de canaux de communication associés au processus créé.

Puis il envoie le message d'initialisation (créé par le gestionnaire de processus) qui active le processus gardé d'initialisation et le message de configuration (créé par le processus créateur) qui active le bloc séquentiel.

Le système d'exécution (gestionnaire de nom) associe au processus nouvellement créé un nom (idp), qui sera le nom local du processus. Le nom global associé est la concaténation du numéro de site et de ce nom local.

Primitive : CREER_PROCESSUS(site, modèle, prio, config, réponse)

Paramètres d'entrée :

- **site** : nom du site sur lequel doit être créé le processus.
- **modèle** : nom du modèle du processus à créer (ce nom est celui qui a été associé lors de l'étape de création du modèle).
- **prio** : priorité d'exécution du processus.
- **config** : message à transmettre au processus nouvellement créé.

Paramètre de sortie :

- **réponse** : nom du processus créé (idp) ou code d'erreur.

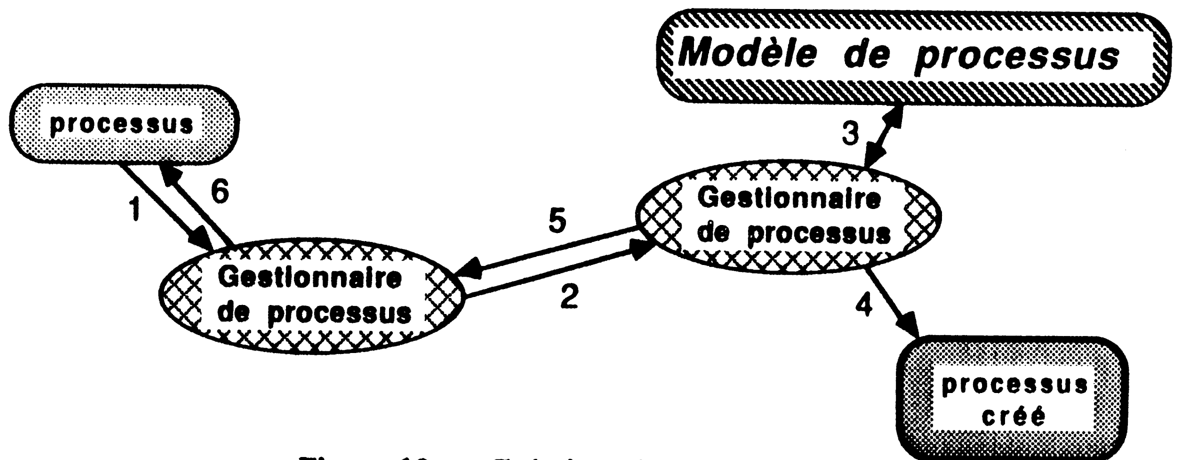


Figure 13. — Création d'un processus.

Ce schéma permet de visualiser les différentes communications nécessaires pour créer un processus :

1) demande de création de processus : le processus demandeur transmet au serveur de gestion des processus : le nom du site sur lequel doit être créé le processus, le nom du modèle, la priorité d'exécution du processus et le message de configuration à transmettre au processus nouvellement créé.

2) si le site de création du processus n'est pas le site du processus demandeur, le gestionnaire local transmet la demande au gestionnaire du site concerné.

3) le gestionnaire du site où doit être créé le processus, le crée par consultation d'une base de modèles de processus ou par coopération avec d'autres gestionnaires qui possèdent ce modèle.

4) le gestionnaire est arrivé à créer le processus.

5) si le processus a été créé sur un autre site que celui ou réside le processus demandeur, le gestionnaire du site de création donne au gestionnaire du site de demande un compte rendu : processus créé ou non (dans ce cas, le gestionnaire donne les raisons de l'échec).

6) le gestionnaire du site de demande transmet au processus demandeur le compte rendu qu'il a reçu.

2.4 Attente sur la terminaison d'un processus

Pour pouvoir programmer des instructions de synchronisation de haut niveau comme le constructeur PAR d'OCCAM, il est nécessaire de pouvoir synchroniser un processus sur la terminaison d'un ou de plusieurs processus.

Par exemple, le processus OCCAM suivant :

```
PROC essai =
  -- déclaration variables propres au processus essai
  SEQ
    -- variables propres au processus P1
    P1
    -- variables propres au processus parallèle
    PAR
      -- variables propres au processus P2
      P2
      -- variables propres au processus P3
      P3
    -- variables propres au processus P4
    P4 :
```

sera construit de la façon suivante (dans cet exemple nous ne tiendrons pas compte de l'initialisation des variables propre à chaque processus, ni du mécanisme d'héritage entre processus offert par OCCAM, ceux-ci seront discutés à la fin du chapitre 6, mais nous nous attacherons à montrer comment des connecteurs et le modèle CONKER peuvent permettre de synchroniser des processus) :

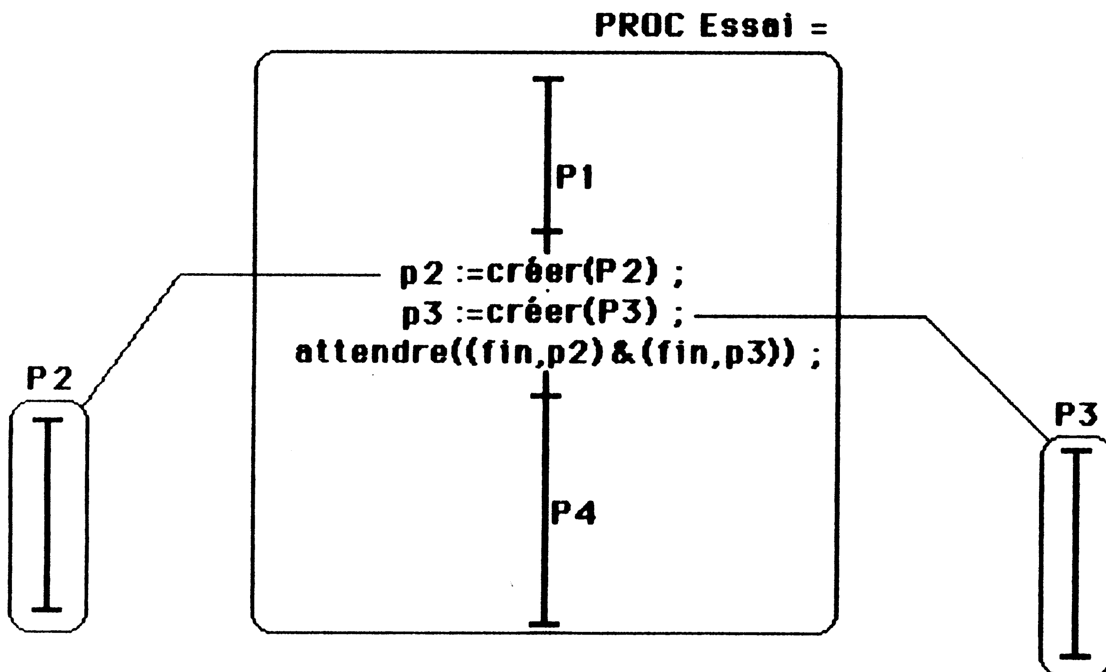


Figure 14. — Programme parallèle.

Deux types de synchronisation sont possibles. Le processus se synchronise soit sur la terminaison de tous les processus, soit sur la terminaison d'un seul parmi la liste définie.

Primitive : ATTENTE_TERMINAISON(liste, et/ou, tempo, réponse)

Paramètres d'entrée :

- **liste** : ensemble des processus dont on attend la terminaison, chaque processus est identifié par son nom global.
- **et/ou** : booléen qui permet de savoir si la synchronisation se fera sur la terminaison

de tous les processus ou sur celle d'un seul parmi ceux de la liste (la construction de synchronisation plus complexe est proposée dans le chapitre 5).
- **tempo** : temps avant lequel le processus désire avoir une réponse.

Paramètres de retour :
- **liste** : contient l'ensemble des processus ayant terminé et leur mode de terminaison (normal ou forcé).
- **réponse** : primitive correctement exécutée ou code d'erreur.

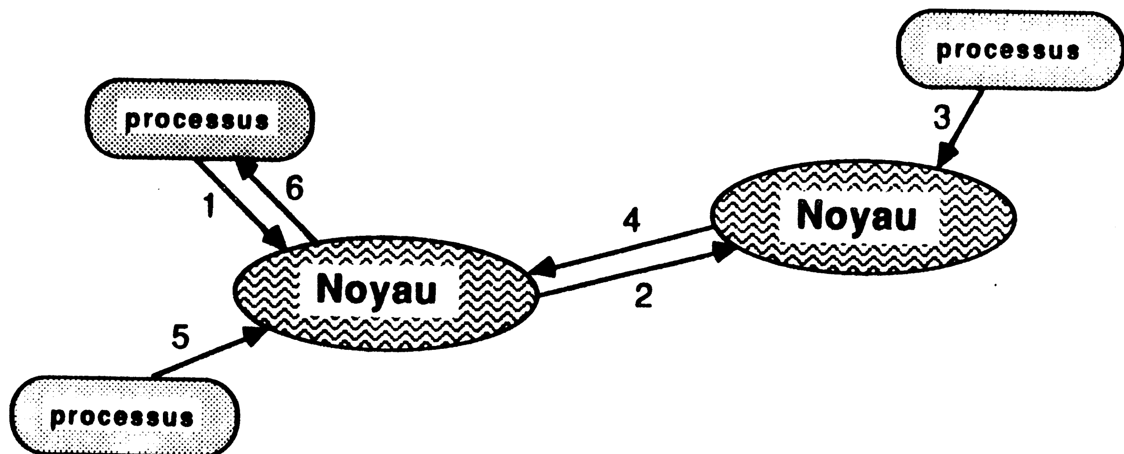


Figure 15. — Attente de terminaison.

Ce schéma permet de visualiser les différentes communications nécessaires pour qu'un processus attende la terminaison de différents processus :

1) demande d'attente : le processus transmet au noyau la liste des processus dont la terminaison est attendue, le booléen permettant de faire l'attente soit sur la première terminaison soit sur l'ensemble et le temps que le processus est prêt à attendre avant de recevoir une réponse.

2) le noyau envoie à chaque site où une terminaison est attendue le nom du (ou des) processus.

3) Chaque fois qu'un processus distant se termine, il envoie un message à son noyau.

4) à chaque terminaison de processus, distante par rapport au site demandeur, le noyau de terminaison prévient le noyau du site demandeur.

5) chaque fois qu'un processus local se termine, il envoie un message au noyau.

6) dés que le noyau peut rendre une réponse positive ou lorsque le délai est dépassé, le processus demandeur reçoit un message réponse contenant le détail (ensemble des processus ayant terminé et compte rendu).

2.5 Détruire un processus

La terminaison normale d'un processus est sa propre destruction quand il arrive à la fin de son exécution.

Pour permettre de forcer la terminaison d'un processus incorrect, on introduit une terminaison anormale qui s'accompagne de la fermeture de toutes les liaisons que ce processus a ouvertes. Chaque processus pourra détruire un autre processus, à condition de connaître son nom (idp).

Note : chaque processus connaît son nom et celui de tous les processus qu'il a créés et il peut en acquérir d'autres par communication.

Primitive : DETRUIRE_PROCESSUS(idp, réponse)
Paramètre d'entrée :
- **idp** : nom global du processus à détruire.

Paramètre de sortie :
- **réponse** : processus détruit ou code d'erreur.

2.6 Créer un connecteur

Tout processus peut créer des connecteurs à partir de modèles connus par le système CONKER. Pour cela, le processus doit nommer le connecteur, et le gestionnaire des connecteurs associera ce nom à l'instance du modèle qu'il va créer (un nom de connecteur est un identificateur unique sur tout le système CONKER).

Lors de la création, le processus configure le connecteur de deux manières :

- la première s'adresse au gestionnaire de connecteurs, et lui permet de fixer certains paramètres du connecteur indépendant du modèle : nombre de processus lecteurs, nombre de processus écrivains.
- la seconde s'adresse uniquement au connecteur et constitue le message d'initialisation du connecteur (par exemple : la liste des processus qui auront le droit de se lier au connecteur, la longueur de la file d'attente à l'intérieur du connecteur pour un modèle qui gère une file, le temps pendant lequel les messages ont un sens pour l'application, ...). Cette configuration dépend étroitement du modèle de connecteur et ne concerne pas le gestionnaire des connecteurs.

La création d'un connecteur n'est pas liée à un site puisque celui-ci assure le transport des messages entre les sites.

Primitive : CREER_CONNECTEUR(nom, prio, modèle, lec, ecr, init, réponse)
Paramètres d'entrée :
- **nom** : nom du connecteur à créer (chaine de caractères).
- **prio** : priorité d'exécution de l'entité connecteur.
- **modèle** : nom du modèle de connecteur dont on désire créer une instance.
- **lec** : nombre de lecteurs désirés (si lec=∞ nombre de lecteurs indéterminé).
- **ecr** : nombre d'écrivains désirés (si ecr=∞ nombre d'écrivains indéterminé).
- **init** : message d'initialisation du connecteur.

Paramètre de sortie :
- **réponse** : connecteur créé ou code d'erreur.

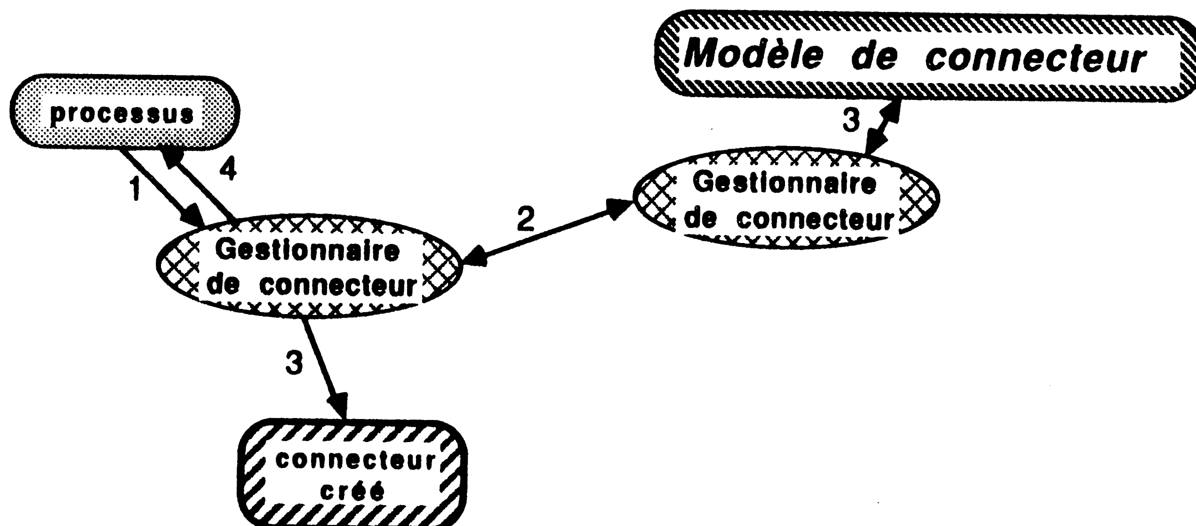


Figure 16. — Création de connecteur.

Ce schéma permet de visualiser les différentes communications nécessaires pour créer un connecteur :

1) demande de création de connecteur : le processus transmet au serveur de gestion des connecteurs le nom du connecteur à créer, la priorité d'exécution du connecteur, le nom global du modèle de connecteur, le nombre de processus pouvant être connectés en lecture et en écriture, le texte du message de configuration pour le connecteur.

2) si le gestionnaire de connecteurs ne connait pas ce modèle, il essaie de l'obtenir par une coopération avec les gestionnaires de connecteurs des autres sites.

3) quand le gestionnaire du site où doit résider le connecteur connait le modèle, il crée une instance de ce modèle et lui associe le nom donné en paramêtre.

4) le gestionnaire du site de demande transmet au processus demandeur un compte rendu : connecteur créé ou non (dans ce cas le processus connaît la raison qui a empéché cette création).

2.7 Détruire un connecteur

Pour détruire un connecteur, un processus s'adresse au gestionnaire de connecteurs local en lui donnant le nom (global) du connecteur à détruire.

La terminaison correcte d'un connecteur est sa destruction par un processus connaissant son nom lorsque plus aucun processus ne lui est lié. Nous appellerons cette destruction : destruction du connecteur sans purge.

Mais il peut être nécessaire de pouvoir détruire un connecteur alors que des processus lui sont encore liés (ces processus sont prévenus, mais ils ne peuvent pas savoir quels sont les messages qui ont été traités ou non). Dans ce cas, seul le processus créateur est autorisé à en forcer la destruction (destruction avec purge). L'exécution avec succès de cette commande implique la purge du connecteur (destruction de tous les messages contenus dans le connecteur, puis destruction du connecteur).

La destruction d'un connecteur peut s'accompagner d'opérations annexes visant à conserver l'intégrité des ressources présentes sur le site (par exemple si le connecteur permet l'accès à un fichier, sa destruction s'accompagnera toujours de la fermeture du fichier).

Primitive : DETRUIRE_CONNECTEUR(nom, purge, réponse)

Paramètres d'entrée :

- **nom :** nom du connecteur à détruire.
- **purge :** destruction forcée ou non du connecteur.

Paramètre de sortie :

- **réponse :** connecteur détruit (émis lorsque le connecteur est effectivement détruit) ou code d'erreur.

2.8 Se lier à un connecteur

Si deux processus veulent communiquer par l'intermédiaire d'un connecteur, ils doivent, après avoir créé ce connecteur (opération exécutée par un seul des deux processus), se lier à celui-ci par l'intermédiaire du nom global du connecteur (nom unique sur l'ensemble du système CONKER).

Si un processus désire se lier à un connecteur déjà créé mais qui n'est pas présent sur le site du processus, le gestionnaire des connecteurs du site, en liaison avec ceux des autres sites, se charge d'étendre le connecteur à ce nouveau site.

Primitive : LIER_CONNECTEUR(nom, mode, réponse)

Paramètres d'entrée :

- **nom :** nom du connecteur auquel le processus désire se lier.
- **mode :** type de connection désiré : lecture, écriture ou signalement d'événement.

Paramètre de sortie :

- **réponse :** nom du canal permettant de communiquer avec le connecteur (lien) ou code d'erreur.

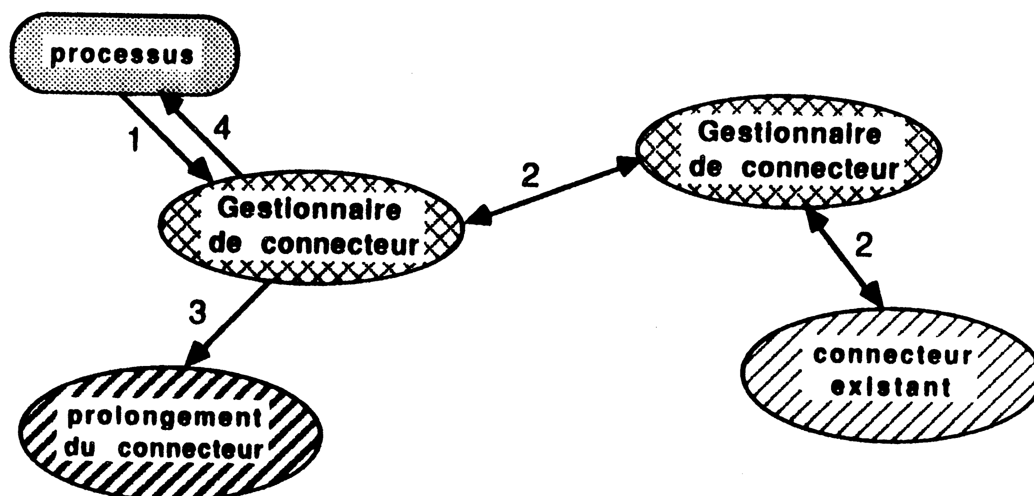


Figure 17. — Liaison d'un processus à un connecteur.

Ce schéma permet de visualiser les différentes communications nécessaires pour qu'un processus se lie à un connecteur :

1) demande de liaison à un connecteur : le processus transmet au serveur de gestion de connecteurs une requête dans laquelle il spécifie le nom du connecteur (nom associé au connecteur lors de sa création) et le mode de liaison désiré (lecteur, écrivain ou traitement).

2) si le connecteur n'est pas présent sur le site de demande, le gestionnaire de connecteurs local, par dialogue avec le gestionnaire d'origine du connecteur (celui ou il a été créé initialement), étend le connecteur sur ce site.

3) quand le connecteur est étendu sur ce site, le gestionnaire de connecteurs prévient le morceau de connecteur local qu'un processus s'est lié dans tel mode.

4) le constructeur du site de demande transmet au processus demandeur un compte rendu d'exécution : opération de liaison réussie (dans ce cas le processus reçoit un identificateur qui correspond au canal de communication que ce processus utilisera pour communiquer avec le connecteur) ou code d'erreur (dans ce cas le processus connaît la raison de l'échec).

2.9 Se délier d'un connecteur

Si un processus ne désire plus utiliser les services d'un connecteur, il peut alors se délier et le gestionnaire de connecteurs, en liaison avec ceux des autres sites, se charge de reconfigurer le connecteur si le modèle le nécessite.

La déliaison supprime le point d'accès (lien) précédemment créé.

Primitive : DELIER_CONNECTEUR(lien, réponse)

Paramètre d'entrée :

- **lien** : nom du canal de communication que le processus désire abandonner.

Paramètre de sortie :

- **réponse** : déliaison effectuée ou code d'erreur.

2.10 Lecture d'un message

Tout processus lié à un connecteur en lecture peut lire les messages que celui-ci contient. Le processus désirant lire un message doit fournir une zone mémoire pour le stockage du message (un message ne sera considéré comme lu que si la taille de la zone mémoire est supérieure à la taille du message). Selon le protocole d'échange que réalise le connecteur, celui-ci supprimera ou non le message lu.

La lecture d'un message sera temporisée de façon à permettre différents types de communications : si la temporisation est égale à l'infini alors la lecture sera bloquante, par contre avec une temporisation égale à 0, la lecture sera non bloquante (un message nul et un code d'erreur signalent dans ce cas, l'absence de message).

Primitive : LECTURE(lien, taille, ptr, tempo, réponse)

Paramètres d'entrée :

- **lien** : nom du canal de communication.
- **taille** : taille maximum du message que le processus accepte de recevoir.
- **ptr** : adresse du tampon pour la réception du message.
- **tempo** : temps avant lequel le processus désire que la lecture soit effectuée.

Paramètres de sortie :

- **taille** : taille effective du message lu.
- **réponse** : lecture du message ou code d'erreur.

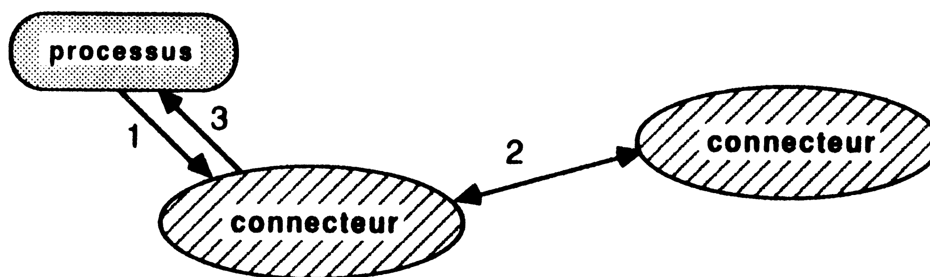


Figure 18. — Lecture d'un message.

Ce schéma permet de visualiser les différentes communications nécessaires pour qu'un processus de l'application puisse lire un message :

1) le processus transmet au connecteur, par le canal de communication qu'il a obtenu lors de l'étape de liaison, une demande de lecture.

2) le morceau de connecteur, par coopération avec les autres morceaux du connecteur, achemine un message sur le site de demande.

3) le connecteur retourne au processus soit le message, soit un code d'erreur, permettant au processus de diagnostiquer la panne.

2.11 Ecrire un message

Tout processus lié à un connecteur en écriture peut lui confier des messages. Si le connecteur n'est pas saturé, le message est lu dans son intégralité. Dans le cas contraire le processus est bloqué au maximum jusqu'à écoulement de la valeur du temporisateur passé en paramètre.

Suivant la valeur du temporisateur, on aura, comme en lecture, différents types d'écriture : si la temporisation est nulle, l'écriture sera non bloquante et si la temporisation est égale à l'infini alors l'écriture sera bloquante.

Primitive : ECRITURE(lien, taille, ptr, tempo, réponse)

Paramètres d'entrée :

- **lien** : nom du canal de communication.
- **taille** : taille du message à écrire.
- **ptr** : adresse du message à écrire.
- **tempo** : temps avant lequel le processus désire que son message soit lu par le connecteur.

Paramètre de sortie :

- **réponse** : message écrit dans le connecteur ou code d'erreur.

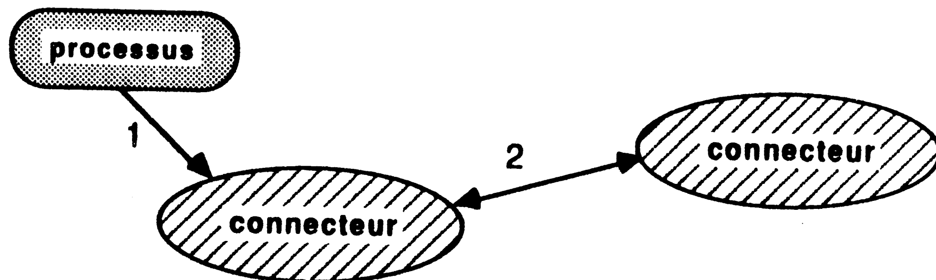


Figure 19. — Ecriture d'un message.

Ce schéma permet de visualiser les différentes communications nécessaires pour qu'un processus envoie un message dans un connecteur (même scénario que pour une lecture).

2.12 Attendre plusieurs événements

Tout processus peut demander à un connecteur, auquel il est lié, de lui signaler sous forme de messages, certains événements propres au connecteur. Les événements associés à un connecteur pourront être des événements simples (connecteur vide ou plein, écriture ou lecture, ...) mais aussi des événements plus spécifiques à tel ou tel connecteur. Ces événements pourront être mémorisés ou non selon leur nature et les possibilités du connecteur qui les gèrent (le chapitre 5 discute des différentes façons d'utiliser les

événements).

Chaque processus a la possibilité de se mettre en attente de l'arrivée d'un ensemble d'événements associés, chacun à un connecteur. A la terminaison de la procédure, le processus possède la liste des événements survenus sur chaque connecteur.

Par exemple : si un connecteur réalise une horloge, il pourra signaler des événements correspondant à l'instruction TIME ? AFTER date du langage OCCAM. De la même façon, un connecteur qui reçoit les valeurs d'un capteur pourra envoyer un message, chaque fois que la valeur du capteur dépasse un certain seuil.

Note : le système ne pouvant pas garantir toutes les cohérences, le programmeur devra prendre quelques précautions pour utiliser cette primitive. Par exemple, si un connecteur comporte N processus lecteurs, et qu'il vient de signaler à un processus une écriture, rien ne garantit que lorsque ce processus fera une lecture, un message sera encore présent.

Primitive : ATTENTE_MULTIPLE(liste, et/ou, tempo, réponse)

Paramètres d'entrée :

- **liste** : liste des événements attendus pour chaque connecteur <lien,évé>; {<lien,évé>}. "lien" est le nom du canal de communication obtenu lors de l'étape de liaison et "évé" est la description de l'événement attendu.
- **et/ou** : attente de l'arrivée de tous les événements ou attente de un parmi tous.
- **tempo** : temps avant lequel le processus désire que les événements lui soient signalés.

Paramètres de retour :

- **liste** : contient la liste des événements survenus sur chaque connecteur.
- **réponse** : primitive correctement exécutée ou code d'erreur.

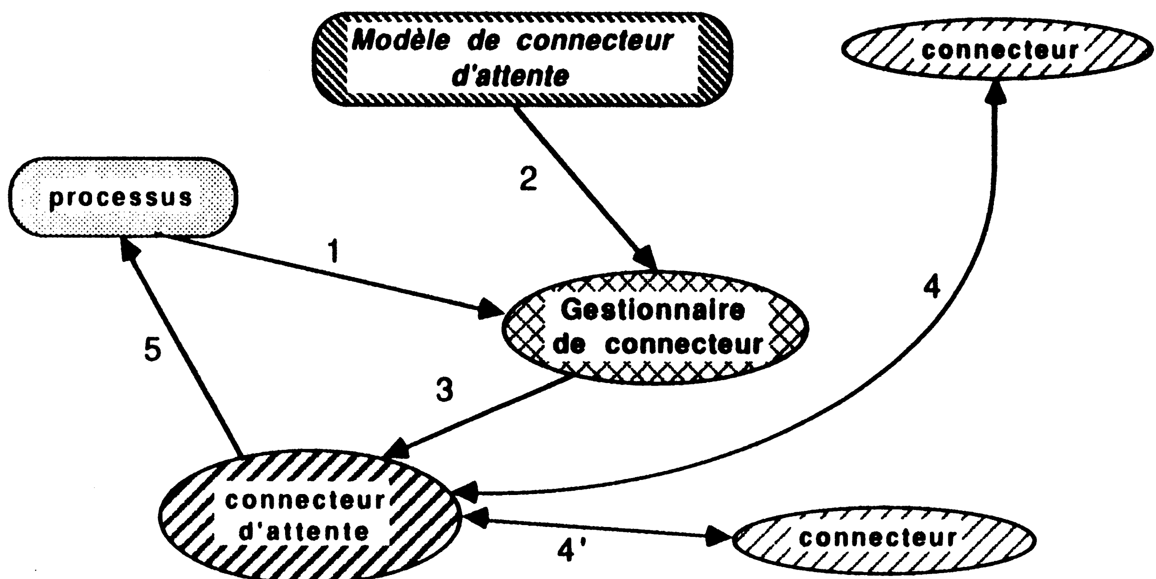


Figure 20. — Attente de plusieurs événements.

Ce schéma permet de visualiser les différentes communications nécessaires pour qu'un processus se mette en attente de plusieurs événements.

1) le processus demandeur envoie sa requête au gestionnaire des connecteurs, requête dans laquelle il spécifie la liste des événements attendus.

2) le gestionnaire des connecteurs recherche dans sa base de connecteurs, un modèle de connecteurs permettant de réaliser cette attente.

3) le gestionnaire des connecteurs crée le connecteur qui va réaliser pour le processus l'attente des événements spécifiés. Ce connecteur reçoit comme message d'initialisation la liste des événements à attendre et l'identification du processus demandeur.

4) le connecteur qui gère l'attente prévient l'ensemble des connecteurs, spécifiés dans le message d'attente, qu'il attend tel événement. Ceux-ci, dès que l'évènement est arrivé, préviennent le connecteur d'attente comme celui-ci l'aurait fait si l'événement était attendu par un processus.

5) quand l'événement attendu par le processus est arrivé ou lorsque le délai imposé est terminé, le connecteur donne au processus un compte rendu d'exécution et se détruit.

2.13 Activer un traitement

Un processus peut demander aux connecteurs auxquels il est lié d'associer un traitement à l'arrivée de certains événements (mécanisme d'interpellation par message). A chaque arrivée d'événements, le connecteur génère un message à destination des processus qui lui ont demandé un traitement.

Sur un même événement, plusieurs traitements indépendants peuvent être activés, le connecteur génère alors un message par traitement demandé.

En retour de cette primitive, le processus reçoit un identificateur de traitement. Si le processus associe au même événement plusieurs traitements (nécessairement indépendants), il recevra pour chacun d'eux un identificateur de traitement différent.

Primitive : ACTIVER_TRAITEMENT(lien, évé, trait, prio, réponse)

Paramètres d'entrée :

- **lien** : nom du canal de communication obtenu lors de l'étape de liaison.
- **évé** : description de l'événement attendu.
- **trait** : adresse du point d'entrée associé au traitement désiré. Cette adresse est le nom du canal de communication appartenant au processus et sur lequel il désire recevoir le message.
- **prio** : priorité d'exécution du traitement.

Paramètre de retour :

- **réponse** : traitement enregistré (idt = identificateur de traitement) ou code d'erreur.

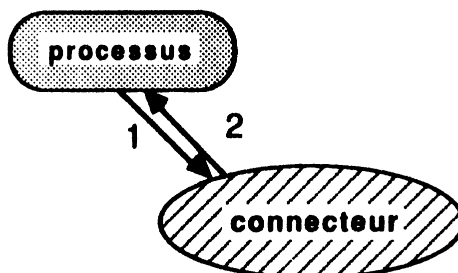


Figure 21. — Activer un traitement.

Ce schéma permet de visualiser les différentes communications nécessaires pour qu'un processus demande à un canal l'activation d'un traitement :

1) demande d'activation de traitement.

2) compte rendu de la demande d'activation : le processus reçoit soit un identificateur de traitement, soit un code d'erreur permettant d'en détecter la cause.

2.14 Supprimer un traitement

Chaque processus ayant demandé le traitement d'un événement, peut, à tout moment supprimer ce traitement.

Primitive : SUPPRIMER_TRAITEMENT(lien, trait, réponse)

Paramètres d'entrée :

- **lien** : nom du canal de communication.
- **trait** : identificateur du traitement à supprimer. Cet identificateur est celui obtenu lors de l'étape d'activation.

Paramètre de sortie :

- **réponse** : traitement de l'événement supprimé ou code d'erreur.

2.15 Masquer un traitement

Le masquage du traitement d'un événement signifie simplement que le processus désire traiter ultérieurement les événements qui pourraient arriver.

Primitive : MASK_TRAITEMENT(lien, trait, réponse)

Paramètres d'entrée :

- **lien** : nom du canal de communication.
- **trait** : identificateur du traitement à masquer.

Paramètre de sortie :

- **réponse** : masquage du traitement de l'événement ou code d'erreur.

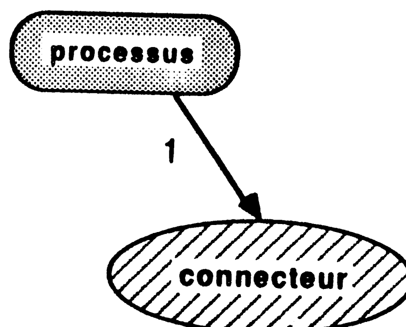


Figure 22. — Masquer un traitement.

2.16 Démasquer un traitement

Lors du démasquage, le nombre de traitements à activer dépend de l'événement lui même. C'est le connecteur qui prend l'initiative d'activer zéro, une ou plusieurs fois le traitement selon que l'événement qu'il contrôle est fugitif ou rémanent.

Primitive : DEMASK_TRAITEMENT(lien, trait, réponse)

Paramètres d'entrée :

- **lien** : nom du canal de communication.
- **trait** : identificateur du traitement à démasquer.

Paramètre de sortie :

- **réponse** : démasquage du traitement effectué ou code d'erreur.

2.17 Connaître l'état d'un connecteur

Cette primitive permet à chaque processus de connaître la valeur actuelle d'un des paramètres du connecteur désiré. Ces paramètres, qui dépendent du modèle de connecteur, peuvent être le nombre de lecteurs ou d'écrivains, la longueur du connecteur, la priorité d'exécution, la liste des processus gardés du connecteur, ...

Primitive : ETAT_CONNECTEUR(nom, param, valeur, réponse)

Paramètres d'entrée :

- **nom** : nom du connecteur.
- **param** : paramètre dont le processus désire la valeur.

Paramètres de sortie :

- **valeur** : valeur du paramètre spécifié.
- **réponse** : état du connecteur ou code d'erreur.

3. Conclusion

Pour implanter CONKER à l'aide d'un système de traitement hôte, il est nécessaire de réaliser le noyau, puis l'ensemble des primitives précédentes.

Afin de pouvoir respecter le modèle CONKER, le système de traitement hôte devra :

- permettre la création d'un nombre suffisant de tâches. Si le noyau n'offre pas cette possibilité, il sera nécessaire de construire un "multiplexeur de processus" [TRICO87].

- posséder une horloge battant à une fréquence suffisante pour l'application supportée par le noyau.

- offrir au noyau CONKER, si l'application nécessite un contrôle du temps d'exécution des processus, la possibilité d'assurer seul le cadencement des processus (par exemple, le système Unix n'offre pas cette possibilité et ne peut pas, par conséquent, respecter des contraintes de temps).

Les utilisations de CONKER que nous avons faites, et qui sont décrites dans les chapitres 5 et 6 suivants, nous ont permis d'en définir une méthodologie d'utilisation :

- la première étape consiste à décomposer l'application en processus. Chaque processus constitue une entité, possédant des variables propres, et coopérant avec d'autres processus par échange de messages selon des protocoles à déterminer.

- une seconde étape consiste en la programmation séquentielle de l'ensemble des processus de traitement, en définissant pour chacun l'ensemble des communications/synchronisations nécessaires (soit avec les autres processus, soit avec l'environnement).

- la troisième étape doit permettre de caractériser les connecteurs utilisés par l'application. Un connecteur est une entité spécialisée qui se charge du transport des messages et de la synchronisation des processus qui lui sont connectés. Si le connecteur est réparti (les processus qu'il relie sont sur différents sites), il assure alors le transport des messages entre ces sites.

- l'étape suivante permet au programmeur de préciser les traitements désirés, liés à l'environnement. Celui-ci interagit avec l'application par échange de messages ou de signaux : des connecteurs spécifiques permettent la communication avec l'environnement. A l'arrivée de chaque message de l'environnement, ces connecteurs peuvent activer des traitements qui seront effectués dans le contexte d'un processus donné.

- une autre étape peut être constituée par la programmation, si c'est nécessaire, des protocoles de reprise en cas de panne ou d'erreur. Ces protocoles pourront être utilisés par chaque entité effectuant une communication.

- puis, si le programmeur le désire, il pourra vérifier un certain nombre d'assertions écrites dans un calcul dérivé de CSP et basées sur les connecteurs

(protocoles réalisés effectivement par les connecteurs décrits) et sur les processus (correction partielle ou totale de l'application).

V. Réalisation de connecteurs pour une application robotique

Résumé

Dans ce chapitre, nous allons suivre la démarche que nous avons suivie pour implémenter la partie du système CONKER nécessaire à l'exéçution d'une application de commande de robot. Après avoir étudié les fonctionnalités de l'application, nous avons structuré celle-ci en un ensemble de processus communiquants entre eux par des protocoles particuliers. Suite à cette étape de définition nous avons écrit le noyau de CONKER en ASM86, les processus en Fortran 77, et les connecteurs réalisant les protocoles en PLM86.

Ce chapitre présentera donc :

- les grandes lignes du langage LM développé à l'IMAG-LIFIA et commercialisé par la société ITMI. Ce langage, qui permet la manipulation de robots, est basé sur un nombre réduit de concepts de base.

- un sous-ensemble du noyau de communication CONKER nécessaire à l'exécution de cette application. L'écriture de ce noyau de communication (réduit), permet d'assurer une portabilité et une répartition aisée de la machine LM.

- un ensemble de connecteurs, spécifique aux besoins en communication et synchronisation de cette application. L'ensemble des connecteurs ainsi défini permet l'interconnexion des différents processus de l'application définie.

Cette réalisation a été effectuée à l'aide du système de traitement hôte iRMX86 [INTEL82, HEIDE82] (défini par INTEL et s'exécutant sur un micro-processeur 8086) et écrite en PLM86 [INTEL82a].

1. LM : langage de manipulation de robots

Le langage LM est un langage de manipulation de robots, développé initialement à l'IMAG par E. Mazer et J-F. Miribel, et commercialisé actuellement par la société ITMI.

Nous présenterons succintement le langage LM de façon à permettre au lecteur de mieux saisir le cadre de cette étude (pour plus de précision voir le manuel de référence [MAZER85]).

Puis, nous nous attacherons à présenter la machine LM, à partir du travail effectué par l'équipe robotique de l'IMAG [MIRIB84]. Chaque fois que ce sera possible, nous indiquerons la réalisation de ces outils à l'aide du noyau CONKER.

La machine LM est la structure logicielle qui permet l'interprétation du langage LM. Elle est actuellement implantée sur de nombreux calculateurs hôtes. Mais les réalisations actuelles comportent un nombre réduit de tâches et n'autorisent pas la répartition de l'application.

Les systèmes actuels de robotique sont constitués d'une machine (fréquemment) monoprocesseur qui pilote un ou plusieurs robots. Dans un proche avenir, il sera nécessaire de pouvoir répartir ces architectures afin de pouvoir faire coopérer :

- un système de robotique complexe constitué de plusieurs robots.
- un système de vision.
- un système d'Intelligence Artificielle, permettant au système de s'adapter au mieux à l'environnement.

1.1 Concepts

LM est un langage de programmation de robots qui repose sur un petit nombre de concepts de base et qui possède un ensemble de constructions syntaxiques expressives permettant de combiner ces concepts. LM a été défini en vue d'être complètement implanté, ce qui fait que, la technologie évoluant, le langage n'est pas figé et de nouvelles composantes peuvent être intégrées. Une nouvelle "mouture" de LM (inspirée du langage CSP) est en cours de définition à l'IMAG-LIFIA par J.M Lefebvre [LEFEB82] de façon à permettre d'exprimer un parallélisme d'exécution. C'est cette version de LM qui devra utiliser les facilités offertes par CONKER.

Un langage de programmation de robots comme LM est un mélange d'instructions algorithmiques, habituelles dans les langages de programmation de type PASCAL ou ALGOL, et d'instructions spécifiques pour la commande du robot : gestion des déplacements et manipulation de l'outil terminant le bras.

1.1.1 Communication avec l'environnement

Dans une application robotique la communication avec l'environnement recouvre deux aspects distincts :

- il s'agit d'obtenir des informations sur les objets de l'environnement immédiat des robots. D'une façon générale, ces informations peuvent être obtenues soit

par des systèmes de vision qui peuvent identifier et localiser des objets, soit par des capteurs de force qui mesurent les efforts exercés par les robots. Nous appellerons ce premier type de communication : **perception de l'environnement.**

- l'autre aspect recouvre la lecture d'informations qui ne dépendent pas directement du système de robotique lui-même, mais qu'il est nécessaire de prendre en compte lors d'une exécution. Par exemple : l'état d'un dispositif d'amenée de pièces caractérise la possibilité, pour un robot d'assemblage, d'être alimenté. Cet ensemble d'informations caractérise **l'état de l'environnement.**

Le but de la communication avec l'environnement est de permettre au programme de manipulation de robots, d'adapter les traitements et les actions à l'environnement (même si celui-ci évolue). Par exemple, une donnée d'un système de vision permettra de calculer la trajectoire (optimale?) permettant d'éviter un objet ou de mettre au rebut une pièce si l'inspection visuelle a montré qu'elle était défectueuse. Un capteur de force permet d'identifier les contacts avec la pièce lors d'une opération de saisie et permet "d'être persuadé" que la pièce a été effectivement prise.

La prise en compte de l'environnement dans un programme de commande de robots a une importance croissante, car il est reconnu qu'elle permet de réduire considérablement les coûts de mise en oeuvre de robots, notamment en ce qui concerne les outillages spécialisés.

1.1.2 LM et l'environnement

Le langage LM offre deux constructions principales pour la communication avec l'environnement : les fonctions capteurs et les variables d'état.

Une **fonction capteur** s'évalue à une valeur de type entier, booléen, réel, ... Cette valeur n'est pas sous le contrôle direct du programmeur, c'est par exemple un réel mesurant la force exercée par un robot le long d'une direction donnée, un booléen donnant l'état opératoire d'un dispositif périphérique (marche ou arrêt)...

> *Tous les capteurs de l'application LM seront connectés à des connecteurs capables de donner la valeur courante du capteur et de générer des événements liés aux valeurs du capteur (traitement asynchrone d'événement par les processus) : par exemple, dépassement de valeur limite.*
>
> *Ces connecteurs, gestionnaire des capteurs, pourront être activés de différentes manières : soit comme un connecteur de communication habituel dans le cas où un processus désire obtenir la valeur courante du connecteur, soit directement par le noyau afin que le connecteur lise la valeur du capteur et produise les événements liés à la valeur du capteur.*

Une **variable d'état** peut aussi désigner un repère, un réel, un entier, un booléen, ... de la même façon qu'une fonction capteur. La valeur de cette variable n'est pas sous le contrôle direct du programmeur, mais les liaisons éventuelles de cette variable avec d'autres variables du programme le sont. Par exemple ROBOT(i) est le repère attaché à l'extrémité du robot numéro i, TEMPS_DEPL(i) est une variable d'état indiquant le temps écoulé depuis le début du dernier déplacement du robot numéro i.

Chaque variable d'état sera "véhiculée" sur le système par un connecteur spécifique permettant à l'ensemble des processus d'en connaître sa valeur courante. Par exemple ROBOT(i) sera un connecteur capable d'acquérir la position courante du robot pour la donner au processus, TEMPS_DEPL(i) sera un connecteur qui aura un comportement proche d'un chronomètre (chaque début de déplacement génèrera un message remettant le chronomètre à zéro, et chaque lecture de message permettra à un processus d'acquérir la valeur courante de la variable).

L'évaluation d'une fonction capteur ou d'une variable d'état peut nécessiter des calculs plus ou moins complexes (lecture d'un port d'entrée/sortie ou analyse d'une image). Si cette évaluation s'effectue en même temps qu'un déplacement, il faudra en tenir compte car elle pourra interagir avec cette exécution. Dans le cas d'une architecture multi-processeurs ces calculs pourront être faits sur une autre machine, mais il faudra toujours pouvoir les prendre en compte s'ils doivent modifier le déplacement en cours : mécanisme de traitement d'événements implémenté par CONKER.

1.1.3 La programmation des déplacements

L'objectif est de donner au programmeur des constructions symboliques lui permettant de décrire de façon explicite les mouvements désirés. La notion de mouvement recrouvre plusieurs concepts qu'il convient de distinguer :

- la **géométrie du mouvement** (c'est à dire la trajectoire). Une façon naturelle de décrire une trajectoire est de la décrire dans un espace cartésien, indépendant de la géométrie du robot qui va l'effectuer. Le programmeur doit pouvoir choisir entre différentes géométries, de façon à pouvoir minimiser la longueur du chemin à parcourir, tout en évitant les collisions avec les objets présents sur son parcours.

- la **cinématique du mouvement.** Elle concerne l'évolution du robot le long de la trajectoire et dans le temps. La vitesse à laquelle s'effectue une trajectoire peut avoir un effet sur la précision (effet dû en particulier à l'inertie du robot). Elle peut aussi être imposée par le programmeur de façon à effectuer une action le long de la trajectoire : par exemple la soudure de deux plaques ou la dépose d'un cordon de colle le long d'un joint doit être effectuée à vitesse constante quelque soit la géométrie de la plaque.

- l'**interaction du mouvement avec les capteurs.** De nombreux mouvements sont contrôlés uniquement en position, cela suppose une connaissance de l'environnement qu'il n'est pas toujours possible d'avoir, y compris au moment où débute la trajectoire. Il est donc parfois nécessaire d'adapter les réactions d'un robot en mouvement à des données sensorielles (force, vision, toucher, ...). L'idéal serait de pouvoir adapter la géométrie du mouvement à des données transmises par des capteurs (de tels mouvements sont dit "à compliance").

- un **mouvement peut aussi dépendre d'autres robots.** En effet, de plus en plus fréquemment un robot n'est plus une machine isolée. Il interagit avec d'autres dispositifs, qui peuvent être d'autres robots (sous le contrôle du même programme ou non) ou des automates assurant des services annexes. Il s'agit

alors de synchroniser et de faire coopérer dans l'espace et dans le temps ces différents robots.

Pour gérer le déplacement d'un robot, LM possède une seule instruction dont l'exécution provoque l'initialisation et le suivi d'une trajectoire. Sa construction fait intervenir les concepts introduits précédemment.

- **Trajectoire** : une trajectoire est composée de plusieurs segments qui peuvent être libres (seule la position d'arrivée du repère est spécifiée), cartésiens (le repère se déplace le long d'une droite), circulaires (le repère suit un arc de cercle), planifiés (le repère suit une séquence de position précédemment enregistrée). L'interpréteur de la machine LM effectue la transition entre les différents segments pour assurer une continuité de la vitesse.

Dans l'exemple suivant l'interpréteur LM doit effectuer le raccord entre deux déplacements cartésiens. Quand le robot arrive à proximité de la zone de changement de trajectoire, l'interpréteur arrête de suivre le premier déplacement pour effectuer un changement de trajectoire à vitesse continue. Ceci permet au robot de changer de trajectoire sans s'arrêter et de reprendre la seconde trajectoire dans la zone de proximité avec la vitesse désirée.

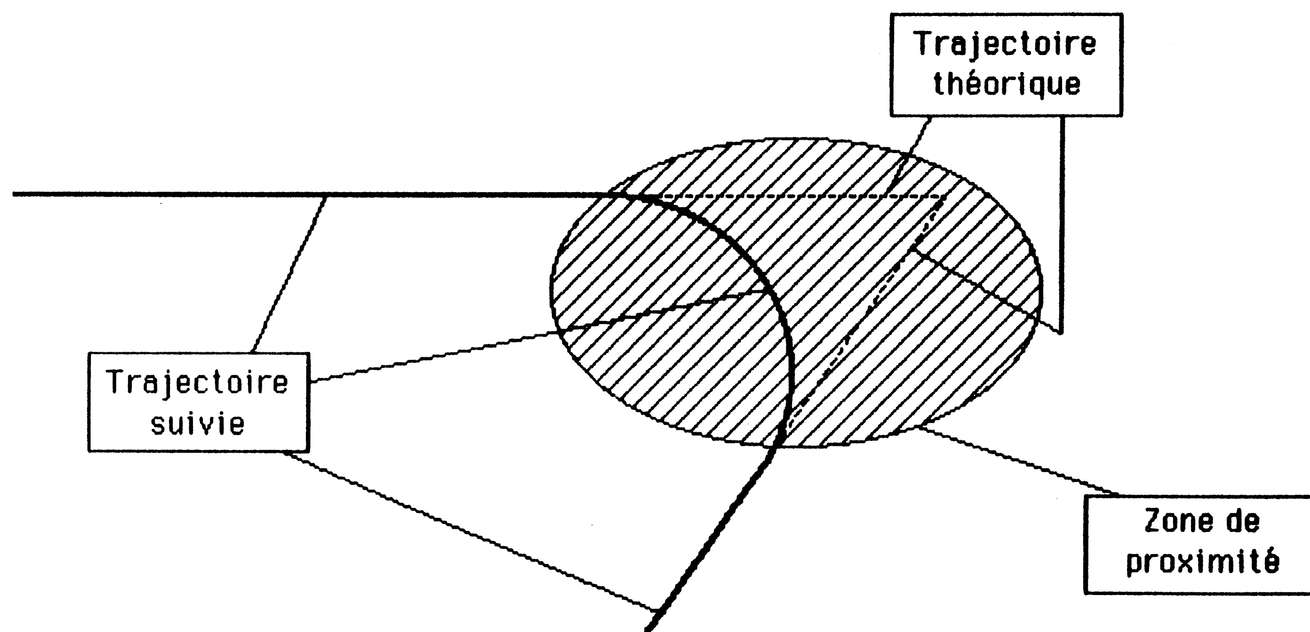


Figure 1. — Raccord entre deux mouvements.

- **Cinématique** : le programmeur peut contrôler la vitesse du déplacement pour chaque segment de la trajectoire. On peut aussi contrôler indirectement la vitesse en fixant la durée d'une trajectoire.

- **Interaction avec les capteurs** : un déplacement peut être gardé par un certain nombre de conditions reposant soit sur des variables d'état (temps d'exécution), soit sur des valeurs données par des capteurs. Par exemple Fz<5 : la force exercée par le robot selon l'axe Z, doit être inférieure à 5 Newtons. Dans le cas où cette garde est vraie, on exécute alors l'action correspondante.

Nous aurons par exemple :

DEPLACER cylindre **A** trou_cube **EN CARTESIEN JUSQU'A** FX>5 **OU** FY>5 ;

DEPLACER cube **VIA** cube*TRANSLATION(VZ, 50) **EN CARTESIEN,**
VIA destination*TRANSLATION(VZ, 50) **EN LIBRE**
A destination **EN CARTESIEN AVEC** VITESSE=0.1 ;

Remarque : de la même façon que l'on peut spécifier des déplacements, LM offre des constructions permettant de gérer les outils terminaux. Nous aurons par exemple :

FERMER pince **JUSQU'A** F>3.

1.2 L'architecture de la machine LM

Dans ce chapitre nous nous attacherons à structurer l'application LM, de façon à ce qu'elle puisse s'exécuter à l'aide du noyau CONKER.

La machine LM reçoit en entrée un code LM_e (code LM déjà compilé et assemblé) et qui est interprété par une tâche LM pour réaliser l'application. Le code intermédiaire LM_e correspond au niveau d'interface sur lequel sont à l'étude de nombreuses propositions de normes.

L'exécutif LM remplit trois fonctions principales :

- **interprétation du code LM_e** (instructions de calcul, de contrôle, de communication avec l'environnement). C'est cet interpréteur, réalisé sous la forme d'un processus de CONKER qui lit un fichier source, qui crée (ou plus exactement active) un ensemble de processus et de connecteurs qui permettent de gérer les instructions complexes de LM (celles spécifiques à la commande de robots) et qui interprètent directement les instructions de type algorithmique classique.

- **gestion de robots** (commande des mouvements et des actions des outils). Cette fonction sera réalisée par un ensemble de processus coopérants et de connecteurs assurant l'interface entre l'armoire de commande et l'équipement informatique.

- **surveillance** des déplacements et des conditions (gestion des gardes). Cette fonction primordiale si l'on désire réaliser des mouvements à compliance, sera réalisée par un ensemble de processus et de connecteurs spécifiques assurant l'interface entre l'équipement informatique et le système qui recueille les données sensorielles (système de vision, capteurs, ...).

Les deux dernières fonctions remplies par l'exécutif de LM, sont nécessairement tributaires de l'interprétation du code LM_e.

1.2.1 Le séquenceur

Pour réaliser le séquencement des processus gardés liés à l'interprétation d'une instruction LM, l'exécutif LM utilisera directement les possibilités du noyau CONKER :

- attribution des priorités, de la façon la plus judicieuse possible.
- description de l'application en processus et connecteurs, les uns s'occupant du traitement et les autres du transfert de l'information. Cette répartition n'est pas toujours évidente. En effet, des connecteurs peuvent effectuer une opération de traitement lors du transport de l'information.

i) Séquencement

L'application LM, comme de nombreuses applications temps-critique, utilise un séquencement des tâches sur événements. Une tâche LM perd le contrôle du processeur uniquement si elle se trouve en manque de ressource ou si un événement externe active une tâche LM plus prioritaire.

ii) Système de priorité réalisé

Afin de privilégier la communication-synchronisation entre les processus par rapport au traitement des messages (ce qui permet à l'application de prendre en compte le plus rapidement possible les messages d'arrêt), nous avons fait le choix de rendre prioritaires les connecteurs par rapport aux processus.

Si l'on désire affiner l'analyse, il devient nécessaire de définir quelles doivent être les priorités du système LM :

- **privilégier la sécurité** : l'arrêt opérateur doit être immédiat. Le connecteur qui récupère ce signal et les processus d'arrêt qu'il doit activer doivent être exécutés quelle que soit la charge du système.
- **préserver l'intégrité du matériel** : le respect de certaines gardes doit être impératif. Par exemple : enfoncer un goujon dans une plaque ne doit se faire que si l'effort à exercer est inférieur à un seuil dépendant de la limite de déformation de la plaque. La fréquence d'évaluation des gardes correspondantes doit être suffisante et peut être calculée en fonction de la vitesse de déplacement du bras du robot.

1.2.2 Portabilité

Il est évident que pour un industriel, le critère de portabilité est essentiel. La portabilité d'un système de programmation de robots se juge non seulement sur la facilité de transport de ce logiciel d'un calculateur à un autre, mais aussi sur la facilité d'adaptation de ce logiciel à de nouveaux robots.

Jusqu'à ce jour, les comités de normalisation dans le domaine de la programmation des robots ne semblent pas s'être intéressés à la définition d'interfaces de raccordement entre l'armoire de commande des robots et les systèmes informatiques extérieurs.

Le noyau CONKER a permis de reprendre directement les différentes interfaces de raccord utilisées par les autres implémentations de la machine LM. Il a suffi de définir pour chaque interface, un connecteur assurant le protocole de connexion : par exemple, un connecteur assurant la transformation entre l'espace cartésien dans lequel est défini un mouvement et l'espace des coordonnées propres (ou articulaires) utilisé par un robot particulier.

2. Architecture de la tâche de déplacement

Nous allons essayer de définir dans cette partie, l'architecture de la tâche de déplacement qui gère les mouvements d'un robot. Cette tâche, composée d'un ensemble de processus et de connecteurs de CONKER, est activée lors de l'interprétation d'une instruction DEPLACER de LM. De même que cette instruction se décrit à l'aide d'un nombre réduit de concepts, son interprétation sera réalisée par un nombre réduit de processus de traitement, connectés entre eux par des connecteurs assurant le transport des messages selon des protocoles que l'on définira.

Remarque : à une tâche de l'application LM correspondra plusieurs processus de CONKER. C'est pourquoi nous garderons la terminologie de tâche pour parler du découpage effectué par J-F. MIRIBEL [MIRIB84] et nous parlerons de processus pour l'implantation que nous avons réalisée.

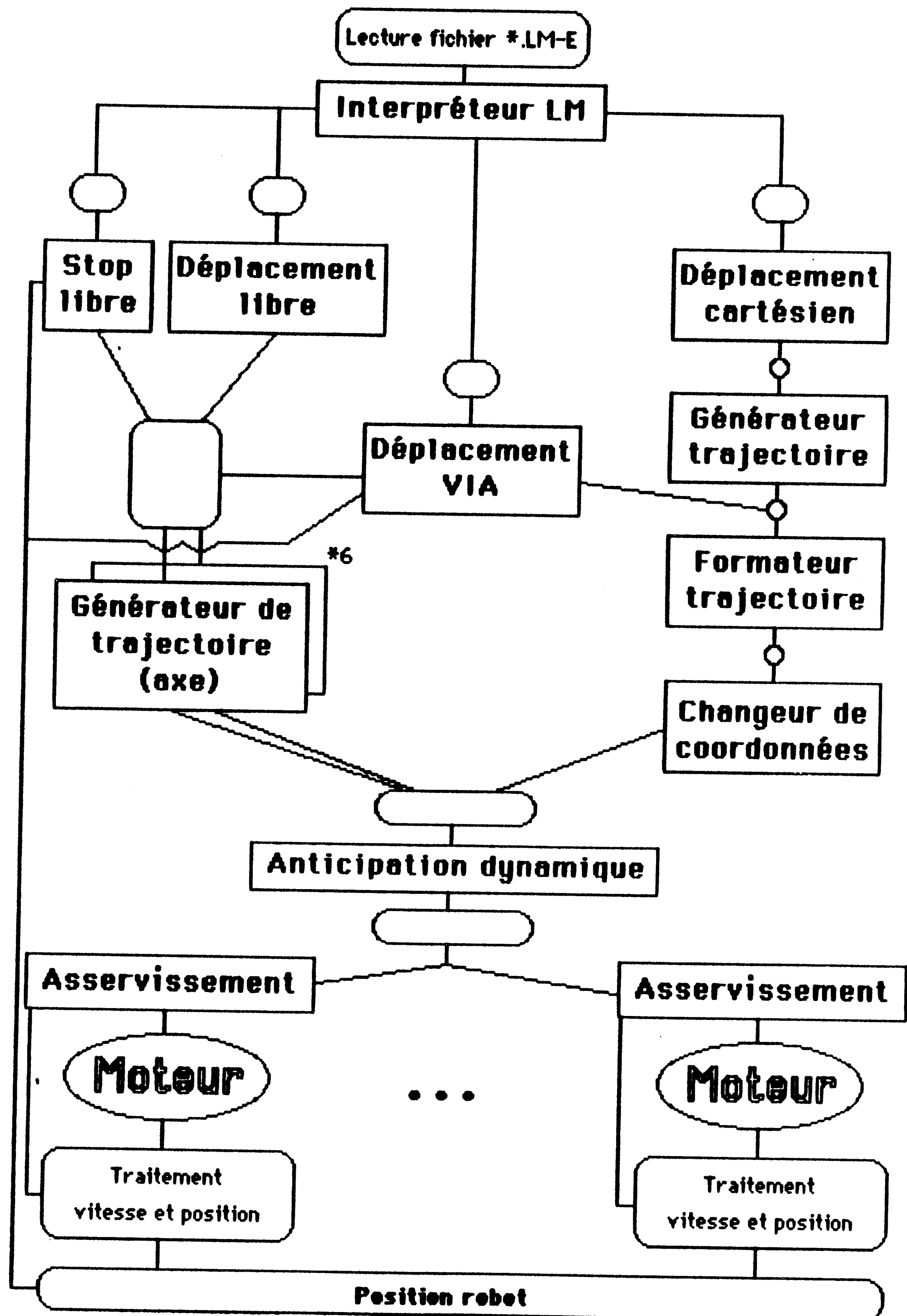
La généralité et la complexité de la tâche de déplacement nous serviront d'exemple pour l'utilisation du noyau CONKER.

2.1 Principe de l'architecture

L'architecture développée s'inspire des concepts suivants :

- **la programmation modulaire** : le système est construit à partir de modules de traitement qui peuvent s'échanger des messages. Certains de ces modules peuvent être des instances d'un même type d'unité de traitement (par exemple le générateur de trajectoire par axe).

- **la programmation concourante** : commander plusieurs degrés de liberté et assurer leur synchronisation dans le temps est typiquement un problème de parallélisme.



Lecture fichier *.LM-E
Interpréteur LM
Stop libre
Déplacement libre
Déplacement cartésien
Générateur trajectoire
Déplacement VIA
*6
Générateur de trajectoire (axe)
Formateur trajectoire
Changeur de coordonnées
Anticipation dynamique
Asservissement
Asservissement
Moteur
...
Moteur
Traitement vitesse et position
Traitement vitesse et position
Position robot

La figure précédente présente sous forme schématique, les principaux processus qui composent la tâche de déplacement. Nous n'avons représenté les processus et les connecteurs que pour deux modes de déplacement : LIBRE (l'interpréteur LM effectue le déplacement en minimisant les déplacements selon chaque degré de liberté) et CARTESIEN (l'interpréteur LM effectue le déplacement de chaque degré de liberté de façon à ce que le repère associé à l'outil terminal suive un segment de droite). Le module gestion VIA gère les raccords entre les différentes trajectoires.

- L'interpréteur est connecté à un connecteur d'accès à un fichier qui lui permet de lire le code à interpréter.
- Les processus "Stop libre", "Déplacement libre" et "Déplacement cartésien" sont en attente d'une lecture sur le connecteur qui les relie à l'interpréteur.
- Le processus "Anticipation dynamique" assure la synchronisation entre les différents modules qui calculent la trajectoire. Par exemple, il existe un "Générateur de trajectoire" par axe pour le mode de déplacement libre, et chacun effectue l'échantillonnage selon son axe ; le processus "Anticipation dynamique" regroupe l'échantillonnage pour donner à son connecteur de sortie un échantillonnage de la trajectoire.
- Un connecteur spécifique "position robot" regroupe toutes les données issues des connecteurs de "Traitements vitesse et position" de façon à maintenir à jour un ensemble d'informations cohérentes concernant la position et la vitesse de l'outil terminal selon chaque degré de liberté.

2.2 Principaux modes de communication entre processus

Dans l'application LM, les communications entre les processus qui réalisent la gestion des déplacements peuvent prendre diverses formes.

Nous supposerons que tous les processus ne renvoie pas d'erreur (le programme LM est correct et peut "tourner" sur cette architecture). Si l'on désire rajouter ce traitement il faut prévoir des connecteurs chargés de transporter cette information et des processus capables de traiter cette information (soit pour arrêter le robot, soit pour effectuer des corrections de trajectoire en ligne).

2.2.1 Activation simultanée

Ce mode de communication permet d'activer simultanément plusieurs traitements (voir figure suivante).

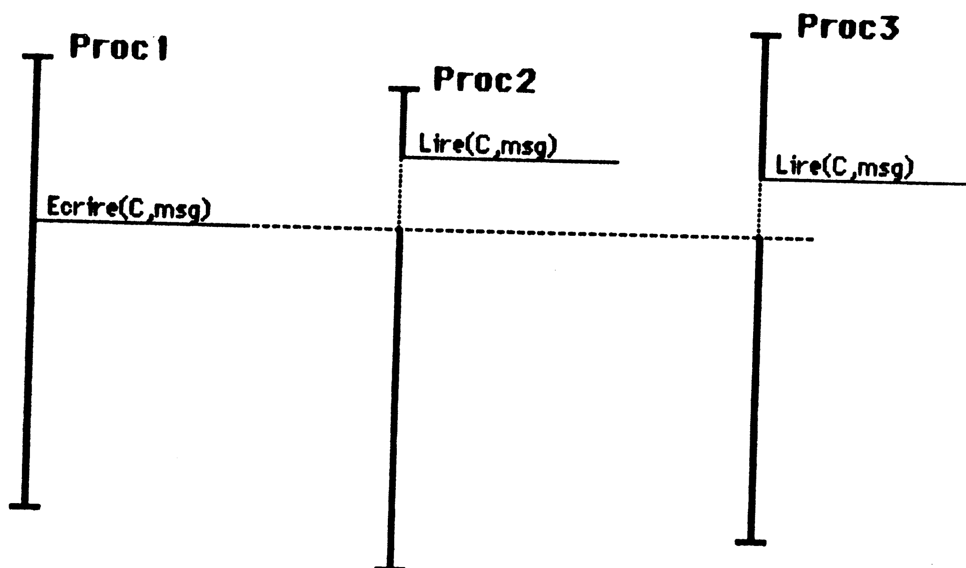


Figure 3. — Activation simultanée.

Dans le schéma précédent, trois processus "Proc1", "Proc2" et "Proc3" sont liés au même connecteur. Le processus "Proc1" est lié en écriture et les processus "Proc2" et "Proc3" en lecture. Ce connecteur attend que chaque processus connecté en lecture soit prêt à recevoir un message, puis se met en attente d'une écriture du processus "Proc1". Quand il a reçu le message du processus "Proc1", ce connecteur transforme le message reçu en un ensemble de nouveaux messages qu'il transmet aux processus qui lui sont connectés en lecture.

Exemple d'utilisation : communication entre le processus "Déplacement libre" et les processus "Générateurs de trajectoire". Les six processus "Générateurs de trajectoire" (un pour chaque degré de liberté) doivent démarrer l'échantillonage de la trajectoire au même instant avec des messages d'initialisation différents (position initiale et finale, coefficient de vitesse).

Pour réaliser ce mode de communication, nous utiliserons "nbre[lec]" processus gardés de lecture de messages (un par processus devant lire), et un processus gardé d'écriture (les processus Stop libre et Déplacement libre utiliseront le même processus gardé d'écriture). A l'initialisation du connecteur, le port du processus gardé d'écriture sera fermé et les ports des processus gardés de lecture, ouverts. L'écriture ne sera autorisée que lorsque tous les processus gardés de lecture auront été activés, puis suspendus jusqu'à la terminaison du processus gardé d'écriture. Quand une écriture est réalisée elle remet les ports des différents

processus gardés dans l'état initial avant de se terminer.

Processus gardé d'initialisation :

```
PROC_GARDE initialisation =
 SEQ
  SEND( my.écriture, simple, fermer )
  SEQ i = [ 0 FOR nbre[ lec ] ]
    SEND( my.lecture[ i ], simple, ouvert )
  n := 1 :
```

Processus gardé de lecture :

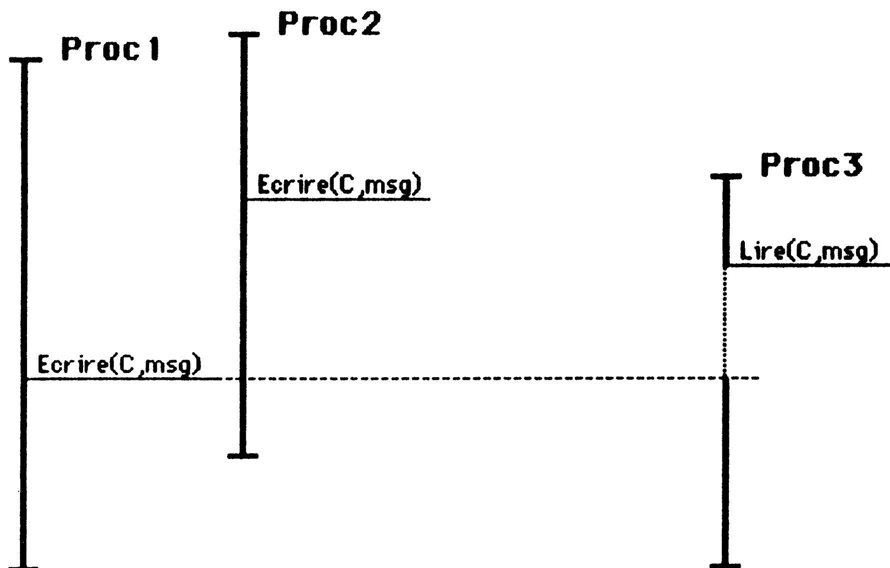
```
PROC_GARDE lecture_i ( VALUE message ) =
 SEQ
  IF
    n = nbre[ lec ]
      SEQ
        SEND( my.écriture, simple, ouvrir )
        n := 1
    n < nbre[ lec ]
      n := n+1
  SEND( my.écriture, synchronisation )
  SEND( reponse, msg[ i ] ) :
```

Processus gardé d'écriture :

```
PROC_GARDE écriture ( VALUE message ) =
 SEQ
  decouper( message, msg )
  SEND( my.écriture, simple, fermer ) :
```

2.2.2 Synchronisation multiple

Ce mode de communication permet d'activer un traitement dès l'arrivée de plusieurs messages permettant de le définir.



Dans le schéma précédent, trois processus "Proc1", "Proc2" et "Proc3" sont liés à un même connecteur. Les processus "Proc1" et "Proc2" sont liés en écriture et le processus "Proc3" en lecture. Ce connecteur attend une écriture de chaque processus qui lui est connecté en écriture, transforme l'ensemble des messages reçus en un nouveau message, qu'il transmet au processus qui lui est connecté en lecture, lorsque celui-ci lui en fera la demande.

Exemple d'utilisation : communication entre les processus "Générateurs de trajectoire" et le processus "Anticipation dynamique". Le processus "Anticipation dynamique" démarre son exécution uniquement quand il a reçu la définition complète d'un point (c'est-à-dire, pour chaque degré de liberté : position et vitesse à atteindre).

Pour réaliser ce mode de communication, nous utilisons "nbre[ecr]" processus gardés d'écriture de messages (un par processus devant écrire), et un processus gardé de lecture. A l'initialisation du connecteur, le port du processus gardé de lecture est fermé et les ports des processus gardés d'écriture, ouverts. La lecture ne sera autorisée que lorsque tous les processus gardés d'écriture auront été activés puis suspendus jusqu'à la terminaison du processus gardé de lecture. Quand une lecture est réalisée elle remet le port du processus gardé de lecture dans l'état initial (fermé) puis se termine.

Processus gardé d'initialisation :

```
PROC_GARDE initialisation =
 SEQ
  SEND( my.lecture, simple, fermer )
  SEQ i = [ 0 FOR nbre[ ecr ] ]
   SEND( my.écriture[ i ], simple, ouvert )
  n := 1 :
```

Processus gardé de lecture :

```
PROC_GARDE lecture ( VALUE message ) =
 SEQ
  SEND( réponse, msg )
  SEND( my.lecture, simple, fermer ) :
```

Processus gardé d'écriture :

```
PROC_GARDE écriture_i ( VALUE message ) =
 SEQ
  IF
   n = N
    SEQ
     SEND( my.lecture, simple, ouvrir )
     fabriquer( message, msg )
     n := 1
   n < N
    SEQ
     noter( message, file )
     n := n+1
  SEND( my.lecture, synchronisation ) :
```

Note : dans cette réalisation, le constructeur autorisait un asynchronisme entre les processus "Générateur de trajectoire" et le processus "Anticipation dynamique". Cet asynchronisme a été réalisé en introduisant entre le connecteur précédemment décrit et le processus "Anticipation dynamique", un connecteur "Buffer" qui gérait un tampon de la longueur désirée.

2.2.3 Appel procédural multiple

Il s'agit d'appels simultanés de plusieurs traitements qui s'exécuteront en parallèle. Leurs terminaisons correctes conditionnent la continuation du processus appelant. Ce mode de communication permet d'effectuer un appel de procédure multiple.

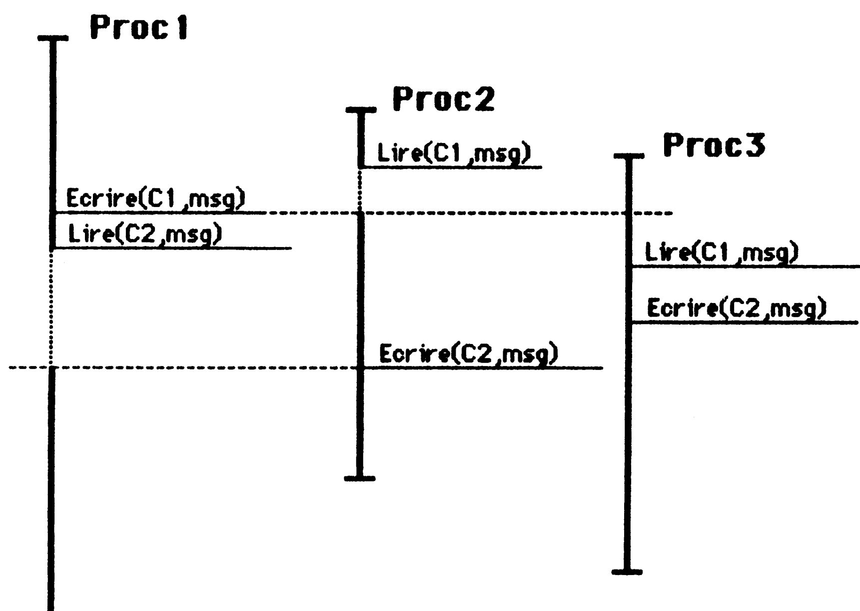


Figure 5. — Appel procédural multiple.

Dans le schéma précédent, trois processus "Proc1", "Proc2" et "Proc3" sont liés aux mêmes connecteurs. Le processus "Proc1" est lié en écriture au connecteur "C1" et en lecture au connecteur "C2". Les processus "Proc2" et "Proc3" sont liés en lecture au connecteur "C1" et en écriture au connecteur "C2". Le connecteur "C1" est du type "activation simultanée" et le connecteur "C2" du type "synchronisation multiple". Le connecteur, composé des connecteurs "C1" et "C2", attend que les processus "Proc2" et "Proc3" soient en attente de lecture, puis attend un message du processus "Proc1" qu'il transforme en deux messages pour les processus "Proc2" et "Proc3" (fonctionnement du connecteur "C1"). Ce même connecteur attend les réponses des processus "Proc2" et "Proc3", qu'il regroupe, pour former un nouveau message qu'il envoie au processus "Proc1" (fonctionnement du connecteur "C2").

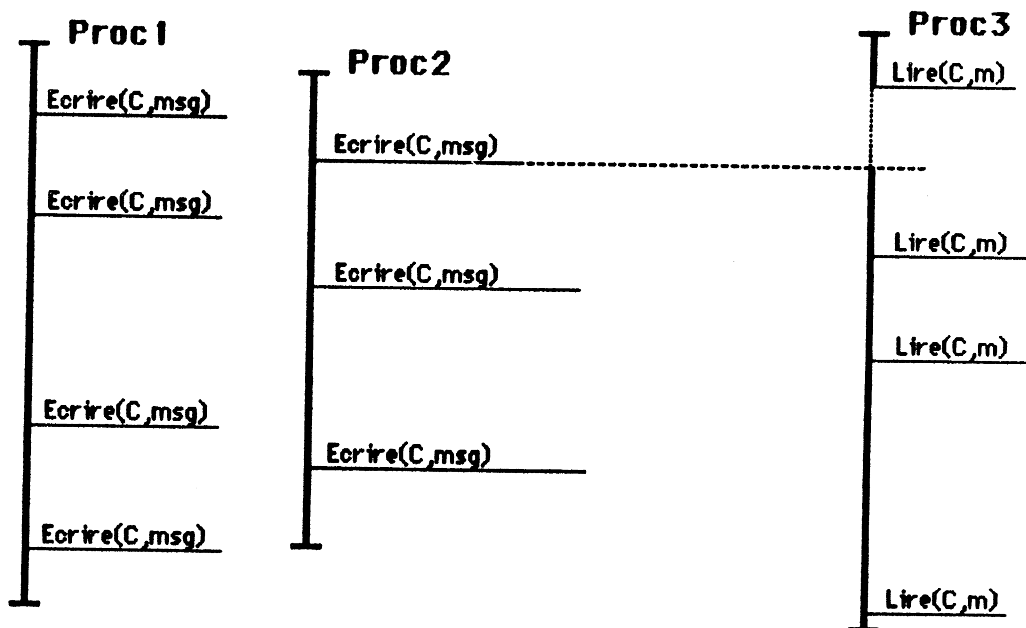
Exemple d'utilisation : communication entre le processus "Déplacement libre" et les processus "Calcul temps de parcours". Chaque processus "Calcul temps de parcours" donne, pour le degré de liberté qu'il gère, le temps de parcours estimé en fontion de la position et

de la vitesse initiale et finale et du coefficient de vitesse désiré pour l'axe concerné.

Ce type de connecteur sera réalisé par deux connecteurs, nous utiliseront pour cela une possibilité de CONKER qui permet de composer des connecteurs pour réaliser un nouveau mode de communication. Sur le premier connecteur de type "Activation Multiple", le processus faisant l'appel écrira le message d'activation, puis il se mettra en attente (par une lecture) sur un connecteur de type "Synchronisation Multiple". Par contre les processus réalisant l'appel de procédure seront en attente de lecture sur le connecteur de type "Activation Multiple", puis écriront, une fois le traitement demandé effectué, sur le connecteur "Synchronisation Multiple".

2.2.4 Prise asynchrone de valeurs

Ce mode de communication permet à un processus de lire une valeur, produite par un autre processus, sans se synchroniser sur la fin de traitement. Le processus ne lit pas obligatoirement toutes les valeurs.



Ce connecteur, auquel est lié deux processus en écriture ("proc1" et "Proc2") et un processus en lecture ("Proc3"), note continuellement le dernier message écrit par "Proc1" ou "Proc2", et pour chaque demande de lecture de "Proc3", lui fabrique un message à l'aide des derniers messages reçus.

Exemple d'utilisation : ce mode de communication est utilisé par le processus "Interpréteur" pour la connaissance de la position courante du robot. En effet l'interpréteur a besoin de connaître uniquement la dernière position connue du robot, et non pas l'historique de la trajectoire. Pour cela il doit lire sur ce type de connecteur un message, construit à partir de N messages qui sont pris au même instant sur N connecteurs élémentaires.

Processus gardé d'initialisation :

```
PROC_GARDE initialisation =
 SEQ
  SEND( my.lecture, simple, ouvert )
  SEQ i = [ 0 FOR nbre[ lec ] ]
    SEND( my.écriture[ i ], simple, ouvert ) :
```

Processus gardé de lecture :

```
PROC_GARDE lecture ( VALUE message ) =
 SEQ
  concaténer( message, msg )
  SEND( réponse, msg ) :
```

Processus gardé d'écriture :

```
PROC écriture_i ( VALUE message ) =
 noter( message, i ) :
```

2.2.5 Activation périodique de traitement

La machine LM, a besoin de pouvoir activer périodiquement des traitements. La gamme de fréquence utilisée est assez large : de 50 hz pour la surveillance des capteurs à 1000 hz pour assurer l'asservissement du robot.

Pour réaliser cela on utilise le mécanisme de CONKER qui permet d'activer périodiquement un processus gardé. Lorsque l'on désire suspendre l'activation périodique de ce processus gardé, il suffit de fermer le port correspondant.

Ce mode d'activation sera utilisé pour la gestion des capteurs et la gestion des variables d'état.

i) Gestion des capteurs et des gardes associés

Nous ne ferons pas ici une étude de la programmation de capteur, mais nous renvoyons le lecteur à [ESPIA86] pour une étude de différents capteurs utilisés en robotiques.

On trouve des gardes dans le langage LM essentiellement dans les instructions de déplacement de robots ou de manipulation d'outils terminaux.

Par exemple :

```
DEPLACER cube A destination JUSQU'A Fz>5 ;

ou

FERMER pince JUSQU'A Force>2 ;
```

Pour réaliser l'interprétation d'une telle instruction à l'aide du noyau CONKER, nous avons choisi de séparer l'interprétation du mouvement (du robot ou de l'outil terminal) du contrôle des gardes.

L'interprétation du mouvement sera réalisée par un ensemble de processus coopérants qui accepteront de traiter de façon asynchrone des événements produits par différents connecteurs.

La gestion des gardes sera réalisée par des connecteurs de CONKER. Nous avons fait le choix d'associer à chaque capteur de l'application un connecteur. Un processus gardé de ce connecteur sera chargé de l'acquisition de la valeur courante du capteur, et d'autres processus gardés, appartenant au connecteur, seront chargés de la gestion des gardes associées au capteur et produiront un événement, chaque fois que la garde deviendra vraie, à destination des processus d'interprétation de l'instruction du mouvement.

Une des stratégies possible d'évaluation des gardes est de les évaluer périodiquement (en LM, il est possible de fixer la fréquence d'évaluation).

ii) Gestion des variables d'état

De la même façon que pour la gestion des gardes, la valeur de chaque variable d'état sera véhiculée par un connecteur permettant d'en connaître la valeur courante. Ce processus sera généralement exécuté de façon périodique en utilisant ce mode d'activation.

2.3 Exemple d'interprétation de l'instruction DEPLACER

Le déplacement suivant se décompose en deux parties distinctes. La première s'effectue dans le mode LIBRE, et le déplacement est contrôlé par deux gardes, la seconde s'effectue dans le mode CARTESIEN sans aucun contrôle particulier. Entre les deux parties l'interpréteur doit gérer avec l'aide du processus "Gestion via" le changement de trajectoire.

```
DEPLACER robot VIA pt_intermédiaire EN LIBRE JUSQU'A Fz>5 OU Fy>4
               A destination      EN CARTESIEN ;
```

i) Initialisation du déplacement

L'interpréteur LM initialise le déplacement dans le mode libre. Pour cela il envoie un message de description au processus "Déplacement libre". Ce message comporte le point de départ de la trajectoire, le point d'arrivée et la vitesse maximale.

De façon simultanée l'interpréteur envoie un message au processus "Stop libre", afin que celui-ci soit prêt à traiter, par un arrêt du déplacement, l'arrivée à vrai d'une des deux gardes (le processus "Stop libre" demande aux deux connecteurs "Gestion capteurs" d'activer un traitement si le capteur dépasse le seuil précisé).

ii) Gestion de la transition entre les deux mouvements

Dès que la première partie de la trajectoire a été initialisée, l'interpréteur prépare la poursuite de la trajectoire : gestion de la transition entre les deux mouvements.

L'interpréteur envoie au processus "Gestion VIA" un message contenant la description du mouvement en cours et la description du mouvement suivant. A partir de cet instant, ce processus contrôle de façon permanente la position du robot. Arrivé à proximité du but du premier mouvement, le processus "Gestion VIA" inhibe le

déplacement et assure le changement de trajectoire à vitesse constante.

iii) Activation du deuxième mouvement

Dès que le processus "Gestion VIA" est prévenu, l'interpréteur envoie un message au processsus "Déplacement CARTESIEN", en lui indiquant le point de départ et le point d'arrivée de la trajectoire, sa vitesse et le fait qu'elle fait suite à un précédent mouvement. Le processus "Déplacement CARTESIEN" initialise le mouvement en tenant compte des données précédentes et se met en attente, prêt à relayer le processus "Gestion VIA" à la réception d'un message.

iv) Poursuite de l'interprétation

Une fois l'ensemble du déplacement initialisé, l'interpréteur poursuit l'interprétation du code LM_e.

3. Conclusion

Dans ce chapitre nous avons décrit une partie d'une implantation complète de CONKER réalisée à l'aide du système "temps réel" iRMX86. Ce travail est décrit dans le rapport de fin de contrat [MUNTE85a] et a nécessité la réalisation :

- d'un serveur pour la gestion des processus (création, destruction, attente de synchronisation, ...) en PLM86.

- d'un serveur pour la gestion des canaux (création, destruction, connexion, ...) en PLM86.

- d'un ensemble de modèle de connecteurs, dont les principaux ont été décrit en OCCAM dans ce chapitre.

Toute l'application CONKER a été écrite en PL/M86 et fait environ 6000 lignes de codes (y compris les connecteurs décrits précédemment en OCCAM et les serveurs systèmes gestionnaires de processus et gestionnaires de canaux).

le modèle CONKER réalisé, utilise comme noyau, le noyau iRMX86 auquel ont été apportées quelques modifications afin de réaliser les communications selon le modèle de CONKER. Les processus application qui réalisent l'application robotique, sont programmées en Fortran77.

Cette réalisation a permis une première évaluation du modèle CONKER, pour ce type d'application et a été à l'origine de l'implantation d'une machine LM répartie sur l'architecture César et Cléopâtre.

VI. Des exemples de connecteurs

Résumé

Après avoir présenté le modèle CONKER, et une utilisation de cette architecture pour une application robotique, ce chapitre présentera deux autres utilisations du modèle CONKER.

La première utilise l'architecture CONKER pour la construction de deux "constructeurs" du langage OCCAM sur une architecture multi-processeurs (réalisation répartie des constructeurs ALTernatif et PARallèle du langage OCCAM).

La deuxième présente un modèle de connecteur utilisé par les automaticiens pour la gestions d'événements simples et/ou combinés lors de la surveillance de procédés industriels (ces réalisations sont généralement réalisées sur de petits microprocesseurs interconnectés par un réseau local).

Ces deux réalisations, bien que présentées indépendement du chapitre précédent, auraient leur place dans la réalisation d'un système robotique complexe nécessitant l'utilisation de plusieurs calculateurs. En effet, il serait alors nécessaire de pouvoir contrôler l'exécution parallèle de différentes tâches surtout dans le cas de coopération de deux robots, ou d'interconnexion entre un système de robotique et un système de vision.

La deuxième de ces réalisations prend immédiatement sa place, dès que les événements attendus par l'application robotique deviennent combinés. Par exemple pour le suivi de surface non plane, la trajectoire s'effectue à force constante, cette force étant la résultante de trois composantes orthogonales.

1. Implémentation répartie de constructeurs OCCAM

Le modèle CONKER permet l'implantation aisée d'un langage parallèle de type OCCAM sur différentes machines, car les connecteurs qu'il offre permettent de réaliser des mécanismes de synchronisation multiples. Cette courte partie va nous permettre de montrer comment nous avons utilisé CONKER pour implémenter le constructeur PARallèle et le constructeur ALTernatif du langage OCCAM sur une architecture multi-processeurs (SM90) et sur un système spécifique SMX. Pour plus de précision nous invitons le lecteur à lire [TRICO87a]

Remarque : Cette réalisation bien que spécifique, peut être étendue à différentes réalisations faites à l'aide du noyau CONKER, ou d'un autre noyau qui offre la même interface.

Nous ne présenterons pas ici, l'implémentation réparti du constructeur séquentiel, bien que celle-ci représente le cas de migration d'un processus séquentiel au cours de son exécution d'un site vers un autre.

1.1 Principe de l'implémentation du PAR

Les règles pour le partage des "variables" entre des processus PARallèle du langage OCCAM permettent :

- à un processus du bloc PARallèle, d'utiliser les variables du processus englobant le constructeur, comme des constantes initialisées à la valeur qu'elles contenaient avant le début de l'exécution du constructeur PARallèle.
- à un seul processus du bloc PARallèle, d'utiliser une variable du processus englobant, comme une variable initialisée à la valeur qu'elle contenait avant le début de l'exécution du constructeur PARallèle. D'une fac,on schématique, ce processus est, pour cette variable, un processus qui se poursuit de fac,on séquentielle avec le processus précédant le constructeur PARallèle. Dans ce cas, les autres processus du bloc parallèle ne peuvent plus l'utiliser.

Dans les implantations que nous avons réalisées, le compilateur OCCAM effectue une analyse statique lui permettant de détecter de quelles "constantes" et de quelles "variables" doivent hériter les processus du bloc parallèle. Une fois cette phase d'analyse effectuée, le noyau CONKER et le processus englobant le constructeur PAR, se chargent de transmettre aux différents processus les valeurs d'initialisation des "constantes" et des "variables" qu'ils héritent, puis de récupérer les valeurs des variables à la fin de l'exécution du bloc PAR.

Exemple de programme :

```
PROC exemple =
 VAR a, b :
 SEQ
  a := 1
  b := 2
  CHAN connecteur :
  PAR
   VAR c :
   SEQ
    c := 4
    c := b + c
    connecteur ! c
   connecteur ? a
  SCREEN ! a :
```

Ce processus exemple se décompose en quatre processus qui s'exécutent de fac,on séquentielle.

```
SEQ
 P1 = { a := 1 }
 P2 = { b := 2 }
 P3 = { processus construit }
 P4 = { SCREEN ! a }
```

Nous allons nous intéresser uniquement aux mécanismes d'héritage des variables lors de l'activation du processus P3 et lors de sa terminaison.

Le processus P3 est composé de deux processus :

```
PAR
 P3.1 = { processus construit }
 P3.2 = { connecteur ? a }
```

i) Héritage lors de l'activation

Avant de créer les différents processus composant le processus P3, le processus englobant (le processus exemple) crée un connecteur réparti qui permettra de véhiculer l'ensemble des valeurs qui seront héritées par les processus composants P3.

ii) Héritage lors de la terminaison

Avant de terminer, chaque processus composant de P3 réécrit dans le connecteur les nouvelles valeurs des variables qu'il a héritées. Si deux processus modifient la même variable, le connecteur génère un événement d'erreur et le processus P3 termine de fac,on incorrecte (violation des règles d'héritage).

iii) code du processus exemple

```
SEQ
  a := 1
  b :=2
  CREER_CONNECTEUR( P_connecteur, type=héritage )
  ECRIRE( P_connecteur, ( a, b ) )
  CREER_PROCESSUS( P3.1, P3.2 )
  ATTENTE_TERMINAISON( P3.1 et P3.2 )
  LIRE( P_connecteur, ( a, b ) )
  DETRUIRE_CONNECTEUR( P_connecteur )
```

iv) code d'un processus composant le bloc parallèle

```
SEQ
  Pour toutes variables héritées
    LIRE( P_connecteur, variable )
  code séquentiel
  Pour toutes variables héritées et modifiées
    ECRIRE( P_connecteur, variable )
  TERMINER( Pid )
```

1.2 Implémentation du ALT

La réalisation du processus ALTernatif d'OCCAM peut se faire en utilisant la primitive de CONKER, qui permet de se mettre en attente sur un ensemble d'événements.

Le constructeur ALTernatif d'OCCAM se conc,oit sur des connecteurs possédant un seul processus émetteur et un seul processus récepteur, et assurant des communications par rendez-vous. Pour réaliser cette alternative, le processus se met en attente sur un ensemble d'événements -- écriture --, pouvant survenir sur les différents connecteurs. Le processus fait alors, localement, le choix entre les différents connecteurs sur lequel un message est prêt à être écrit.

Le noyau CONKER permet d'étendre la notion d'alternative :

1) il est possible d'introduire des sorties dans les gardes (l'événement à attendre étant alors une lecture). Cependant la restriction habituelle demeure : à savoir que si sur un même connecteur réalisant un rendez-vous, deux processus veulent réaliser une alternative, un avec une garde de sortie et l'autre une garde d'entrée, l'algorithme d'implémentation introduit un interblocage. Chaque processus effectue le choix localement, le système ne permet pas à deux processus de se mettre d'accord pour utiliser le bon canal et résoudre ainsi chacun de leur côté l'alternative de fac,on cohérente).

2) si le connecteur ne réalise pas un rendez-vous entre deux processus, mais une file (quelque soit la relation d'ordre utilisée pour ordonner les messages), le système permet d'étendre la notion d'alternative. Ainsi, l'alternative sera pour le processus qui désire émettre de savoir si le connecteur n'est pas plein (et l'evénement attendu sera : "connecteur non plein") et pour le processus qui désire recevoir un message, si le connecteur n'est pas non vide ?

- si le connecteur précédent est point à point on pourra réaliser des gardes d'entrée et de sortie.

- si le connecteur relie un processus émetteur et N processus récepteurs, il ne pourra réaliser que des gardes de sortie (restriction indroduite parce que le choix de la communication à effectuer est faite localement par chaque processus (voir commentaire 1, ci-dessus).
- si le connecteur relie un processus récepteur et N processus émetteurs, il ne pourra réaliser que des gardes d'entrée (restriction indroduite parce que le choix de la communication à effectuer est faite localement par chaque processus (voir commentaire 1, ci-dessus).

2. Connecteurs de gestion d'événements

Pour pouvoir être utilisé de fac,on agréable, CONKER nécessite l'écriture de modèles de connecteurs permettant par la suite de construire les connecteurs nécessaires. Dans ce travail nous ne nous sommes pas attachés à définir quel devrait être l'ensemble des connecteurs de base que CONKER se doit d'offrir, mais nous avons voulu proposer un ensemble de connecteurs utilisables pour l'écriture de diverses applications.

Dans les chapitres précédents et dans l'annexe jointe, nous avons présenté différents connecteurs qui réalisent différent mode de synchronisation et de communication entre des processus. Afin de décrire complètement les possibilités de CONKER, il manquait à ce tour d'horizon, quelques connecteurs gérants des événements. C'est cette lacune que nous comblons ici.

2.1 Gestion des événements complexes

La description d'une application à l'aide d'un langage de structuration se fait généralement en trois phases [GRIES80]. On définit d'abord les outils qu'utilisent les processus, puis les processus et enfin la structure de contrôle. Ces trois phases sont généralement modifiables indépendament les unes des autres. En effet, pour les automaticiens, les procédés qu'ils doivent commander sont très souvent connus de fac,on incomplète. Le concepteur doit donc s'adapter à l'évolution de la connaissance du procédé qu'il a à traiter.

Dans les langages de structuration (SYGARE [NONN79], GAELIC [MADAU77], STR/MP [LADET82]) pour la programmation d'application "temps réel", il existe de nombreuses possibilités pour exprimer l'activation d'un traitement dès l'arrivée d'un événement.

Avec CONKER, le concepteur d'application adopte une démarche similaire : structuration de l'application en processus communicants, décomposition d'un processus en traitement séquentiel et traitement asynchrone d'événements, puis définition de la structure de contrôle (choix des protocoles d'échange et écriture des connecteurs). Mais pour que CONKER puisse être utilisé agréablement par les automaticiens, il est nécessaire d'étendre les possibilités d'expression des événements.

i) Notion d'événement

Dans ce type d'application on peut distinguer deux types d'événements :

- les événements propre au système informatique : fin d'entrée-sortie par exemple.

- les événements qui correspondent à un changement de l'état du procédé : vanne d'alimentation ouverte, robot en déplacement...

Si l'on peut distinguer deux types d'événements, on constate que leur nature n'est pas très éloignée [DESCH77]. Dans l'un et l'autre cas, il s'agit de changement d'état, tels que "libération d'une ressource" ou "franchissement d'un seuil". Que la grandeur provoquant l'événement par changement d'état soit informatique ou non importe peu. Nous pouvons donc assimiler les deux types d'événements et nous les traiterons de la même manière.

Si le traitement des événements et la synchronisation des processus sont toujours effectués par le système informatique dans une application "temps critique", l'utilisateur doit pouvoir gérer lui-même les événements (en particulier, ceux liés au procédé). Les connecteurs de CONKER, permettent d'offrir à l'utilisateur des primitives de synchronisation qui donnent la possibilité de gérer les événements qu'il aura lui-même définis.

Un événement pourra prendre deux valeurs logiques : il sera arrivé ou non arrivé et le système associera un événement différent à chaque occurence. Ainsi le temps est découpé en tranche : pour chaque occurence d'un événement il y a l'instant passé ou' l'événement n'est pas arrivé et l'instant futur ou' l'événement est arrivé. Le système ne connait pas l'instant présent, ce qui peut se dire : le système ne peut pas savoir qu'un événement se produit.

Les événements présentés ci-dessus seront rangés en deux catégories : ceux qui sont **rémanents (ou mémorisés),** le système conserve une trace de toutes leurs occurences et ceux qui sont **fugitifs (ou non mémorisés),** et qui sont perdus si lorqu'ils sont déclenchés aucun processus ne les attend (l'application ne garde pas trace de ces événements).

ii) Définition des modes d'activation des traitements

L'origine du temps est pour chaque connecteur l'instant de sa création. Depuis son existence dans le système et jusqu'à sa destruction par un processus, un connecteur perc,oit les occurences des événements qu'il surveille.

D'une fac,on générale, on peut vouloir activer un traitement pour un événement survenu dans le passé ou qui va survenir dans le futur. Ce traitement peut se faire pour une occurence de l'événement ou pour chaque occurence. La construction a la forme suivante :

(PASSE)
POUR (CHAQUE) <événement> ou FAIRE <traitement> ;
(FUTUR)

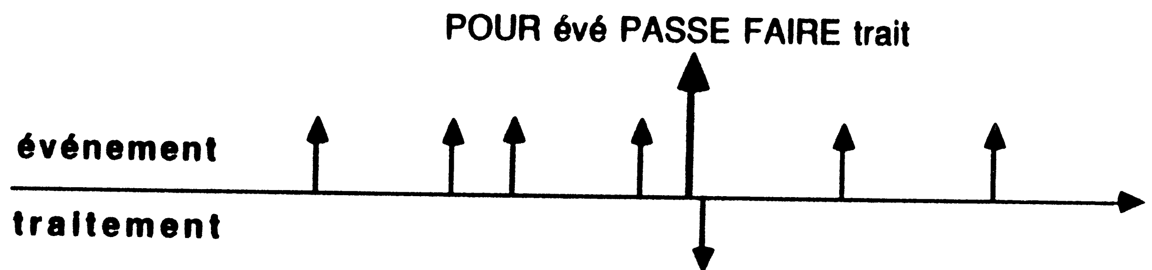
La distinction entre le passé et le futur se fait par rapport à la date à laquelle s'effectue la connexion entre l'événement attendu et le traitement à activer.

Note1 : si l'événement est rémanent le système peut s'intéresser à toutes les occurences de l'événement ou simplement à son apparition. Dans le premier cas il s'agit de lancer une action si l'événement a eu lieu (quelque soit le nombre d'occurences de cet événement), dans le deuxième cas l'action devra être lancée autant de fois qu'il y a eu d'occurences de l'événement. Ces deux types d'événement seront similaires pour CONKER : dans le premier cas, le système ne s'intéressera qu'à la première apparition de l'événement jusqu'à son traitement qui l'annulera, et dans le second cas, le système entretiendra un compteur permettant de compter le nombre d'échéances survenues. C'est le connecteur implémentant l'événement qui fera la distinction entre l'événement dont chaque occurence à une importance, et celui pour lequel une seule occurence sera comptabilisée.

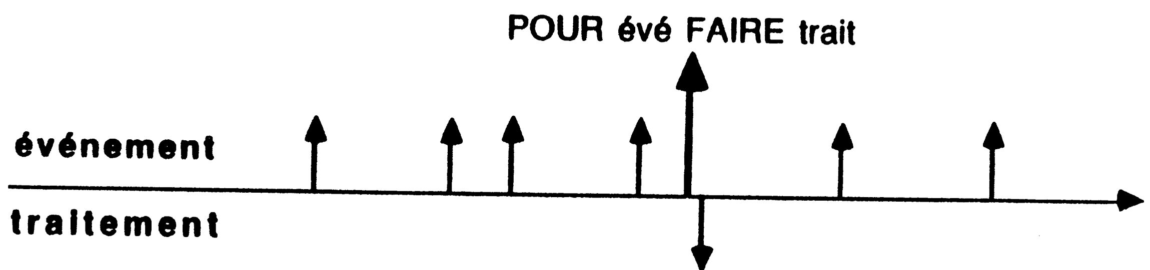
Note2 : si l'événement est fugitif le système ne s'intéresse qu'aux occurences futures et une partie de l'écriture ci-dessus n'a aucun sens. Par exemple, "POUR évé PASSE FAIRE trait" est équivalent au processus SKIP si l'événement est fugitif.

iii) Exemple d'utilisation

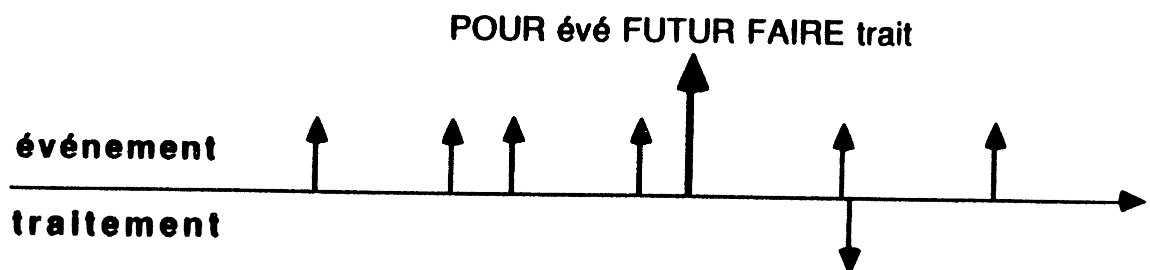
POUR événement PASSE FAIRE traitement ; signifie que l'application désire activer une seule fois le traitement indiqué si l'événement attendu est déjà survenu une fois dans le passé. Si le système n'a pas perc,u une occurence de cet événement, alors cette construction est sans effet. Cette construction n'a aucun sens si l'événement attendu est fugitif, et cette construction sera sans effet.



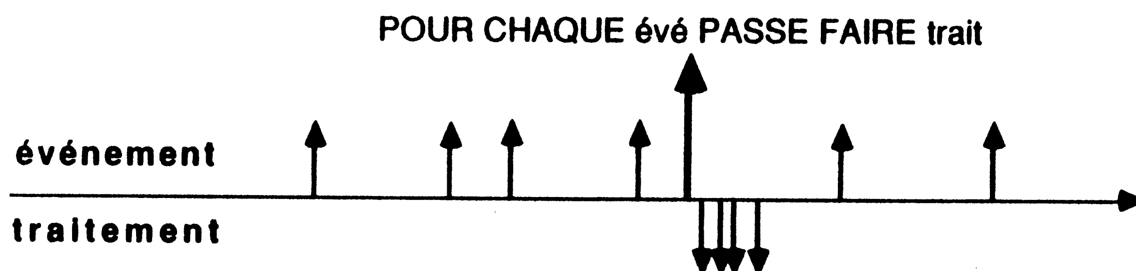
POUR événement FAIRE traitement ; signifie que l'application désire activer une seule fois le traitement indiqué, que l'occurence de l'événement apparaisse dans le futur ou qu'elle soit déjà apparue. Si l'événement attendu est fugitif, cette construction est équivalente à la construction suivante "POUR événement FUTUR FAIRE traitement".



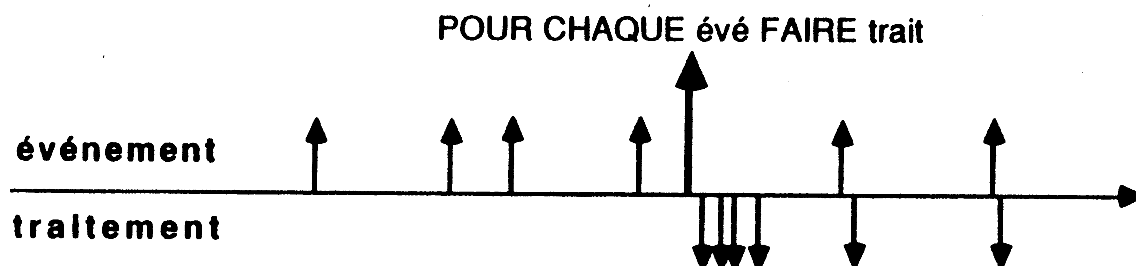
POUR événement FUTUR FAIRE traitement ; signifie que l'application désire activer une seule fois le traitement indiqué dès qu'une occurence de l'événement survient.



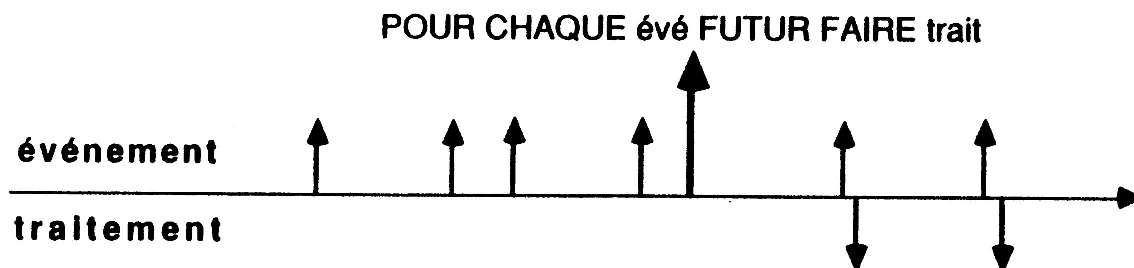
POUR CHAQUE événement PASSE FAIRE traitement ; signifie que l'application désire activer un traitement pour chaque occurence de l'événement déjà connue. Si le système n'a pas perc,u une occurence de cet événement, alors cette construction est sans effet. Cette construction n'a aucun sens si l'événement attendu est fugitif, cette construction sera alors sans effet.



POUR CHAQUE événement FAIRE traitement ; signifie que l'application désire activer un traitement pour chaque occurence de l'événement déjà connue et pour toutes les occurences à venir. Si l'événement attendu est fugitif, cette construction est équivalente à la construction suivante "POUR CHAQUE événement FUTUR FAIRE traitement".



POUR CHAQUE événement FUTUR FAIRE traitement ; signifie que l'application désire activer un traitement pour toutes les occurences de l'événement à venir.



iv) Evénements exclusifs

Dans une application "environnementale", il est nécessaire de pouvoir faire des traitements d'événements de fac,on exclusive. Un processus de l'application se met en attente sur différents événements et choisit de traiter uniquement le premier arrivé.

POUR ev1 FUTUR FAIRE traitement1
OU
POUR ev2 FAIRE traitement2
OU
POUR ev3 PASSE FAIRE traitement3 ;

L'activation d'un des trois traitements demandés, désactives les deux autres.

Exemple : un contrôleur de robot mobile peut se mettre en attente sur trois événements à venir. Selon que l'obstacle rencontré se trouve à droite ou à gauche ou devant, le contrôleur réagira différemment.

v) Définition des événements composés

Un événement peut aussi être une donnée composée d'autres événements combinés par les opérateurs ET et OU.

ev = ev1 ET (ev2 OU ev3)

Un événement composé est arrivé, si l'évaluation de l'expression logique donne un état arrivé. Pour savoir si l'événement, correspondant à l'expression logique, est fugitif ou rémanent, on applique des règles de composition immédiate :

OU	Fugitif	Remanent
Fugitif	Fugitif	1
Remanent	1	Remanent

ET	Fugitif	Remanent
Fugitif	2	Remanent
Remanent	Remanent	Remanent

1 : dans ce cas l'événement combinaison possède les deux types : il sera rémanent si l'événement arrivé est rémanent ; il sera fugitif, si l'événement arrivé est fugitif.

2 : la combinaison ET de deux événements fugitifs est impossible, sauf si le système arrive à spécifier que deux événements sont simultanés, et, dans ce cas, l'événement combiné est de type fugitif.

2.2 Réalisation de ces connecteurs

Un processus qui veut exécuter la commande "POUR <CHAQUE> événement <PASSE, ou FUTUR> FAIRE traitement" envoi au connecteur qui gère l'événement un message contenant : l'identification de l'événement attendu (numéro du canal associé au processus gardé du connecteur qui gère cet événement, ce numéro a été obtenu lors de l'étape de

liaison décrite à la fin du chapitre 3), l'identification du traitement à exécuter (numéro du canal associé au processus gardé qui assurera le traitement de l'événement pour le compte du processus), et un descriptif de la commande permettant de présiser : CHAQUE, PASSE, FUTUR.

Le connecteur gérant ces événements comportera au moins les quatre processus gardés décrits ci-après. Il peut traiter des événements fugitifs ou rémanents. Pour les événements à niveau, i. e. qui ont une durée, le connecteur sera légèrement différent puisque cet événement pourra être traduit par deux événements : début de l'événement, fin de l'événement.

```
CONNECTEUR gestion_événement (VALUE nombre) =
 VAR type, compteur :
 VAR chaque[ nombre ], trait[ nombre ], temps[ nombre ] :

 PROC_GARDE initialisation =
  SEQ
   IF
     message = rémanent
       type := rémanent
     message = fugitif
       type := fugitif
   compteur := 0
   SEQ i = [ 0 FOR nombre ]
     trait[ i ] := ANY
   SEND( noyau, simple, (événement, ouvert) )
   SEND( noyau, simple, (demande_traitement, ouvert) )
   SEND( noyau, simple, (suppression_traitement, ouvert) ) :

 PROC_GARDE événement =
  SEQ
   IF
     type = rémanent
       compteur := 1
     type = rémanent
       compteur := compteur+1
   SEQ i = [ 0 FOR nombre ]
     IF
       (trait[ i ]<>ANY)
         SEQ
           SEND( trait[ i ], simple, ANY )
           IF
             NOT CHAQUE[ i ]
               trait[ i ] := FALSE :
```

```
PROC_GARDE demande_traitement =
  VAR j :
  SEQ
    noter( message, j )
    IF
      (temps[ j ]<=0) AND chaque[ j ]
        SEQ i = [ 0 FOR compteur ]
          SEND( trait[ j ], simple, ANY )
      (temps[ j ]<=0) AND NOT chaque[ j ]
        SEQ
          SEND( trait[ j ], simple, ANY )
          trait[ i ] := FALSE :

PROC_GARDE suppression_traitement =
  SEQ
    trait[ message ] := ANY :
```

3. Remarques

Ce chapitre, terminant la présentation de CONKER, nous a permis de présenter deux points mis a l'écart dans les chapitres précédents.

Le premier a été la réalisation répartie, à l'aide de CONKER, des constructeurs OCCAM. Le constructeur ALTernatif réparti est nécessaire chaque fois que les processus doivent exécuter un traitement sur l'arrivée d'événements produits sur différents sites. Le constructeur PARallèle permet la réalisation du contrôle de l'exécution répartie de processus. Ce contrôle est nécessaire chaque fois que les processus s'exécutant en parallèle coopèrent (contrôle du déplacement de deux robots manipulant le même objet, par exemple).

Le deuxième point, nous a permis d'aborder la réalisation de connecteurs pour la gestion des événements. Ces connecteurs sont nécessaires pour l'écriture d'application "temps critique", ou l'environnement interagit avec l'application.

Pour terminer, nous allons présenter un connecteur qui permet de mémoriser la dernière valeur d'une donnée physique (température, pression, heure, ...).

Cette valeur est transmise au connecteur par un processus qui scrute cette donnée (processus capteur). Le connecteur permet à un ensemble de processus de lire la dernière valeur écrite, et peut envoyer, à un (autre) processus, un message d'alarme chaque fois qu'elle dépasse le seuil qu'il a fixé.

```
CONNECTEUR gestion_événement (VALUE nombre) =
  VAR valeur :
  VAR trait, seuil :

  PROC_GARDE initialisation =
   SEQ
    seuil := ∞
    trait := ANY
    valeur := 0
    SEND( noyau, simple, (lecture, fermer) )
    SEND( noyau, simple, (écriture, ouvert) )
    SEND( noyau, simple, (demande_traitement, ouvert) )
    SEND( noyau, simple, (suppression_traitement, fermer) ) :

  PROC_GARDE lecture =
   SEND( ANY, réponse, valeur ) :

  PROC_GARDE écriture =
   SEQ
    IF
     valeur = 0
      SEND( noyau, simple, (lecture, ouvrir) )
    valeur := message
    IF
     valeur > seuil
      SEND( trait, simple, ANY ) :
```

```
PROC_GARDE demande_traitement =
 SEQ
  seuil := message.seuil
  trait := message.trait
  SEND( noyau, simple, (demande_traitement, fermer) )
  SEND( noyau, simple, (suppression_traitement, ouvert) )
  IF
   (seuil<=valeur)
    SEND( trait, simple, ANY ) :

PROC_GARDE suppression_traitement =
 SEQ
  SEND( noyau, simple, (demande_traitement, ouvert) )
  SEND( noyau, simple, (suppression_traitement, fermer) )
  seuil := ∞
  trait := ANY :
```

Conclusion et Perspectives

Après avoir dans le chapitre 1 présenté différents systèmes répartis et montré comment CONKER s'insérait dans ce foisonnement de propositions, nous avons dans les chapitres 3 et 4, présenté le modèle CONKER (et quelques règles d'implémentation) et sa filiation avec les travaux effectués par HOARE sur CSP.

Le chapitre suivant, plus directement tourné vers les problèmes d'implémentation (et de description de connecteur) nous a permis de cerner un peu mieux les problèmes "temps critique" des applications robotiques, et de montrer comment CONKER permettait de résoudre agréablement les problèmes de communication dans cette classe d'application.

Le chapitre 6, nous a permis d'introduire de nouveaux types de connecteurs, nécessaires à tous ceux qui utilisent l'informatique pour commander et contrôler des processus industriels.

Avec l'expérience acquise lors de ces réalisations, nous pouvons à présent définir, en guise de conclusion, une méthodologie d'utilisation de CONKER :

- la première étape consiste à décomposer l'application en processus. Chaque processus constitue une entité possédant des variables propres, et coopérant avec d'autres processus par échange de messages selon des protocoles à déterminer.
- une seconde étape consiste à programmer de façon séquentielle l'ensemble des processus de traitement, en définissant pour chacun d'eux l'ensemble des communications/synchronisations nécessaires (soit avec les autres processus, soit avec l'environnement).
- la troisième étape doit permettre de caractériser les connecteurs utilisés par l'application. Un connecteur est une entité spécialisée qui se charge du transport des messages et de la synchronisation des processus qui lui sont connectés. Si le connecteur est réparti (les processus qu'il relie sont sur différents sites), il assure alors le transport des messages entre ces sites.
- l'étape suivante permet au programmeur de préciser les traitements désirés liés à l'environnement. Celui-ci interagit avec l'application par échange de messages ou de signaux : des connecteurs spécifiques permettent la communication avec l'environnement. A l'arrivée de chaque message, ces connecteurs peuvent activer un traitement qui sera effectué dans le contexte d'un processus donné.
- une autre étape peut être constituée par la programmation. Si c'est nécessaire, des protocoles de reprise en cas de panne ou d'erreur pourront être utilisés par chaque entité effectuant une communication.

- puis, si le programmeur le désire, il pourra vérifier un certain nombre d'assertions écrites dans un calcul dérivé de CSP et basées sur les connecteurs (protocoles réalisés effectivement par les connecteurs décrits) et sur les processus (correction partielle ou totale de l'application).

Ce projet ne s'achève pas avec cette thèse et se poursuit dans différentes directions :

- **Programmation du temps.** Cette étape doit nous permettre de définir un langage de programmation "temps-critique", proche du langage OCCAM par ses primitives de communication et, reposant sur une extension du modèle CSP de HOARE. Nous pouvons renvoyer le lecteur à [MUNTE85b] pour une première approche du modèle et à [MUNTE86b] pour la définition de la sémantique du langage.

- **Synthèse automatique de connecteurs.** Construire pour chaque application utilisant CONKER une base de connecteurs spécifiques nous paraît aberrant. Nous proposons de réaliser pour CONKER une base "minimale" de connecteurs et par la suite de construire, de façon incrémentale les connecteurs nécessaires à l'aide de cette base. Nous avons déjà utilisé cette approche pour construire le connecteur "Appel de procédure multiple" à l'aide des connecteurs "Synchronisation Multiple" et "Activation Simultanée" et pour désynchroniser les processus "Générateur de trajectoire" et le processus "Anticipation dynamique" (dans le 4ème chapitre, consacré au langage LM) mais il reste à systématiser cette approche.

- **Allocation de processus et routage.** Cette recherche devrait nous permettre de répondre à la question : "comment allouer, de la meilleure façon possible, les processus aux processeurs ?" La meilleur façon est celle qui permet de minimiser certains paramètres spécifiques d'une application particulière (parallélisme, communication, temps d'exécution, ...)

- **Construction d'outils systèmes pour une architecture multi-Transputer reconfigurable.** D'ici peu, il sera possible de posséder un réseau reconfigurable de Transputers. Il reste à construire un ensemble d'outils systèmes afin de pouvoir exploiter au mieux les caractéristiques de ce type d'architecture.

Annexe : Base de connecteurs

1. Exemple de connecteurs

Nous allons, dans cette annexe, décrire à l'aide d'une extension du langage OCCAM différent types de connecteurs. Certains seront d'un type simple, utilisés habituellement dans les applications informatiques, d'autres nous permettront de montrer quelques types non triviaux permettant de résoudre agréablement certains problèmes.

1.1 Connecteur rendez-vous

Ce connecteur, qui ne peut pas être réparti, dans son implèmentation actuelle, réalise un rendez-vous, au sens de CSP, entre deux processus : l'un sera l'émetteur du message et l'autre, le récepteur. Contrairement aux canaux d'OCCAM, ce connecteur permet de choisir l'émetteur du message parmis N émetteurs potentiels (i. e. les N processus liés au connecteur en écriture), et le processus récepteur parmis M récepteurs potentiels (i. e. les M processus liés au connecteur en lecture).

Algorithme : à chaque instant, le connecteur accepte un message en écriture si et seulement si un processus est prêt à le recevoir (i. e. un processus attend un message en lecture).

Lors de son initialisation, le connecteur ouvre le port correspondant au processus gardé de lecture, et ferme celui correspondant au processus gardé d'écriture.

La première demande de lecture d'un message entraîne l'ouverture du port associé au processus gardé d'écriture et, une suspension de l'exécution de ce processus gardé jusqu'à la terminaison du processus gardé d'écriture (cette suspension est exécutée par la commande de demande de synchronisation).

La première demande d'écriture entraîne la fermeture du port associé au processus gardé d'écriture. Lorsque le processus gardé d'écriture termine, le noyau réveille le processus gardé de lecture. Celui-ci envoie au processus récepteur le message attendu et demandé précédemment.

Note : à un instant donné, il ne peut y avoir qu'une seule demande de lecture acceptée par le connecteur. La demande suivante ne sera acceptée que lorsque le connecteur aura délivré le message précédemment attendu.

```
CONNECTEUR rendez_vous =
  VAR buffer :

  PROC_GARDE initialisation =
    SEQ
      conker ! ( lecture, ouvrir )
      conker ! ( écriture, fermer ) :

  PROC_GARDE lecture =
    SEQ
      conker ! ( écriture, ouvrir )
      conker ! ( synchro, écriture )
      conker ! ( réponse, buffer )
      I := I+1 :
```

```
PROC_GARDE écriture( VALUE message ) =
 SEQ
  buffer := message
  conker ! ( écriture, fermer ) :
```

1.2 Connecteur égalité

Ce connecteur réalise un tampon de longueur bornée : l'ensemble des messages reçus par le connecteur est égal à l'ensemble des messages qu'il a émis plus l'ensemble des messages qu'il stocke. Ce connecteur réalise ce protocole entre un ensemble de processus émetteurs et un ensemble de processus récepteurs. Chaque message reçu par le connecteur, n'est émis par celui-ci qu'une seule fois.

Algorithme : lorsque le connecteur est vide, celui-ci ne peut accepter que des demandes d'écriture ; lorsqu'il est plein, que des demandes de lecture. Si le connecteur n'est pas dans un de ces deux cas, il accepte des demandes de lecture et d'écriture, il les traite dans leur ordre d'arrivée. Les messages reçus pourraient être réémis dans n'importe quel ordre. Nous avons choisi d'implémenter la politique "FIFO" : premier message arrivé, premier message servi. D'autres choix auraient pu être possibles : "LIFO" (voir paragraphe suivant), à priorité, en fonction des dates de péremption des messages (non prises en compte ici).

Lors de l'initialisation, le connecteur ferme le port associé au processus gardé de lecture et ouvre celui associé au processus d'écriture.

A chaque demande de lecture, le connecteur ouvre le port associé au processus gardé d'écriture si le connecteur est plein, renvoie au processus qui lui en a fait la demande, le plus ancien message qu'il ait reçu, et ferme le port associé au processus gardé de lecture si le connecteur est vide.

A chaque demande d'écriture, le connecteur ouvre le port associé au processus gardé de lecture si le connecteur est vide, note dans son tampon "buffer" le message qu'il vient de recevoir, et ferme le port associé au processus gardé d'écriture si le connecteur est plein.

```
CONNECTEUR égalite( VALUE i ) =
 VAR buffer[ i ] :
 VAR l, e :

 PROC_GARDE initialisation =
  SEQ
   l := 0
   e := 0
   conker ! ( lecture, fermer )
   conker ! ( écriture, ouvrir ) :
```

```
PROC_GARDE lecture =
 SEQ
  IF
   e = (l + i)
    conker ! ( écriture, fermer ) :
  conker ! ( réponse, buffer[ (li) ] )
  l := l+1
  IF
   e = l
    conker ! ( lecture, fermer ) :

PROC_GARDE écriture( VALUE message ) =
 SEQ
  IF
   e = l
    conker ! ( lecture, ouvrir )
  buffer[ ei ] := message
  e := e+1
  IF
   e = (l + i)
    conker ! ( écriture, fermer ) :
```

1.3 Connecteur pile

Ce connecteur gère une pile ou N processus émetteurs peuvent venir déposer un message et M processus récepteurs peuvent venir en chercher.

Algorithme : lorsque le connecteur est vide, celui-ci ne peut accepter que des demandes d'écriture ; lorsqu'il est plein, que des demandes de lecture. Si le connecteur n'est pas dans un de ces deux cas, il accepte des demandes de lecture et d'écriture, et les traite dans leur ordre d'arrivée. Les messages sont émis vers les processus demandeurs dans l'ordre contraire à celui de leur d'arrivée.

Lors de l'initialisation, le connecteur ferme le port associé au processus gardé de lecture et ouvre celui associé au processus d'écriture.

A chaque demande de lecture, le connecteur ouvre le port associé au processus gardé d'écriture si le connecteur est plein, renvoie au processus qui lui en a fait la demande le message le plus récent qu'il a reçu, et ferme le port associé au processus gardé de lecture si le connecteur est vide.

A chaque demande d'écriture, le connecteur ouvre le port associé au processus gardé de lecture si le connecteur est vide, note dans son tampon "buffer" le message qu'il vient de recevoir, et ferme le port associé au processus gardé d'écriture si le connecteur est plein.

```
CONNECTEUR pile( VALUE i ) =
 VAR buffer[ i ] :
 VAR s :

 PROC_GARDE initialisation =
  SEQ
   s := 0
   conker ! ( lecture, fermer )
   conker ! ( écriture, ouvrir ) :

 PROC_GARDE lecture =
  SEQ
   IF
    s = i
     conker ! ( écriture, ouvrir )
   conker ! ( réponse, buffer[ s-1 ] )
   s := s-1
   IF
    s = 0
     conker ! ( lecture, fermer ) :

 PROC_GARDE écriture( VALUE message ) =
  SEQ
   IF
    s = 0
     conker ! ( lecture, ouvrir )
   buffer[ s ] := message
   s := s+1
   IF
    e = i
     conker ! ( écriture, fermer ) :
```

1.4 Connecteur aléatoire

Ce connecteur réalise le protocole utilisé pour gérer une horloge. Il renvoie au processus récepteur le dernier message reçu, si aucun message n'a été écrit dans le connecteur la lecture est impossible. Il est possible de consommer plusieurs fois le même message.

Lors de l'initialisation, le connecteur note qu'il n'a pas encore reçu de message, ferme le port associé au processus gardé de lecture et ouvre celui associé au processus d'écriture.

A chaque demande de lecture, le connecteur renvoie au processus demandeur, le dernier message reçu.

A chaque demande d'écriture, le connecteur ouvre le port associé au processus gardé de lecture si le connecteur est vide, et note dans son tampon "buffer" le message qu'il vient de recevoir.

```
CONNECTEUR aleatoire =
 VAR buffer :

 PROC_GARDE initialisation =
  SEQ
   buffer := ANY
   conker ! ( lecture, fermer )
   conker ! ( écriture, ouvrir ) :

 PROC_GARDE lecture =
  conker ! ( réponse, buffer ) :

 PROC_GARDE écriture( VALUE message ) =
  SEQ
   IF
    buffer = ANY
     conker ! ( lecture, ouvrir )
   buffer := message :
```

1.5 Connecteur rafraichi

Ce connecteur réalise le protocole utilisé pour récupérer les valeurs d'un capteur. Un processus scrute régulièrement la valeur du capteur et la note dans le connecteur. Un ensemble de processus peut lire en une seule fois, afin de la traiter, la valeur stockée dans le connecteur qui correspond à la dernière valeur du capteur.

Lors de l'initialisation, le connecteur note qu'il n'a pas encore reçu de message, ferme le port associé au processus gardé de lecture et ouvre celui associé au processus d'écriture.

A chaque demande de lecture, le connecteur renvoie au processus demandeur, le dernier message qu'il a reçu, et ferme le port associé au processus gardé de lecture.

A chaque demande d'écriture, le connecteur ouvre le port associé au processus gardé de lecture si le connecteur est vide, et note dans son tampon "buffer" le message qu'il vient de recevoir.

```
CONNECTEUR rafraichi =
 VAR buffer :

 PROC_GARDE initialisation =
  SEQ
   buffer := ANY
   conker ! ( lecture, fermer )
   conker ! ( écriture, ouvrir ) :

 PROC_GARDE lecture =
  SEQ
   conker ! ( réponse, buffer )
   buffer := ANY
   conker ! ( lecture, fermer ) :
```

```
PROC_GARDE écriture( VALUE message ) =
 SEQ
  IF
   buffer = ANY
    conker ! ( lecture, ouvrir )
  buffer := message :
```

1.6 Connecteur diffusion-partielle

Ce connecteur diffuse un message reçu par un processus émetteur si au moins N sur M processus recepteurs sont prêts à le recevoir. Les autres processus, s'ils ne sont pas prêts quand le message est reçu par le connecteur, ne recevront jamais ce message.

Contrairement aux connecteurs précédents, qui ne possédaient qu'un processus gardé de lecture et un processus gardé d'écriture, celui-ci possède un processus gardé d'écriture et M processus gardés de lecture.

Lors de l'initialisation, le connecteur ouvre l'ensemble des ports associés aux processus gardés de lecture et ferme celui associé au processus d'écriture.

A chaque demande de lecture faite par un processus émetteur différent qui active un processus gardé de lecture différent, le connecteur ouvre le port associé au processus gardé d'écriture si cette demande de lecture est la Nième, demande au noyau de réactiver ce processus gardé sur la terminaison du processus gardé d'écritur et renvoie au processus qui lui en a fait la demande le message qu'il vient de recevoir.

A chaque demande d'écriture, le connecteur note dans son tampon "buffer" le message qu'il vient de recevoir, et ferme le port associé au processus gardé d'écriture.

```
CONNECTEUR diffusion_partielle( VALUE N, M ) =
 VAR buffer :
 VAR n :

 PROC_GARDE initialisation =
  SEQ
   SEQ i = [ 0 FOR M ]
    conker ! ( lecture[ i ], ouvrir )
   conker ! ( écriture, fermer )
   n := 0 :

 PROC_GARDE lecture[] =
  SEQ
   n := n+1
   IF
    n = N
     conker ! ( écriture, ouvrir )
   conker ! ( synchro, écriture )
   conker ! ( réponse, buffer ) :
```

```
PROC_GARDE écriture( VALUE message ) =
 SEQ
  buffer := message
  conker ! ( écriture, fermer )
  n := 0 :
```

1.7 Connecteur rendez-vous-multiple

Ce connecteur réalise un rendez-vous entre N processus émetteurs et M processus récepteurs. Le message reçu par chaque processus récepteur est une combinaison des N messages émis.

Comme le connecteur précédent, ce connecteur possède M processus gardés de lecture, de plus, il possède N processus gardés d'écriture.

Lors de l'initialisation, le connecteur ouvre l'ensemble des ports associés aux processus gardés de lecture et ferme celui associé aux processus d'écriture.

A chaque demande de lecture faite par un processus émetteur différent qui active un processus gardé de lecture différent, le connecteur ouvre l'ensemble des ports associés aux processus gardés d'écriture si cette demande de lecture est la Nième, exepté le port associé au processus gardé d'écriture[0], demande au noyau de réactiver ce processus gardé sur la terminaison du processus gardé d'écriture[0] et renvoie au processus qui lui en a fait la demande le message construit à partir des M messages que le connecteur vient de recevoir.

A chaque demande d'écriture, le connecteur note dans son buffer le message qu'il vient de recevoir, ouvre le port associé au processus gardé d'écriture[0] si cette demande de lecture est la (M-1)ième, demande au noyau de réactiver ce processus gardé sur la terminaison du processus gardé d'écriture[0], et ferme le port associé à ce processus gardé d'écriture.

```
CONNECTEUR rendez_vous_multiple( VALUE N, M ) =
 VAR buffer[ N ] :

 PROC_GARDE initialisation =
  SEQ
   SEQ i = [ 0 FOR N ]
    conker ! ( lecture[ i ], ouvrir )
   SEQ i = [ 0 FOR M ]
    conker ! ( écriture[ i ], fermer )
   n := 0 :

 PROC_GARDE lecture[] =
  SEQ
   n := n+1
   IF
    n = N
     SEQ i = [ 1 FOR M ]
      conker ! ( écriture[ i ], ouvrir )
   conker ! ( synchro, écriture[ 0 ] )
   conker ! ( réponse, fabriquer( i, buffer ) ) :
```

```
PROC_GARDE écriture[] ( VALUE messsage )=
 SEQ
  buffer[ i ] := message
  m := m+1
  IF
   m = (M - 1)
    conker ! ( écriture[ 0 ], ouvrir )
  IF
   m < M
    conker ! ( synchro, écriture[ 0 ] )
  conker ! ( écriture[ i ], fermer ) :
```

1.8 Connecteur échange

Dans ce connecteur, les deux processus qui communiquent sont émetteurs et récepteurs puisqu'ils échangent leurs messages. Les deux ports associés aux processus gardés d'échange sont ouverts lors de l'initialisation du connecteur.

A chaque activation d'un processus gardé d'échange, le connecteur note le message reçu, et si c'est le premier connecteur d'échange activé il demande au noyau de se synchroniser sur la terminaison de l'autre processus gardé d'échange, puis renvoie au processus le message reçu par l'autre processus gardé d'échange. Si c'est le second processus gardé d'échange il renvoie au processus le message reçu par l'autre processus gardé d'échange.

```
CONNECTEUR échange =
 VAR bool, msg1, msg2 :

 PROC_GARDE initialisation =
  SEQ
   bool := FALSE
   conker ! ( échange1, ouvrir )
   conker ! ( échange2, ouvrir ) :

 PROC_GARDE échange1 ( VALUE message ) =
  SEQ
   msg1 := message
   IF
    bool
     bool := FALSE
    NOT bool
     SEQ
      bool := TRUE
      conker ! ( synchro, écriture2 )
   conker ! ( réponse, msg2 ) :
```

```
PROC_GARDE échange2 ( VALUE message ) =
 SEQ
  msg2 := message
  IF
   bool
    bool := FALSE
   NOT bool
    SEQ
     bool := TRUE
     conker ! ( synchro, écriture1 )
  conker ! ( réponse, msg1 ) :
```

2. Connecteur réparti

Tous les connecteurs présentés précédemment ne peuvent pas être répartis. En effet, ils utilisent tous des variables communes aux processus gardés d'un même connecteur. Dans ce paragraphe, nous allons présenter un connecteur implémentant la gestion d'une file de messages. Cette proposition possède les caractéristiques suivantes :

- le même modèle de connecteur sera utilisé quel que soit la gestion de file implémentée, et le nombre de processus connectés en lecture et en écriture. L'ordonnancement de la file des messages produits (et des messages consommables) sera fait à partir de la date de production du message, et d'une estampille. La combinaison de ces deux éléments permet de réaliser les différents types de communication habituellement utilisés : FIFO, LIFO, à priorité, ...

- le même modèle de connecteur sera utilisé quel que soit le réseau d'interconnexion réalisé entre les sites, et par le système hôte de chaque site. Le connecteur devant s'adapter aux différentes possibilités du médium de communication.

Face à ce choix, dicté par le souci d'offrir un même modèle de connexion quelle que soit l'architecture, il est bien évident que notre proposition ne peut être optimale. Pour une gestion de file spécifique et pour une architecture donnée, des optimisations pourront (et devront) être trouvées si le programmeur désire une certaine efficacité.

Ce travail, visant à définir un modèle général de connecteur réparti, pour une gestion de file, s'est effectué en trois étapes :

- la première fut le choix d'un langage et d'une architecture permettant de simuler aisément le comportement du connecteur réparti (gestion cohérente de la file des messages par chacun des morceaux du connecteur). N'ayant dans notre environnement aucun réseau, nous avons choisi d'écrire le noyau de CONKER[10] à l'aide du langage OCCAM. Celui-ci permet de simuler sur une architecture mono-processeur différents types de réseau d'interconnexion.

- la seconde étape fut la définition du modèle de connecteur réparti, puis sa programmation (la première description du connecteur fut effectuée à l'aide d'un langage "graphique" : réseau de Nutt ; nous l'avons ensuite réécrit à l'aide

---

10. *Le noyau CONKER a déjà été implémenté sur différents systèmes très divers :*

V.G SANCHEZ l'a porté sur le système UNIX. Dans un premier temps à l'aide du système UNIX V7, puis du système UNIX 4.2 BSD. Ce travail a fait l'objet d'une thèse [SANCH85].

J.M LEFEBVRE en réalise une implantation sur une architecture multi-processeurs afin de l'utiliser comme support d'exécution pour une machine LM.

Nous mêmes avons réalisé l'implantation du noyau sur le système temps réel d'INTEL : iRMX86. Cette réalisation sert de base à une répartition de l'application LM.

du langage OCCAM).

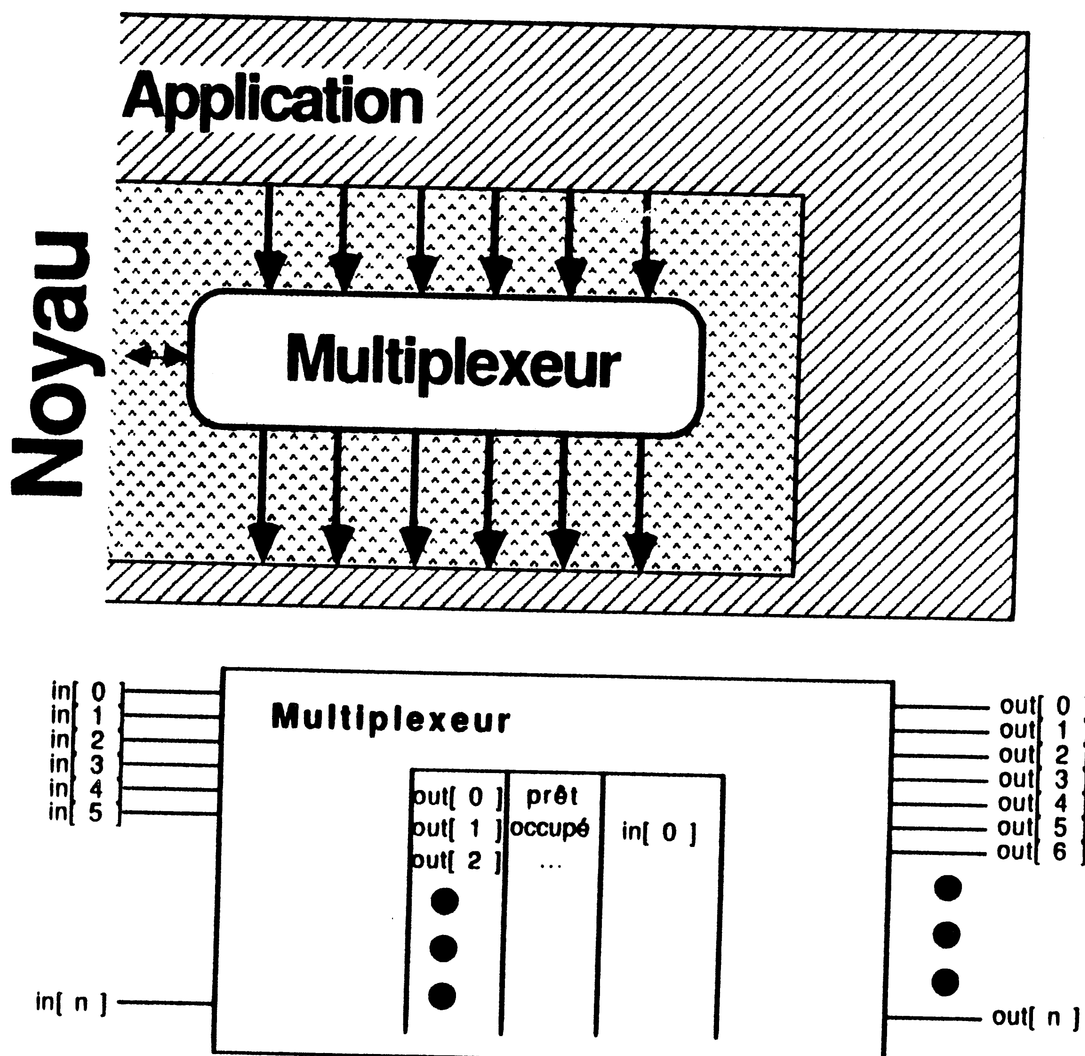
- la troisième étape fut l'analyse, et la réalisation de la partie "routage" du connecteur réalisé selon l'architecture du système d'interconnexion, et selon la gestion de file.

2.1 Programmation du noyau de CONKER

Nous ne reviendrons pas ici sur la définition du noyau CONKER qui a été faite au chapitre 3 et 4. Ce court paragraphe nous permettra de mettre en évidence quelques problèmes liés à la programmation en OCCAM (problèmes qui sont liés à la structure du langage).

i) Réalisation d'un multiplexeur de communication non-bloquant

Le mode de connexion réalisé par les canaux du langage OCCAM est un rendez-vous unidirectionnel entre deux processus. Dans CONKER, l'utilisation de la primitive "SEND", nécessite la réalisation de communication par rendez-vous entre plusieurs processus gardés émetteurs et plusieurs processus gardés récepteurs de CONKER. Pour cela, il a été nécessaire de construire un processus multiplexeur qui permet de recevoir des messages sur un canal, et de le renvoyer sur un autre, sans créer de nouveaux interblocages.



L'architecture actuelle de ce processus multiplexeur est simple :

- **chaque processus gardé possède trois canaux,** le premier sera utilisé pour émettre des messages (canal "out" de la primitive "SEND") et l'autre pour en recevoir (canal "in" de la même primitive). Le dernier canal sera utilisé pour suspendre l'exécution d'un processus gardé sur l'arrivée d'un événement externe.
- **le processus multiplexeur connaît l'état de chaque canal de réception** (prêt, si le processus gardé correspondant est en attente de lecture sur ce canal ; occupé, dans le cas contraire).
- **le processus multiplexeur est prévenu du changement de l'état d'un canal** (la transition "prêt->occupé" est détectée par le multiplexeur à chaque émission de message sur le canal ; la transition "occupé->prêt" est détectée par le processus gardé associé au canal qui prévient le multiplexeur).
- **le processus multiplexeur essaie de résoudre un par un les rendez-vous :** lecture sur un canal d'émission d'un processus gardé, puis tentative d'écriture sur un canal de réception (s'il est libre) ou stockage de la demande. A chaque changement de l'état d'un canal, le processus multiplexeur essaie de résoudre les rendez-vous suspendus.

ii) Gestion des temporisations

Le langage OCCAM ne possède qu'une seule horloge qui donne le temps de l'environnement et chaque communication est toujours blocante jusqu'à sa résolution.

Dans CONKER, le noyau doit résoudre des communications dans un délai précis. Pour cela, en parallèle au processus multiplexeur, nous avons construit un processus échéancier qui classe les demandes dans leur ordre de priorité temporelle, et qui prévient le processus multiplexeur si le rendez-vous n'a pas été résolu avant la date spécifiée.

iii) Réalisation du noyau

Le noyau de CONKER a été réalisé en utilisant les deux processus définis précédemment : processus "multiplexeur" et processus "échéancier". Il a comme entrée, l'entrée du multiplexeur et comme sortie la sortie du multiplexeur. Le noyau gère le graphe d'état des processus gardés en fonction des messages échangés, ce qui lui permet d'émettre un message uniquement à destination d'un processus gardé libre.

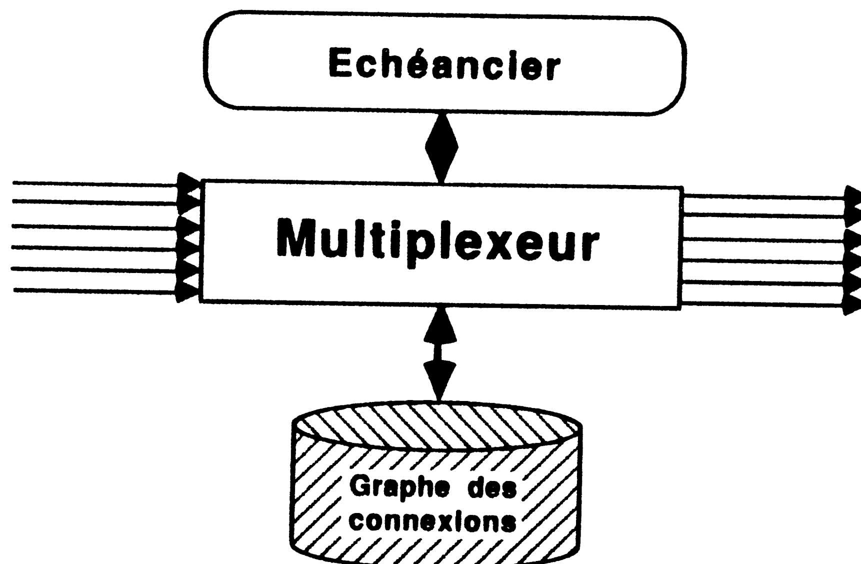


Figure 2. — Noyau CONKER en OCCAM.

2.2 Les connecteurs répartis

Dans l'architecture CONKER, un connecteur réparti est constitué d'un ensemble de connecteurs présents sur chacun des sites que le connecteur réparti relie. Chacun des morceaux du connecteur offre les mêmes services que le connecteur réparti. Pour cela l'ensemble des morceaux du connecteur coopèrent par échange de messages.

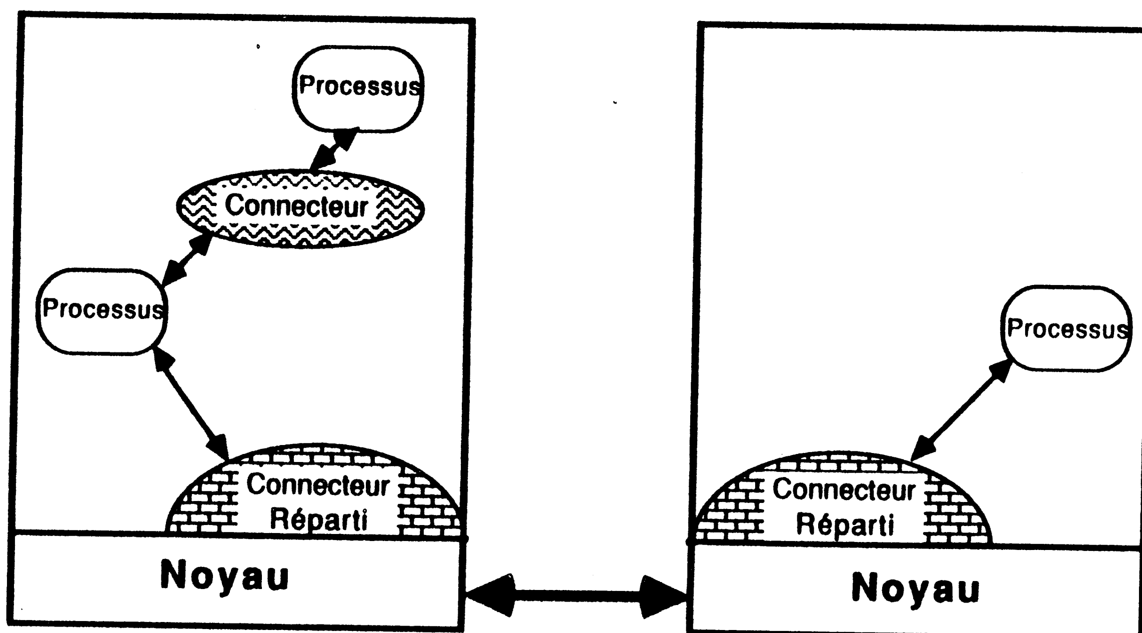


Figure 3. — Connecteurs répartis.

Pour pouvoir s'exécuter, un connecteur réparti utilise les services offerts par une entité système de CONKER qui permet l'échange de messages entre différents sites, indépendamment de l'architecture du système d'interconnexion : le serveur système "station de transport".

Dans l'implémentation réalisée, chaque morceau d'un connecteur réparti connaît les autres morceaux. Cette hypothèse d'implémentation, nous semble réaliste si le connecteur implémenté est relativement statique (i. e. il s'exécute presque toujours sur le même ensemble de sites)

2.3 Un connecteur réparti de gestion de file

Ce modèle de connecteur réparti permet d'implémenter une file composée d'une suite ordonnée de messages dont la production et la consommation sont faites sur un ensemble borné de sites. Chaque site stocke les messages produits par les processus qui s'exécutent sur le site.

Pour définir complètement le mode de connexion réalisé par un connecteur de gestion de file (que ce connecteur soit réparti ou non), il faut définir les relations temporelles entre l'ensemble des messages produits, l'ensemble des messages consommables et l'ensemble des messages consommés [PERRI85].

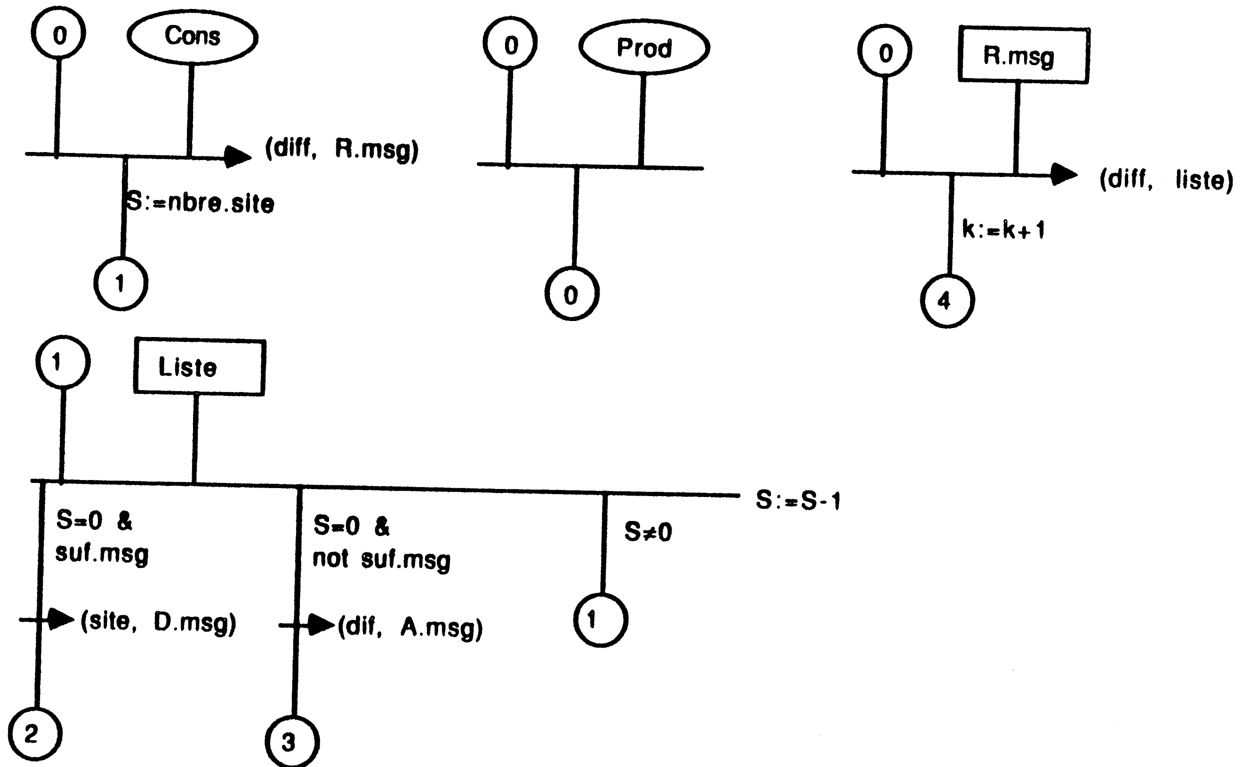
La programmation de cette relation se fera par l'intermédiaire des deux processus qui manipulent la file globale des messages. Le processus d'insertion d'un message permet de définir comment est modifié l'ensemble des messages consommables après chaque production de messages, et le processus de suppression permet de définir comment est modifié le même ensemble de messages après chaque consommation.

Le modèle de connecteur réparti que nous avons implémenté permet à chaque morceau du connecteur d'acquérir la vision partielle de l'état global du connecteur. Ceci lui permet de prendre la décision voulue (acceptation d'une demande de production ou localisation du message à consommer) sans l'hypothèse qu'il ne peut y avoir deux demandes simultanées sur le même site. Par contre, à l'échelle du système, il peut y avoir plusieurs demandes simultanées mais une relation d'ordre assure un ordre complet sur l'ensemble des événements (production et consommation des messages).

Note : Dans l'implémentation réalisée, qui n'est qu'une première proposition, chaque morceau de connecteur doit acquérir la vision globale de la file des messages et des demandes en cours de résolution, avant de prendre la décision. Mais, dans certain mode de connexion spécifique, il n'est pas nécessaire de posséder la vision globale de la file pour prendre une décision. Une vision partielle ou même locale peut être suffisante.

La réalisation de ce type de connecteur repose sur l'utilisation d'algorithme de maintien de cohérence dans un ensemble réparti. Notre objectif n'étant pas de présenter une étude complète sur ce type d'algorithme, ni de détailler complètement l'algorithme utilisé nous renvoyons le lecteur à [MUNTE86a] pour une description complète de l'implémentation et à [WILMS79, HERMA79, ELLIS77, BOKSE84] pour une étude bibliographique de différents algorithmes de maintien de cohérence.

Le réseau de Nutt [NUTT72] ci-dessous décrit partiellement l'algorithme utilisé.



0 Etat du morceau de connecteur

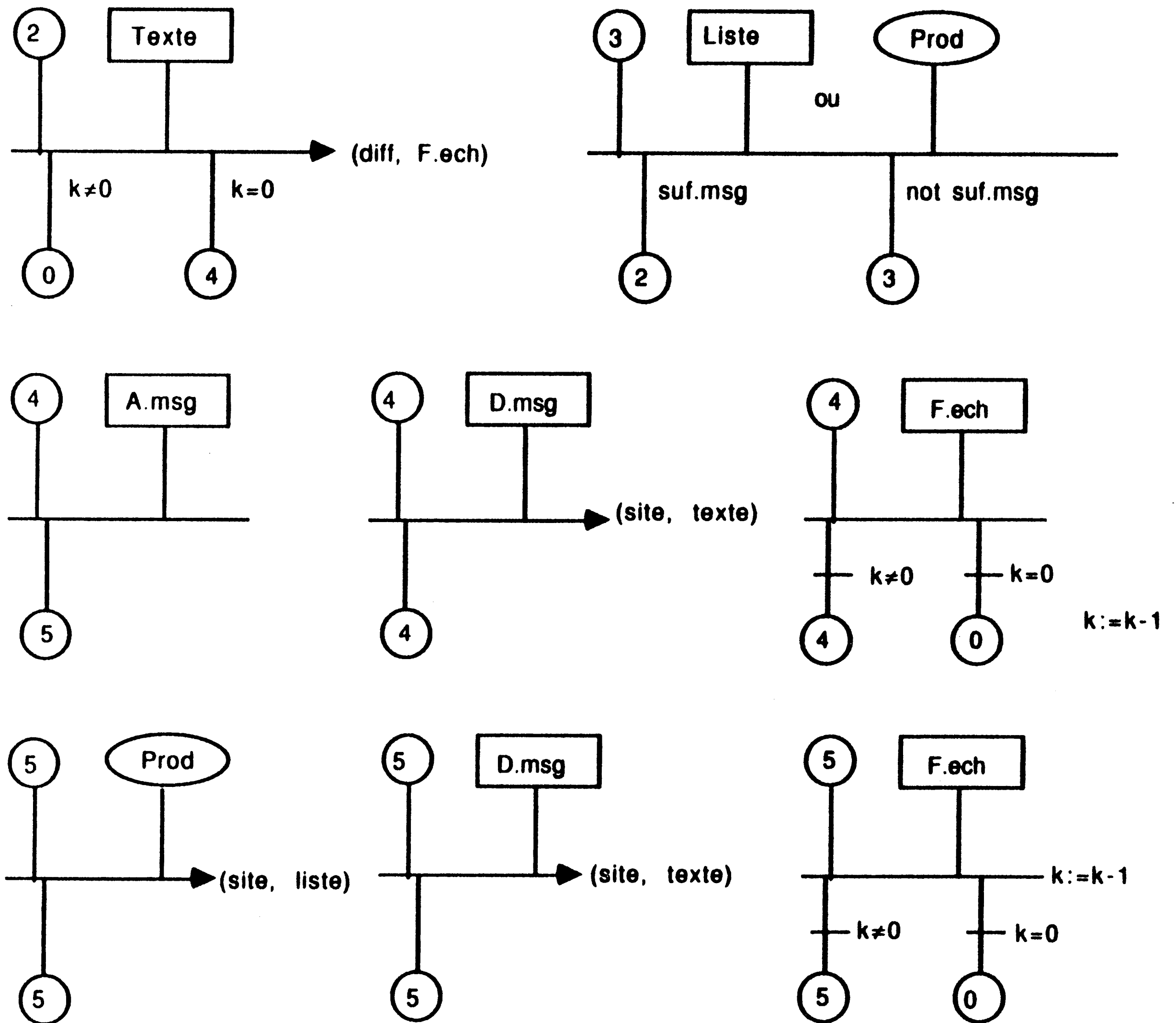
Cons Message émis par un processus lié au connecteur

R.msg Message émis par un autre morceau du connecteur

(diff, R.msg) Message émis par le morceau de connecteur (destinataires, message)

S:=nbre.site Opération sur des variables propres au morceau de connecteur

Etat 0 : Connecteur libre
Etat 1 : Le morceau de connecteur a reçu une demande de consommation et reconstruit la file des messages produits et la file des messages consommés
Etat 2 : Le morceau de connecteur demande à un autre morceau un message
Etat 3 : Le morceau de connecteur demande aux autres morceaux de lui signaler les productions de messages
Etat 4 : Un morceau de connecteur est en train de construire la file des messages produits et la file des messages consommés
Etat 5 : Un morceau de connecteur demande que lui soit signalé toutes les productions de messages



A.msg : le site demande aux autres morceaux de lui signaler les productions de messages
D.msg : le site demande au morceau destinataire de lui renvoyer le message spécifié
F.msg : le morceau de connecteur a reçu le message attendu
R.msg : le morceau de connecteur émetteur désire reconstruire la file des messages produits et la file des messages consommés
liste : le morceau de connecteur renvoie la liste des messages produits et de messages consommés sur ce site
texte : le morceau de connecteur renvoie le message demandé

Figure 4. — Algorithme de gestion de file.

2.4 Le serveur système station de tranport

Dans chaque site de l'architecture CONKER réparti il existe une entité qui offre les services généralement attendus d'une station de transport. Dans les implémentations que nous avons réalisées, cette entité permet :

- **l'accès au réseau d'interconnexion** (interface avec les liens ou les ports physiques de la machine).
- **le routage des messages** dans le cas ou le site est un site noeud du réseau.
- **la localisation des entités distantes** (choix du lien d'émission).
- l'accès à un service de diffusion pour minimiser au mieux le nombre de messages qui doivent circuler sur les différentes voies de l'architecture (ce service est dépendant des caractéristiques du service d'interconnexion).

2.5 Réseau d'interconnexion

Afin de simuler différents réseaux d'interconnexion, nous avons réalisé deux maquettes complémentaires :

- la première permet de construire un connecteur réparti sur une architecture composée de sites reliés par un réseau possédant la diffusion : un site peut communiquer directement avec tous les autres sites. Cette diffusion peut être soit directe (le réseau possède une commande de diffusion) soit indirecte (le réseau simule une diffusion par l'envoi de messages en point à point). L'accès au réseau se fait de façon asynchrone. Cette réalisation peut aisement simuler : réseau Ethernet, architecture multi-processeurs construite sur un Bus (SM90, César et Cléopatre, ...).
- la seconde permet de construire un connecteur réparti au dessus d'une architecture multi-Transputers où chaque site possède 4 sites voisins. La communication entre deux Transputers voisins se fait par rendez-vous. Cette réalisation peut être étendue à toute architecture de type hypercube.

Afin de pouvoir réaliser aisément cette construction, le serveur système station de transport défini dans le paragraphe précédent utilise les services d'un processus d'interconnexion (qui n'appartient pas directement au système CONKER). Ce processus permet de simuler au-dessus d'une architecture spécifique (mono-processeur dans notre cas) une autre architecture (réseau de type Ethernet ou architecture Multi-Transputer).

Note : le processus d'interconnexion est présent uniquement sur la maquette de simulation. En phase d'utilisation ce processus est directement réalisé par le matériel : réseau pour le premier cas ou interconnexion directe des Transputers dans le deuxième cas.

Remarque : pour adapter un connecteur réparti à une architecture spécifique, il suffit de construire l'entité système devant offrir l'interface station de transport (et le processus d'interconnexion, si l'on désire simuler sur une architecture physique particulière une autre architecture). Le connecteur lui-même n'est pas modifié, sauf si l'on désire utiliser au mieux les spécificités de chaque architecture.

2.5.1 Caractéristiques de l'architecture réseau à diffusion

Simuler un réseau à diffusion "atomique" (sans perte de messages) est assez aisé en OCCAM. L'architecture se décompose : en un **module de simulation du réseau** qui gère le multiplexage des messages entre les différents émetteurs et les différents récepteurs et en N **modules (un par site simulé) d'accès au réseau** qui introduisent éventuellement des retards dans l'émission et la réception des messages et assure ainsi l'asynchronisme entre le processus émetteur et le réseau (et par conséquent entre le processus émetteur et le processus récepteur).

Les caractéristiques du réseau à diffusion que nous avons simulé sont :

- pas de perte de message.
- retard aléatoire borné dans les communications.
- deux messages émis par le même site vers le même site destinataire sont reçus dans leur ordre d'émission.
- deux messages émis par 2 sites vers un troisième sont reçus dans un ordre aléatoire.

2.5.2 Caractéristiques d'une architecture à base de Transputers

Simuler une architecture à base de Transputers à l'aide du langage OCCAM n'est plus une simulation, puisque ce langage peut être considéré comme le langage de programmation du Transputer. Nous avons décrit deux types d'architectures : une architecture hypercube qui présente de nombreux avantages et une matrice de Transputers qui peut être utilisée par exemple dans des applications systoliques.

i) Architecture Hypercube

L'architecture d'un hypercube [ASBUR85] est régulière : un n_cube comporte 2^n noeuds comportant un processeur ou un ensemble de processeurs. Chaque noeud est relié à ses n voisins par n connecteurs bidirectionnels (un connecteur par voisin). Ceci permet des communications point à point à très haut débit et chaque noeud possède un système d'exécution de taille réduite qui assure, outre le séquencement des processus, le routage des messages entres les sites. Ce type d'architecture présente différentes caractéristiques intéressantes :

- résistance aux pannes de liaisons (il existe différents chemins entre deux noeuds).

- existence d'un chemin maximal entre deux sites quelconques : la distance entre deux noeuds est toujours inférieure à n.

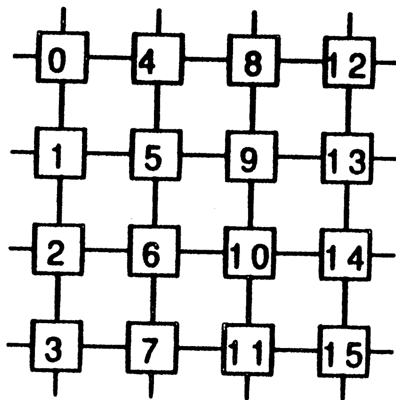
Chaque noeud possède sa propre autonomie et l'architecture ne possède pas de mémoire commune (cette spécificité supprime la gestion d'une mémoire commune partagée). La communication s'effectue uniquement par messages envoyés entre les différents noeuds et chaque processeur déroule un code séquentiel (dans le cas ou le noeud est réalisé par un Transputer, il peut exécuter du code parallèle). Les hypercubes font partie de la classe des machines Multiples Intructions Multiples Données (MIMD).

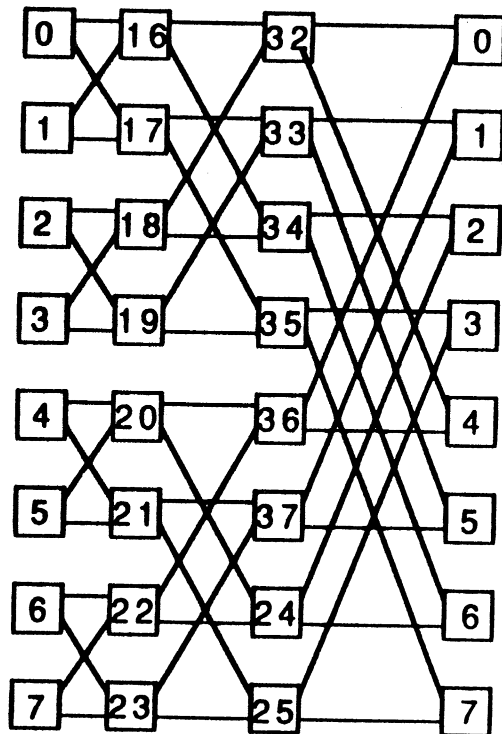
Parmi les machines de ce type déjà en vente nous pouvons citer outre les différentes architectures en cours de développement à l'aide du Transputer, le "COSMIC CUBE" [SEITZ85] conçu par une équipe du "California Institute of Technologie" et commercialisé depuis peu par la firme INTEL sous le nom iPSC, c'est un 6_cube.

La principale difficulté de cette réalisation a été le système de routage des messages, si l'on désire qu'il s'adapte aux pannes et, si l'on désire minimiser à la fois le nombre de messages et le nombre de noeuds traversés (un message à destination de deux noeuds sur le même chemin est envoyé une seule fois).

ii) Matrice de Transputer

Dans ce type d'architecture, chaque processeur communique avec ses 4 voisins, et les processeurs aux extrémités gèrent les entrées/sorties avec la structure matricielle. Cette structure est classique dans les réseaux systoliques et on peut, par un rebouclage astucieux des processeurs aux bords de la matrice (voir figure suivante), constituer un tore permettant de simuler un plan infini sur un nombre fini de processeurs.





De le même façon que pour une architecture hypercube chaque processus possède un système de traitement qui lui permet d'assurer le routage des messages en transit et le séquencement des processus.

2.6 Connecteur file_reparti

```
CONNECTOR canal.reparti =
 -- type des messages échanges entre les morceaux du connecteur
 DEF
  -- Attente de message
  A.msg='A',
  -- Demande de message
  D.msg='D',
  -- Fin d'echange
  F.ech='F',
  -- Recherche de message
  R.msg='R',
  -- J'ai bien reçu le message
  OK='O',
  -- Liste des messages
  liste='l',
  -- Envoi du message de texte demandé
  texte='t' :
 VAR
  -- Le connecteur attend des messages suite à une demande de recherche
  attendre,
  -- Un morceau de connecteur recherche des messages
  recherche,
  -- Un message a été écrit sur le connecteur depuis la demande de recherche
  arrivee,
  -- numero du prochain message à écrire dans le morceau du connecteur
  l,
  -- numero du prochain message à lire dans le morceau du connecteur
  L,
  -- ensemble des messages consommables sur le morceau de connecteur : (texte ; numero de priorite)
  msg.texte[ size ],
  msg.prio[ size],
  -- liste des sites qui sont en attente de message
  msg.site[ size ],
  -- nombre de processus consommateurs désirant consommer un message
  k,
  -- nombre de messages consommables sur le connecteur
  n,
  -- numero d'ordre pour la consommation
  m,
  -- nombre de sites qui n'ont pas repondu pour la consommation
  C,
  -- nombre de sites qui n'ont pas repondu pour la production
  P,
  -- un processus du site desire consommer un message
  dem.cons,
```

```
   -- ensemble des messages consommables du connecteur (site ; numero de priorite)
   buff.site[ size ],
   buff.prio[ size ],
   -- zone des messages en cours d'envoi ou de reception
   buffer,
   tmp[ size ] :

 PROC noter.message VALUE site, prio ) =
   -- recréer la file des messages consommables sur le connecteur
   VAR i :
   SEQ
     i := 0
     WHILE i < n
       IF
         (buff.site[i]=site) AND (buff.prio[i]=prio)
           i := size+1
         TRUE
           i := i+1
     IF
       i = n
         SEQ
           buff.site[ n ] := site
           buff.prio[ n ] := prio
           n := n+1 :

PROC recevoir( VALUE my.pg ) =
  SEQ j = [ 1 FOR tmp[ 0 ] ]
    in[ my.pg ] ? tmp[ j ] :

PROC emettre( VALUE my.pg, d.site, msg ) =
  VAR d.pg :
  SEQ
    tmp[ 1 ] := l
    tmp[ 2 ] := msg
    IF
      msg = D.msg
        tmp[ 0 ] := 4
      msg = liste
        SEQ
          tmp[ 0 ] := l+3
          SEQ j = [ 0 FOR l ]
            tmp[ j+3 ] := msg.prio[ j ]
      msg = texte
        tmp[ 0 ] := 3
      TRUE
        tmp[ 0 ] := 2
    IF
      msg = texte
        d.pg := pg.message
      TRUE
        d.pg := pg.communication
    SEND( my.pg, d.site, d.pg, simple, 0, tmp ) :
```

```
PROC recherche.msg( VALUE my.pg ) =
 PROC choix.site( VALUE i, VAR j ) =
  j := buff.site[ i ] :

 PROC choix.prio( VALUE i, VAR j ) =
  j := buff.prio[ i ] :

 IF
  m < n
   SEQ
    choix.site( m, tmp[ 3 ] )
    choix.prio( m, tmp[ 4 ] )
    emettre( my.pg, tmp[ 1 ], D.msg )
  TRUE
   SEQ
    emettre( my.pg, diffusion, A.msg ) :

PROC_GARDE initialisation( VALUE my.pg ) =
 SEQ
  recevoir( my.pg )
  -- chaque morceau de connecteur peut être :
  --       soit producteur
  --       soit consommateur
  --       soit producteur et consommateur
  -- mise a jour du nombre de morceaux consommateurs
  proc.c := tmp[ 2 ]
  -- mise a jour du nombre de morceaux producteurs
  proc.p := tmp[ 3 ]
  tmp[ 1 ] := 2
  tmp[ 2 ] := pg.écriture
  tmp[ 3 ] := ouvrir
  SEND( my.pg, I, noyau, simple, 0, tmp )
  tmp[ 2 ] := pg.lecture
  SEND( my.pg, I, noyau, simple, 0, tmp )
  tmp[ 2 ] := pg.communication
  SEND( my.pg, I, noyau, simple, 0, tmp )
  tmp[ 2 ] := pg.message
  SEND( my.pg, I, noyau, simple, 0, tmp )
  SEQ j = [ 0 FOR size ]
   msg.site[ j ] := size
  attendre := FALSE
  recherche := FALSE
  arrivee := FALSE
  l := 0
  a := 0
  dem.cons := FALSE
  attend := FALSE
  k := 0
  TERMINAISON( my.pg ) :

PROC_GARDE écriture( VALUE my.pg ) =
 VAR prio :
 PROC noter.msg( VALUE msg, prio ) =
```

```
     -- manipule la file des messages consommables sur le morceau de connecteur
     SEQ
      IF
       I < size
        SEQ
         msg.texte[ I ] := msg
         msg.prio[ I ] := prio
         I := I+1 :

   SEQ
    recevoir( my.pg )
    TIME ? prio
    noter.msg( tmp[ 1 ], prio )
    IF
     recherche
      arrivee := TRUE
    IF
     attendre
      SEQ k = [ 0 FOR size ]
       IF
        msg.site[ k ] <> size
         emettre( my.pg, msg.site[ k ], liste )
    TERMINAISON( my.pg ) :

PROC_GARDE lecture( VALUE my.pg ) =
  SEQ
   emettre( my.pg, diffusion, R.msg )
   C := proc.c-1
   P := proc.p-1
   n := 0
   m := 0
   k := 1
   SEQ j = [ 0 FOR I ]
    noter.message( I, msg.prio[ j ] )
   tmp[ 0 ] := 2
   tmp[ 1 ] := fermer
   tmp[ 2 ] := pg.lecture
   SEND( my.pg, I, noyau, simple, 0, tmp )
   dem.cons := TRUE
   SYNCHRO( my.pg, 0, I, pg.message )
   tmp[ 0 ] := 1
   tmp[ 1 ] := buffer
   REPONSE( my.pg, tmp ) .
   TERMINAISON( my.pg ) :

PROC_GARDE communication( VALUE my.pg ) =
 PROC noter.site( VALUE site ) =
  VAR i :
  SEQ
   i := 0
   attendre := TRUE
   WHILE i < size
    IF
```

```
      msg.site[ i ] = size
        SEQ
          msg.site[ i ] := site
          i := size
      TRUE
        i := i+1 :

PROC supprimer.site( VALUE site ) =
  VAR i, attend :
  SEQ
    i := 0
    attend := TRUE
    WHILE i < size
      IF
        msg.site[ i ] = site
          msg.site[ i ] := size
        msg.site[ i ] <> size
          attend := FALSE
        TRUE
          i := i+1
    IF
      attend
        -- plus aucun site est en attente de message
        SEQ
          attendre := FALSE
          recherche := FALSE
          arrivee := FALSE :

PROC chercher.message( VALUE m, VAR texte ) =
  -- manipule la file des messages consommables sur le morceau de connecteur
  --
  -- assure l'opération inverse de noter.message
  -- rend en paramêtre le texte du message qui a le numero d'ordre m
  --
  -- actuellement renvoi le dernier message produit
  --
  SEQ
    texte := msg.texte[ L ]
    l := l-1
    L := l :

SEQ
  recevoir( my.pg )
  IF
    tmp[ 1 ]=R.msg
      SEQ
        k := k+1
        IF
          k = 1
            SEQ
              tmp[ 0 ] := 2
              tmp[ 1 ] := fermer
              tmp[ 2 ] := pg.lecture
```

```
        SEND( my.pg, I, noyau, simple, 0, tmp )
      IF
        dem.cons
          SEQ
            C := C-1
            IF
              tmp[ 0 ] < 1
                m := m+1
            IF
              (P+C) = 0
                recherche.msg( my.pg )
        TRUE
          emettre( my.pg, site, OK )
      emettre( my.pg, site, liste )
      recherche := TRUE
  tmp[ 1 ] = A.msg
    SEQ
      noter.site( tmp [ 2 ] )
      IF
        arrivee
          emettre( my.pg, tmp[ 2 ], liste )
  tmp[ 1 ] = D.msg
    SEQ
      supprimer.site( tmp[ 2 ] )
      chercher.message( m, tmp[ 3 ] )
      emettre( my.pg, tmp[ 1 ], texte )
  tmp[ 1 ] = F.ech
    SEQ
      supprimer.site( tmp[ 2 ] )
      k := k-1
      IF
        k = 0
          SEQ
            tmp[ 0 ] := 2
            tmp[ 1 ] := ouvrir
            tmp[ 2 ] := pg.lecture
            SEND( my.pg, I, noyau, simple, 0, tmp )
  tmp[ 1 ] = liste
    SEQ
      SEQ j = [ 3 FOR tmp[ 0 ]-2 ]
        noter.message( tmp[ 2 ], tmp[ j ] )
      IF
        P <> 0
          SEQ
            P := P-1
            IF
              (P+C) = 0
                recherche.msg( my.pg )
      ((P+C) = 0) and (m < n)
      -- tous les morceaux de connecteurs ont déjà donné leur réponse
      -- le morceau a demandé que lui soit signalé toute demande de production
      --
      -- attend que tous les messages en train d'être acheminés soient arrivés
```

```
        -- tous les morceaux de connecteurs auront la même vision de la file des consommables
         SEQ
           tmp[ 0 ] := 2
           tmp[ 1 ] := ouvrir
           tmp[ 2 ] := pg.temporisation
           SEND( my.pg, I, noyau, simple, 0, tmp )
    tmp[ 1 ] = OK
      SEQ
        C := C-1
        IF
          (P+C) = 0
            recherche.msg( my.pg )
  TERMINAISON( my.pg ) :

PROC_GARDE message( VALUE my.pg ) =
  SEQ
    recevoir( my.pg )
    buffer := tmp[ 3 ]
    emettre( my.pg, diffusion, F.ech )
    k := k-1
    IF
      k = 0
        SEQ
          tmp[ 0 ] := 2
          tmp[ 1 ] := ouvrir
          tmp[ 2 ] := pg.lecture
          SEND( my.pg, I, noyau, simple, 0, tmp )
    dem.cons := FALSE
    TERMINAISON( my.pg ) :

PROC_GARDE temporisation( VALUE my.pg ) =
  -- processus garde active périodiquement par le noyau quand il est ouvert
  -- la frequence d'activation est de (2*transfert)
  SEQ
    tmp[ 0 ] := 2
    tmp[ 1 ] := fermer
    tmp[ 2 ] := pg.temporisation
    SEND( my.pg, I, noyau, simple, 0, tmp )
    recherche.msg( my.pg ) :
```

Bibliographie

[ALLEN85] **ALLEN L.W :** *Design of a kernel for ARGUS ;* thesis, MIT, Massachusetts, June 85.

[ALMES83] **ALMES G.T, BLACK A.P, LAZOWSKA E.D, NOE J.D :** *The EDEN system : a technical review ;* University of Washington (Departement of Computer Science), Washington, October 1983. *(Technical report).*

[ANDRE81] **ANDREWS G.R :** *Synchronizing Resources ;* ACM Transactions on Programming Languages and Systems, vol. 3, no. 4, October 1981.

[ASBUR85] **ASBURY R, FRISON S.G, ROTH T :** *Concurrent computers ideal for inherently parallel problems ;* Computer Design, September 1, 1985.

[BARRO83] **BARRON I, CAVILL P, MAY D, WILSON P :** *Transputer does 10 or more MIPS even when not used in parallel ;* ELECTRONICS, November 17, 1983.

[BASKE77] **BASKETT F, HOWARD J.H, MONTAGUE J.T :** *Task communication in DEMOS ;* Proc. of the sixth Symp. on Operating Systems Principles, November 1977.

[BOKSE84] **BOKSENBAUM C, CART M, FERRIE J, PONS J.F :** *Le contrôle de cohérence dans les bases de données réparties ;* RR 17, CRIM, Montpellier, Novembre 1984.

[BRAIN84] **BRAIN S :** *The Transputer : 'exploiting the opportunity of VLSI' ;* Electronics Product Design, December 1983, january 1984.

[BRETE85] **BRETEUIL H :** *Le Transputer de INMOS, un processeur peu banal : présentation générale ;* Minis et Micros, no. 224, 1985.

[BRETE85a] **BRETEUIL H :** *Une machine qui a du coeur : le processeur systolique GAPP de NCR ;* Minis et Micros, no. 228, 1985.

[BRINC78] **BRINCH HANSEN P :** *Distributed Processes : a concurent programming concept ;* Communications of the ACM, vol. 21, no. 11, November 1978.

[BROWA84] **BROWAEYS F, DERRIENNIC H, DESCLAUD P, FALLOUR H, FAULLE C, FEBVRE J, HANNE J.E, KRONENTAL M, SIMON J.J, VOJNOVIC D :** *SCEPTRE : proposition de noyau normalisé pour les exécutifs temps réel ;* TSI, vol. 3, no. 1, 1984.

[CARTO84] **CARTON P, GILLON J :** *Manuel de présentation de CESAR V1.00 ;* Mai 1984.

[CARTO84a] **CARTON P, GILLON J :** *Manuel descriptif du système CESAR V1.00 ;* Mai 1984.

[CII80] **CII Honeywell :** *Reference manual for the ADA programming language ;* 1980.

[COOK80] **COOK R.P :** **MOD - A language for distributed programming ;* IEEE Transactions on Software Engineering, vol. 6, no. 6, November 1980.

[DELAT84] **DELATTRE E, GEIB J.M, MEHAUT J.F :** *The omphale distributed system architecture : communication and concurrency issues ;* LIFL, Lille, 1984.

[DESCH77] **DESCHIZEAUX P, LADET P :** *Primitives de synchronisation pour langages évolués de commande de procédés ;* RAIRO Automatique, vol. 11, no. 3, 1977.

[DIX83] **DIX T.I :** *Exceptions and interrupts in CSP ;* Sc. of comp. prog, no. 3, 1983.

[Digit79] **Digital Equipement System :** *RSX-11M ;* Digital Equipement Corporation, 1979. *(Reference Manual).*

[ELLIS77] **ELLIS C.A :** *Consistency and correctness of duplicate database systems ;* Proc of the 6th ACM Symposium on Operating Systems Principles, November 1977.

[ESPIA86] **ESPIAU B :** *Etat de l'art en capteurs d'environnement local pour la robotique ;* PI, no. 296, IRISA, Rennes, Mai 1986.

[FINGE82] **FINGER U, MEDIGUE G :** *Architectures multiprocesseurs : l'exemple de la SM90 ;* Minis et Micros, no. 173, 1982.

[GIEB84] **GIEB J.M :** *L'architecture omphale ;* LIFL, Lilles, 1984. *(Rapport interne).*

[GRAY76] **GRAY J.N, LORIE R.A, PUTZOLU G.F, TRAIGER I.L :** *Granularity of locks and degrees of consistency in a shared data base ;* Modeling in data base management systems, Elsevier North Holland, New York, 1976.

[GRIES80] **GRIESNER R :** *Conception et réalisation d'un système temps réel réparti pour la commande décentralisée de procédé ;* Thèse Docteur 3ème cycle, INPG, Grenoble, 4 avril 1980.

[GUILL82] **GUILLEMONT M :** *Intrégration du système réparti CHORUS dans le langage de haut niveau PASCAL ;* Thèse Docteur Ingénieur, USMG, Grenoble, Mars 1982.

[HEHNE83] **HEHNER E.C.R, HOARE C.A.R :** *A more complete model of communicating processes ;* Theorical Computer Science, vol. 26, 1983.

[HEIDE82] **HEIDER G :** *Let operating systems aid in component designs ;* Computer Design, September 1982.

[HERMA79] **HERMAN D, VERJUS J.P :** *Un algorithme de maintien de la cohérence de copies multiples ;* RI 105, IRISA, Rennes, Fevrier 1979. *(Version provisoire).*

[HOARE78] **HOARE C.A.R :** *Some properties of predicate transformers ;* Journal of the ACM, vol. 25, no. 3, July 1978.

[HOARE81] **HOARE C.A.R :** *A calculus of total correctness for communicating processes ;* Science of Computer Programming, no. 1, North-Holland Publishing Compagny, 1981.

[HOARE84] **HOARE C.A.R, ROSCOE A.W :** *Programs as executable predicates ;* Proceedings of the international conference on fifth generation computer systems, ICOT, 1984.

[INMOS84] **INMOS Limited :** *OCCAM programming manual ;* Prentice-hall international, 1984.

[INMOS85] **INMOS :** *TRANSPUTER ;* 72-TRN-006-001, September 1985. *(Reference manual).*

[INTEL82a] **INTEL :** *PLM86 ;* INTEL Corporation, 1982.

[INTEL82] **INTEL :** *iRMX86 ;* INTEL Corporation, 1982.

[JACKS83] **JACKSON K :** *MASCOT and multiprocessor systems ;* Distributed Computing Systems, Academic Press London, 1983.

[JOLOB84] **JOLOBOFF V, QUINT V :** *Two-dimensional editing ;* RR TIGRE 16, IMAG Bull, Grenoble, Juin 1984.

[KRAME81] **KRAMER J, MAGEE J, SLOMAN M :** *Intertask communication primitives for distributed computer control systems ;* IEEE, 1981.

[KRAME85] **KRAMER J, MAGEE J :** *Dynamic configuration for distributed systems ;* IEEE Transactions on Software Engineering, vol. 11, no. 4, April 1985.

[KUNG82] **KUNG H.T :** *Why systolic architectures? ;* IEEE Computer, vol. 15, no. 1, January 1982.

[LADET82] **LADET P :** *Contribution à l'étude des systèmes informatiques répartis pour la commande des procédés industriels ;* Thèse Docteur d'état es science, USMG-INPG, Grenoble, Juin 1982.

[LAZOW81] **LAZOWSKA E.D, LEVY H.M, ALMES G.T, FISCHER M.J, FOWLER R.J, VESTAL S.C :** *The architecture of the EDEN system ;* Proceeding of the eighth symposium on operating systems principles, ACM, 1981.

[LEFEB82] **LEFEBVRE J.M :** *Nouvelle définition de l'instruction DEPLACER dans le langage LM ;* Rapport de DEA, IMAG, Grenoble, 1982.

[LISKO79] **LISKOV B :** *Primitives for distributed computing ;* Proc 7th ACM Symposium on Operting Systems Principles, Pacific Grove, California, December 1979.

[LISKO83] **LISKOV B, SCHEIFLER R :** *Guardians and Actions : linguistic support for robust, distributed programs ;* ACM Transactons on Programming Languages and Systems, vol. 5, no. 3, July 1983.

[MADAU77] **MADAULE, MENDELBAUM, CALVEZ :** *GAELIC ;* Journée AFCET, 1977.

[MAY83] **MAY D :** *OCCAM ;* ACM SIGPLAN Notices, vol. 18, no. 4, April 1983.

[MAZER85] **MAZER E, MIRIBEL J.F :** *Le langage LM ;* CEPADUES, Janvier 1985. *(Manuel de référence).*

[MIRIB84] **MIRIBEL J.F :** *Conception et implantation d'un système de programmation de robots ;* Thèse Docteur 3ème cycle, USMG-INPG, Grenoble, Octobre 1984.

[MUNTE85] **MUNTEAN T, RIVEILL M, SANCHEZ V :** *CONKER : Noyau réparti pour OCCAM ;* RR 529, IMAG-Laboratoire de Génie Informatique, Grenoble, Mai 1985.

[MUNTE85a] **MUNTEAN T, RIVEILL M :** *Noyau de communication pour systèmes répartis en robotique ;* Juin 1985. *(Rapport final de contrat ITMI-IMAG).*

[MUNTE85] **MUNTEAN T :** *Temps pour TCCP ;* IMAG, Grenoble, 1985. *(Draft).*

[MUNTE86] **MUNTEAN T, RIVEILL M, TRICOT C :** *OCCAM-TO : langage pour la programmation temps-critique ;* W.P 20, vol. Projet Esprit 1085, Grenoble, mai 1986.

[MUNTE86] **MUNTEAN T, RIVEILL M :** *TO : an extended OCCAM model for time critical programming - Application to environmental robot systems ;* W.P 37, vol. Esprit Project 1085, Grenoble, October 1986.

[MUNTE86a] **MUNTEAN T, RIVEILL M :** *An extended OCCAM model for timed parallel systems : application to environmental robot programming ;* Intelligent Autonomous Systems, Amsterdam, december 1986.

[NATAR85] **NATARAJAN N :** *Communication and synchronization primitives for distributed programs ;* IEEE Transactions on Software Engineering, vol. 11, no. 4, April 1985.

[NONN79] **NONN P :** *Le système d'exploitation dans le projet SYGARE ;* Thèse de Docteur Ingénieur, Nancy, 1979.

[NUTT72] **NUTT G.J :** *Evaluation nets for computer performance analysis ;* Proc AFIPS FJCC, 1972.

[OLSON85] **OLSON R :** *Parallel processing in a message based operating system ;* IEEE SOFTWARE, July 1985.

[PERRI85] **PERRIN G.R :** *La communication : un outil pour la spécification, la construction et la vérification de systèmes parallèles ;* Thèse Docteur es Sciences Mathématiques, Université de Nancy I, 3 octobre 1985.

[POSTE80] **POSTEL J. :** *Standart transmission control protocol ;* Internet Working Group, vol. IEN 129, DOD, January 1980.

[RITCH74] **RITCHIE D.M, THOMPSON K :** *The UNIX time-sharing system ;* Communications of the ACM, vol. 17, no. 7, July 1974.

[ROSCO85] **ROSCOE A.W :** *Denotational semantics for OCCAM ;* Lecture Notes in Computer Science, no. 197, Springer-Verlab, 1985.

[SANCH85] **SANCHEZ ARIAS V.G :** *Un noyau pour la communication et la synchronisation de processus répartis ;* Thèse Docteur Ingénieur, IMAG, Grenoble, Janvier 1985.

[SEITZ85] **SEITZ C.L :** *The COSMIC cube ;* Communications of the ACM, vol. 28, no. 1, January 1985.

[SENAY83] **SENAY C :** *Un système de désignation et de gestion de portes pour l'architecture répartie CHORUS ;* Thèse Docteur Ingénieur, Conservatoire National des Arts et Métiers, Décembre 1983.

[SHATZ84] **SHATZ M :** *Communication mechanisms for programming distributed systems ;* IEEE Computer, vol. 17, no. 6, June 1984.

[SIMPS79] **SIMPSON H.R, JACKSON K :** *Process synchronisation in MASCOT ;* The Computer Journal, vol. 22, no. 4, 1979.

[TRICO87a] **TRICOT C, RIVEILL M :** *Implémentation du langage OCCAM sur une architecture multi-processeurs ;* RR, IMAG, Grenoble, 1987. *(à paraitre).*

[TRICO87] **TRICOT C :** *Mux : a lightweight multiprocessor Subsystem under Unix ;* European Unix systems User Group, Helsinki and Stockholm, May 1987.

[WALKE85] **WALKER P :** *The Transputer ;* BYTE, May 1985.

[WILMS79] **WILMS P :** *Etnde et comparaison d'algorithmes de maintien de la cohérence dans les bases de données réparties ;* Thèse Docteur Ingénieur, INPG, Grenoble, 23 novembre 1979.

[ZIMME84] **ZIMMERMANN H, GUILLEMONT M, MORISSET G, BANINO J.S :** *CHORUS : a communicating and processing architecture for distributed systems ;* RR 328, INRIA, Rocquencourt, Septembre 1984.

TABLE DES MATIERES

Introduction et plan de l'étude 3

I. Communication et synchronisation dans des systèmes distribués 7

1. Les systèmes distribués 8
   1.1 Brève description 8
   1.2 Quelle communication/synchronisation ? 9
   1.3 Un complément nécessaire : le temps 11
   1.4 Comment désigner le correspondant 11

2. ARGUS 13

3. CONIC 15
   3.1 Configuration statique 15
   3.2 Configuration dynamique 15
   3.3 Le système CONIC 16

4. DEMOS, EMBOS, OMPHALE 19
   4.1 Architecture 19
   4.2 Communication/synchronisation 19
   4.3 Protection 19
   4.4 Contrôle de la dispersion des liens 21

5. EDEN 22

6. MASCOT 24
   6.1 Structure d'une application MASCOT 24
   6.2 Communication 26
   6.3 Répartition 27

7. Proposition de [Natar85] 29
   7.1 Primitives de base 29
   7.2 Primitives pour la gestion des erreurs de communication 32

8. SCEPTRE 33

9. Comparaison et synthèse 35
   9.1 Présentation générale 35
   9.2 La communication 35

9.3 La synchronisation 37
9.4 Désignation et protection 37
9.5 Conclusion 37

II. Présentation du langage OCCAM et de ses extensions 45

1. Langage et modèle de départ 46

2. Description du langage OCCAM 47
2.1 Processus élémentaires 47
2.2 Les constructeurs d'OCCAM 48
2.3 Les extensions au langage OCCAM 50

3. Le Transputer 52

4. Exemples 54
4.1 Exemple de programme OCCAM : un compteur d'essieux 54
4.2 Un autre exemple de programme OCCAM : un diviseur de fréquence 57

III. CONKER : définition du noyau 59

1. Démarche 60

2. Caractéristiques des applications visées 61

3. Le modèle de communication CONKER 65
3.1 La communication par canal en OCCAM 65
3.2 Les connecteurs 70

4. Définition du noyau de CONKER 73
4.1 Communication entre entités 75
4.2 Protection et confidentialité 79
4.3 Utilisation de CONKER dans un environnement réparti 80

IV. CONKER : description du système 85

1. Architecture du système CONKER 86
1.1 Structure d'implémentation du noyau 86
1.2 Structure d'implémentation d'une entité CONKER 88
1.3 Le noyau et les serveurs du système CONKER 93

1.4 Communication entre processus de l'application 96
1.5 Connecteurs 97
1.6 Programmation des processus gardés 99

2. Ensemble des primitives 107
2.1 Construire un modèle d'entité 107
2.2 Supprimer un modèle d'entité 109
2.3 Créer un processus 109
2.4 Attente sur la terminaison d'un processus 110
2.5 Détruire un processus 113
2.6 Créer un connecteur 113
2.7 Détruire un connecteur 114
2.8 Se lier à un connecteur 115
2.9 Se délier d'un connecteur 116
2.10 Lecture d'un message 117
2.11 Ecrire un message 118
2.12 Attendre plusieurs événements 118
2.13 Activer un traitement 120
2.14 Supprimer un traitement 121
2.15 Masquer un traitement 121
2.16 Démasquer un traitement 122
2.17 Connaître l'état d'un connecteur 122

3. Conclusion 123

V. Réalisation de connecteurs pour une application robotique 125

1. LM : langage de manipulation de robots 126
1.1 Concepts 126
1.2 L'architecture de la machine LM 130

2. Architecture de la tâche de déplacement 133
2.1 Principe de l'architecture 133
2.2 Principaux modes de communication entre processus 135
2.3 Exemple d'interprétation de l'instruction DEPLACER 142

3. Conclusion 144

VI. Des exemples de connecteurs 145

1. Implémentation répartie de constructeurs OCCAM 146
   1.1 Principe de l'implémentation du PAR 146
   1.2 Implémentation du ALT 148

2. Connecteurs de gestion d'événements 150
   2.1 Gestion des événements complexes 150
   2.2 Réalisation de ces connecteurs 154

3. Remarques 157

Conclusion et Perspectives 159

Annexe : Base de connecteurs 161

1. Exemple de connecteurs 162
   1.1 Connecteur rendez-vous 162
   1.2 Connecteur égalité 163
   1.3 Connecteur pile 164
   1.4 Connecteur aléatoire 165
   1.5 Connecteur rafraichi 166
   1.6 Connecteur diffusion-partielle 167
   1.7 Connecteur rendez-vous-multiple 168
   1.8 Connecteur échange 169

2. Connecteur réparti 171
   2.1 Programmation du noyau de CONKER 172
   2.2 Les connecteurs répartis 174
   2.3 Un connecteur réparti de gestion de file 175
   2.4 Le serveur système station de tranport 178
   2.5 Réseau d'interconnexion 178
   2.6 Connecteur file_reparti 181

INPG département des études doctorales

AUTORISATION de SOUTENANCE

VU les dispositions de l'article 15 Titre III de l'arrêté du 5 juillet 1984 relatif aux études doctorales

VU les rapports de présentation de Messieurs

. J. FERRIE, Professeur
. A. SCHIPER, Professeur

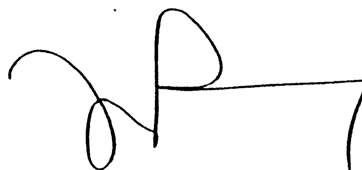
Monsieur Michel RIVEILL

est autorisé à présenter une thèse en soutenance en vue de l'obtention du diplôme de DOCTEUR de L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE, Spécialité "Informatique".

Fait à Grenoble, le 20 janvier 1987

D. BLOCH
Président
de l'Institut National Polytechnique
de Grenoble

P.O. le Vice-Président.



CONKER : un modèle de répartition pour processus communicants. Application à OCCAM.

Résumé

CONKER est un noyau de communication construit à l'aide du modèle CSP. Une application décrite à l'aide de CONKER sera composée de connecteurs qui sont les objets chargés de la communication-synchronisation et de processus qui se chargent du traitement.

Chaque connecteur du noyau réalise un protocole de communication particulier entre les processus "applications". Ceux-ci n'utilisent que des primitives d'envoi et de réception de messages, de manière homogène et transparente à la réalisation des connecteurs qui les relient.

Cette caractéristique de transparence, ainsi que la possibilité d'implémenter CONKER sur un ensemble de systèmes "hôtes hétérogènes", assurent une transportabilité aisée des applications en environnement multi-processeurs, ou sur un réseau hétérogène.

De part sa conception, CONKER permet de mettre à la disposition des processus "applications", des types de connecteurs, construits de façon incrémentale à la demande, et utilisant pour cela, les calculs de processus issus du modèle CSP.

Mots clés : modèle de communication, système réparti, application "temps critique", CSP, OCCAM, Transputer.