INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE

E.N.S.E.M NANCY

SYGARE : UNE STRUCTURATION POUR LA CONCEPTION D'APPLICATIONS EN TEMPS REEL ET REPARTIES

THESE

présentée et soutenue publiquement le 9 Mai 1980
devant la Commission d'Examen

á L'Institut National Polytechnique de Lorraine

pour l'obtention du grade de

DOCTEUR ES-SCIENCES MATHEMATIQUES

par

Jean-Pierre THOMESSE

R. HUSSON
C. KAISER

MM. P. HUGOT
M. SINTZOFF

INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE

E.N.S.E.M NANCY

SYGARE : UNE STRUCTURATION POUR LA CONCEPTION D'APPLICATIONS EN TEMPS REEL ET REPARTIES

THESE

présentée et soutenue publiquement le 9 Mai 1980
devant la Commission d'Examen

à L'Institut National Polytechnique de Lorraine

pour l'obtention du grade de

DOCTEUR ES-SCIENCES MATHEMATIQUES

par

Jean-Pierre THOMESSE

Président M. C. PAIR

Rapporteurs MM J.C. DERNIAME
R. HUSSON
C. KAISER

MM. P. HUGOT
M. SINTZOFF

A mes Parents

A ma Femme

A Guillaume et Anne-Sophie

Je voudrais ici remercier tous ceux qui ont contribué directement ou non au travail que je présente aujourd'hui, tous ceux qui m'ont formé et aidé tout au long de mes études et de mes recherches.

Je remercierai vivement

Monsieur C. PAIR, Président de l'Institut National Polytechnique de Lorraine, non seulement pour l'honneur qu'il me fait en présidant ce Jury mais aussi pour tout ce qu'il m'a apporté au cours de mes études, mes débuts de chercheur et d'enseignant.

Monsieur J.C. DERNIAME, Professeur à l'Université de NANCY I pour ses enseignements, ses conseils et son aide permanente tout au long de ce travail.

Monsieur R. HUSSON, Professeur à l'Ecole Nationale Supérieure d'Electricité et de Mécanique de NANCY pour son accueil chaleureux au Laboratoire d'Automatique et ses conseils amicaux.

Monsieur C. KAISER, Professeur au Conservatoire National des Arts et Métiers qui a bien voulu accepter de juger ce travail et qui par ses critiques et conseils a contribué à son amélioration.

Monsieur P. HUGOT, Directeur d'Unité à la STERIA et Monsieur M. SINTZOFF, Ingénieur à la Société PHILIPPS qui se sont intéressés à ce travail et qui me font un grand honneur en participant à ce Jury.

Je voudrais aussi remercier

- ceux qui par leur collaboration ont contribué à ce résultat et spécialement

Alain COCHET-MUCHY, Philippe NONN, Raymond SCHNEIDER et Pascal CAUMONT.

Dominique GERLL, Ingénieur à SGN qui m'a intéressé aux applications en conduite de procédés et qui m'a fait profité de sa grande expérience industrielle.

Tous mes Collègues et Amis du L.E.E.A. et du C.R.I.N. avec lesquels j'ai pu travailler et mener de fructueuses études.

Madame DALBOURG et Madame MARCHAND qui avec un grand dévouement et beaucoup de compétence ont assuré la réalisation de cet ouvrage.

SOMMAIRE

CHAPITRE 1

POURQUOI SYGARE* ?

* *SYGARE* *Système de conception et gestion d'applications en temps réel et réparties.*

Une application en conduite de procédés industriels a pour objet, la surveillance et la commande d'un phénomène physique, à partir de grandeurs mesurées, d'événements pris en compte, de consignes, selon une loi et des règles prévues à l'avance.

Une telle application se distingue des autres applications informatiques par le fait qu'elle est en prise directe sur le monde extérieur réel représenté par le procédé industriel. Le procédé est souvent composé de plusieurs activités physiques auxquelles correspondent plusieurs processus logiques qui, selon les activités, coopérent en étant parfois concurrents quant à l'utilisation de certaines ressources. Il faudra exprimer cette coopération et la gestion de la concurrence.

Comme tous les grands programmes, il est important de les écrire de façon modulaire. De plus, dans le cas de la conduite de procédés, on n'a pas d'unité de lieu en ce qui concerne la prise d'informations et la commande d'actions sur le procédé physique. Le procédé est en général réparti géographiquement.

La modularité de l'application devra donc aussi pouvoir tenir compte de la répartition du procédé.

Le fait que le système informatique soit en prise directe sur le procédé se traduit aussi par le respect de contraintes de temps parfois très sévères. Ces contraintes peuvent impliquer une répartition des actions à entreprendre pour profiter au maximum d'un parallélisme vrai.

Pour des raisons de dimensionnement du système informatique, compte tenu de ces contraintes, et pour des raisons liées à la sûreté de fonctionnement, en autorisant des fonctionnements en performances dégradées, il est important de concevoir des applications pouvant être implantées de façon répartie. Mais au moment même de la conception, on aimerait ne pas avoir à tenir compte de ces contraintes.

De même pour permettre des reconfigurations de l'application, il nous faudra pouvoir faire abstraction de l'implantation future

des programmes au moment de leur écriture. Ainsi, il serait agréable de disposer d'un langage de haut niveau qui autorise que la même description d'une application, dans ce langage, puisse conduire par une étape, qu'on pourrait qualifier d'implantation, à diverses réalisations, complètes voire dégradées.

Nous allons étudier ci-après, les divers concepts qu'il est nécessaire d'exprimer dans un langage permettant une programmation modulaire et structurée d'applications en temps réel et réparties. Compte tenu des objectifs et des solutions habituelles, nous en déduirons les lignes directrices de notre proposition (§§ 1.3 et 1.4) que nous comparerons alors aux travaux actuels dans le même domaine ou dans des domaines voisins. (§§ 1.5 et 1.6)

1.1 LES CONCEPTS A EXPRIMER DANS UNE APPLICATION TEMPS REEL REPARTIE

Les concepts que nous voulons exprimer dans une application en conduite de procédé sont relatifs :

- au calcul arithmétique, logique et en général l'expression des algorithmes ;
- à la programmation des opérations d'entrée-sortie sur des périphériques très variés, avec ou sans attente de fin d'opération pour continuer l'exécution du programme ;
- à des temporisations pour retarder certaines actions, pour détecter des absences de réponse, et plus généralement une absence d'événement pendant un délai ;
- la synchronisation avec d'autres éléments de l'application : activation d'une tâche sur un événement positionné, activation périodique, attente de l'arrivée d'une tâche à un point donné ;
- communication de données entre "morceaux" de programme qu'ils soient résidents ou non sur le même site ;

- synchronisation des tâches quant à l'utilisation d'une ou plusieurs ressources ;
- conditions d'exécution éventuellement spécifiées par des commandes de l'opérateur.

En fait on peut regrouper en trois classes les concepts ci-dessus énumérés :

- programmation séquentielle et algorithmique
- synchronisation
- communication impliquant aussi une synchronisation.

On ne distingue souvent que deux classes, les actions et le contrôle ([PETERSON - 74] par exemple, on verra plus loin que cette idée est à la base de nombreux autres systèmes).

Bien qu'il soit possible effectivement de ne pas distinguer synchronisation (impliquant une communication, ne serait ce qu'un signal) et communication, impliquant une synchronisation, nous ferons cette distinction. Nous pensons en effet qu'il serait pratique de les spécifier différemment, même si ce sont les mêmes outils de base qui sont utilisés pour l'implantation. Ceci nous permet d'envisager une description plus "naturelle" de l'application car plus proche des concepts manipulés par l'utilisateur.

- Enfin, il faut exprimer la répartition sur les sites des "morceaux" de programme qui constituent l'application.

1.2 COMMENT SONT RESOLUS HABITUELLEMENT CES PROBLEMES ?

Si on excepte les systèmes relevant de la catégorie des automates programmables dans lesquels une "tâche" monolithique est exécutée en permanence, la programmation de telles applications est, dans les systèmes commercialisés, réalisée sous forme de tâches temps réel.

Une tâche temps réel est un module de chargement résident ou non en mémoire centrale, construit de façon monolithique, bien que mettant en oeuvre des sous-programmes et parfois des techniques de recouvrement. A l'intérieur d'une telle tâche on trouve des instructions relatives aux concepts énumérés précédemment.

a) Les algorithmes sont programmés la plupart du temps en assembleur, parfois en Fortran. On ne peut pas en général exprimer d'algorithmes parallèles. Les inconvénients sont :

- le langage n'est pas structurant par lui-même
- la programmation est une activité intégrant la connaissance du problème, d'une solution et de la machine
- la programmation et la mise au point sont longues, coûteuses et peu sûres
- les programmes sont difficilement portables, inconvénient majeur dans le cadre des systèmes répartis.

b) Les opérations d'entrée et de sortie sont réalisées par des requêtes au système d'exploitation, un paramètre indique le mode de synchronisation désiré : attente ou non de la fin de l'opération.

c) Les temporisations sont aussi programmées par des requêtes du type positionnement d'un événement au bout d'un délai donné. Il en est de même pour les synchronisations, soit à base de sémaphores, soit à base d'événements.

d) La programmation de la communication de données dépend essentiellement de l'implantation envisagée

- "send" et "receive" quand les programmes sont sur des sites différents
- passage de paramètres quand les programmes exécutables sont résidents dans une même mémoire
- tampon commun résident quand ils ne le sont pas a priori.

La gestion des accès au tampon est alors à la charge de l'utilisateur ; on retrouve les problèmes de synchronisation d'accès à une ressource.

e) A ce sujet, les outils couramment utilisés sont les sémaphores et les primitives P et V de DIJKSTRA [1966].

Les principaux inconvénients sont liés au fait que tous ces éléments sont programmés à l'intérieur d'une même tâche dans un "morceau" de texte souvent compilé en une seule fois (à l'exception des sous-programmes). Ceci nécessite de se préoccuper en même temps de tous les problèmes répertoriés ci-dessus (§ 1.1).

L'application est relativement figée. Un changement d'implantation est impossible car remettant en cause tout ce qui est communication et synchronisation. Ceci implique aussi qu'une tâche écrite correctement à un instant donné ne l'est plus quand ultérieurement on lui en adjoint une autre (pour des problèmes de communication, d'accès à une ressource... qui n'avaient pas été prévus au préalable).

Les primitives de synchronisation sont multiples et diffuses dans les programmes. On n'a aucune vue d'ensemble, et il n'est pas facile de dire que des interblocages n'interviendront pas. Une modification remet en cause l'ensemble de l'application. Il est quasi impossible dans un tel contexte de prouver que la moindre solution est correcte. Il est en plus impossible d'appréhender le temps de réponse du système à une sollicitation extérieure, variable selon la charge du système. Des systèmes de priorité ont été proposés, qui en permettant le respect de certaines contraintes de temps, n'en modifient pas moins le déroulement spécifié par les primitives de synchronisation et l'enchevêtrement de diverses tâches.

De plus, les outils utilisés sont de niveau relativement bas (les sémaphores). S'ils sont toutefois adaptés à la programmation d'application sur des matériels à mémoire commune, ils ne le sont plus en milieu réparti.

On remarquera que ce qu'on appelle "Fortran temps réel" est plus un système de multiprogrammation dont les requêtes sont

exprimables en Fortran, qu'un véritable langage temps réel. D'autres langages ont vu le jour, et ont les mêmes caractéristiques, construits autours d'Algol 60. Il y a des liens étroits entre le langage et le système d'exploitation de la machine. Cette tendance fut particulièrement accentuée avec des produits comme PROCOL* [RITOUT et Al - 72], [STERIA] ou PEARL** [ELZER - 75], [BRANDES et Al - 72] dans lesquels le langage n'est qu'une composante intégrée dans un système complet. Le "produit" PROCOL comporte ainsi

- un compilateur
- un générateur de système, qui permet de construire un moniteur adapté, comme pour tous les systèmes d'exploitation, à la configuration matérielle, mais aussi à l'application qu'elle est chargée de supporter,
- un chargeur, éditeur de liens
- un moniteur temps réel.

PROCOL et PEARL ne sont pas les seuls représentants de cette génération de produits. On pourrait aussi citer CORAL-66 [WEBB - 75] développé au Royaume-Uni, LTR [PARAYRE et Al - 75] langage de la marine française.

Une excellente comparaison des langages temps réels et des systèmes a été faite par GERTLER et SEDLAK [1975]. Ils permettent une meilleure expression des concepts qu'un langage comme Fortran. Toutefois, leur principal inconvénient est leur manque de modularité conduisant à une certaine lourdeur et à d'importantes difficultés de mise au point. En effet, les algorithmes, les opérations d'entrée et sortie, les synchronisations sur événement, les attentes etc.. sont programmés à l'intérieur des mêmes textes, des mêmes "morceaux" de l'application. Toute modification peut conduire à des erreurs difficiles à déceler ; le coût est dans certains cas tellement élevé que dès que la taille des programmes est importante, il devient prohibitif de les modifier. De plus, les opérations de communication n'étant pas spécifiées explicitement, elles ne sont pas synchronisantes.

---

* PROCOL Process Control Oriented Language

** PEARL Process and Experiment Automation Real time Language

C'est à partir du recensemment des besoins d'expression de tous les concepts déjà cités que nous avons proposé la structuration et le langage de SYGARE.

1.3 QU'EST CE QUE S Y G A R E ?

Dans le projet SYGARE [THOMESSE - 77], [COCHET - MUCHY et Al - 77] nous proposons une structuration pour la conception et la programmation d'applications réparties en conduite de procédés industriels. Cette structuration doit nous permettre de faire abstraction des matériels utilisés pour l'implantation de l'application et de son caractère réparti ou centralisé. Elle s'apppuie sur un langage spécifique et un système support.

1.31 Algorithmes

Pour la programmation des algorithmes, nous voulons disposer d'un langage de haut niveau, qui permette effectivement une programmation des traitements, indépendante de l'implantation ultérieure. Nous voulons d'autre part pouvoir exprimer que certaines parties de l'algorithme sont parallèles , par exemple pour respecter des contraintes de temps. Les algorithmes seront exprimés de façon modulaire au sens classique du terme. On pourra découper ou assembler des morceaux d'algorithmes selon certaines règles.

Comme nous nous adressons aussi à des utilisateurs automaticiens ou électroniciens, nous avons structuré ces morceaux d'algorithme, que nous appellerons des modules, sous forme d'automate. Un module sera une "boîte noire" avec des entrées, des sorties et des variables d'état. A la limite, on doit pouvoir assembler des modules comme on assemble des circuits intégrés digitaux. C'est la première modularité qu'offre notre langage.

L'expression de l'assemblage de ces morceaux se fera indépendamment des modules eux mêmes. On voit ici apparaître une seconde modularité moins classique qui consiste à écrire séparément des éléments relevant de centres d'intérêt différents.

1.32 Synchronisation

Pour l'expression des synchronisations, nous voulons pouvoir utiliser indifféremment le concept de condition ou d'événement, contrairement aux systèmes qui n'offrent en général que l'un ou l'autre de ces concepts.

Cette expression concerne aussi bien la synchronisation de programmes entre eux, que la synchronisation de programmes avec le milieu extérieur, par l'intermédiaire des messages avec l'opérateur, et celui des entrées-sorties industrielles, analogiques et numériques, tout ceci constituant l'environnement.

1.33 Communications

Nous spécifions complétement le flux de données entre les divers constituants de l'application. Nous pourrons ainsi acquérir une large indépendance vis à vis de l'implantation. Ceci est très important car le respect des contraintes de temps peut obliger à multiplier les processeurs pour répartir les charges, après que l'application ait été écrite. Il est alors agréable de ne pas avoir à la modifier.

La sureté de fonctionnement de plus en plus recherchée nécessite des possibilités de reconfiguration et de fonctionnement en performances dégradées qui se traduisent dans la pratique par une nécessité de programmation indépendante de l'implantation.

De plus, nous pourrons exprimer directement la synchronisation des éléments communicants, sans avoir à se ramener à un problème de synchronisation à résoudre avec les outils adéquats.

1.34 Définition statique du parallélisme

Les opérations que nous introduirons pour exprimer les synchronisations naturelles et les synchronisations à propos de communication seront identiques, qu'il s'agisse d'événement, de condition, de données communiquées, à savoir : la production, la consommation, la consultation et la duplication.

C'est cette similitude de traitement qui nous permettra d'envisager une description relativement naturelle de l'application.

Les buts d'indépendance vis à vis de l'implantation et d'expression de synchronisation à partir d'une définition de données communiquées, d'évènement, de condition nous ont conduit à proposer une programmation du type "déclaratif" plutôt que sous forme de séquences d'instructions, en particulier en ce qui concerne la spécification du parallélisme, extérieure au texte des actions. Cette caractéristique importante fait que notre système offre une définition statique des synchronisations, au sens d'indépendance de toute exécution. Elle implique aussi qu'on ne puisse autoriser la création dynamique de processus, ce qui est vrai dans la majeure partie des cas en conduite de procédés industriels.

On veut plutôt disposer d'un ensemble de modules et de tâches qu'on puisse assembler de diverses manières afin d'obtenir diverses réalisations, complètes ou dégradées. On n'acceptera pas de créer en ligne de nouveaux processus, mais seulement d'établir ou de détruire un lien entre une tâche existante et des conditions d'activation.

1.4 CONSTITUANTS DE SYGARE ET PLAN DE CE TRAVAIL

Un élément de SYGARE concerne la spécification de l'implantation des morceaux qui constituent l'application.

Nous allons passer en revue ces constituants qui permettent en fait la programmation des concepts nécessaires à la définition d'une application temps réel répartie.

Ces constituants sont au nombre de quatre :

- les modules qui expriment les algorithmes de traitement. Ils contiennent des instructions dont certaines structures de contrôle dites de niveau 1 ;

- l'enchaînement de modules coopérant à une même tâche sans aucune notion de concurrence. Cet enchaînement est spécifié par des structures de contrôle dites de niveau 2 ;

- les déclarations de connexion qui expriment la communication des données entre les divers modules de l'application. Cette expression permettra la spécification de certaines synchronisations, grâce au concept de données transmissibles et aux opérations de consommation et consultation ;

- la définition d'événements et de conditions qui associées à certaines structures permettent une déclaration des règles d'activation des tâches qui constituent l'application. Les structures utilisées seront dites de niveau 3.

A chacun de ces éléments est associé un langage.

Nous avons jusqu'ici utilisé des termes comme, module, tâche, événement, condition dans leur acceptation courante. Des définitions précises seront données ultérieurement.

La première partie de ce travail est consacrée aux propositions que nous faisons en ce qui concerne la structuration de l'application : les notions de module, de structure d'une tâche et des objets échangés sont présentés au chapitre 2.

Les problèmes de parallélisme et de synchronisation sont traités aux chapitres 3 et 4, d'abord en liaison avec la communication de données puis avec les concepts d'événements, de condition.

L'ensemble des deux volets

- structuration
- spécification des synchronisations

fait que notre langage peut servir directement comme langage de spécification de solution à un problème posé. On peut alors considérer notre proposition comme une alternative à d'autres outils tels que le GRAFCET [BLANCHARD - 78] ou les Réseaux de PETRI [1965] quand, ils sont utilisés comme outils de synthèse.

Par ailleurs, il se veut un langage de programmation et bien que toutes les propositions qui figurent dans cette thèse ne soient pas complètement implantées, les réalisations actuelles permettent de dire que SYGARE est un langage de programmation.

Les réalisations partielles sont présentées, ainsi que quelques autres possibles dans la seconde partie. On y étudie d'abord (chapitre 5) l'implantation des modules, puis celles de leur enchaînement par les structures de contrôle de niveau 2, ainsi que celle des événements.

Le chapitre 6 est consacré à l'implantation des données communiquées entre les modules. On y présente plusieurs solutions possibles. Nous y montrons aussi comment on peut en SYGARE décrire le fonctionnement des entrées-sorties physiques.

Enfin, dans une troisième partie, nous donnons quelques éléments d'aide à la conception par l'intermédiaire de règles de découpage d'une application en disant dans certains cas, ce qui constitue une tâche ou un module.

Il y a encore beaucoup à faire dans ce domaine. Nous citerons toutefois à ce niveau les travaux en cours de SCHNEIDER [1980] qui utilise les réseaux de PETRI comme outil de modélisation des divers éléments de notre structuration selon SYGARE, modélisation à partir de laquelle une conception assistée par ordinateur peut être envisagée. Nous utilisons donc les réseaux de PETRI plutôt comme un outil d'analyse d'une solution spécifiée par ailleurs que comme un outil de spécification, comme le notent DEMUYNCK et MEYER [1979].

Une application est donc composée d'éléments de 5 types. Nous allons donné ici un exemple purement informel afin d'illustrer déjà ce que pourra être l'architecture de l'application.

a) Un ensemble de modules. Notons les :
M1, M2, M3, M4, M5, M6 ;

b) Un ensemble de structures de contrôle de niveau 2,
notées $ST2_1$, $ST2_2$, $ST2_3$
chacune d'entre elles exprime la structure d'une tâche

fonctionnelle T1, T2, T3 ;

par exemple on voudra exprimer que la tâche

T1 est composée de l'enchaînement séquentiel de M1, puis M2 et M3 dans cet ordre,

on voudra exprimer que la tâche T2 est composée de deux modules M4 et M5 s'exécutant en parallèle et que T3 n'est composée que du module M6.

Les structures de niveau 2 décrivent ces divers enchaînements indépendamment des modules. Ceci permet de répartir les modules sur des sites différents, en particulier quand il y a parallélisme vrai ce qui ne serait plus possible si les structures étaient intégrées dans le corps des modules. Les deux éléments a) et b) permettent la programmation de tous les algorithmes.

c) Un flux de données entre les modules d'une même tâche et entre les modules de tâches différentes. Le flux de données exprimera d'ailleurs certaines règles de synchronisation grâce aux opérations de consommation et consultation ;

par exemple on voudra exprimer que

une sortie de M1 est connectée à une entrée de M2

une sortie de M2 est connectée à une entrée de M4 et à une entrée de M6,

nous exprimons ici toutes les communications de données entre les constituants. Il ne s'agit que de déclarations ; la communication effective sera assurée de diverses façons selon l'implantation des modules par exemple. Il pourra s'agir de transmission de la valeur comme d'un accès à une mémoire commune.

d) Description des événements et ces structures de contrôle de niveau 3; nous définissons ici d'abord des liens entre les événements extérieurs ou intérieurs au système informatique, et les tâches fonctionnelles concernées à titre d'exemple, on peut vouloir déclarer

un événement "alarme" quand une sortie de M3 dépasse un seuil S

un événement "démarrage" est signalé quand une entrée binaire passe de 0 à 1

un événement "fin de lecture" est une interruption émise par un convertisseur analogique numérique.

On reliera les événements et les tâches fonctionnelles par des structures de contrôle de niveau 3 :

par exemple

sur alarme faire T3

sur fin de lecture faire T1

sur démarrage faire T2

e) Une spécification de l'implantation des éléments décrits ci-dessus, nous ne proposerons ici qu'une esquisse de langage d'implantation :

sur le site 1 implanter M1, M4

sur le site 2 implanter M5 etc ...

Les divers éléments peuvent être spécifiés dans un ordre quelconque

- flux de données avant les modules concernés
- structure d'une tâche avant les modules qui la composent
- modules avant leur enchaînement
- événements et structures de niveau 3 avant les tâches concernées, etc...

1.5 OUTILS POUR LA CONDUITE DE PROCEDES INDUSTRIELS

D'autres travaux sont issus de constatations et analyses voisines des nôtres. Ainsi, se sont développées d'autres études sur ce même sujet.

Il nous semble que la principale amélioration soit due à deux idées successives :

d'abord le concept de modularité et d'abstraction introduit par PARNAS [1972], mais aussi sous diverses formes dans les travaux de DERNIAME [1974], KRAKOWIAK et son équipe [CHEVAL et Al - 76]. Ce concept est maintenant bien connu. L'introduction des types abstraits par LISKOV [1974 - 1975] marque la dernière étape dans cette démarche.

. Puis la séparation des expressions du traitement et du contrôle qui apparaît sous diverses formes : secrétaires, puis moniteurs [HOARE - 74], centralisant dans une capsule appelée "moniteur" l'ensemble des objets et de leurs procédures de manipulation. Les expressions de chemins d'HABERMANN [1975], les modules de contrôle de ROBERT et VERJUS [1977] sont dans cette ligne de recherches où le but essentiel est de pouvoir prouver qu'une solution est correcte.

D'autres structurations et outils ont déjà été proposés ; deux en particulier ont des objectifs semblables ou voisins des nôtres ; celle proposée par MENDELBAUM [1976] et celle de LADET [1976].

Dans [MENDELBAUM - 76] puis [LE CALVEZ et Al - 77], [LE CALVEZ - 79] sont présentées

d'une part une spécification par graphe

d'autre part le langage GAELIC de programmation associé.

Un graphe représente un schéma de contrôle qui contient tous les liens qui unissent les différents morceaux de l'application aussi bien algorithmiques que temporels. Un schéma est unique et global, pour une application donnée, et, bien qu'il comporte des

sous-schémas, on ne dispose pas de modularité pour l'expression du contrôle. La modularité n'existe qu'au niveau des textes des modules qui, ici aussi, représentent les traitements. Bien que n'existe pas en GAELIC notre notion de tâche, le schéma de contrôle peut être en première approche, considéré comme comportant l'ensemble des informations contenues dans nos structures de contrôle de niveau 2 et de niveau 3. Les déclarations de connexion n'ont pas d'équivalent. Les variables sont globales, la synchronisation des accès étant réalisée par une définition de section critique. Cette définition est assez peu facile à utiliser dans le graphe de spécification. Par ailleurs, la structure obtenue ne parait pas se prêter facilement à une répartition, à cause du manque de modularité.

Dans [LADET - 76] puis [DESCHIZEAUX et Al - 77], [LADET - 78] on trouve une proposition de structuration par uniquement une spécification de liens entre événements et processus. Une structure de contrôle POUR < événement > FAIRE < processus > avec des attributs permet d'utiliser six formes de liens. Les opérations d'entrées/sorties sont exprimées par ailleurs et font l'objet d'autres formes de synchronisation, cette structuration se prête bien à la répartition mais pour le moment, tous les problèmes de synchronisation ne sont pas résolus, en particulier en ce qui concerne la transmission de données. On accepte de ne pas traiter tous les messages sans autre forme d'indication. On n'a donc pas de modèle producteur - consommateur. Le choix a été fait délibérement pour une première réalisation [GRIESNER - 80] et des extensions sont en cours.

En ce qui concerne les méthodes de production et les systèmes opératoires, supports du langage on pourrait envisager, dans le cas d'une application répartie, un noyau de système sur chaque site, du type de celui proposé par VOJNOVIC [1977]. Ce noyau assurerait la communication et la synchronisation avec les autres. Il contiendrait un algorithme de signalisation [LE LANN - 79] analogue par exemple à celui de [DARGENT - 79]. Nous avons fait une première étude dans ce sens [NONN - 78]. C'est ce qu'a réalisé GRIESNER,

au contraire des travaux de LE CALVEZ ou l'application est implantée soit sous forme d'automates, soit grâce à PROCOL. Pour la production sûre et efficace de ces noyaux, il semble que les méthodes développées par BEZIVIN et son équipe [BEZIVIN et Al - 77], [BEZIVIN et Al - 78] soient particulièrement intéressantes.

1.6 EXPRESSION DU PARALLELISME ET DES SYNCHRONISATIONS

C'est bien évidemment le problème majeur dans la programmation d'une application en temps réel.

Les solutions à ce problème particulièrement étudié depuis une quizaine d'années [DIJKSTRA - 65] seront nombreuses et variées chacune présentant un avantage par rapport aux autres, mais aucune n'est considérée satisfaisante.

Le premier outil de programmation du parallèlisme et des synchronisations fut certainement le masquage et de la démasquage des interruptions, mécanismes permettant le parallèlisme, mais aussi cause de nombreux maux....

DIJKSTRA [1968] en introduisant les sémaphores et les primitives P.V. apportait les premiers outils véritablement utiles pour la programmation de ces problèmes. On remarquera d'ailleurs qu' au niveau matériel, ce sont toujours les seuls outils implantés sur la plupart des ordinateurs.

Notre but ici n'est pas de retracer l'histoire et l'évolution de l'expression du parallélisme, de nombreux auteurs l'ont déjà très bien fait [BRINCH HANSEN - 73] et plus récemment sous des formes différentes [ANDLER - 78], [MOSSIERE - 77], [KRONENTAL - 79]. Nous retiendrons seulement que l'ensemble des mécanismes proposés est adapté à une implantation sur monoprocesseur. Que ce soient les moniteurs [HOARE - 74] et [BRINCH HANSEN - 73], les régions critiques conditionnelles etc... A la limite, ces mécanismes peuvent être implantés sur des multiprocesseurs avec mémoire

commune comme la machine NARCIS [LAGIER - 76] et [VERNEL - 77], [LAGIER et Al - 78] ou la machine MICRAL de REE [GERNELLE - 77] en utilisant astucieusement les blocages matériels au niveau de l'accès à la mémoire commune.

Les recherches actuelles vont dans le sens de la définition de mécanismes les plus naturels possibles, les plus aptes à faire l'objet de preuves [ROBERT et Al - 77], [SINTZOFF et Al - 75], dans le sens de la spécification complète des règles d'accès aux ressources, mais aussi des processus de l'ordonnancement [ROUCAIROL - 78], [ROUCAIROL - 79] ; elles vont aussi dans le sens de la définition de mécanismes adaptés à l'implantation sur des réseaux de processeurs sans mémoire commune : [HOARE - 78], [BRINCH - HANSEN - 78].

La synchronisation de processus distants n'ayant pas accès à une même mémoire commune pose un problème dû au temps de transfert des informations synchronisantes.

Il n'y a plus de blocage élémentaire assuré par le matériel qui interdit deux opérations simultanées sur le même élément.

Notre préoccupation dans ce domaine va dans le sens de la spécification des synchronisations indépendamment de la façon dont les processus seront implantés. Nous proposerons donc une description des règles de synchronisation qui permette diverses implantations pouvant utiliser l'un des mécanismes proposés : habituellement les sémaphores, les moniteurs, les modules de contrôle.

Cette abstraction de l'implantation nous a conduit à définir une description de type statique. Cette description doit être la plus naturelle possible en entourant particulièrement la description des synchronisations indépendamment de celle des traitements. Habituellement la synchronisation est décrite en termes d'événements, de conditions, de partage de données. Le fait qu'on considère des applications réparties nous a amenés à introduire la transmission de données comme un concept élémentaire. La conception d'applications réparties nécessite en fait la spécification des échanges entre les "morceaux" constitutifs de l'application.

Les communications entre processus ne sont véritablement spécifiées que depuis que les réseaux se développent. A part, les sémaphores avec message [SAAL et Al - 70] et les mécanismes du système MU5 [MORRIS - 69], on a souvent supposé que la communication de données se faisait par mémoire commune, et le problème de communication devenait un problème de synchronisation des accès à un tampon partagé. Il est d'ailleurs intéressant de noter que les sémaphores avec messages ont été introduit parce qu'on s'est rendu compte qu'une activation d'un processus P2 par un processus P1 pouvait nécessiter une transmission d'informations. C'est en retournant le problème : une information étant produite (ici par P1), elle déclenche le processus qui doit la traiter (P2) , que les promoteurs des systèmes basés sur le "pilotage" par les données (data driven) ont été conduits à proposer leurs solutions [DENNIS - 75], [KOSINSKI - 73], [SYRE et Al - 78], concept que l'on retrouve dans SYGARE avec les connexions entre sortie et entrée de modules avec en plus, les opérations de consommation, consultation, production, duplication.

Dans notre système c'est cette programmation des connexions entre les modules qui servira de base à leur synchronisation, selon que l'utilisation des données sera une consommation, une consultation, une production.

Pour implanter de telles opérations sur des machines non prévues pour être effectivement pilotées par les données, on doit introduire des événements ou des conditions (par exemple : donnée produite) qui déclenchent l'action ou l'exécution du module (par exemple : consommation). La programmation de ces liens conditions, actions pourra être exprimée dans les structures de contrôle de niveau 3 d'une façon analogue à ce qui est fait dans la structure if des commandes gardées de DIJKSTRA [1974] ou des parc's de GRIEM [1976].

Dans les deux cas, que la synchronisation soit exprimée par des connexions entre sortie et entrée de module, ou par des événements ou des conditions, les opérations sur ces trois types d' objets seront les mêmes.

En ce qui concerne la formulation des communications ce sont essentiellement des liaisons virtuelles bipoint qui étaient établies entre processus communicants autour du concept de porte [WALDEN - 72], [ZIMMERMANN - 74]. La formulation que nous proposons permet des communications sur des liaisons virtuelles de type multipoint ou du type diffusion (un producteur, plusieurs consultants).

Il nous reste à signaler deux points que nous n'aborderons pas dans ce travail.

Le premier qui est lié au traitement des défaillances.

Le second qui est lié à la signalisation dans les réseaux et à leur fonctionnement. En particulier, nous supposons l'existence d'un protocole de bout en bout [LORRAINS].

En ce qui concerne le traitement des défaillances nous ne proposons pas de procédure d'exception au sens de ADA [ICHBIAH - 79] ni de directive d'abandon comme cela est proposé dans [LE CALVEZ - 79]. Grâce toutefois à nos déclarations d'événement (cf. chapitre 4) nous pourrons détecter des pannes du type :

"un module ne s'est pas terminé au bout de < durée >"

ceci est un événement. Il reste à lier cet événement à un traitement. Compte tenu de l'état du système à cet instant, on pourra spécifier des traitements différents, notre définition des modules permettant de connaître parfaitement l'état de l'application par la connaissance des sorties.

En ce qui concerne la signalisation dans les réseaux nous faisons l'hypothèse chaque fois qu'il le sera nécessaire qu'un certain algorithme permet d'ordonner des requêtes ou de résoudre une exclusion mutuelle entre processus distants.

REFERENCES DU CHAPITRE I

[ANDLER - 78]

[BEZIVIN et Al - 77], [BEZIVIN et Al - 78], [BLANCHARD - 78],

[BRINCH-HANSEN - 73], [BRINCH-HANSEN - 78], [BRANDES et Al - 72]

[CHEVAL et Al - 76], [COCHET - MUCHY et Al - 77]

[DARGENT - 79], [DEMUYNCK et Al - 79], [DENNIS - 71],

[DENNIS - 75], [DESCHIZEAUX et Al - 77], [DIJKSTRA - 65],

[DIJKSTRA - 68], [DIJKSTRA - 74]

[ELZER - 75]

[GERTLER et Al - 75], [GRIEM - 74], [GRIESNER - 80]

[HABERMANN - 75], [HOARE - 74], [HOARE - 78]

[ICHBIAH - 79]

[KOSINSKI - 73], [KRONENTAL - 77] , [LISKOV - 74, 75]

[LADET - 76], [LADET - 78],[LAGIER - 76], [LAGIER et Al - 78]

[LE CALVEZ et Al - 77], [LE CALVEZ - 79], [LE LANN - 79],

[MENDELBAUM - 76], [MORRIS - 69], [MOSSIERE - 77]

[NONN - 78]

[PARAYRE et Al - 75], [PARNAS - 72], [PETRI - 65]

[PLEYBER et Al - 77], [PRUNET et Al - 74]

[RITOUT et Al - 72], [ROBERT et Al - 77], [ROUCAIROL - 78]

[ROUCAIROL - 79]

[SAAL et Al - 70], [SCHNEIDER et Al - 75], [SINTZOFF et Al - 75]

[STERIA], [SYRE et Al - 78]

[TACONET - 78], [THOMESSE - 77]

[VOJNOVIC - 77]

[WALDEN - 72], [WEBB - 75]

[ZIMMERMANN - 74]

CHAPITRE 2

LA STRUCTURATION D'UNE APPLICATION

2.1 INTRODUCTION

Dans le premier chapitre nous avons présenté les grandes lignes de la structuration d'une application selon notre système. Nous allons ici détailler les éléments qui composent l'application. Cette structure pourrait rapidement être schématisée par des couches. Mais alors que lorsqu'on parle de couches, on pense souvent à des couches superposées et ordonnées comme l'envisagent CHININ [1978] ou NEBUT [1974], dans notre cas, il s'agit plutôt d'un découpage en unités non ordonnées correspondant à des centres d'intérêt différents, ici nous n'avons pas défini d'ordre entre les divers éléments de la structuration. A chacune des couches, correspond en fait le traitement d'un problème particulier

a - algorithmique
b - enchaînement de modules
c - parallélisme
d - liens entre tâches et environnement
e - communication de données
f - implantation des divers éléments constitutifs de l'application.

Pour chacun de ces divers points nous avons défini un langage particulier. De cette façon nous pouvons nous préoccuper d'un problème particulier en ignorant les autres.

En effet nous voulons pouvoir préciser les conditions d'activation d'une tâche sans l'avoir déjà écrite.

Nous voulons également pouvoir spécifier des connexions entre modules sans les avoir écrites.

A l'implantation elle-même de l'application correspond une couche de langage. On pourra en effet préciser comment les éléments correspondants aux points a, b, c, d, e doivent être implantés, sur quels matériels. Dans ce chapitre 2, toutefois, nous ne nous intéresserons pas à ce problème qui sera étudié aux chapitres 5 et 6. Nous verrons ce que pourrait être un langage de spécification d'implantation.

Nous étudierons successivement les concepts suivants :
- module (§ 2.2)
- structures de contrôle de niveau 2 (§ 2.3)

- connexions entre modules et tâches fonctionnelles (§ 2.4)
- structures de contrôle de niveau 3 (§ 2.5)

grâce auxquels les points a) à e) sont programmés.

Nous rappelons que les structures de contrôle de niveau 2 expriment les enchaînements de modules dans une même tâche, et que les structures de contrôle de niveau 3 expriment les liens entre les événements, les conditions et les tâches.

Nous avons dit que les couches n'étaient pas ordonnées ; ceci signifie qu'elles peuvent être écrites dans n'importe quel ordre en faisant abstraction de la manière dont les autres le sont. On peut dire en ce sens que les couches sont indépendantes entre elles. C'est cette indépendance, déjà soulignée au chapitre 1, qui fut un des points de départ du projet [THOMESSE - 77], [COCHET - MUCHY et AI - 77].

L'ordre de présentation de ces objets peut correspondre à une méthode de programmation ascendante. Il n'y a aucun a priori dans ce choix. A partir du moment où on permet de définir les éléments d'une application dans n'importe quel ordre, il n'y a pas de raison d'en privilégier un en particulier.

2.2 LE CONCEPT DE MODULE

2.21 Définitions - Généralités

Dans notre système, un module est l'expression logicielle d'un automate d'état fini.

Ainsi, une exécution d'un module revient à évaluer des sorties et un état final de module à partir d'entrées et d'un état initial. Il peut donc être défini comme représentant la programmation de deux fonctions et telles que

$$X_{t+1} = \varphi (X_t, E_t)$$

$$S_{t+1} = \psi (X_{t+1}, E_t)$$

où X est le vecteur d'état de l'automate
E représente les entrées et S les sorties
t, t+1 représentent la discrétisation du temps.

Nous avons dit "expression logicielle" car l'automate est décrit dans un langage de programmation, bien qu'au niveau de la description, on ne prédéfinit nullement l'implantation qui en sera faite.

Un module est une unité de texte composée de la déclaration du nom unique du module

- d'une partie de déclarations
- d'une partie d'instructions qui seront toujours exécutées séquentiellement. Les instructions représentent toujours les fonctions φ et ψ .

Le nom du module est unique en ce sens que deux exemplaires du même texte porteront des noms différents. Ce seront donc deux modules différents. Un nom désigne donc sans aucune ambiguïté, un module précis de l'application.

Comme l'entité "module" est entièrement définie : Entrées, Sorties, Variables d'état et fonction φ et ψ, le module sera considéré comme unité atomique pour notre système, à la fois dans son contexte de compilation et son contexte d'exécution.

Un module sera aussi une unité de répartition. Dans son contexte d'exécution, c'est aussi une entité indivisible [CROCUS - 75].

Un module est donc toujours exécuté séquentiellement à la vitesse du processeur sur lequel il est implanté. Le processeur doit posséder toutes les ressources nécessaires à l'exécution d'un module. Nous ne nous en préoccupons pas au moment de l'écriture. Toutefois, des facilités de décomposition de modules en d'autres modules permettent d'adapter la structuration de l'application aux matériels, selon leurs ressources respectives.

Dès qu'un module est lancé, il se déroule sans intervention sur son exécution, jusqu'à sa terminaison. Les ressources ne sont libérées qu'à la fin de son exécution. Aucun point de synchronisation n'existe à l'intérieur du module ; les seuls points de synchronisation possible sont à l'entrée et à la sortie du module. (cf. chapitre 3).

Par rapport aux modules de CIVA [DERNIAME - 74], les nôtres ne précisent pas seulement quels sont les objets utilisés mais aussi comment on les utilise, en entrée, en sortie, comme variables d'état.

Comme nous n'avons pas de variable globale, nous ne définissons dans un module que des noms locaux. Comme un module est l'expression d'un automate nous ne pouvons observer son comportement entre le moment où les entrées sont affichées et celui où les sorties sont évaluées. Ceci nous permet d'envisager simplement une séparation de l'expresssion du parallélisme de celle des traitements. Toutes les données doivent être allouées en entrée avant de pouvoir activer un module. Par contre, cette contrainte peut conduire à un découpage très fin chaque fois qu'on désire synchroniser un module avec un autre.

2.22 Les objets manipulés dans un module

Les objets manipulés dans un module sont :

- des entrées
- des sorties
- des variables d'état
- des variables auxiliaires.

Tous les identificateurs sont locaux au module dans lequel ils sont définis et seuls ces objets sont accessibles dans un module, par contre, les objets manipulés peuvent avoir des durées de vie différentes.

La durée de vie d'un objet désigné par un identificateur est caractérisée par l'un des attributs

ENTREE
SORTIE
ETAT ou l'absence d'attribut, il s'agit alors d'une variable auxiliaire.

Les entrées, sorties sont respectivement déclarées par :

ENTREE < liste d'identificateurs > ;
SORTIE < liste d'identificateurs > ;

Un identificateur déclaré en entrée est un nom interne au module ; il désigne un objet qui existe avant l'exécution du module et qui aura dû recevoir une valeur avant l'exécution.

Un identificateur déclaré en sortie désigne un objet qui reçoit éventuellement une valeur pendant l'exécution du module (par exemple pour une affectation). Seuls ces deux types de valeurs sont disponibles à l'extérieur du module. Ces valeurs d'entrée sont collectées avant l'exécution. Les valeurs de sortie sont disponibles après l'exécution du module.

Vis à vis de l'extérieur du module, seules ses entrées et ses sorties sont accessibles. Nous n'avons jusqu'ici défini aucune notion assimilable à un Global.

Les informations seront transmises entre modules par des connexions de sortie à entrée comme on le verra au paragraphe 2.4.

Les variables d'état sont déclarées par

ETAT < liste d'identificateurs > INIT < liste de valeurs >;

Une variable d'état se distingue d'autre part des entrées et des sorties en ce sens que l'objet qu'elle désigne n'est jamais accessible à l'extérieur du module ; Il s'agit de rémanents d'Algol. On verra toutefois qu'on peut implanter une variable d'état sous forme de variables d'entrée, sortie et de connexions.

Les valeurs indiquées derrière INIT sont les valeurs qui sont affectées aux objets variables d'état à la première exécution du module.

La durée de vie d'une variable d'état va de cette initialisation à la fin du système ou a une réinitialisation.

Exemple 2.1

Soit le module suivant

```
MODULE M ;
  ENTREE   E1, E2, X ;
  SORTIE   S1, S2, X ;
  ETAT     ST1, ST2   INIT   val1, val2 ;
           Algorithme
             :
             :
FIN DU MODULE M ;
```

A la première activation de M , ST1 et ST2 prendront respectivement les valeurs val1 et val2.

E1, E2 sont des entrées. Avant toute exécution E1 et E2 reçoivent une valeur. Tout se passe comme si on ajoutait avant l'algorithme les deux affectations E1 := ;
E2 := ;

S1 et S2 sont des sorties. Leurs valeurs sont calculées dans le module. Dans l'algorithme S1 et S2 doivent apparaître en partie gauche d'une affectation S1 := ..... ;
S2 := ;

X apparait à la fois en entrée et en sortie. Son comportement est donc celui d'une entrée, puis celui d'une sortie. Toutefois comme X a une valeur par ENTREE, il peut ne pas recevoir de valeur par affectation. C'est la valeur d'entrée qui sera communiquée en sortie si aucune affectation ne la modifie.

Nous n'avons pas ici appliqué les règles qu'on rencontre dans les langages à assignation unique [SYRE et A1 - 78], [COMTE et A1 - 79], dans lesquels E1 et E2 (comme X) ne pourraient pas être modifiés durant l'exécution du module. Ceci nous permet d'utiliser des entrées comme variables internes. La valeur ainsi calculée ne sera ni accessible à l'extérieur du module, ni sauvegardée d'une exécution à la suivante.

Nous ne pouvons donc pas assurer les mêmes contrôles que dans les langages à assignation unique. Par contre nous pouvons ne pas créer d'objets auxiliaires, créations coûteuses en temps et en mémoire.

Par contre ST1 et ST2 voient leurs valeurs conservées entre deux exécutions successives.

On verra plus loin (cf. § 2.4) qu'on peut transformer une variable d'état en une entrée et une sortie (comme X) moyennant quelques précautions.

2.23 Les instructions dans un module

Le langage de programmation d'un algorithme n'était pas un des points importants de notre projet. De nombreux langages existent, qui remplissent très bien cette fonction. Pour des simplifications d'implantation nous avons choisi [COCHET - MUCHY - 78] un langage assez proche de Pascal. L'ensemble des structures de contrôle utilisées pour l'écriture des algorithmes à l'intérieur des modules est désigné par "structures de contrôle de niveau 1". Il contient la conditionnelle simple et deux formes d'itération (Pour et tantque). C'est déjà plus que le minimum permettant la programmation de tout algorithme [LEDGARD - 75]. Nous reviendrons en détail sur ce point au chapitre 5. Ici, il est seulement important de noter que nous avons voulu que tous les algorithmes de l'application soient écrits dans le même langage.

Nous faisons abstraction de la machine sur laquelle l'algorithme sera exécuté, abstraction des ressources nécessaires et du contexte d'exécution.

> Il sera ainsi possible de modifier un algorithme sans se préoccuper de ses conditions d'activation (en particulier en ne les changeant pas), et réciproquement changer les conditions d'activation sans modifier l'algorithme lui-même.

Toutefois, particulièrement, pour des raisons de performance tout en respectant les déclarations d'entrée, de sortie, de variable d'état, il est possible d'utiliser tout langage de programmation existant pour écrire le code des modules (à condition que le système de production de programmes soit adapté).

2.3 LES STRUCTURES DE CONTROLE DE NIVEAU 2

La structure d'un module permet la décomposition d'un algorithme en structures purement séquentielles.

Pour décrire l'enchaînement de ces séquences, nous introduirons de nouvelles structures de contrôle qui seront dites de niveau 2. Elles comprendront les enchaînements séquentiel, parallèle, conditionnel et itéré.

L'exécution d'un module est purement séquentielle. Elle ne met en oeuvre qu'un seul processeur. Dès qu'une application est destinée à être implantée sur un réseau de processeurs, il est naturel d'exprimer le parallèlisme vrai qui peut être réalisé entre programmes séquentiels.

Ce parallélisme peut être réalisé pour accélérer le traitement afin de respecter des contraintes de temps. Il peut être également réalisé dans l'acquisition "quasi simultanée" d'un grand nombre d'informations représentant la photographie du procédé à un "instant" donné.

Nous dirons "quasi simultané" car l'instant sur un réseau n'est pas équivalent à un instant sur une seule et même machine. La photographie sera prise entre t et t + ε selon les machines.

Certains modules seront amenés à s'exécuter soit en série, soit en parallèle. Nous l'exprimerons par des structures inspirées des expressions de chemin [HABERMANN - 75] ;

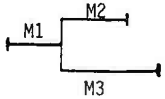
- parallélisme de deux modules

M1 // M2 ;

- modules séquentiels

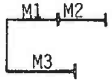
M1 - M2 ;

Le parenthésage est introduit pour exprimer toute combinaison sans ambiguïté



M1 - (M2 // M3) exprime : M1 est suivi de deux modules parallèles M2 et M3

(M1 - M2) // M3 exprime : il y a deux branches parallèles, l'une est constituée de M3, l'autre de deux modules séquentiels M1 et M2



Dans la réalisation actuelle [NONN - 78 b] nous avons considéré que l'opérateur // était prioritaire par rapport à l'opérateur-.(C'est une priorité du même type que celle de la multiplication par rapport à l'addition dans les expressions arithmétiques).

Ainsi, nous interprétons

M1 - M2 // M3 comme M1 - (M2 // M3).

La durée d'exécution d'un ensemble de modules parallèles est celle de la branche la plus longue en temps.

Si M2 et M3 sont prévus pour être exécutés en parallèle ils n'ont pas à se synchroniser entre eux en dehors du début et de la fin d'exécution.

Le parallélisme est ici sans concurrence. Les branches parallèles se déroulent effectivement en parfait parallélisme, si elles sont sur des processeurs différents. Dans le cas où elles seraient implantées sur un seul processeur, elles pourraient être exécutées dans n'importe quel ordre.

Les activités sont donc indépendantes. M2 peut être exécuté en parfait parallélisme avec M3 ou M2 peut être exécuté avant M3 voire M3 avant M2. Nous rappelons que les problèmes de concurrence seront traités aux chapitres 3 et 4.

2.32 La conditionnelle et l'itération

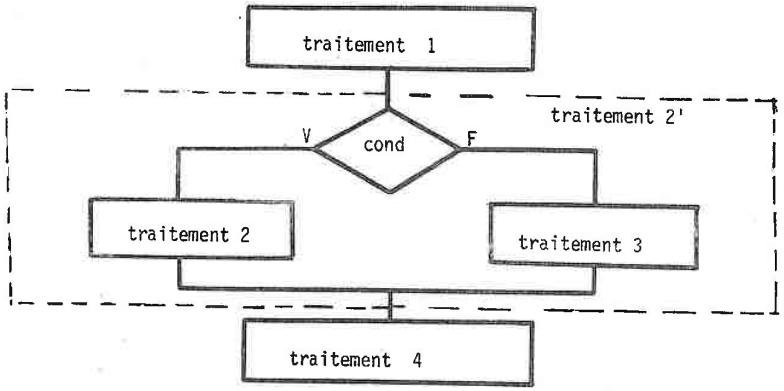
Nous avons réintroduit les structures de contrôle de niveau 1 de niveau 2. Ceci, afin de permettre toute décomposition (ou recomposition) de modules en plusieurs (ou à partir de plusieurs) modules.

Ces opérations ayant une incidence sur les entrées, sorties et variables d'état des modules concernés, nous reviendrons sur ce problème au paragraphe 2.5.

Signalons seulement que l'introduction d'une telle possibilité permet d'écrire des enchaînements conditionnels et des itérations sur un ou plusieurs modules extérieurement aux modules concernés.

Exemple 2.2

Soit un module M dont l'algorithme est constitué comme suit :



Une décomposition séquentielle permettrait de construire trois modules M1, M'2, M4, dont les algorithmes respectifs seraient traitement1, traitement2', traitement4.

La structure de contrôle au niveau 2 serait M1 - M'2 - M4
Pour diverses raisons, on peut avoir envie de décomposer traitement 2' en deux modules M2 et M3, contenant respectivement les traitements 2 et 3.

Les raisons de découpage sont multiples

- contraintes de répartition
- maintenabilité du module et de l'application
- réutilisation de petits composants.

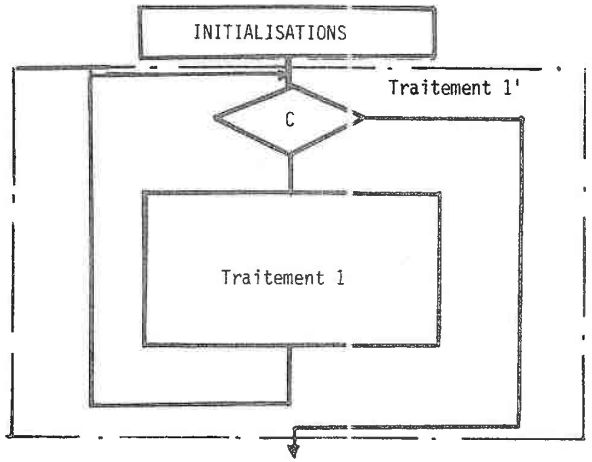
On est donc amené à écrire l'algorithme d'enchaînement des traitements à l'extérieur du texte des modules.

La structure de contrôle au niveau 2 devient alors :

M1 - si C alors M2 sinon M3 fsi - M4

Exemple 2.3

Soit un module M dont l'algorithme est constitué comme suit :



Une décomposition séquentielle permettrait de construire deux modules INIT et M1' dont les algorithmes seraient "initialisations" et "traitement1'" respectivement. A supposer que traitement 1

soit composé de modules parallèles s'exécutant sur un multiprocesseur.

Pour pouvoir l'exprimer nous devons écrire la structure de contrôle suivante au niveau 2 :

INIT - tant que C faire M1 ftq

où M1 est le module contenant l'algorithme "traitement1". Ainsi, si M1 est lui-même composé par exemple de M2 // M3 // M4, on écrira

INIT - tant que C faire M2 // M3 // M4 ftq

Comme nous l'avons déjà signalé, nous reviendrons sur ce sujet dans les chapitres 4 et 7 afin de préciser ce qu'est C dans les exemples 2.2 et 2.3, et ce que sont les entrées, les sorties, les variables d'état des divers modules intervenant.

2.33 La tâche fonctionnelle

On appellera "tâche fonctionnelle" l'algorithme formé par l'enchaînement de modules décrit par une structure de contrôle de niveau 2.

2.4 LES CONNEXIONS DE MODULES D'UNE MEME TACHE FONCTIONNELLE

Une connexion est un lien entre une sortie d'un module et une entrée d'un module. Elle représente une transmission de valeur entre la sortie et l'entrée précisées. Cette transmission pourra être réalisée de diverses manières, mais nous ne faisons aucune hypothèse à ce sujet.

On notera une connexion par une flèche indiquant le sens de transmission

sortie → entrée.

Nous avons dit que les noms des entrées et des sorties étaient locaux aux modules dans lesquels ils étaient déclarés.

Les connexions sont exprimées à l'extérieur des modules et elles mettent en oeuvre des objets déclarés à l'intérieur. Il convient donc de revenir sur la notion de localité des identificateurs. Les identificateurs d'entrée et de sortie d'un module sont locaux à ce module en ce sens qu'ils ne peuvent être manipulés et qu'ils ne peuvent être soumis à des traitements qu'à l'intérieur de ce module. Ils pourront toutefois être nommés à l'extérieur du module afin d'exprimer les liens entre sortie et entrée.

Afin de décrire sans ambiguïté, quelle sortie et quelle entrée sont connectées, on précisera une connexion en utilisant les noms locaux des entrées et des sorties avec le nom du module.

< identificateur de sortie > de < nom du module >
→ (signifie "connecté à")
< identificateur d'entrée > de < nom du module >

Exemple 2.4

Soient deux modules M1 et M2 définis comme suit :

Module M1	Module M2
entrée E1	sortie S2
⋮	⋮
fin du Module M1	fin du Module M2

Pour exprimer que M1 doit être exécuté avec comme valeur d'entrée E1, la valeur de S2 évaluée dans M2 on indiquera

S2 de M2 → E1 de M1

Cette déclaration de connexion est exprimée
- extérieurement aux modules M1 et M2
- extérieurement à la structure de contrôle de niveau 2 reliant les deux modules.

Toutefois, il est évident que des liens sémantiques existent entre une déclaration de connexion et la structure de contrôle de niveau 2 qui régit l'enchaînement des modules concernés.

Si on reprend l'exemple précédent, il est logique que le module M2 soit exécuté avant M1. Dans le cas général le contraire devrait être détecté comme une erreur ou tout au moins comme une déclaration incomplète.

Remarque :

On peut dire que la portée des identificateurs d'entrée et de sortie est variable selon l'endroit où ils sont utilisés. Dans un module la portée est le corps du module. Mais lié au nom d'un module dans une déclartion de connexion, la portée est l'ensemble des déclarations d'une même tâche fonctionnelle.

On verra au chapitre suivant qu'en fait la portée est l'ensemble des déclarations de connexion de l'application dès qu'on peut connecter des modules n'appartenant pas à une même tâche.

2.5 LES STRUCTURES DE CONTROLE DE NIVEAU 3

Les structures de niveau 3 établissent les liens entre l'environnement et les tâches fonctionnelles de l'application. Elles sont écrites indépendamment des tâches des modules qui composent celles-ci et des structures de niveau 2 qui les définissent.

Exemple 2.5

- Définition d'une tâche T1 (niveau 2)

  T1 : M0 - M1 - si C alors M2 sinon M3 fsi ;

- Exemple d'une structure de niveau 3

  Tous les 10 secondes faire T1 ;

Ces deux définitions étant séparées, on peut :

- modifier le corps de T1 sans avoir à se préoccuper de la structure de niveau 3 ;
- modifier la structure de niveau 3 sans modifier la définition de la tâche fonctionnelle T1 ;

Dans l'exemple 2.5, on peut changer la période ou même compléter la structure par exemple

Après Evénement tous les 11 secondes faire T1 ;

Nous rappelons qu'il est toujours possible de modifier le corps d'un module sans modifier la définition de la tâche et bien sûr la structure de niveau 3 associée.

Plusieurs structures de niveau 3 peuvent faire référence à la même tâche. Ces structures sont en fait l'expression d'un langage de commande de l'exécution de tâches en temps réel. Nous ne donnerons pas dans ce chapitre une liste des structures de niveau 3. Elle dépend des possibilités de définition d'événements. Le chapitre 4 répond à ces problèmes.

Toutefois, il faut souligner que toutes les tâches fonctionnelles sont implicitement parallèles et concurrentes, les points de synchronisation sont les seuls début et fin des modules qui les composent. La communication de données se fera également à chacun de ces points.

2.6 EXEMPLE D'APPLICATION CONSTITUEE D'UNE SEULE TACHE FONCTIONNELLE

Nous ne nous préoccupons pas ici de savoir comment l'application a été conçue. Nous nous contentons de donner le résultat de la conception afin de montrer comment les divers éléments peuvent être utilisés pour décrire une application.

Enoncé

- Quatre grandeurs doivent être mesurées en "même" temps. Un calcul s'en suit. Cette opération se répète périodiquement (toutes les Tsecondes).

Solution

. Niveau 3

Toutes les T secondes faire Tâche

. Niveau 2

Tâche : (M1 // M2 // M3 // M4) - Calcul

. Module M1

Sortie S

⋮

S :=

fin du module M1

Les Modules M2, M3, M4 sont supposés identiques à M1.

. Module Calcul

Entrée E1, E2, E3, E4

Etat I INIT 0 ; I compte les itérations

⋮

. fin du Module Calcul

Connexions

S de M1 → E1 de Calcul
S de M2 → E2 de Calcul
S de M3 → E3 de Calcul
S de M4 → E4 de Calcul

Remarque D'autres solutions étaient envisageables en définissant par exemple deux tâches fonctionnelles périodiques de période T : une constituée de M1 // M2 // M3 // M4 et l'autre de Calcul. On

verra au chapitre suivant que la synchronisation est alors réalisée par les déclarations de connexion.

2.7 CONCLUSION

Dans ce chapitre nous avons vu, les divers éléments qui constituent une application.

- Les modules qui, à la base du système permettent d'exprimer des traitements sous une forme très élémentaire, celle de l'automate d'états finis dans laquelle à une évaluation des équations correspond une évaluation du nouvel état et des sorties.

Ce point est très important comme on le reverra au chapitre 7. Il nous permettra en particulier de préciser des règles quant au découpage en modules.

- Les structures de contrôle de niveau 2 qui permettent de relier les modules entre eux sous forme d'un flot de contrôle.

Nous n'avons qu'entrevu les déclarations de connexion et les structures de contrôle de niveau 3.

- Structure de contrôle de niveau 3 exprimant les liens entre procédé industriel, événements et tâches.

- Les déclarations de connexion qui, complétées permettront d'exprimer une synchronisation entre modules de tâches différentes à propos des échanges de données (cf. chapitres 3 et 4).

Nous verrons enfin, au chapitre 4 comment ces deux éléments sont en fait deux expressions des mêmes concepts.

Comme nous l'avons signalé à propos des langages d'assignation unique, les déclarations d'entrée et de sortie sont à

rapprocher des déclarations INPUT et OUTPUT des systèmes "guidés par le flux de données" [DENNIS - 71], [DENNIS - 75]. Leur utilisation pour gérer la synchronisation des modules concurrents les en distingue toutefois. A ce sujet, nos déclarations seraient peut être plus proches des primitives GET et PUT de [KAHN et Al - 76] voire les primitives INPUT et OUTPUT de [HOARE - 78]. La différence essentielle avec ces systèmes sera détaillée au moment de la description du parallélisme.

Contrairement aux primitives d'entrée et de sortie pouvant être exécutées n'importe quand dans un corps de programme ou de procédure, nos spécifications d'entrée et de sortie sont des déclarations. La spécification est alors statique. L'interprétation de ces déclarations sera effectivement assurée par une transmission de données entre les modules connectés. Cette transmission toutefois, ne pourra se faire qu'à des instants privilégiés et parfaitement connus dès la spécification du système : avant et après l'exécution des modules. Ainsi, nous saurons quand et où faire les contrôles éventuels pour s'assurer du bon déroulement de l'application.

REFERENCES DU CHAPITRE 2

[CHININ - 78]

[COCHET MUCHY et Al - 77]

[COMTE et Al - 79]

[CROCUS - 75]

[DENNIS - 71], [DENNIS - 75]

[DERNIAME - 74]

[HABERMANN - 75]

[HOARE - 78]

[KAHN et Al - 76]

[LEDGARD - 75]

[NEBUT - 74]

[NONN - 78 b]

[SYRE et Al - 78]

[THOMESSE - 77]

CHAPITRE 3

LES CONNEXIONS ET LE PARALLELISME

3.1 INTRODUCTION

Nous avons signalé que les tâches fonctionnelles sont a priori parallèles et concurrentes éventuellement, en l'absence de spécification contraire (en imposant un ordre par exemple). Chaque exécution d'un module constitue un processus au sens de CROCUS [1975]. Les modules de tâches fonctionnelles différentes peuvent avoir à se communiquer des informations. Cette communication sera, comme dans le cas de modules d'une même tâche, spécifié par des connexions entre sortie et entrée, mais la forme des connexions présentées au chapitre précédent ne suffit plus à cause des conditions d'activation différentes de chacune des tâches. Nous proposerons donc les opérations supplémentaires de consommation, consultation et duplication avec un attribut précisant l'exemplaire concerné.

Ces opérations seront introduites à propos d'exemples.

Aprés avoir présenté quelques problèmes classiques de synchronisation (§ 3.2) nous définirons les opérations sur les données concernées par les connexions (§§ 3.3 à 3.5).

Nous présenterons ensuite des solutions en SYGARE aux problèmes de production - consommation, exclusion mutuelle, rendez-vous, lecteur-rédacteur (§§ 3.6 et 3.7). On ne traitera pas les problèmes d'ordonnancement dans ce chapitre parce qu'ils

s'expriment plus facilement en termes d'événements ou de conditions. On les traitera au chapitre 4 à la fin duquel on reviendra sur les connexions afin de montrer l'équivalence des concepts, donnée transmissible, événement, condition lorsqu'ils sont munis des mêmes opérations.

3.2 QUELQUES PROBLEMES ET LEURS SOLUTIONS CLASSIQUES

3.21 Producteur - Consommateur

Un des problèmes les plus classiques dans le domaine du parallélisme est celui des producteurs et des consommateurs.

Enoncé du problème [CROCUS - 75].

Nous reprenons textuellement ici l'énoncé qui figure dans l'édition de 1975 p. 37 et 38 :

Le schéma connu sous le nom de "modèle du producteur et du consommateur" permet de présenter les principaux problèmes de la communication entre processus par accès à des variables communes avec synchronisation. On considère deux processus, le producteur et le consommateur, qui se communiquent de l'information à travers une zone de mémoire, dans les conditions suivantes :

- *l'information est constituée par des messages de taille constante,*
- *aucune hypothèse n'est faite sur les vitesses respectives des deux processus.*

La zone de mémoire commune, ou tampon, a une capacité fixe de n messages (n > 0). L'activité des deux processus se déroule schématiquement suivant le cycle décrit ci-après :

PRODUCTEUR	*CONSOMMATEUR*
PROD : Produire un message ;	*CONS : Prélever un message*
Déposer un message	*dans le tampon ;*
dans le tampon ;	*Consommer le message ;*
aller à PROD ;	*aller à CONS ;*

On souhaite que la communication se déroule suivant les règles ci-après :

- *exclusion mutuelle au niveau du message : le consommateur ne peut prélever un message que le producteur est en train de ranger ;*
- *le producteur ne peut pas placer un message dans le tampon si celui-ci est plein (on s'interdit de perdre des messages par surimpression) ; le producteur doit alors attendre ;*
- *le consommateur doit prélever tout message une fois et une seule ;*
- *si le producteur est en attente parce que le tampon est plein, il doit être prévenu dès que cette condition cesse d'être vraie ; il en est de même pour le consommateur et la condition "tampon vide".*

A propos de cet énoncé nous ferons quelques remarques :

+ La séquence "produire un message" dans le processus producteur, et la séquence "consommer le message" dans le processus consommateur sont les séquences propres à l'application ; le dépôt et le retrait des messages sont des séquences purement techniques dont le texte est lié aux choix d'implantation des processus en coopération et du tampon.

+ Les deux instructions aller à PROD et aller à CONS expriment que les processus sont cycliques et infinis. La seule synchronisation est assurée par la présence ou l'absence de données ce qui est typique des systèmes "data drivens".

  En général les processus de production et de consommation ont d'autres conditions d'activation et il faut pouvoir les exprimer.

Ces remarques nous conduisent à la formulation suivante :

Le processus producteur (PROD dans l'énoncé) est en fait composé d'un premier module PROD1.

Le processus consommateur (CONS dans l'énoncé) est en fait composé d'un module CONS1.

Chaque processus a des conditions d'activation qui peuvent lui être propres et qui seront exprimées indépendamment des traitements : production et consommation.

En ce qui concerne uniquement la synchronisation entre PROD1 et CONS1, on a envie d'écrire que le message produit dans PROD1 est une sortie de PROD1, et que le message consommé dans CONS1 en est une entrée.

Ainsi l'expression de la synchronisation pourrait être quelque chose comme :

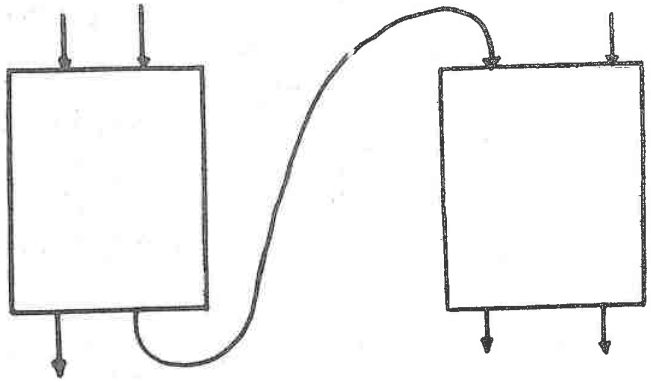
message produit de PROD1 → message de CONS1

avec

```
module PROD1
    sortie message produit
      ⋮     production
      ⋮
    message produit := ...
    fin du module PROD1
```

et


```
module CONS1
  entrée message
    ⋮
    ⋮ consommation
    ⋮
fin du module CONS1
```



Modèle Producteur - Consommateur

3.22 Exclusion mutuelle

Deux processus s'exécutent en exclusion mutuelle vis à vis d'une ressource dès que la ressource est à un seul point d'accès. La section critique d'un processus pour cette ressource est une phase du processus pendant laquelle la ressource est utilisée donc inaccessible aux autres processus.

Ce problème au niveau logiciel est dû au fait que la section critique d'un processus est en général une phase interruptible. C'est un problème habituellement résolu au niveau matériel par diverses méthodes (ex : un système séquentiel qui ne prend en compte qu'un changement d'entrée à la fois). Comme le faisait remarquer ANCEAU [1978] de nombreux problèmes ont des solutions matérielles simples.

Ainsi prenons deux processus P1 et P2 faisant à un certain moment respectivement I := f1 (I) et I := f2 (I).

P1 et P2 seraient programmés aussi en terme de sémaphores. Initialisation à I_0

P1	P2
⋮	⋮
1ère phase	1ère phase
P (sema$_i$)	P (sema$_i$)
phase critique : I := f1 (I)	phase critique : I := f2 (I)
V (sema$_i$)	V (sema$_i$)
⋮	⋮
3ème phase	3ème phase

P1 peut être décomposé en trois modules séquentiels correspondant respectivement aux trois phases 1, critique, et 3. Nous les noterons $P1_1$. $P1_2$; $P1_3$. Si nous considérons uniquement P1 le module correspondant à la modification de I dans P1 peut être considéré comme ayant I en variable d'état.

Module $P1_2$

état I Init I_0

I := f1 (I)

fin du module $P1_2$

Si nous ne considérons plus P1 comme unique processus, dès qu'un deuxième module ($P2_2$) utilise une variable d'état d'un autre module, il peut y avoir exclusion mutuelle entre les deux modules, alors qu'on aurait envie d'écrire :

Module $P2_2$

état I init I_0

I := f2 (I)

fin du module $P2_2$

et une connexion exprimant que I de Module $P1_2$ et I de Module $P2_2$ ne représentent qu'un seul et même objet. Il se pose alors un problème de cohérence dû à la copie d'une valeur d'un module dans un autre. En termes d'automates, dire qu'une variable d'état d'une machine est la même variable que celle d'une autre machine, n'a pas de sens.

En fait, on voit que le système est mal découpé et que les modules $P1_2$ et $P2_2$ devraient faire partie du même automate (ce qui simplifierait aussi la gestion de l'initialisation). Cette démarche a donné naissance aux moniteurs [HOARE - 74], le choix entre les deux procédures f1 et f2 internes au moniteur serait désigné par une entrée supplémentaire de l'automate.

Normalement le module équivalent au moniteur sera alors une tâche fonctionnelle à lui seul, devant être exécuté dès qu'une entrée positionnée le lui demande. Nous verrons que ce problème peut être résolu en termes de communication de données, qu'il s'agit d'un cas particulier, d'un cas général et que le concept de moniteur est trop restrictif, car ramenant tous les problèmes à des problèmes d'exclusion mutuelle.

3.23 Caractéristiques de notre solution

La solution que nous proposons pour la description des synchronisations appartient à l'ensemble des solutions où le "contrôle" est séparé du traitement. Il semble normal, en effet que le programmeur, au moment où il écrit un algorithme n'ait pas à se préoccuper de la synchronisation de cet algorithme avec un événement un phénomène, un autre algorithme, qui de plus peuvent ne pas être localisés au même endroit. Il en est de même quand on écrit les règles de synchronisation vis à vis des algorithmes.

Encore faut-il préciser ce qu'on entend par "séparément". Ainsi dans [ROBERT et Al - 77 a] il est précisé :

"Nous faisons l'hypothèse qu'il existe une solution totale entre les données du système de contrôle et celles des processus concernés".

Or un système de contrôle est là pour gérer des processus qui se communiquent ou se partagent de l'information. Notre idée est basée sur le fait que la synchronisation n'est là que pour régler les accès, l'usage, des informations communiquées et partagées. Cette synchronisation doit être issue de la spécification des traitement qui sont faits. Nous en proposerons une extérieure aux traitements.

L'hypothèse de ROBERT est contraire à celles qui permettent la description en termes de commandes gardées [DIJKSTRA - 74]. En effet, tous les systèmes qui utilisent ce mécanisme sont obligés de manipuler dans les actions, les données qui apparaissent dans les "gardes". Toutes les opérations sont permises. Une erreur dans une action peut ne pas être facilement détectée.

Nous pensons donc qu'il est nécessaire de réglementer les opérations autorisées sur les objets de module de traitement qui servent au contrôle. D'autre part, compte tenu du fait que le système de contrôle n'est parfois nécessaire que parce que les traitements sont implantés sur des machines aux primitives trop élémentaires, il est normal que le système de contrôle ne soit pas, de façon artificielle, indépendant des traitements.

Dans notre système nous autorisons l'usage des données communiquées dans l'expression du contrôle. Ces usages sont les mêmes que ceux qui sont autorisés dans les traitements et doivent être spécifiés au moment de leur utilisation.

C'est dans cette optique qu'il faut considérer le système que nous proposons dans ce chapitre.

On remarquera immédiatement que si les processus ne se communiquent pas explicitement des données, il sera difficile d'exprimer le schéma de contrôle. C'est le cas des problèmes qu'on qualifie habituellement de problèmes d'ordonnancement ; c'est pourquoi dans notre système une certaine forme de parallélisme est exprimée à un autre niveau par les structures de contrôle de niveau 3. Nous verrons d'ailleurs en fin du chapitre 4 que ces concepts sont équivalents et peuvent être interchangeables. On aperçoit à ce sujet comment on pourrait passer d'une application conçue selon une méthode "data driven" à la même application conçue en termes plus classiques d'événements et de structures de contrôle associées.

Si on s'en tient à la description de la communication des données dans une application conçue selon notre système, elle s'apparente aux langages "data flow" [DENNIS - 75], [SYRE et AL - 78]

Le système de coroutines de [KAHN et Al - 76] introduit des canaux virtuels entre processus alimentés par la primitive PUT et dans lesquels on puise par une primitive GET. Le concept d'entrée-sortie avec ou sans anticipation apparaît maintenant comme un des mécanismes de base pour la communication entre processus ; nous rappellerons que ce concept nous était déjà apparu intéressant [SCHAFF - 74], [THOMESSE - 74] pour exprimer la collaboration de deux ordinateurs par l'intermédiaire d'un protocole d'exécution de sous-programmes à distance.

3.3 A PROPOS D'UN EXEMPLE

Au chapitre précédent nous avons introduit les connexions entre sortie d'un module et entrée d'un autre module d'une même tâche fonctionnelle. Nous avons brièvement vu les structures de contrôle de niveau 3 qui liaient les événements aux tâches fonctionnelles. Il est bien évident que les événements, dans une application,

ne sont pas tous indépendants. Les tâches ne sont donc pas non plus indépendantes. Elles ont à collaborer entre-elles. Cette collaboration se fait par échanges de données en partageant des ressources. Or qui dit échange de données, dit synchronisation entre l'émetteur et le récepteur. Nous exprimerons la synchronisation de tâches parallèles par des connexions. Nous allons voir comment il faut compléter les connexions précédemment vues (chapitre 2) pour exprimer les différents cas de synchronisation.

Exemple 3.1

Deux modules M1 et M2 s'enchaînent séquentiellement dans une même tâche. Soient S une sortie de M1 évaluée à chaque exécution de M1, et E une entrée de M2, telles que S de M1 soit connectée à E de M2 :

Module M1	Module M2
sortie S	entrée E
:	:

connexion : S de M1 → E de M2

structure de contrôle de niveau 2 : M1 - M2.

Dans cet exemple simple, où l'exécution de M2 suit inconditionnellement celle de M1, la valeur de S communiquée à E est toujours la dernière valeur évaluée. Il n'y a a priori aucune ambiguïté sur la valeur de S transmise de M1 à M2.

Par contre, si on ne suppose plus que les modules appartiennent à la même tâche, mais à deux tâches différentes, il existe une ambiguïté quant à la valeur transmise entre les modules.

Exemples 3.2

Les modules M1 et M2 sont les mêmes que dans l'exemple 3.1, la connexion également ; ils sont dans deux tâches différentes T1 et T2 réduites respectivement à M1 et M2 :

structure de niveau 2 T1 : M1
T2 : M2

structures de niveau 3

A partir de 10 heures toutes les 2 secondes faire T1
sur événement1 faire T2

Toutes les deux secondes, une valeur de S est produite. Sur un axe gradué de 2 secondes en 2 secondes, notons les productions de S et l'arrivée de l'événement 1 supposé arrivé entre la ième et i + 1 ième production.

Quelle valeur prendra l'entrée E de M2 au moment de l'activation de la tâche T2 ? s_1, s_2 ou s_i ? Que signifie la connexion S de M1 → E de M2 ?

Nous sommes amenés à préciser d'autres renseignements qu'un simple lien entre le nom d'une sortie de module et celui de l'entrée d'un autre module. Les simples connexions vues au chapitre précédent sont insuffisantes à la description de la solution du problème de l'exemple 3.2. Nous allons voir aux paragraphes suivants comment les connexions seront complétées pour exprimer les synchronisations.

3.4 OPERATIONS DE TRANSMISSION ENTRE DEUX MODULES

3.41 Valeurs transmissibles

Une déclaration de connexion entre une sortie et une entrée de modules de tâches différentes définit la transmission d'un flot de valeurs successives : la suite $v_1, v_2, v_3, \cdots, v_\ell$ des valeurs de la sortie aux instants $t_1 < t_2 < t_3 \cdots\cdots < t_\ell$.

Une entrée prend ses valeurs successives dans l'ensemble des valeurs produites. Cet ensemble est initialement vide. Il est éventuellement infini : on ne fait aucune hypothèse sur l'implantation de cette suite de valeurs. On peut rapprocher cette suite de valeurs du flot de données mémorisées à un instant donné dans un "canal" selon [KAHN et A1 - 76] dans un mode d'exécution parallèle des coroutines. Le nombre de données mémorisées est le coefficient d'anticipation. Nous avons considéré que ce nombre est un problème d'implantation. On le verra plus loin, il apparaîtra quand on définira l'implantation d'une connexion. Ce sera par exemple la taille du tampon associé à une connexion.

Remarque :

Si on rapproche ces concepts des différents protocoles de transmission [LORRAINS - 79], on remarquera que ce coefficient d'anticipation permet une régulation du flux entre l'émetteur et le récepteur. Il correspond effectivement au nombre de blocs d'information qui peuvent être émis avant d'avoir reçu un quelconque acquittement. C'est la taille du tampon alloué à l'émetteur, dans la mémoire du récepteur.

Si à un instant donné, les valeurs transmissibles sont les valeurs successivement évaluées $v_k, v_{k+1} \cdots v_\ell$, nous dirons que cette suite est l'état de la connexion. Diverses opérations peuvent intervenir et sont susceptibles de modifier l'état de la connexion -

Ces opérations sont :

- la production d'une valeur
- la consommation d'une valeur
- la consultation d'une valeur

Nous considérons dans la suite les opérations sur la connexion type S de M1 → E de M2.

3.42 Production

Une production a lieu chaque fois que se termine le module M1 ; la dernière valeur de S est rangée dans la file des valeurs ; comme la file est toujours de longueur finie, on verra que selon les autres connexions, il pourra y avoir blocage ou non du producteur.

état initial de la connexion avant production

$$v_k \; v_{k+1} \cdots v_\ell$$

état final de la connexion après production

$$v_k \; v_{k+1} \cdots v_\ell \; v_{\ell+1}$$

où $v_{\ell+1}$ est égal à la dernière valeur de S.

3.43 Consommation

Une consommation est l'opération qui consiste à donner une valeur de la file à une entrée et à retirer cette valeur de la file. Le module demandant une consommation sera bloqué si la file est vide. Il pourra être débloqué dès qu'une production sera intervenue.

état initial avant consommation de la ième valeur

$$v_k \cdots v_{i-1} \; v_i \; v_{i+1} \cdots v_\ell$$

état final après consommation de la ième valeur

$$v_k \cdots v_{i-1} \; v_{i+1} \cdots v_\ell$$

état de l'entrée $E = v_i$

Quand on parle de la consommation, on sous entend habituellement "consommation du plus ancien exemplaire produit et non encore consommé" (ici v_k)

état final après consommation du plus ancien exemplaire

$$v_{k+1} \cdots v_{i-1} \; v_i \; v_{i+1} \cdots v_\ell$$

état de l'entrée $E = v_k$

état final après consommation du plus récent exemplaire

$$v_k \; v_{k+1} \cdots v_{\ell-1}$$

état de l'entrée $E = v_\ell$

3.44 Consultation

Une consultation est une opération qui consiste à lire une valeur dans la file sans la retirer de la file (au contraire de la consommation).

L'état de la file est donc inchangé ; l'entrée prend la valeur spécifiée : le plus ancien exemplaire, le dernier exemplaire produit, ou tout autre intermédiaire.

3.5 CAS DE PLUSIEURS MODULES - DONNEES TRANSMISSIBLES

3.51 Définition des données transmissibles

L'ensemble des valeurs transmissibles représente en fait les valeurs d'une donnée que nous appellerons donnée transmissible entre les deux modules concernés [THOMESSE et Al - 79]. Nous n'avons pas défini de noms pour désigner chacune des données transmissibles de l'application. Chaque donnée est parfaitement définie par le couple

(sortie d'un module, entrée d'un module). Lui donner un nom reviendrait à définir une pseudo variable globale puisque l'identificateur aurait pour portée, l'ensemble de l'application.

On n'avait pas besoin de définir de tels identificateurs tant que les modules n'étaient connectés que deux à deux. Or il est courant que plus de deux modules collaborent ou se partagent une même ressource. On a donc besoin de connecter plusieurs sorties à plusieurs entrées, sous une forme voisine du multiplexage / démultiplexage.

Nous donnerons donc un nom à une donnée transmissible, une connexion de sortie à entrée se transformera en deux connexions

sortie → donnée transmissible
donnée transmissible → entrée.

Ce que nous avons appelé "état de la connexion" devient l'état de la donnée transmissible. Il sera possible, plus généralement de définir des données transmissibles quand seuls deux modules sont connectés, mais ce ne sera pas nécessaire. Les connexions vues jusqu'à maintenant sont des cas particuliers d'une connexion définie comme suit.

Une connexion se décompose donc en deux parties

< sortie > de < nom de module > → < nom de donnée transmissible >

< nom de donnée transmissible > → < entrée > de < nom de module >

, { avec / sans } consommation

, exemplaire { récent / ancien }

Exemple 3.3

S1 de M1 → $\mathcal{D}$ 1

$\mathcal{D}$ 1 → E2 de M2, avec consommation ancien exemplaire

$\mathcal{D}$ 1 → E1 de M3, avec consommation ancien exemplaire

Ainsi plusieurs modules peuvent être connectés à la même donnée transmissible en entrée, ou en sortie.

M2 et M3 sont ici deux modules en concurrence pour la consommation des valeurs successives de S1 produites par le module M1.

3.52 Cas de plusieurs productions

Nous étudions ici le cas où plusieurs modules P_1, P_2 ... P_n possèdent chacun une sortie S connectée à une même donnée transmissible

S de P_1 → $\mathcal{D}$

S de P_2 → $\mathcal{D}$

⋮

S de P_n → $\mathcal{D}$

P_1, P_2 ... P_n se déroulent a priori en parfait parallélisme si chaque P_i est implanté sur un site particulier S_i.

Au fur et à mesure qu'un P_i se termine, la valeur de sa sortie S est rangée dans la file de $\mathcal{D}$ conformément au § 3.42.

Nous faisons l'hypothèse que toutes les productions sont ordonnées selon une relation d'ordre total. Cette relation n'existe pas dans un réseau. Par contre si toutes les terminaisons des modules P_i sont transmises à un même site S_c, elles peuvent être ordonnées par le site S_c. Ceci revient à ranger les valeurs successives en exclusion

mutuelle dans la file de la donnée transmissible. L'ordre de rangement est donc un ordre fixé par un site S_c.

Si l'état de la donnée $\mathcal{D}$ est avant les n productions v_k .... v_ℓ, après les n productions et en l'absence de toute consommation, l'état final de $\mathcal{D}$ est

$$v_k \cdots v_\ell \; v_{\ell+1} \cdots v_{\ell+n}$$

où $v_{\ell+1}$ ... $v_{\ell+n}$ est une permutation des sorties S de P_1 ... P_n,

si chaque P_i n'est exécuté qu'une fois et une seule. Si ce n'est plus vrai, après m productions, l'état final de $\mathcal{D}$ est $v_k \cdots v_\ell \; v_{\ell+1} \cdots v_{\ell+m}$ où $v_{\ell+1}$, ... $v_{\ell+m}$ est un arrangement avec permutation des sorties S de P_1 ... P_n.

3.53 Cas de plusieurs consommations

Nous étudions ici le cas où plusieurs modules C_1 ... C_m possèdent chacun une entrée E connectée à une même donnée transmissible $\mathcal{D}$

$\mathcal{D}$ → E de C_1 , avec consommation

$\mathcal{D}$ → E de C_2 , avec consommation

⋮

$\mathcal{D}$ → E de C_m ; avec consommation

C_1, C_2 ... C_m peuvent se dérouler a priori en parfait parallélisme si chaque C_j est implanté sur un site S_j et que toutes les ressources nécessaires à chacun d'entre eux sont disponibles à la fois. En particulier, il faut qu'au moins m valeurs soient dans la file de $\mathcal{D}$ pour que les m modules C_1 ... C_m puissent être actifs en même temps. Comme il s'agit de consommation dès qu'une valeur a été donnée à une entrée, elle est retirée de la file. Il ne faut pas que la même valeur soit donnée à deux entrées différentes. Ceci oblige à ordonner les requêtes de consommation au même titre qu'on a ordonné les productions.

3.54 Cas de plusieurs consultations

Une consultation ne modifie pas l'état d'une donnée.

La seule condition requise est que la file ne soit pas vide, pour satisfaire une demande de consultation. Toutes les consultations voulues peuvent avoir lieu en même temps.

Les modules étant écrits, entrées et sorties étant définies les liens entre les modules ne sont pas connus. Réciproquement les liens entre modules peuvent être définis sans que les modules soient écrits.

Cette indépendance des déclarations de connexion et des modules permet d'établir diverses connexions entre les mêmes modules.

Exemple

```
Module M1                    Module M2
  entrée E1                    entrée E2
  sortie S1                    sortie S2
    ⋮                            ⋮

E1 de M1 ← ℬ1 ← S1 de M1
avec consommation      ↖ S2 de M2

E2 de M2 ← ℬ2 ← S1 de M1
avec consommation      ↖ S2 de M2

              ou bien

E1 de M1 ← S2 de M2
sans consommation

E2 de M2 ← S1 de M1
avec consommation
```

Ces connexions n'expriment pas qu'un transfert de valeurs entre des modules. En précisant d'une part s'il s'agit d'une consommation ou d'une consultation et d'autre part quel exemplaire est concerné, nous avons introduit diverses possibilités de synchronisation.

Ainsi seule l'écriture des connexions suffit à exprimer la synchronisation des modules. On retrouve ainsi la plupart des

situations classiques de parallélisme
producteur - consommateur, exclusion mutuelle etc...
que nous allons étudier dans le paragraphe suivant.

3.6 SITUATIONS DE PARALLELISME

3.61 Modèle producteur - consommateur

3.611 producteur - consommateur uniques

Un module producteur est un module qui évalue une sortie pendant son exécution.

Un module consommateur est caractérisé par une entrée dont les valeurs successives seront les valeurs de la sortie du module producteur, qui lui est connectée.

```
Module PROD
  Sortie S
    ⋮
  S :=
Fin du Module PROD

Module CONS
  Entrée E
    ⋮
Fin du module CONS

connexion   E de CONS ← S de PROD
            avec consommation
            ancien exemplaire.
```

Quelles que soient les structures de contrôle de niveau 3 qui régissent l'exécution de CONS et PROD, CONS est bloqué tant qu'une

valeur au moins n'a pu être produite pour PROD. Le producteur a priori n'est jamais bloqué puisque la file est supposée infinie. Si son implantation devient un tampon de taille finie, un blocage du module PROD pourra intervenir car la sortie ne pourra être rangée.

3.612 Producteurs et consommateurs multiples

Supposons que n modules producteurs $PROD_i$, analogues au module PROD, produisent concurrement des messages à destination de m modules consommateurs $CONS_j$ analogues à CONS. Ils seront tous connectés à la même donnée transmissible $\mathfrak{D}$.

Les déclarations de connexion seront

$\forall$ i de 1 à n S de $PROD_i$ → $\mathfrak{D}$

$\forall$ j de 1 à m $\mathfrak{D}$ → E_j de $CONS_j$, avec consommation ancien exemplaire

Initialement $\mathfrak{D}$ n'a aucune valeur, donc si un consommateur désire s'exécuter, il sera bloqué en attendant qu'au moins un producteur soit terminé. Les valeurs des S de $PROD_i$ sont rangées dans une file associée à $\mathfrak{D}$ selon les règles de l'opération de production. Les producteurs ne sont pas bloqués a priori puisque la file est infinie. Leur blocage est susceptible d'intervenir comme dans le cas du producteur simple (cf. paragraphe précédent).

3.62 Exclusion mutuelle

Deux modules M1 et M2 s'exécutent en exclusion mutuelle pour une donnée transmissible $\mathfrak{D}$ quand ils modifient tous deux cette donnée :

(forme 1)

```
MODULE M1                    MODULE M2
ENTREE I                     ENTREE E
SORTIE I                     SORTIE S
  :  I := I + 1                :  S := f (E)
FIN DU MODULE M1             FIN DU MODULE M2
```


Il s'agit en fait de deux modules ayant une même variable d'état. (cf. § 3.22)

(forme 2)

```
Module M1                    Module M2
  Etat I                       Etat K
    :                            :
  I := I + 1                   K := f (K)
Fin du Module M1             Fin du Module M2
```

K du module M2 a été défini comme une entrée et une sortie.

Mais au cours de la programmation progressive d'une application, on peut en effet être amené à écrire d'abord le module M1 sous la forme 2. Lors de l'adjonction de M2, il faudra alors se rendre compte que K et I sont en fait un seul et même objet. Ils représentent la même donnée d'où l'écriture selon une forme analogue à la forme 1, avec des connexions

I de M1 → $\mathfrak{D}$

S de M2 → $\mathfrak{D}$

$\mathfrak{D}$ → I de M1

$\mathfrak{D}$ → E de M2

On pourrait aussi convenir qu'il est possible de connecter une variable d'état à des entrées et sorties d'autres modules, ou deux variables d'état ensemble. Ce cas correspond en fait à une variable d'état d'un automate qui a été décomposé de telle façon que la variable d'état reste présente dans les deux sous-automates. Dans les systèmes basés sur le concept de moniteur, [HOARE - 74] , cette variable d'état serait extraite des deux modules M1 et M2 pour être "encapsulée" dans un moniteur. Les opérations I := I +1 et S := E + 1 seraient remplacées par des appels de procédure du moniteur.

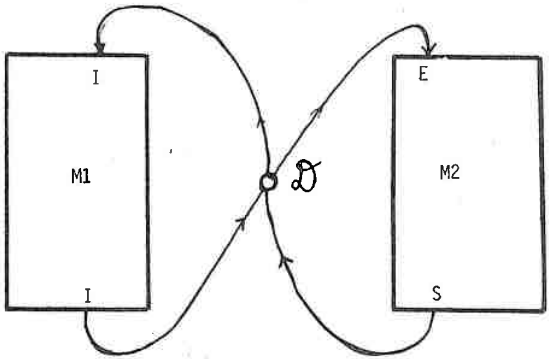
L'exclusion mutuelle sera décrite par les connexions suivantes :

I de M1 → $\mathcal{B}$

S de M2 → $\mathcal{B}$

$\mathcal{D}$ → I de M1, avec consommation
, ancien exemplaire

$\mathcal{D}$ → E de M2, avec consommation
ancien exemplaire



Exclusion mutuelle sur $\mathcal{D}$

En effet, soit un module M qui pour une donnée transmissible $\mathcal{D}$ se comporte d'abord comme un consommateur, puis comme un producteur.

Module M

Entrée E

Sortie S

⋮

fin du Module M

E et S sont connectés à $\mathcal{D}$ selon les connexions suivantes :

$\mathcal{D}$ → E de M, avec consommation
, ancien exemplaire

S de M → $\mathcal{D}$

Soit $v_j \ldots v_\ell$ l'état de $\mathcal{D}$ avant l'exécution de M. Avant l'activation de M, v_j est donné à E d'une façon analogue à E := v_j. Après l'entrée dans le module M, l'état de $\mathcal{D}$ est :

$v_{j+1} \ldots\ldots v_\ell$ (v_j a été retiré de la file)

Si aucune autre production, ni consommation n'interviennent pendant l'exécution de M, à la sortie de M la valeur de S est rangée dans la file associée à $\mathcal{D}$; l'état de $\mathcal{D}$ est donc

$v_{j+1} \ldots\ldots v_\ell \; v_{\ell+1}$

Supposons maintenant, que plusieurs modules M_j soient définis comme suit.

Module M_j

ENTREE E

SORTIE S

⋮

S := f (E)

FIN DU MODULE M_j

Supposons que les connexions soient pour tout j

$\mathcal{B}$ → E de M_j avec consommation
ancien exemplaire

S de M_j → $\mathcal{D}$

Si l'état initial de la donnée $\mathcal{D}$ est $v_1 \ldots\ldots v_\ell$, c'est-à-dire si la longueur de la file associée à $\mathcal{D}$ est ℓ, au maximum, ℓ Modules M_j pourront être actifs en même temps.

Dans ce cas la donnée $\mathcal{D}$ représente une ressource à ℓ points d'accès.

Si ℓ est égal à 1 on retrouve l'exclusion mutuelle.

Supposons maintenant que l'état initial de $\mathcal{D}$ soit réduit à une valeur v_j. Dès l'entrée dans M tout autre module qui voudrait consommer une valeur de $\mathcal{D}$ sera bloqué, puisque la file sera vide. Il ne pourra être débloqué que par la production de M.

Ce cas où l'état initial est réduit à une valeur v_j sera effectivement obtenu quand un seul module producteur INIT initialise $\mathcal{D}$.

MODULE INIT

Sortie S

⋮

Fin du module INIT

MODULE M_j

ENTREE E

SORTIE S

⋮

S := f (E)

FIN DU MODULE M_j

Connexions S de INIT → $\mathcal{D}$

$\forall$ j = 1...N S de M_j → $\mathcal{D}$

$\mathcal{D}$ → E de M_j avec consommation ancien exemplaire

$\forall$ j = 1, 2 ... N

Il est à noter que l'attribut "ancien exemplaire" n'a pas de signification ici puisque la longueur de la file des valeurs de $\mathcal{D}$ est soit nulle, soit égale à 1.

En effet, avant l'exécution de INIT, la file est vide, aucun module M_j ne peut être activé. Dès la fin de INIT, la longueur de la file est égale à 1. Un seul module M_k peut être activé. Dès qu'il l'est, la file est vide donc tous les autres sont bloqués jusqu'à ce que M_k soit terminé auquel cas la longueur de la file est redevenue 1.

D'après [CROCUS - 75] page 19, une solution au problème de l'exclusion mutuelle doit présenter quatre propriétés :

a) à tout instant un processus au plus peut se trouver en section critique (par définition de la section critique),

b) si plusieurs processus sont bloqués en attente de la ressource critique, alors qu'aucun processus ne se trouve en section critique, l'un d'eux doit pouvoir y entrer au bout d'un temps fini (en d'autres termes, il faut éviter qu'un blocage mutuel des processus puisse durer indéfiniment),

c) si un processus est bloqué hors d'une section critique, ce blocage ne doit pas empêcher l'entrée d'un autre processus en sa section critique,

d) la solution doit être la même pour tous les processus, c'est-à-dire qu'aucun processus ne doit jouer de rôle privilégié.

On a vu que a) était toujours vérifié dès qu'un seul module INIT pouvait s'exécuter une seule fois.

La propriété b) est automatiquement vérifiée dans notre système : si un module doit être activé, et que la donnée en entrée soit disponible, elle lui sera allouée si la ressource est libre, aucun module ne peut être en attente de cette seule ressource.

La propriété c) est aussi vérifiée. Si un module est en attente de la donnée en entrée, mais aussi d'une autre ressource R, et qu'il soit bloqué parce que R n'est pas disponible, alors la donnée en entrée est libre car les entrées sont toutes allouées en même temps au moment d'entrer dans le module. Donc un autre module peut prendre la donnée en entrée. Tous les modules M_j sont écrits de la même façon, il n'y en a pas de privilégié, la propriété d) est alors vérifiée.

3.63 Données utilisées uniquement en consultation

Soient un ou plusieurs modules producteurs notés $Prod_i$ (i = 1 ... N) tels que chacun ait une sortie si :

Module $Prod_i$

sortie S_i

⋮

fin du module $prod_i$

Soient des modules consultants notés $Consult_j$ (j = 1 ... M)

Module $Consult_j$

entrée E_j

⋮

fin du module $Consult_j$

Soient les déclarations de connexion suivantes :

$\forall$ i = 1, 1 S_i de $Prod_i$ $\mathfrak{D}$

$\forall$ j = 1, M $\mathfrak{D} \rightarrow E_j$ de $Consult_j$ (sans consommation) (plus récent exemplaire).

Lors de la terminaison de $prod_i$, la valeur de S_i est rangée dans la file de la donnée d au rang $\ell + 1$ si l'état antérieur de $\mathfrak{D}$ était $v_j \ldots v_\ell$.

Avant la terminaison de $prod_i$, toute activation d'un module consultant se traduit par la prise par E_j de la valeur v_ℓ.

Après la terminaison de $prod_i$, toute activation d'un module consultant se traduit par la prise par E_j de la valeur $v_{\ell+1}$.

3.64 Consommations et consultations d'une même donnée transmissible

Nous étudions ici le cas où à une même donnée

transmissible sont connectées plusieurs entrées de modules, certains avec consommation, d'autres sans consommation.

3.641 Consultations bloquées et non bloquées

Du fait que l'opération de consommation se traduit par un retrait de l'exemplaire de la donnée transmissible, il est possible qu'une consultation ne puisse jamais avoir lieu si l'exemplaire est consommé aussitôt après sa production. Il est des cas où effectivement une consommation doit bloquer les consultants éventuels.

Exemple : Dans le problème de gestion d'un carrefour (CROCUS - 75), le module décrivant la traversée d'un véhicule aura en entrée une donnée "feu vert". Le processus de changement de feu consommera cette même donnée, bloquant ainsi les nouveaux véhicules qui se présentent, processus consultant la donnée feu vert.

Il est d'autres cas où ce blocage n'a pas de raison d'être :

Exemple : Une valeur x est calculée dans un module M1. Cette valeur doit être imprimée dans un module M2 de telle façon que chaque valeur produite soit imprimée une fois et une seule. Sur une demande d'un opérateur, la valeur courante de x doit être affichée sur un écran.

Module M1

sortie x

x :=

⋮

Fin du module M1

```
Module M2
   entrée y
      ⋮
      ⋮
    impr y
fin du Module M2

Module M3
   entrée z

    impr z
Fin du Module M3
```

Les modules M1, M2, M3 sont supposés être dans des tâches fonctionnel-différentes.

Soient les connexions

x de M1 → $\mathfrak{D}$1
$\mathfrak{D}$1 → y de M2 avec consommation, ancien exemplaire ;
$\mathfrak{D}$1 → z de M3 sans consommation, récent exemplaire ;

Il est possible que, compte tenu des vitesses relatives d'exécution, le plus ancien exemplaire et le plus récent ne fassent qu'un à un moment donné. Si la consommation a lieu avant la consultation, la consultation sera bloquée, or dans notre cas, il n'y a pas lieu d'introduire de blocage à ce niveau.

Pour cette raison, on devrait distinguer des consultations bloquées ou non bloquées par une consommation.

3.642 Consommations multiples d'une même donnée

Considérons le problème suivant :
soient les mêmes modules M1, M2, M3 que dans l'exemple ci-dessus. Leurs fonctions respectives sont maintenant M1 produit toujours une valeur, M2 et M3 impriment chacun sur un périphérique différent chaque valeur produite par M1. On a envie d'écrire les déclarations de connexion suivantes pour exprimer les règles de synchronisation.

x de M1 → $\mathfrak{D}$1
$\mathfrak{D}$1 → y de M2 avec consommation ancien exemplaire
$\mathfrak{D}$1 → z de M3 avec consommation ancien exemplaire

connexions qui ne résolvent pas le problème.

En effet, une valeur imprimée dans M2 ne pourra plus l'être dans M3 et réciproquement.

Une solution à ce problème, comme au précédent, réside dans la définition de copies de données. On peut la noter par une affectation du type $\mathfrak{D}$2 := $\mathfrak{D}$1 où $\mathfrak{D}$2 représente la copie de la donnée $\mathfrak{D}$1 définie par ailleurs comme la suite des valeurs produites par des modules producteurs.

3.643 Solution au problème de consultations non bloquées

Sortie de Prod → donnée 1
donnée 1 → entrée de cons 1 avec consommation ancien exemplaire
donnée 2 := donnée 1
donnée 2 → entrée de cons 2 sans consommation récent exemplaire

3.644 Solutions au problème de consommations multiples

sortie de Prod → donnée 1
donnée 1 → entrée de cons 1
avec consommation
ancien exemplaire
donnée 2 := donnée 1
donnée 2 → entrée de cons 2
avec consommation
ancien exemplaire

Dupliquer une donnée signifie dupliquer chaque exemplaire v_i dès que l'exemplaire est rangé dans la file initiale. En particulier la donnée dupliquée n'est pas soumise à une quelconque production directe. On est ainsi assuré que l'ordre des exemplaires dans la donnée dupliquée est le même que dans la donnée initiale. On pourrait rapprocher la duplication de la définition de deux ou plusieurs données transmissibles à partir d'une ou de plusieurs sortes de modules producteurs. Nous verrons ci-dessous qu'il existe une différence.

Exemple :

sortie de Prod → $\mathcal{D}$1
sortie de Prod → $\mathcal{D}$2
sortie de Prod → $\mathcal{D}$3

$\mathcal{D}$1 → entrée de cons 1 avec consommation
ancien exemplaire

$\mathcal{D}$2 → entrée de cons 2 sans consommation
récent exemplaire

Cette formulation présente toutefois l'inconvénient d'autoriser plusieurs producteurs sur des données différentes. Il faudra s'assurer que toutes les productions sont bien les mêmes pour chaque donnée, et ceci peut être fait à la compilation des déclarations de connexion ; Mais aussi assurer à l'exécution que les données seront produites dans le même ordre pour les diverses données. C'est pourquoi,

nous préférons la formulation sous forme de duplication qui a l'avantage de spécifier une seule donnée maître et les copies de cette donnée maître.

De nombreux contrôles pourront ainsi être effectués à la compilation et même, compte tenu d'une description, la sémantique des connexions, traduisant la synchronisation, pourra être indiquée au programmeur ; cette sémantique sera par exemple précisée sous la forme d'un ensemble de couples

(modules bloqués - modules bloquants).

Ceci permet aussi au programmeur de vérifier que les déclarations de connexion écrites reflètent bien les règles de synchronisation qu'il a voulu exprimer et le cas échéant d'introduire des copies ou de modifier les connexions ou les paramètres des connexions existantes.

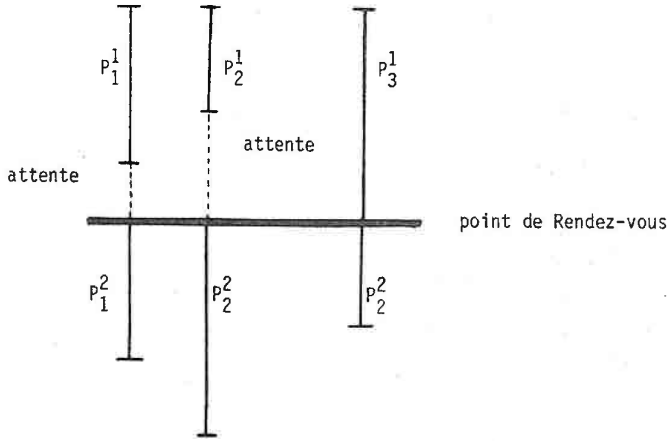
3.7 LE RENDEZ-VOUS

Nous dirons que deux ou plusieurs processus ont rendez-vous quand chacun d'entre eux étant arrivé à un certain point appelé RV, il doit attendre pour continuer que tous les autres y soient arrivés. Les processus évoluent ensuite de façon parallèle [ABRIAC - 78 a]. Ce mécanisme est utilisé par HOARE [1978] dans les C.S.P. quand il précise qu'un émetteur doit attendre que le récepteur soit arrivé à la primitive "INPUT", pour continuer (et réciproquement). Dans ADA, c'est le mécanisme de base pour exprimer la synchronisation de processus.

Nous montrons ici comment on réalise un rendez-vous entre processus à partir de nos outils.

Chaque processus (noté P_i) est décomposé en P_i^1, RN_i, P_i^2 où P_i^1 représente l'exécution avant RV_i et P_i^2 l'exécution après RV_i. Pour tout i, les séquences P_i^1 s'exécutent parallélement. Les séquences P_i^2 ne peuvent démarrer que lorsque chaque P_i est arrivé au point RV

Exemple :



Dans notre système, les points de synchronisation ne peuvent être que l'entrée et la sortie d'un module. Le rendez-vous entre processus aura donc lieu soit enfin, soit en début de module. Ils seront synchronisés par connexions entre entrée et sortie.

3.71 Rendez-vous de deux processus P1 et P2

Soit P1 composé de P_1^1 et P_1^2 ;

Soit P2 composé de P_2^1 et P_2^2.

P_1^1 et P_1^2 d'une part, P_2^1 et P_2^2 d'autre part s'enchaînent séquentiellement.

MODULE P_1^1

SORTIE O_1, O_2

⋮

MODULE P_1^2

ENTREE I_1, I_2

MODULE P_2^1

SORTIE O_1, O_2

⋮

MODULE P_2^2

ENTREE I_1, I_2

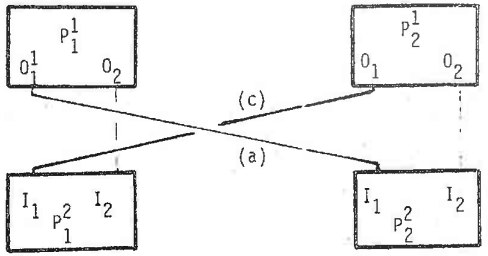
Les connexions sont les suivantes :

a) O_1 de P_1^1 → I_2 de P_2^2 , avec consommation

b) O_2 de P_1^1 → I_2 de P_1^2

c) O_1 de P_2^1 → I_1 de P_1^2 , avec consommation

d) O_2 de P_2^1 → I_2 de P_2^2



Rendez-vous de P1 et P2

. Les connexions a), b), c), d) ont été exécutées sans faire intervenir de données transmissibles car seuls interviennent deux processus P1 et P2, on aurait pu écrire par exemple :

a) O_1 de P_1^1 → (a)

(a) → I_2 de P_2^2, avec consommation
, ancien exemplaire

à chaque connexion on aurait pu associer une telle donnée.

. Les connexions b) et d) peuvent être superflues car redondantes avec les structures de contrôle de niveau 2 :

P_1^1 - P_1^2 , P_2^1 - P_2^2 qui expriment la séquentiabilité des composantes de P_1 et de P_2.

. Supposons initialement les données (a) et (c) vides. Supposons également que P_1^1 se termine le premier. Dès que P_1^1 est terminé, la donnée (a) contient une valeur de O_1 de P_1^1. Tant que P_2^1 n'est pas terminé, O_2 n'étant pas produit (O_2 de P_2^1), P_2^2 est bloqué. De même P_1^2 est bloqué car O_1 de P_2^1 n'est pas produit. La terminaison de P_1^2 se traduit donc par le déblocage de P_1^2 et de P_2^2. Les valeurs de (a) et (c) sont consommées ; l'état des données est donc vide pendant l'exécution de P_1^2 et P_2^2.

. Compte tenu des structures de contrôle de niveau 3 et de la répartition des modules, il sera possible que les données relatives aux connexions a), b), c) et d) prennent plusieurs valeurs. On pourra vérifier qu'une attente infinie ne risque pas de se produire.

3.72 Généralisation

On pourra étendre facilement le rendez-vous à n processus P_i (i = 1, ... n). Chaque P_i est décomposé en P_i^1 et P_i^2. Chaque P_i^1 possède n sorties O_1 ... O_n, et chaque P_i^2 possède n entrées I_1 ... I_n.

La sortie O_j de P_i^1 sera connectée à l'entrée I_i de P_j^2 pour tout j et tout i variant de 1 à n.

Cette solution présente toutefois un inconvénient majeur : son nombre de connexion très élevé (n^2) dû au fait qu'on impose dans chaque module précédant (resp : suivant) le rendez-vous, autant de sorties (resp : entrées) que de modules concernés. De plus, si on désire ajouter un processus supplémentaire, ceci revient à ajouter 1 sortie supplémentaire dans n modules, 1 entrée supplémentaire dans n autres et 2 n + 1 connexions supplémentaires.

Un avantage toutefois est à signaler. Cette description du rendez-vous entre N processus permet non seulement la synchronisation des N processus, mais aussi le transfert de messages (ou de données,

plus simplement) de chacun de ces N processus vers les N-1 autres.

Si on ne désire exprimer que la synchronisation sans préciser des liaisons de données particulières, on préférera la solution suivante.

3.73 Module "Rendez-vous"

Nous introduisons ici un module spécial chargé d'assurer la synchronisation des N processus P_i ayant rendez-vous. Le module recevra en entrée les messages des modules précédant le point de rendez-vous et délivrera en sortie des messages pour les modules qui suivent le point de rendez-vous.

```
MODULE P_i^1                    MODULE P_i^2
  ENTREE -                        ENTREE I_1
  SORTIE O_1                      SORTIE .
    :
    :
      MODULE RENDEZ-VOUS
            ENTREE  E1, E2, ..., En
            SORTIE  S1, S2, ..., Sn
              :
              :
```

Connexions

O_1 de P_i^1 → E_i de RENDEZ-VOUS, avec consommation ancien exemplaire

pour tout i de 1 à n.

Si de RENDEZ-VOUS → I_1 de P_i^2, avec consommation ancien exemplaire

Le module RENDEZ-VOUS ne sera activé que lorsque tous les modules P_i^1 seront terminés. Dès la fin de RENDEZ-VOUS, les sorties sont validées en même temps donc tous les modules P_i^2 sont activés en parallèle.

Le modèle du rendez-vous étant parfaitement défini, on peut envisager, une écriture plus condensée des connexions par une déclaration

RENDEZ-VOUS (P_1^1, P_2^1 ... P_n^1 ; P_1^2, P_2^2 ... P_n^2)

(O_1, O_1 .... O_1 ; I1, I1, I1 ... I_1)

en précisant dans l'ordre, les modules précédant le rendez-vous, puis les modules suivant le rendez-vous et respectivement dans le même ordre, les sorties et les entrées des modules concernés.

3.8 LES PROBLEMES LECTEURS - REDACTEURS [COURTOIS - 71]

Nous rappelons ici l'énoncé de [CROCUS - 75] :

Le modèle des lecteurs et rédacteurs schématise une situation rencontrée dans la gestion des fichiers partageables. Dans ce modèle, on considère des processus parallèles (n au plus) divisés en deux classes : les lecteurs et les rédacteurs. Ces processus peuvent se partager une ressource unique, le fichier. Ce fichier peut être lu simultanément par plusieurs lecteurs, tandis que les rédacteurs doivent y avoir un accès exclusif (un seul rédacteur peut y écrire et aucun lecteur ne peut lire pendant ce temps).

Cas 1 - Priorité des lecteurs sur les rédacteurs, sans réquisition. Les lecteurs ont toujours priorité sur les rédacteurs sans réquisition ; le seul cas où un lecteur doive attendre est celui où un rédacteur occupe le fichier. Un rédacteur ne peut donc accéder au fichier que si aucun lecteur n'est en attente ou en cours de lecture. On autorise toute coalition de lecteurs pour occuper indéfiniment le fichier et en interdire l'accès aux rédacteurs.

Cas 2 - Priorité des lecteurs sur les rédacteurs si et seulement si un lecteur occupe déjà le fichier. Quand aucun lecteur ne lit, les lecteurs et les rédacteurs ont même priorité. Par contre dès qu'un lecteur lit tous les autres lecteurs peuvent lire, quel que soit le nombre de rédacteurs en attente. Les lecteurs ont toujours le droit de se coaliser pour monopoliser le fichier.

Cas 3 - Priorité égale pour les lecteurs et rédacteurs. Aucune catégorie n'a priorité sur l'autre. Si un lecteur utilise un fichier, tous les lecteurs nouveaux qui arrivent y accèdent jusqu'à l'arrivée d'un rédacteur. A partir de ce moment, tous les nouveaux arrivants attendent, sans distinction de catégorie. Si un rédacteur utilise le fichier, tous les nouveaux arrivants attendent également. Quand le rédacteur a fini, il réveille le premier processus en attente dans l'ordre (ici inconnu) des files d'attente. Si plusieurs lecteurs se suivent dans la file, ils accèdent ensemble au fichier. L'attente infinie est impossible dans ces conditions.

Cas 4 - Priorité des rédacteurs sur les lecteurs. On donne cette fois la priorité aux rédacteurs : dès qu'un rédacteur réclame l'accès au fichier, il doit l'obtenir le plus tôt possible sans réquisition, c'est-à-dire à la fin de l'exécution des processus occupant le fichier au moment de la demande. Donc tout lecteur arrivé après que le fichier ait été demandé par un rédacteur doit attendre, même si des lecteurs utilisent encore le fichier. On notera que les rédacteurs peuvent cette fois se coaliser pour interdire indéfiniment aux lecteurs l'accès au fichier.

Un lecteur est un module qui a une donnée f en entrée uniquement

```
MODULE  Lecteur
ENTREE  F
  :
```

Un rédacteur est un processus qui a donné f en sortie, s'il ne fait qu'une écriture

Module rédacteur

Sortie F

⋮

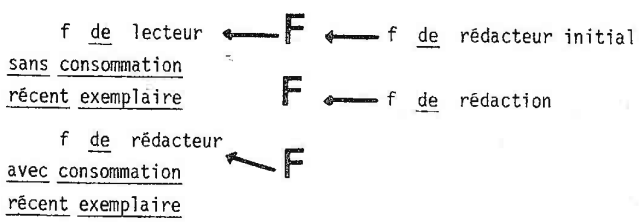
Si un rédacteur fait une modification du fichier à partir de l'état antérieur f sera déclaré en entrée et en sortie

entrée f

sortie f

Il y a toujours un rédacteur initial qui crée une première version f_0.

Les déclarations de connexion sont donc



- S'il n'y a qu'un seul rédacteur initial, le fichier F est en un seul exemplaire

- S'il y a plusieurs rédacteurs initiaux, le fichier F comportera plusieurs valeurs (versions) f_1, f_2 ... f_n dans l'ordre où elles auront été produites. Selon l'implantation, nous aurons effectivement présents en même temps un ou plusieurs exemplaires.

Ici nous ne nous intéressons qu'au cas où il n'y a qu'un exemplaire de F. (problème posé initialement).

Vérifions que les règles fixées dans l'énoncé sont bien respectées.

- Exclusion mutuelle entre rédacteurs.

Comme un rédacteur consomme le seul exemplaire existant, tous les autres sont bloqués tant que celui qui est actif n'est pas terminé. A ce moment, l'exemplaire suivant du fichier est disponible. Un autre rédacteur peut être activé.

- Exclusion mutuelle entre lecteurs et rédacteurs.

Cette exclusion est assurée par le fait que l'opération de consultation garantit que l'entrée garde sa valeur pendant l'exécution du module. l'exclusion est donc assurée par la spécification même des connexions. On notera toutefois que les comportements des processus peuvent être différents quand on tient compte de l'implantation. Si l'entrée est dupliquée dans chaque lecteur, rien n'empêche un parallélisme parfait entre lecteur et rédacteur. Par contre si l'exemplaire de F est unique, le rédacteur attendra la fin de la lecture pour commencer la rédaction. Nous ferons l'hypothèse que le fichier est unique et qu'on ne le recopie pas pour une lecture.

Ainsi, si un lecteur occupe le fichier, tous les autres lecteurs ont la possibilité de lire. Mais tant qu'un lecteur au moins sera actif, aucun rédacteur ne pourra être activé. Les problèmes liés aux autorisations de lecture , si des rédacteurs sont en attente ou liés à des priorités relatives ne sont pas résolus ici.

Les connexions que nous avons présentées nous permettent de gérer naturellement la synchronisation de processus à partir de la définition de leurs inter-communications.

Nous n'avons exprimé que les règles d'accès à des données sans nous préoccuper des conditions d'enchaînement relatif des lecteurs avec les rédacteurs (4 problèmes).

Notre solution est commune aux quatre problèmes. Les régles de priorité seront précisées au chapitre suivant grâce aux structures de contrôle de niveau 3.

3.9 CONCLUSION

Nos données se rapprochent des canaux virtuels de [KAHN et A1 - 76] en particulier par le fait qu'un canal peut contenir une file de valeurs comme dans notre cas l'état d'une donnée peut être une suite d'exemplaires. Toutefois, avec les opérations de consommation, de consultation, de duplication en précisant l'exemplaire concerné, nous disposons d'opérations plus précises que GET et PUT. Les opérations qu'il est possible d'effectuer sur cette file sont précisées au moment de la connexion, les opérations de consommation et de consultation n'étant pas indiquées dans le texte des modules, mais à l'extérieur.

On peut voir que les primitives INput et OUTput de [HOARE - 78] avec les attentes correspondant en fait à notre description du rendez-vous avec la restriction qu'il n'y a aucune anticipation. De plus, le rendez-vous introduit des possibilités de blocage au niveau interne des actions.

Dans un souci d'assurer une meilleure sécurité, nous pensons qu"il est préférable de connaître exactement les points où la synchronisation peut avoir lieu, car en chacun de ces points, il sera possible d'exercer un contrôle qui est impossible quand les points de synchronisation peuvent être internes aux modules.

Les données transmissibles permettent de décrire quelques cas typiques de synchronisation de façon relativement naturelle. On leur a donné un nom pour pouvoir les désigner au fur et à mesure de l'évolution de l'application. Le nom pourrait être supprimé si on précisait en une seule fois les entrées et les sorties intervenant dans la connexion.

On pourra faire remarquer qu'il s'agit de données globales nous pensons que ce n'en sont pas au sens habituel de global. En effet, ces données ne sont accessibles qu'à un nombre précis et connu de modules explicitement nommés. De plus, il est indiqué comment les modules peuvent et doivent y accèder.

D'autre part, comme il ne s'agit que d'un concept indépendant de toutes les implantations, nous verrons que dans certains cas,

la donnée n'existe pas physiquement, il s'agit uniquement d'une transmission entre deux variables locales (une sortie et une entrée).

Nous retrouvons ici la liaison INput - OUTput de [HOARE - 78]. On ne peut alors parler de variable globale.

Rappel de la SYNTAXE

< sortie > de < nom de module >
⟶ < nom de donnée transmissible >

< non de donnée transmissible >
⟶ < entrée > de < nom de module >
, { avec / sans } consommation
, { ancien / récent } exemplaire

REFERENCES DU CHAPITRE :

[ABRIAL - 78 a]

[ANCEAU - 78]

[COURTOIS - 71]

[CROCUS - 75]

[DENNIS - 75]

[HOARE - 74], [HOARE - 78]

[KAHN et Al - 76]

[LORRAINS - 79]

[ROBERT et Al - 77 a]

[SCHAFF - 74]

[SYRE et Al - 78]

[THOMESSE - 74], [THOMESSE et Al 79]

CHAPITRE 4

LES STRUCTURES DE CONTROLE DE NIVEAU 3 ET LE PARALLELISME

4.1 INTRODUCTION

Dans ce chapitre nous allons présenter le second volet de programmation des synchronisations, à savoir, les événements, les conditions et les structures qui les relient aux tâches. L'ordre de cette présentation montrera comment nous sommes arrivés à formuler nos propositions.

D'abord (§ 4.2), nous rappellerons les structures de contrôle et les événements définis dans [NONN - 78 b]. Dans ces propositions, et pour des raisons d'implantation, nous verrons qu'il était largement tenu compte du site à savoir un SOLAR 16 de la SEMS. Les structures de contrôle sont en effet très voisines des requêtes du systèmes RTES [SEMS].

Nous préciserons ensuite ce que nous appelons un événement et proposerons un langage pour leur définition. Ceci nous permettra de ramener toutes les structures de contrôle à une seule forme. Pour tenir compte de l'état du système au moment de l'occurence des évéments, nous introduirons une conditionnelle dans la structure qui aura alors la forme suivante :

SUR événement SI condition FAIRE tâche

Ceci est traité au § 4.3.

Aux §§ 4.4 et 4.5, nous étudierons la dualité événement-condition et la dualité événement-donnée transmissible respectivement, pour récapituler les définitions possibles de condition au paragraphe 4.6.

Il est important de noter que ces propositions n'ont pas encore été implantées. Ceci est en cours [CAUMONT] sur un ordinateur SOLAR et sur microprocesseur. Une validation de ces propositions peut toutefois être acceptée par l'aptitude à la description de solution à certains problèmes.

Nous résoudrons le problème de gestion du carrefour proposé dans [CROCUS - 75] (§ 4.7) ; à ce sujet, nous indiquerons comment on peut intégrer une condition comme une entrée de module, avec les implications occasionnées sur d'autres éléments.

Le dernier problème résolu sera celui des lecteurs-rédacteurs déjà abordé au chapitre précédent (§ 4.8).

Au paragraphe 4.9 nous comparerons nos propositions à d'autres systèmes et outils de spécification, de synchronisation en rapprochant les systèmes "pilotés par les données" des systèmes "pilotés par le contrôle".

4.2 LES STRUCTURES DE CONTROLE DE NIVEAU 3

Les structures de contrôle de niveau 3 représentent les liens entre les événements extérieurs, le ou les opérateurs, le procédé physique et les tâches fonctionnelles de l'application. Chacune de ces structures représente ici les conditions d'activation d'une tâche.

Exemples : SUR < EVENEMENT > FAIRE < TACHE > ;

forme analogue au ON < EVENT > DO du langage PL1, à la différence qu'en PL1 les événements sont prédéfinis

SUR Dépassement de températures FAIRE T1 ;

TOUTES LES 10 SECONDES FAIRE T2 ;

APRES EVENT1 ATTENDRE 10 SECONDES FAIRE T3.

Toutes les tâches ainsi mises en oeuvre sont considérées comme parallèles. Si chacune de ces tâches était implantée sur un processeur différent, et si toutes les conditions étaient vraies en même temps, elles pourraient être activées simultanément.

Si pour diverses raisons deux taches T1 et T2 ne doivent pas se dérouler en même temps, on pourra l'exprimer à ce niveau : une des conditions d'activation de T1 étant la non-activité de T2 et réciproquement. On voit aussi apparaître ici, des informations concernant l'enchaînement relatif des tâches pour lequel on trouve habituellement des priorités dans la plupart des systèmes temps réels classiques par exemple (RTES, [SEMS]).

Compte tenu de l'implantation sur SOLAR, la notion de priorité est maintenue dans notre système sans toutefois obliger l'utilisateur à la mentionner [NONN - 78 b], ce qui impose à résoudre les choix d'ordonnancement à partir d'un ou de plusieurs autres critères.

Une structure de niveau 3 a donc pour rôle d'exprimer statiquement quelle action doit être exécutée quand un événement se produit. Nous avons déjà proposé quelques structures permettant d'exprimer des activations périodiques ou aléatoires [NONN - 78 b].

{ A < HEURE > | SUR < EVT > | IMMEDIATEMENT }

{ FAIRE | REPRENDRE } < TACHE FONCTIONNELLE > [PRIORITE]<p>

{ TUER | SUSPENDRE } < TACHE FONCTIONNELLE >

Dans le cas de FAIRE on pourra donner des ordres de répétition par

REPETER { AUSSITOT | TOUS LES < PERIODE > | SUR < EVT > }

{ INDEFINIMENT | PENDANT < DUREE > | JUSQUA { < EVT > | < HEURE > } }

< N > FOIS

Ces structures permettent d'exprimer n'importe quelles conditions d'arrêt ou de suspensions de tâche et ceci de façon complètement indépendante des tâches et de la façon même dont les événements seront pris en compte. Une telle expression statique est une expression asynchrone dans le sens où on énumère les événements et leur tâche associée, un peu comme on s'exprimait dans des systèmes plus anciens.

Sur Interruption, n° de niveau, n° de bit, faire sous-programme, par exemple dans le système TSX de l'IBM 1800, au moment de l'édition de liens d'un module de chargement, on précisait par une commande INCLD, l'inclusion d'un sous-programme et la condition de son exécution. [IBM]

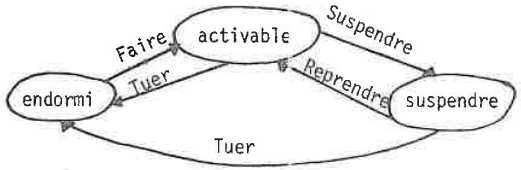
Une telle association était définie pour toute la durée de vie du module de chargement. Les modifications étaient lourdes ; au minimum une nouvelle édition de liens que l'on charge le programme ou la condition d'activation. Les systèmes actuels sont moins contraignants permettant l'expression de certaines conditions d'activation à l'extérieur des tâches :
par exemple la commande TRNON du système RTES du SOLAR 16 [SEMS].

Toutefois le concept d'événement n'y apparait pas. La notion d'interruption est toujours là et liée à la notion de tâche "hardware".

Nous n'avons pas jusqu'ici précisé la notion d'événement (voir § 4.3). Nous verrons que les structures pourront être simplifiées en précisant ce qu'on entend par événement.

Les quatre verbes FAIRE, REPRENDRE, TUER, SUSPENDRE permettent d'exprimer le passage des tâches dans divers états

FAIRE : met la tâche dans l'état activable
SUSPENDRE : met la tâche dans l'état suspendu
TUER : met la tâche dans l'état inactivable ou endormi
REPRENDRE : met la tâche dans l'état activable



Dans la suite nous ne nous intéresserons qu'aux structures avec FAIRE ; les autres commandes permettent de modifier l'ordonnancement des tâches.

4.3 PRECISIONS SUR LA NOTION D'EVENEMENT

4.31 Evénements et occurrences

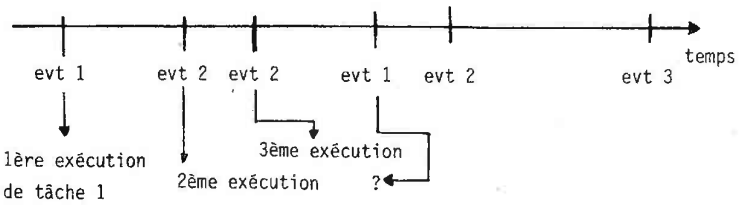
. 4.311 Ambiguïtés entre événement et occurrences

Prenons un exemple de structure de contrôle de niveau 3 :

sur evt1 faire tâche1, répéter sur evt2 jusqu'à evt3

La tâche concernée est tâche 1. Elle devra être exécutée une première fois lors de la première occurrence de l'événement evt1 successives. Les exécutions suivantes seront déclenchées par les arrivées de l'événement evt2. Jusqu'à ce que l'événement evt3 soit arrivé, à chaque occurrence de evt2, la tâche "tâche 1" sera exécutée. Quand l'événement evt3 sera arrivé, on arrêtera de prendre en compte les événements evt2 pour activer tâche 1.

Que se passe t-il si une deuxième occurence de evt1 arrive avant la première de evt3 ? Il faut pouvoir le spécifier.



Ladet ([1977] et [DESCHIZEAUX et AI - 77] a proposé un langage exprimant les divers liens entre événement et processus.

Exemple :

Pour chaque < événement > futur faire < traitement >
Pour < événement > passé faire < traitement >

L'introduction de "chaque" permet de lever l'ambiguïté relative à l'événement ou à l'occurrence de l'événement.

Nous pouvons donc constater que la sémantique d'une structure de contrôle peut varier d'une interprétation à une autre. Ceci est particulièrement gênant et nous avons besoin de préciser la notion d'événement, d'occurrence, d'instant d'arrivée, de reconnaissance et de traitement afin de lever complètement ces ambiguïtés.

Nous considérons ici qu'une occurrence d'événement est un changement d'état d'une grandeur à un instant donné.

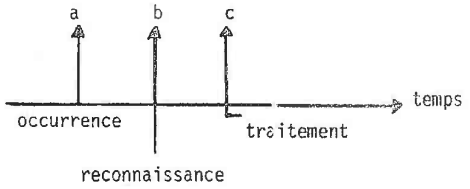
4.312 Instants caractéristiques

Cette grandeur peut être une entrée numérique, analogique comparée à un seuil, mais aussi une donnée transmissible.

Un événement est alors l'ensemble des occurrences d'un même changement. En liant habituellement l'événement et la tâche, on ne précise pas de quelle occurrence il s'agit.

L'ensemble des occurrences d'un même événement est ordonné dans le temps.

Pour chaque occurrence on peut distinguer l'instant d'occurrence, l'instant de reconnaissance et l'instant de traitement



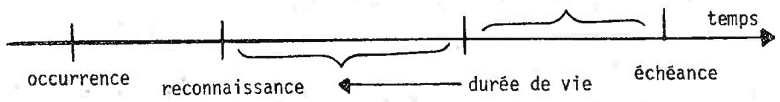
L'instant de reconnaissance doit être assez proche de l'instant d'occurrence afin de ne pas laisser échapper des changements d'état. Le problème de l'implantation de l'occurrence d'un événement se pose à ce sujet. S'il s'agit d'interruption classiquement une deuxième interruption ne sera pas prise en compte si la première n'a pas encore été reconnue. En fait, dans notre système nous ne nous préoccupons que d'occurrences reconnues.

L'instant de traitement est en fait l'instant où la tâche associée à l'événement est activée. En général, le traitement ne peut pas être trop retardé après l'occurrence. Le traitement peut ne pas être lancé dès l'événement par manque de ressources par exemple. Doit-on retarder le traitement jusqu'à ce qu'il puisse être exécuté ou purement et simplement l'annuler après un certain retard. Doit-on alors distinguer divers types d'événements, les événements "fugitifs" ou non [MENDELBAUM - 76] ou les événements "latched" et "unlatched" de langage HAL/S [FLYSTRA - 75].

Nous préférons considérer un seul type d'événement et associer à un événement une durée de vie qui indique le délai maximum qui sépare la reconnaissance de l'occurrence, de traitement. Il s'agit bien du temps qui sépare l'instant de reconnaissance du traitement. Dans le cas où l'occurrence se manifeste par une interruption, le temps qui sépare a et b sera borné supérieurement par la durée de l'exécution de l'instruction la plus longue. Si l'interruption est prioritaire, l'intervalle bc est nul, par contre si elle ne l'est pas cet intervalle bc peut être important.

Plus généralement, si T est la période de scrutation des entrées l'intervalle ab sera maximisé par T, l'intervalle bc n'est pas maximisé en général. Il dépend éventuellement de la charge du processeur devant exécuter le traitement. Il dépend plus généralement de l'état du système au moment de la reconnaissance de l'événement.

On peut rapprocher la durée de vie d'un événement de la notion d'échéance ou "deadline" qui traduit la date limite à laquelle une tâche doit être terminée. En l'absence de précision de durée de vie on supposera infinie. Ceci signifie qu'il n'y a pas d'échéance particulière pour la tâche associée. Seulement, on supposera que le scheduling est tel que la tâche sera activée au bout d'un temps fini, la propriété de délai fini [KARP et Al - 66] doit être ici vérifiée. Le non respect de cette propriété pourra elle-même être considérée comme un événement, ce qui nous permet naturellement de prendre en compte de la même manière, les traitements dits d'exception comme des tâches normales.



L'échéance apparaît ici comme la date telle que :

date de reconnaissance + durée de vie + temps maximum d'exécution ≤ date d'échéance.

4.32 Conditions relatives à l'état du système

4.321 L'état du système

L'état du système est constitué de l'état du procédé industriel et de celui de son dispositif de commande : le système informatique.

L'état du procédé industriel est connu par l'historique de l'ensemble des grandeurs mesurées et des actions qui lui sont commandées. Par exemple, on saura qu'un moteur tourne

soit parce qu'une entrée numérique l'indique

soit parce qu'un capteur de vitesse indique une vitesse non nulle

soit parce qu'on lui a envoyé une commande pour la faire tourner et qu'il n'a pas été arrêté depuis.

L'état du système informatique est connu par les données et par l'état des modules et des tâches

Par exemple, une tâche est active ou non, un module est en attente d'une entrée pour pouvoir être exécuté.

4. 322 La prise en compte de l'état

L'instant d'occurrence d'un événement est une caractéristique de cet événement. Quand on exprime le lien entre événement et tâche, on ne tient pas compte de l'instant. Or, à cet instant, le système est dans un certain état et un même événement ne sera pas forcément cause des mêmes traitements selon l'instant d'occurrence. L'association événement - traitement doit donc s'exprimer de façon conditionnelle.

La reconnaissance de l'état aurait pu être exprimé à l'intérieur du traitement auquel cas nous aurions été obligés de programmer des synchronisations à l'intérieur des algorithmes, ce qui est contraire à nos options initiales. Il faut donc pouvoir exprimer les associations conditionnelles dans les structures de niveau 3.

Exemple :

Dans le problème de gestion du carrefour [CROCUS - 75], les événements "arrivée d'une voiture" donneront lieu à une traversée, on à une attente selon l'état du système :

feu vert ou rouge, et place disponible dans le carrefour ou non.

On a donc envie d'exprimer simplement ces conditions dans une conditionnelle.

sur arrivée de voiture si feu vert et place libre faire traversée.

4. 323 Forme générale d'une structure de contrôle de niveau 3

La forme générale d'une structure de contrôle de niveau 3 sera donc

sur < événement > si < condition > faire < tâche >

Nous n'avons pas défini de condition en tant que telles sinon les conditions "tâche active"
"tâche inactive"
Les autres conditions sont définies relativement aux événements et aux données transmissibles (cf. § 4.6).

4.33 Définition des événements

4.331 Sortes d'événement

On distingue souvent l'événement extérieur au système informatique de l'événement intérieur.

Par exemple, on distingue une interruption d'un événement positionné dans un programme par une primitive "Setevent". Nous voulons ici pouvoir désigner aussi bien l'un que l'autre. Un événement peut être lié au temps, et dans ce cas être relatif à une heure

absolue ou à un delai ou une période. On remarquera d'ailleurs à ce sujet que ces événements peuvent aussi bien être intérieur qu' extérieur. Ceci ne dépend que de l'emplacement géographique de l'horloge. Il s'agit d'une horloge interne ou d'une horloge externe.

Un événement peut être lié à une entrée du système informatique, qu'elle soit analogique, numérique ou même constituée par des ordres de l'opérateur.

Un événement peut être défini à partir d'autres événements en utilisant des fonctions booléennes des relations d'ordre temporel comme "avant" ou "après", des comptages etc... On distinguera donc des événements élémentaires des autres obtenus par composition des premiers.

NONN [1978 a] avait déjà proposé quelques définitions formelles d'événement. Nous précisons dans le paragraphe suivant une syntaxe et un mode de définition des événements.

4.332 Syntaxe de la définition d'événement

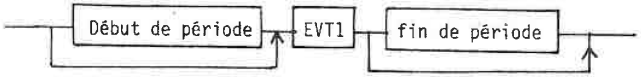
Un nom d'événement est un identificateur de portée globale au niveau de l'application. Il est attaché à un changement d'état et un seul.

On définira un événement en lui associant éventuellement une durée de vie.

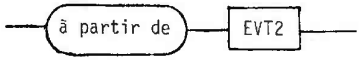
Déclaration d' événement : < nom d'événement > , < durée de vie >

La durée de vie est destinée uniquement à l'ordonnancement des tâches au moment de l'exécution.

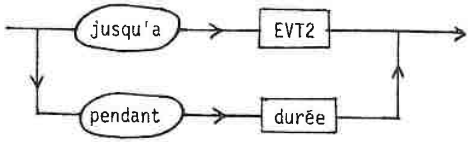
Evénement

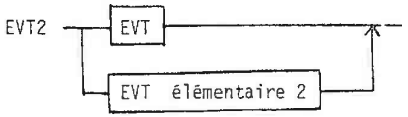


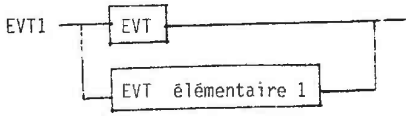
Début de période

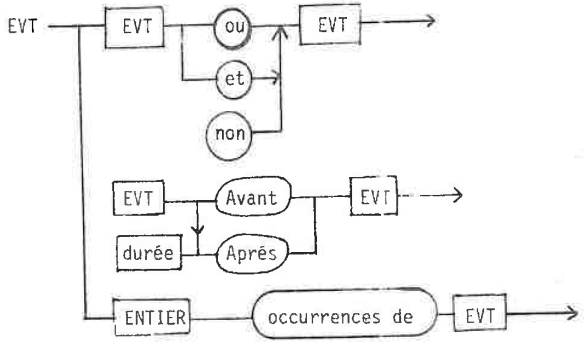


Fin de période

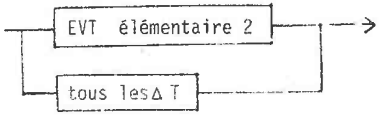






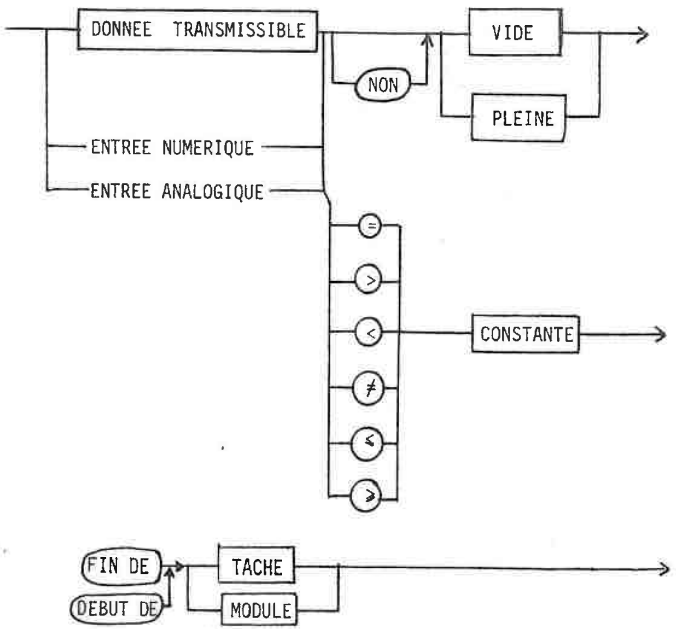


EVT élémentaire 1



EVT élémentaire 2

- HEURE
- TRANSITION VF Entrée Numérique n°
- TRANSITION FV Entrée Numérique n°
- INTERRUPTION N° NIVEAU BIT



Message opérateur : texte

4.333 Autres définitions possibles

Il aurait été possible de définir d'autres éléments de langage permettant la définition d'autres combinaisons d'événements :

Exemple 1 :

Définition de l'événement arrivant entre deux autres événements

EVT : événement 1 entre événement2 et événement3
on l'exprimera par

EVT : événement 2 avant événement 1
et
événement 1 avant événement 3

Exemple 2 :

Définition de l'événement :

EVT : la deuxième fois que événement 1 se produit avant evt 2

Un tel événement peut être défini par

EVT : 2 occurrences de (evt 1 avant evt 2)

ou

EVT : (2 occurrences de evt 1) avant evt 2.

Il y a ambiguïté dans l'expression informelle de l'énoncé. Il serait nécessaire de préciser.

Nous avons ici utilisé des parenthèses pour montrer effectivement l'ambiguïté des expressions.

En obligeant l'utilisateur à définir des événements "intermédiaires" on peut s'en passer :

exemple : evt int 1 : 2 occurrences de evt 1
EVT : evt int 1 avant evt 2

On peut ainsi envisager des définitions par l'utilisateur de nouvelles règles d'évaluation d'événements.

Dans cette optique, il faudrait être vraiment sûr qu'un minimum d'opérateurs permet toute définition. La notion d'événement et de fonction d'événement permet ici de formaliser (en levant les ambiguïtés) des relations introduites par [MIDDLETON - (1977)]. Elle permet de plus une modification d'enchainement des actions sans aucune intervention dans les textes des traitements.

Nous pensons toutefois, que notre syntaxe nous permet à peu près n'importe quelles relations. En ce sens on pourrait peut être démontrer que certaines déclarations représentent le minimum à partir duquel on peut définir n'importe quel événement comme il a été montré que certaines structures de contrôle étaient nécessaires et suffisantes à l'expression de tout algorithme séquentiel [LEDGARD - 75].

- Les relations ne mettant en jeu que des opérateurs booléens pourront toutes être exprimées avec ET, OU, NON, puisque toute fonction booléenne peut s'exprimer avec ces opérateurs.

- La relation d'ordre crée par AVANT et APRES permet d'ordonner les éléments dès qu'on saura l'implanter. Si un seul observateur peut "voir" en même temps les événements concernés, on pourra alors implanter la relation d'ordre ; par exemple : si leur site d'occurrence est le même. De toutes façons, la différence de temps entre les instants d'occurrence est minorée par un ε en dessous duquel on est obligé de considérer les événements comme simultanés. Si les sites d'occurrence sont différents, il en est bien évidemment de même. Si un ordonnancement est alors fait, c'est de manière arbitraire (une entrée est testée avant l'autre par exemple).

4.4 DUALITE EVENEMENT - CONDITION

4.41 Conditions d'exécution d'une tâche

Nous déclarons donc d'une part les événements,
d'autre part le lien entre événement et tâche en incluant les conditions relatives à l'état du système.

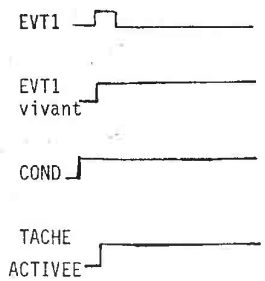
sur EVT1 si COND FAIRE TACHE (expression 1)

Cette structure exprime que TACHE sera exécutée

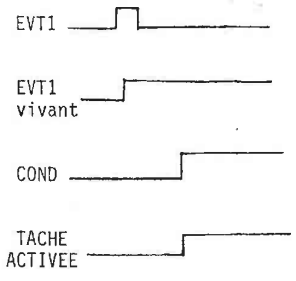
- dès que EVT aura été détecté si COND est vrai à ce moment (1er cas).

Sinon dès que COND vrai aura été détectée si EVT est encore vivant (2ème cas).

Ce qui peut s'exprimer simplement par un diagramme de temps

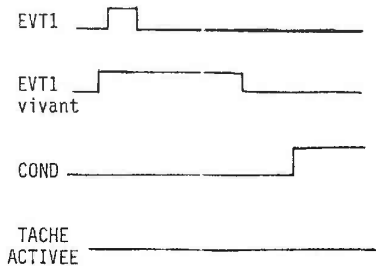


1er cas



2ème cas

Dans le cas où la condition devient vraie alors que l'événement n'est plus vivant, la tâche ne sera pas activée (3ème cas)



3ème cas

On aurait pu exprimer la même chose en considérant comme un événement le passage de faux à vrai de la condition ; appelons "transition vraie" cet événement ; EVT1 vivant sera alors considéré comme une condition.

La structure de contrôle pourra alors être

sur TRANSITION VRAIE si EVT1 vivant faire TACHE (2)

les deux expressions (1) et (2) indiquent que TACHE sera exécutée

quand il y aura coïncidence entre COND et EVT1 vivant indépendamment de la façon dont cette coïncidence sera détectée. On verra en particulier dans la partie implantation que les événements sont toujours détectés par scrutation cyclique (matérielle quand ce sont des interruptions , autrement logicielle).

On verra dans la partie implantation que ces deux formes peuvent être implantées indifféremment selon que les événements sont des interruptions ou non. On se ramènerait ainsi aux deux cas extrêmes :

sur EVT1 et TRANSITION VRAIE FAIRE TACHE (3)

Tous les n unités de temps (4)
si EVT1 vivant et COND et
(ancien (EVT1 vivant et COND) faux) faire Tâche

Cette forme (4) exprime que les événements sont détectés par scrutation périodique programmée de période n. La condition d'activation de la tâche à la ième période de scrutation est que ce soit la première occurrence de EVT1 vivant et COND. L'activation de la tâche rendant l'événement non vivant, on est ainsi assuré qu'il n'y aura qu'une seule exécution de la tâche pour l'occurrence en gestion.

4.42 Conditions et événements partageables

Quand une condition ou un événement n'est lié qu'à une seule tâche l'activation de cette tâche peut rendre non vivant l'événement ou la transition vraie. En fait, dès que plus d'une tâche est concernée par l'événement ou la condition, ce n'est plus tout à fait vrai.

En effet, prenons par exemple deux tâches T1 et T2 telles que les structures régissant leur activation soient :

sur EVT1 si COND faire T1

sur EVT2 si COND faire T2

Par la règle de dualité événement condition, ces formes peuvent se traduire par

sur TRANSITION VRAIE si EVT1 vivant faire T1

sur TRANSITION VRAIE si EVT2 vivant faire T2

Si l'occurrence de EVT1 précède celle de EVT2, il ne faut pas que l'activation de T1 empêche celle de T2.

La condition COND est partagée par les deux tâches. L'événement TRANSITION VRAIE l'est donc également. Nous sommes donc amenés à définir l'événement et la condition d'une façon analogue à celle des données transmissibles. Comme les données sont consultables ou consommables nous avons envie de définir les événements, les conditions et les données de façon équivalente en distinguant d'une façon analogue à celle de HOLT, les ressources réutilisables des autres [HOLT - 71].

4.5 DUALITE EVENEMENT - DONNEE TRANSMISSIBLE

4.51 Un événement est une donnée

Associons une donnée transmissible à chaque événement déclaré.

Nous considérons toute occurrence d'un événement comme une production de donnée. Ainsi, toute structure de contrôle de niveau 3 exprime la consommation ou la consultation par une tâche de la ou des donnée (s) associées (s) à l'événement.

La ième occurrence est en fait la ième valeur de la donnée. Une tâche pourra donc se synchroniser sur un événement par consommation ou consultation du plus ancien exemplaire ou du plus récent donc de la plus ancienne occurrence, ou de la dernière survenue. On exprime ainsi les deux possibilités habituelles [LADET - 76].

Pour chaque événement
et Pour événement.

De plus, on dispose de toutes les autres occurrences entre la première et la dernière. On a également par la file associée à l'événement la possibilité d'exprimer la synchronisation d'une tâche dont la durée d'exécution est supérieure à la période ou pseudo-période d'arrivée des événements.

Cette dualité fait que les structures de contrôle de niveau 3 doivent refléter le type de synchronisation d'une façon analogue aux déclarations de connexions vues au chapitre précédent.

On connectera une ou plusieurs tâches à un ou plusieurs événements de la même façon que nous avons connecté des modules par l'intermédiaire de données transmissibles. La production des événements sera alors une opération liée à la définition même de l'événement. De même, comme suite à la dualité Evénement - condition à toute condition déclarée pourront être associée les événements "transition vraie", "transition fausse".

Une condition comme les événements sera déclarée en faisant intervenir divers éléments. Compte tenu de la dualité, une déclaration de condition implique la déclaration des événements associés. De même une déclaration d'événement se traduira par la connaissance de la condition "événement vivant".

Les éléments intervenant dans l'une ou l'autre de ces déclarations seront :

- l'heure, des délais ou retards
- des valeurs lues sur des périphériques
- des valeurs de données transmissibles.

Nous ne ferons pas de distinction entre ces éléments. Dans tous les cas, on peut se ramener à une donnée transmissible $\mathcal{D}$ dont l'état (état ($\mathcal{D}$)) est défini au fur et à mesure des productions et consommations ($\mathcal{D}$ vide, longueur de la file = x etc...).

Le blocage d'un module consommateur peut être ainsi assuré par entrée E si aucune valeur n'est disponible pour E, ou par une condition de la forme

si longueur de la file $\mathcal{D} \neq 0$ alors M.

Il serait intéressant de définir des conditions sur l'état d'une donnée transmissible.

Nous n'avons pas défini formellement ces conditions ou fonction de l'état $\mathcal{D}$. Toutefois il nous paraît que associé à une

donnée $\mathcal{D}$, on pourrait prédéfinir quelques conditions clés comme

$\mathcal{D}$ vide

$\mathcal{D}$ plein

longueur ($\mathcal{D}$) = x etc...

4.52 Consultation et Consommation

Dans notre système une condition vraie revient à produire une valeur qui transmise à une entrée valide l'exécution du module.

sur evt si cond faire Tâche

est équivalent à

Tâche

entrée evt, COND — evt de Production

↓

evt de tâche

Production

sortie evt, COND — COND de production

→ COND de tâche

Le module production étant un module associé à la reconnaissance de evt, COND, il peut être le système d'interruption comme une tâche périodique.

Comme les données en entrée sont consommables ou non, comme un événement est consommable ou nor, il en est de même pour les conditions.

Une condition est dite consommable par une tâche T quand elle devient fausse dès que T a été activée par le fait même qu'elle était vraie.

Si la condition reste vraie, alors que la tâche a été activée, la condition sera dite consultable par la tâche.

Exemple : Dans le système RTES du Solar 16, cette distinction correspond en fait à la remise ou non à zéro d'un événement qui par définition est partageable entre plusieurs tâches :

T1

```
call sevent (evt1)
a : commentaire positionnement
    d'un événement evt1
```

T2

```
call wevent (evt1)
co attente événement evt1
call revent (evt1)
mise à zéro de l'événement evt1
```

dans T2

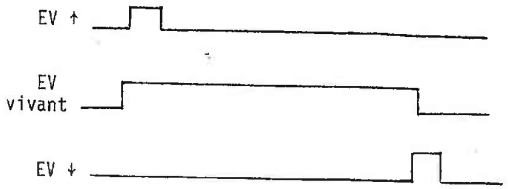
evt1 est un événement consommable

T3

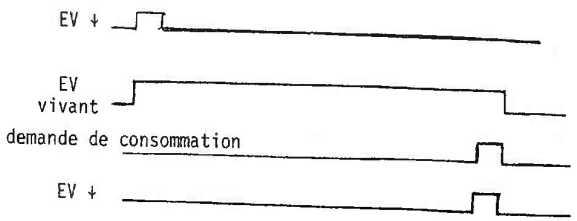
```
call wevent (evt1)
co attente evt1
```

dans T3

evt1 est un événement non consommable il est laissé à son état arrivé ou vivant



Si EV est une entrée numérique par exemple, les signaux EV↑ et EV↓ sont considérés comme envoyés par l'extérieur même s'ils ne le sont pas physiquement. Une telle condition n'est que consultable par une tâche de l'application. C'est en fait une "tâche" du procédé qui en fait la consommation quand EV↓ arrive.



EVENEMENT CONSOMMABLE

4.53 Production des événements

4.531 Evénement élémentaire

Définir un événement élémentaire revient à définir un module qui détecte cet événement et qui produit en sortie une valeur communiquée à une donnée transmissible. Ce module peut être le système d'interruption d'une machine ou une tâche logicielle activée périodiquement qui saute les changements d'état. (cf. § 5.3).

Dans le cas où l'événement est associé à une donnée transmissible il pourra être détecté chaque fois qu'une nouvelle valeur est produite.

4.532 Autres événements

Définir un événement non élémentaire revient aussi à définir un module qui implante la fonction spécifiée par les opérateurs intervenant dans la définition.

En sortie de ce module on trouve la donnée équivalente à l'événement

Exemple : EVT = EV1 AVANT EV2

l'implantation de "AVANT" peut utiliser l'heure d'occurrence de EV1 et EV2.

Ainsi, le module de production de EVT est

```
Module MEVT
  sortie EVT
  entrée EV1, EV2, heure1, heure2
  si heure1 < heure2 alors EVT
fin du Module MEVT
```

Il faut que EV1 et EV2 soient arrivés pour juger si EVT l'est ou non. Des connexions expriment ensuite les liens entre EVT et la tâche concernée d'une part, entre EV1, EV2 et les modules qui les produisent d'autre part.

4.533 Conclusion

Notre langage de déclaration d'événement revient à spécifier des fonctions de modules spécifiques qui traduisent les diverses combinaisons d'événements.

En spécifiant également, la consommation et la consultation d'une part, l'exemplaire concerné d'autre part, on lève les ambiguïtés soulevées au chapitre 3 sur les données qu'on retrouve au niveau des événements, est-ce chaque occurrence ou une seulement qui doit être prise en compte ?

4.6 DONNEES TRANSMISSIBLES, EVENEMENTS ET CONDITIONS

4.61 Expression de conditions et événements à partir de données

Une donnée $\mathcal{D}$ est définie par un ensemble de connexions de sortie et d'entrée de module.

On a vu qu'un événement était aussi une donnée transmissible, qu'une condition pouvait être associée à un événement.

Associée à $\mathcal{D}$ on définira donc les conditions utilisées et les événements en fonction des besoins du concepteur. La syntaxe pourra être quelque chose comme ce qui suit.

< sortie > de Mod → < Donnée >
↳ < entrée > de < Mod >

{avec / sans} consommation,

{récent / ancien} exemplaire

< Cond > : < Donnée > vide | < Donnée > pleine

: longueur (< Donnée >) = x < entier >

: {récent / ancien} exemplaire < Donnée > < opérateur de relation > < référence >

< événement > : transition de < cond > faux, vrai |
transition de < cond > vrai, faux

4.62 Expression de condition à partir d'événements

Si on part maintenant de la définition des événements qui a été donnée au § 4.33, on pourra définir les conditions par

< COND > : Evénément < EVT > VIVANT
: Evénement < EVT > NON VIVANT

La forme générale d'une structure de niveau 3 devient alors

sur < événement > {avec / sans} consommation {ancien / récent} exemplaire

si < COND > {avec / sans} consommation {ancien / récent} exemplaire

faire < Tâche >

Nous avons introduit de plus deux conditions systèmes associées aux tâches - la condition < tâche > active
et l'inverse la condition < tâche > inactive

La condition "tâche active" est vraie pendant la durée de vie de l'événement "début de tâche".

On remarquera que de nombreuses conditions font intervenir les compteurs de synchronisation introduits par VERJUS [ROBERT et Al - 77]. Ce ne sont pas toutefois les seules possibilités. En effet, on a la possibilité d'introduire des comparaisons par rapport

à des références auquel cas, nous précisons de quel exemplaire de la donnée il s'agit. Des comparaisons peuvent introduire des blocages c'est pourquoi au niveau des structures de niveau 3 nous préciserons si la condition est de type consommable ou non.

D'autre part, si toutes les conditions ne sont pas exprimées au niveau structure de contrôle, mais réparties entre une structure et les entrées du premier module de la tâche, le blocage pourra intervenir à l'un ou l'autre des niveaux. Il est important que compte tenu des connexions et du paramètre consommation des conditions les conditions soient encore vraies au moment de l'activation effective de la tâche. Au moment de l'implantation, il faudra tenir compte de cette contrainte.

4.7 PROBLEME DES FEUX DE CIRCULATION

Rappel d'énoncé [CROCUS - 75]

La circulation au carrefour de deux voies est réglée par des feux verts ou rouges. Quand le feu est vert pour une voie, les voitures qui y circulent peuvent traverser le carrefour ; quand le feu est rouge, elles doivent attendre (on admet que les voitures de chaque voie traversent le carrefour en ligne droite). On impose les conditions suivantes :

a) toute voiture se présentant au carrefour doit le franchir au bout d'un temps fini,

b) les feux de chaque voie passent alternativement du vert au rouge, chaque couleur étant maintenue pendant un temps fini,

c) à un instant donné, le carrefour ne doit contenir que des voitures d'une même voie.

Les arrivées sur les deux voies sont distribuées de façon quelconque.

Le fonctionnement de ce système est représenté par un ensemble de processus parallèles.

Un processus changement *gère la commande des feux, et un processus particulier est associé à chaque voiture. La traversée du carrefour par une voiture de la voie i (i = 1,2) correspond à l'exécution d'une procédure* traversée$_i$ *par le processus associé à cette voiture. On demande d'écrire le programme du processus* changement *et des procédures* traversée$_1$ *et* traversée$_2$ *dans les deux cas suivants :*

Cas 1. *Le carrefour peut contenir une voiture au plus à la fois.*

Cas 2. *Le carrefour peut contenir k voitures au plus à la fois.*

Le fonctionnement correct des feux doit être maintenu quel que soit l'ordre d'arrivée des voitures ; la modification des feux par changement *doit donc comporter les opérations suivantes :*

- interdire aux nouvelles voitures arrivant sur la voie où le feu est vert de s'engager dans le carrefour (pour respecter la condition b)),

- attendre que les voitures engagées dans le carrefour en soient sorties avant d'ouvrir le passage sur l'autre voie (pour respecter la condition c)).

Dans le cas 1, on a un problème d'exclusion mutuelle ; dans le cas 2, on pourra remarquer que le problème est analogue à celui des lecteurs et rédacteurs, et adopter une solution du même type.

Nous avons choisi le problème des feux de circulation pour montrer diverses solutions illustrant les dualités événement-condition-donnée transmissible.

Nous verrons d'ailleurs que certaines solutions sont peu naturelles car mettant en oeuvre des données transmissibles qui sont

peu naturelles. On préférera donc certainement les solutions plutôt basées sur les événements.

Les tâches à écrire et leurs synchronisations sont

. les deux traversées que nous appellerons NS et EO
. le changement de feu

les évènements sont :

. arrivée d'une voiture NS et arrivée d'une voiture EO
. volonté de changement de feu (par exemple périodique)

les conditions de traversée sont

. feu vert et une place au moins dans le carrefour.

Nous allons donc écrire

sur arrivée de voiture NS si feu NS vert et place libre
faire traversée NS

sur arrivée de voiture EO si feu EO vert et place libre
faire traversée EO.

Nous considérons qu'une traversée a une donnée en entrée qui est le carrefour, ainsi qu'en sortie puisqu'il y a libération de la ressource. Le carrefour sera une donnée transmissible schématisée par une file de N places, N étant le nombre de places du carrefour.

Une entrée dans le carrefour se traduit donc par une consommation d'une place, une sortie par une production de une place.

Dans le cas où N = 1, on a aussi l'exclusion mutuelle sur la seule place disponible.

La tâche traversée NS (respectivement EO) consiste donc en un module

module traversée
entrée carrefour
sortie carrefour
:
traversée
:
fin du module traversée.

On a défini une donnée transmissible que nous appellerons C

carrefour de traversée NS → C
carrefour de traversée EO → C

C → carrefour de traversée NS avec consommation
C → carrefour de traversée EO avec consommation
avec par exemple l'attribut "ancien exemplaire" : ce pourrait être un autre ;

la condition "place libre" est liée à cette donnée transmissible par

place libre : C non plein.

Nous précisons maintenant la structure de contrôle de niveau 3 :

sur arrivée de voiture NS si feu NS vert et place libre
avec consommation sans cons avec cons
faire traversée NS

sur arrivée de voiture EO si feu EO vert et place libre
avec consommation sans cons avec cons
faire traversée EO

En ce qui concerne le changement de feu, nous avons envie d'écrire

sur volonté de changement de feu si carrefour libre faire changement de feu

où changement de feu est une tâche qui associera le passage des feux de vert à rouge et rouge à vert.

Cette écriture implique qu'il faut la volonté de changement et le carrefour libre pour que le moindre changement ait lieu. Cette écriture n'évite pas la coalition éventuelle des voitures pour interdire le changement. On écrira donc :

sur volonté de changement de feu faire Mise au rouge

sur fin de mise au rouge avec consommation, ancien exemplaire
si carrefour libre récent exemplaire faire Mise au vert
sans consommation

la condition "carrefour libre" est définie par
carrefour libre : C vide

Les tâches Mise au rouge et Mise au vert sont constituées chacune d'un module de même nom que la tâche :

```
Module Mise au rouge
    entrée feu NS, feu EO
    sortie feu NS, feu EO
cas feu NS = vert alors feu NS : rouge
    feu EO = vert alors feu EO = rouge
fin du Module Mise au rouge

Module Mise au vert
    sortie feu NS, feu EO
    état X init feu NS
cas X = feu NS alors feu EO = vert
                     X = feu EO
     X = feu EO alors feu NS = vert
                     X = feu NS
fin du Module Mise au vert
```

Les connexions relatives aux feux seront

```
feu NS de Mise au rouge → FNS
feu NS de Mise au vert  → FNS
```



```
FNS → feu de NS de Mise au rouge
                   avec consommation
                   récent exemplaire
```

condition "feu NS vert" : FNS = vert récent exemplaire.

De cette spécification on peut déduire une autre forme en intégrant les conditions sous forme d'entrée de modules.

Associons à la condition Feu NS vert (respt Feu EO vert) une donnée transmissible de même nom associée à l'événement passage de Feu NS (respectivement EO) de rouge à vert.

Le module traversée devient :

```
Module traversée
    entrée feu NS vert, carrefour
    sortie carrefour
      ⋮
    traversée
      ⋮
Fin du Module traversée
```

Le module Mise au vert devra alors produire la donnée Feu NS vert :

```
Module Mise au vert
    sortie feu NS, feu EO, feu NS vert, feu EO vert
    état X init feu NS
    cas X = feu NS alors feu EO := vert
                         X := feu EO
                         feu EO vert
         X = feu EO alors feu NS := vert
                         X := feu NS
                         feu NS vert  sac
fin du Module Mise au vert.
```

On ajoutera alors les connexions :

feu NS vert de Mise au vert → FNSV

FNSV → feu NS de traversée
sans consommation
récent exemplaire

Si on désire intégrer la condition "carrefour libre" comme une entrée du module Mise au vert, il faut l'évaluer dans les modules qui sont connectés à la donnée C, donc dans traversée que nous sommes par la cause obligés de découper avec un comptage en début et un comptage en fin. Le découpage est obligatoire si on ne veut pas faire toutes les traversées en exclusion mutuelle.

Module Début de traversée NS

entrée Nb places, carrefour, feu NS vert
sortie Nb places
Nb places = Nb places - 1
fin du Module Début de traversée NS.

Module traversée

⋮
traversée
⋮
fin du module traversée

Module fin de traversée NS

entrée Nb places
sortie Nb places, carrefour, carrefour libre
Nb places = Nb places + 1
si Nb places = N alors carrefour libre
Fin du module fin de traversée NS

La sortie carrefour libre produite par la dernière "fin de traversée NS" doit être consommée par le premier module "Début de traversée NS".

On découpera donc le module

Début de traversée NS en deux modules
Deb1 et Deb2.

Module Deb1
entrée carrefour libre
fin de Module Deb1

Module Deb2
entrée Nb places, carrefour, feu NS vert
sortie Nb places
Nb places := Nb places - 1
fin du Module Deb2

L'enchaînement de la tâche est définie par

si NB = N alors Deb1 fsi - Deb2 - traversée - fin de traversée

On ajoutera dans Mise au vert

entrée carrefour libre

et les connexions

carrefour libre de fin de traversée NS → Cℓ

carrefour libre de fin de traversée CO → Cℓ

Cℓ → carrefour libre de Deb1, avec consommation, récent exemplaire
Cℓ → carrefour libre de Mise au vert, avec consommation
récent exemplaire

Nb places de début de traversée NS → NB → Nb places de début de traversée
(récent exemplaire avec cons)
Nb places de fin de traversée NS → NB → Nb places de fin de traversée
(récent exemplaire avec cons)

On remarque, que les conditions définies au niveau des structures ou des données transmissibles allègent considérablement l'écriture. dès qu'on interdit la programmation d'attente à l'intérieur des modules (c'est le cas de notre système).

4.8 PROBLEMES DES LECTEURS-REDACTEURS

Nous avons vu au chapitre précédent, comment, naturellement par les déclarations d'entrée et de sortie, on pouvait définir les processus lecteurs et rédacteurs en assurant l'exclusion mutuelle entre lecteur et rédacteur d'une part, entre rédacteurs d'autre part.

Nous nous attachons ici à résoudre les problèmes liés aux priorités et à l'enchaînement des uns par rapport aux autres.

Nous verrons en particulier que dans ce problème les conditions testées sont plus relatives aux événements déjà survenus qu'aux données transmissibles et à leur état.

4.81 Cas 1 [CROCUS - 75]

Priorité des lecteurs sur les rédacteurs sans réquisition, le seul cas où un lecteur doive attendre est celui où un rédacteur occupe le fichier.

Cette règle est déjà respectée par la gestion des entrées et des sorties.

Les évènements sont

- demande de lecture
- demande de rédaction
- fin de rédaction
- fin de lecture (fin de toutes les lectures en cours) et pas de lecture en attente.

1) sur demande de lecture si pas de rédacteur actif faire lecture sans consommation, récent exemplaire

2) sur demande de rédaction si pas de lecteur actif ou en attente sans consommation, récent exemplaire
faire rédaction

3) sur fin de rédaction si lecteur en attente avec consommation ancien exemplaire
et rédacteur en attente sans consommation récent exemplaire
faire lecture

si lecteur en attente avec consommation ancien exemplaire
et pas de rédacteur en attente sans consommation récent exemplaire
faire lecture

si rédacteur en attente avec consommation ancien exemplaire
et pas de lecteur en attente sans consommation récent exemplaire
faire rédaction

4) sur fin de lecture si rédacteur en attente avec consommation ancien exemplaire
faire rédaction

Les évènements sont tous consommables, dans l'ordre où ils arrivent ; il s'agit donc chaque fois du plus ancien exemplaire.

Il y a des redondances avec les déclarations d'entrée et de sortie.

1) Est inutile
2) expriment bien la priorité des lecteurs
3 et 4) détaillent le traitement du scheduler sur les événements fin de rédaction et fin de lecture.

Nous introduisons ici des conditions du type de celles exprimées par les compteurs de VERJUS et ROBERT [ROBERT et Al - 77] pas de rédacteur actif, pas de lecteur en attente. Ces conditions sont associées à certains événements :

début de rédaction
fin de rédaction
demande de lecture etc... (§ 4.62)

Exemple : pas de rédacteur actif.

Cette condition est définie par

non (début de rédaction et non fin de rédaction)
pas de lecteur en attente

Exemple :

Cette condition est définie par

demande de lecture vivant.

Nous n'avons pas détaillé chacune des conditions pour les laisser sous une forme plus naturelle, montrant ainsi combien il est agréable de pouvoir exprimer les conditions de "scheduling" à partir d'événements.

L'introduction de condition pré-définie associée aux tâches : tâche active et tâche inactive peut également résoudre simplement le problème.

Cas 2 [CROCUS - 75]

Priorité des lecteurs sur les rédacteurs si et seulement si un lecteur occupe le fichier.

Quand aucun lecteur ne lit, les lecteurs et rédacteurs ont même priorité. Par contre dès qu'un lecteur lit, tous les autres lecteurs peuvent lire, quelque soit le nombre de rédacteurs en attente.

1,2 ne changent pas par rapport au cas 1. Dans le cas 1, on avait la structure 3),

sur fin de rédaction
si lecteur en attente et rédacteur en attente faire lecture

qui exprimait bien la priorité de la lecture sur l'écriture.
Ici

sur fin de rédaction si lecteur en attente et rédacteur en attente, s'ils ont tous deux la même priorité, il faut par exemple activer le premier arrivé.

sur fin de rédaction si lecteur en attente avant rédacteur
avec consommation, ancien exemplaire
en attente faire lecture
si contrainte faire rédaction ;

il faut donc se souvenir de l'ordre des évènements, c'est ce qu'a déjà remarqué ROUCAIROL [1978], [1979] en introduisant les mots de synchronisation.

Ces mots sont en fait l'image de la file ordonnée d'arrivée des événements.

Cas 3 [CROCUS - 75]

Priorité égale pour les lecteurs et rédacteurs.

Si un lecteur utilise le fichier, tous les lecteurs nouveaux qui arrivent y accèdent jusqu'à l'arrivée d'un rédacteur. A partir de ce moment, tous les nouveaux arrivants attendent sans distinction de catégorie.

événement fin de lecture : fin d'une lecture

1) sur demande de lecture si pas de rédacteur actif faire lecture
ou en attente
(sans consommation, récent exemplaire)

2) sur demande de rédaction si pas de lecteur actif faire rédaction
et pas de rédacteur actif
sans consommation, récent exemplaire

3) sur fin de rédaction si lecteur en attente avant rédacteur
en attente
(avec consommation, ancien exemplaire)
faire lecture
si rédacteur en attente avant lecteur
en attente faire rédaction
(avec consommation, ancien exemplaire)

4) sur fin de lecture si pas de lecteur actif et rédacteur en
attente faire rédaction
(avec consommation, ancien exemplaire)

On peut montrer qu'il suffit de tester "rédacteur en attente" plutôt que "rédacteur en attente" avant "lecteur en attente".

En effet, si ce n'était pas vrai, les lecteurs en attente auraient été activés et seraient terminés au moment de l'événement "fin de toutes les lectures en cours".

Cas 4 [CROCUS - 75]

Priorités rédacteurs sur les lecteurs.

1) sur demande de lecture si pas de rédacteur faire lecture
   actif ou en attente
   (sans consommation, récent exemplaire)

2) sur demande d'écriture si pas de rédacteur actif faire rédaction
   (sans consommation, récent exemplaire)
   et pas de lecteur actif

3) sur fin des lectures en cours si rédacteur en attente faire
   rédaction
   (avec consommation, ancien exemplaire)

4) sur fin d'écriture si rédacteur en attente faire rédaction
   avec consommation, récent exemplaire
   si pas de rédacteur en attente et lecteur
   en attente faire lecture
   (avec consommation, ancien exemplaire)

4.9 QUELQUES COMPARAISONS AVEC D'AUTRES SYSTEMES

Nous avons donc vu que le parallélisme pouvait être décrit de diverses façons, grâce aux données transmissibles et aux structures de contrôle de niveau 3. Nous avons vu l'équivalence entre les événements, les conditions et les données transmissibles.

Selon le point de vue du concepteur ou du programmeur il pourra utiliser indifféremment l'une ou l'autre des descriptions.

En appliquant les règles de dualité il pourra passer d'une forme à une autre en fonction des contraintes ou de choix d'implantation.

Dans tous les cas la description du parallélisme se fait de façon complètement externe aux modules de traitement.

Ces modules n'ont a priori en entrée que les données nécessaires aux calculs de l'algorithme quand la synchronisation est effectivement liée à la production de ces données d'entrée. Les autres paramètres ou règles d'activation apparaissent alors dans les structures de niveau trois.

Si on ne veut pas répartir la description de la synchronisation entre les données transmissibles et les structures de niveau trois, il faut intégrer les événements et les conditions sous forme d'entrées artificielles de module. Ces entrées ne sont là que pour la synchronisation sans intéresser les calculs à proprement parler. On a ainsi une solution peu naturelle. Ces données transmissibles supplémentaires introduites uniquement pour la description de la synchronisation sont (bien que sous une autre forme) analogues aux variables historiques [HOWARD - 76] ou variables auxiliaires utilisées par exemple dans [BRINCH - HANSEN - 73] ou [OWICKI et GRIES - 76]. Les compteurs d'événements et les séquenceurs de [REED et KANODIA - 79] en sont une autre forme. Ils ont pour rôle d'ordonner les événements d'une classe donnée par l'intermédiaire

d'une primitive "ticket". Cette primitive est implicite dans notre système dès lors que nous avons précisé que production et consommation se traduisent par des opérations de rangement et extraction dans la file de données transmissibles. Ces opérations sont ordonnées. Nous n'avons fait aucune hypothèse sur la façon dont elles l'étaient. Nous pouvons par exemple utiliser la technique de [REED et KANODIA - 79] ou un algorithme comme celui de LE LANN 77 ou [LAMPORT - 78] ou [DARGENT - 79] pour une implantation répartie. Par contre dans le cas d'une implantation sur processeurs à mémoire commune d'autres méthodes sont possibles, par exemple des moniteurs [HOARE - 74] bien que parfois trop contraignants. Cette primitive "ticket" ordonne les événements d'une classe comme les mots de synchronisation de [ROUCAIROL - 79] .

Toutefois on doit distinguer deux niveaux dans ces divers outils, sachant qu'un même outil peut appartenir aux deux niveaux. On peut envisager des outils de description et des outils d'implantation adaptés en particulier aux systèmes répartis. Les mots de synchronisation apparaissent plutôt comme des outils de description et spécification comme les modules de contrôle de [ROBERT et Al - 77], les compteurs de [REED et Al - 79] plutôt comme des outils d'implantation.

Nous supposons toujours un ordonnancement aussi bien dans les données transmissibles que dans les événements, quand nous précisons EVT1 : EVT2 $\left\{\begin{matrix}\underline{\text{AVANT}}\\ \underline{\text{APRES}}\end{matrix}\right\}$ EVT3 sans nous préoccuper de la façon dont ils seront implantés et de la façon dont l'ordonnancement sera assuré.

Un autre point de vue consiste à intégrer le plus possible de variables de synchronisation au niveau des événements, et des conditions des structures de contrôle de niveau 3, en ne laissant naturellement en entrée et sortie que les données dont en fait la valeur est importante pour les calculs. On se rapproche ainsi du point de DIJKSTRA avec les commandes gardées [DIJKSTRA - 74] et les PAR.Cs introduits par [GRIEM - 76] puis développés par HAASE et HALLING [HAASE - 77], [HAASE - 79], [HALLING - 77], [HALLING - 79]

Dans ces systèmes toute la synchronisation est ramenée au niveau des conditions qu'on suppose évaluées en parallèle. Cette hypothèse d'évaluation parallèle est implicite dans notre système à partir du moment où les structures de contrôle de niveau 3 peuvent être évaluées à n'importe quel moment si aucune contrainte d'ordre n'est précisée explicitement.

Toutefois, notre système offre la possibilité de définir des événements en plus, ou à la place des conditions. Nous pouvons ainsi espérer une définition plus souple et plus adaptable aux contraintes d'implantation [CAUMONT - 79].

Les commandes gardées sont reprises par HOARE pour définir les processus séquentiels communiquants [HOARE - 78] et par BRINCH HANSEN pour définir les processus distribués [BRINCH HANSEN - 78].

Le premier utilise des primitives INPUT et OUTPUT internes aux processus à synchroniser pour recevoir et émettre respectivement les informations que nous avons déclassées en ENTREE et SORTIE. Toutefois une différence essentielle avec notre système réside dans le fait que HOARE suppose qu'il y a toujours rendez-vous entre les deux processus au niveau des INPUT - OUTPUT, ou on suppose une coïncidence mutuelle [MUNTEAN - 79] que nous n'avons pas a priori, hypothèse également faite par [CHANDY - 79]. La façon dont les "CSP" s'enchaînent et se synchronisent se rapproche de la façon dont les sous-programmes et programmes s'enchaînent par le protocole d'exécution de sous-programmes à distance défini dans [SCHAFF - 74] et [THOMESSE - 74].

Dans notre système, une donnée transmissible peut être interprétée comme une liaison multipoint, ou un réseau de diffusion sélective alors qu'un couple INPUT - OUTPUT de HOARE peut apparaître comme une liaison bipoint, ou une voie logique unidirectionnelle entre deux ports comme les voies du réseau ARPA [CROCKER - 72].

Quand une donnée est en entrée sans consommation, c'est

exactement une liaison multipoint. Par contre, en cas de consommation, il s'agit d'une liaison multipoint avec le protocole de Sélection - structation [LORRAINS - 79].

La possibilité de HOARE d'inclure des INPUT dans les commandes gardées lui permet la communication de variables entre les actions et l'élaboration des conditions. Nous avons cette facilité dans les conditions associées aux données transmissibles.

Dans les processus distribués, BRINCH HANSEN inclut aussi les commandes gardées, mais il y ajoute les régions gardées [HOARE - 72], [BRINCH HANSEN - 79] qui permettent de mettre en attente un processus jusqu'à ce qu'une variable d'état soit à une certaine valeur ce qui revient à programmer des primitives INPUT au sens de HOARE dans une commande gardée.

Les processus se synchronisent uniquement par exécution d'appels de procédure des processus à l'intérieur desquelles les commandes et/ou régions gardées retardent et synchronisent les processus appelant. Les informations sont communiquées par passage de paramètres en entrée et en sortie, soit par copie dans une mémoire commune, soit par entrée et sortie entre processeurs. Les procédures sont communes aux divers processus. Les processus sont distribués et non les procédures. On a donc encore ici comme dans les moniteurs, une structure de capsule indivisible au sens géographique, structure bien adaptée à la gestion de ressources implantées sur un seul site, même si les requêtes sont distantes mais beaucoup moins souple pour la description de la synchronisation entre processus distants accèdant à des ressources réparties. Si on exprime ceci en termes de réseaux, un moniteur est assez bien adapté à la modélisation d'un serveur d'une ressource locale (quoique peut être trop restrictif). Par contre c'est beaucoup moins vrai quand il s'agit de modéliser un serveur d'une ressource répartie. En particulier, parce que les procédures du moniteur pourraient être exécutées autrement qu'en exclusion mutuelle.

Dans ADA [ICHBIACH - 79 a et b], la synchronisation est réalisée par rendez-vous entre tâches parallèles, ou appel de procédure de point d'entrée déclaré en ENTRY correspondant à une instruction ACCEPT point du rendez-vous. Si nous le désirons, nous pouvons utiliser le rendez-vous (cf. § 3.7), auquel cas nous avons la même possibilité que dans ADA. Dans ADA, une séquence peut être exécutée en exclusion mutuelle dans la procédure appelée. Il nous faut alors un module particulier qui soit un module "Rendez-vous" qui contienne la séquence figurant dans ACCEPT, DO.... END, de ADA.

L'exclusion mutuelle peut être assurée par consommation, production (conformément au § 3.6).

Le côté "temps réel" de ADA apparaît par la gestion des interruptions par la directive FOR .... USE AT ; et par l'instruction DELAY qui permet de retarder l'exécution d'une alternative.

Ces structures sont très simples et devraient permettre des créations de super-structures comme les activations périodiques depuis un certain événement jusqu'à un autre. Toutefois l'introduction des événements n'est pas aisée, elle ne peut être faite directement. Il faut utiliser l'équivalence événement condition pour tester des conditions dans une structure SELECT.

Un autre point de vue nous est conné par les systèmes dits pilotés par les données dont un des premiers fut celui de DENNIS [DENNIS - 71] . A partir de ces premiers travaux furent développés plusieurs systèmes : nous citerons le système LAU de TOULOUSE [DURRIEU - 79], [SYRE et Al - 78]. Ainsi que le système de l'Université d'IRVINE [ARVIND et Al - 77].

Dans les deux cas l'expression de la synchronisation est du type programmation globale avec une structure séparation des synchronisations et des actions à synchroniser à l'intérieur d'un objet analogue en première approximation à une classe de Simula [DAHL - 66]. Les règles de synchronisation sont exprimées sous forme de conditions qui sont vérifiées quand l'action est exécutée

au même titre que dans notre système avant l'activation effective d'une tâche.

Un objet de ce système est donc formé de diverses parties dont les règles de synchronisation. On y retrouve la forme des capsules, des types abstraits de telle sorte que nous n'observons pas la même séparation du contrôle et du traitement que dans notre système. On pourrait faire la même remarque à propos de "Dataflow - monitors" [ARVIND et Al - 77], ou une partie de la capsule contient non seulement les conditions mais les règles de "scheduling" des actions.

Dans le même ordre d'idée on peut associer les acteurs de la théorie de HEWITT, [HEWITT et Al - 77], [YONEZAWA et Al - 78] à la différence que la théorie des acteurs est une expression dynamique alors que les systèmes "dataflow" sont plutôt statiques comme dans notre système où nous n'avons pas défini de création dynamique de processus et de création dynamique de connexions entre modules. Toutefois, un point commun entre les deux systèmes consiste en le fait qu'on suppose toujours un ordre partiel des événements.

Enfin, nous indiquerons un point capital dans notre système qui fait souvent l'objet d'hypothèses contradictoires entre les divers langages et systèmes cités précédemment. Nous ne supposons pas que le moyen de communication est avec ou sans mémoire. Avons-nous une simple ligne ou un canal à mémoire ? Nous ne faisons pas d'hypothèses à ce sujet. Une donnée transmissible sera aussi bien implantée dans un système à mémoire quand ce sera nécessaire, ou dans un système sans mémoire quand ce sera possible.

En particulier, une donnée transmissible implantée sans mémoire impliquera des blocages des producteurs si des consommations du plus ancien exemplaire sont prévues.

On obtiendra ainsi une synchronisation du même type que celle de HOARE avec les INPUT - OUTPUT, avec une mémorisation de

données et rendez-vous. Ou bien, on attendra que producteurs et consommateurs soient en mutuelle coïncidence [MUNTEAN - 79], c'est-à-dire que si le processus émetteur est au rendez-vous avant le processus émetteur, il attend un signal d'arrivée de ce dernier pour émettre les données. Un mécanisme de synchronisation fait que les deux primitives OUTPUT et INPUT sont exécutées en même temps. Nous pensons qu'il s'agit de cas d'implantation d'une donnée transmissible.

Finalement, notre système se révèle apte à décrire le parallèlisme de façon purement déclarative et le plus naturellement possible. Il est particulièrement bien adapté à la programmation d'application en conduite de procédés industriels, aussi bien des algorithmes de régulation avec une synchronisation très simple en début et fin qu'une prise en compte de nombreux événements pour une conduite de procédés séquentiels.

Les contraintes d'implantation seront vues ultérieurement. Elles seront facilement intégrées par le fait même que nous avons montré comment on pouvait passer d'une représentation à une autre grâce à la dualité entre événement - condition et donnée transmissible. En ce sens peut être pourrons-nous rapprocher les points de vue assez lointains en apparence de ceux qui développent les systèmes pilotes par les données et ceux qui pensent plutôt flot de contrôle.

4.10 CONCLUSION

Nous n'avons fait aucune hypothèse sur la façon dont les divers éléments seraient implantés, ni comment les diverses tâches et les modules seraient activés et gérer. Quand nous parlons d'implantation, il s'agit non seulement de connaître le matériel qui supportera les divers éléments, mais aussi la façon dont les

événements et conditions seront respectivement détectés et évalués.

Nous n'avons jusqu'ici que proposé une description pour des applications en temps réels en insistant sur le fait que cette description devait être indépendante de toute implantation. Ceci se retrouve au niveau de la description des synchronisations pour laquelle nous avons été amené à introduire sous forme purement déclarative le fait qu'une entrée se validait avec ou sans consommation et l'équivalent d'un numéro d'exemplaire. Cette description est aussi à un niveau tel que nous pouvons passer de cette spécification à n'importe quel niveau matériel (éventuellement cablé) pour l'implanter.

Il reste peut être un point sur lequel on pourrait encore préciser des options : en ce qui concerne le "scheduling" des tâches et des modules, nous n'avons fait que l'hypothèse d'un "scheduler" sûr, à savoir que tout processus activable sera effectivement activé au bout d'un temps fini. Nous avons introduit, pour les événements la notion de durée de vie associée à la notion d'échéance de la tâche. Il semblerait donc normal que le scheduler organise les enchaînements en fonction de ces paramètres. Nous ne l'avons pas fait de cette façon ; nous nous sommes contentés de traduire ces paramètres en termes de priorité pour rester compatibles et réutiliser des systèmes existants. Toutefois, des travaux en cours [SCHNEIDER - 80] intégrant ces paramètres dans le choix d'une implantation particulière.

La modélisation proposée par SCHNEIDER permet en plus d' assurer certaines verifications de cohérence et donc d'effectuer des preuves de validité de solution, comme [SZLANKO - 77], [DESCHIZEAUX et Al - 79].

REFERENCES DU CHAPITRE 4

[ARVIND et Al - 77]

[BRINCH-HANSEN - 73], [BRINCH-HANSEN - 78], [BRINCH-HANSEN - 79]

[CAUMONT - 79], [CAUMONT], [CHANDY - 79], [CKOCKER - 72]

[CROCUS - 75]

[DAHL - 66], [DENNIS - 71], [DESCHIZEAUX et Al - 77],

[DESCHIZEAUX et Al - 79], [DIJKSTRA - 74], [DURRIEU - 79]

[FLYSTRA - 75]

[GRIEN - 76]

[HAASE - 77], [HAASE - 79], [HALLING - 77], [HALLING - 79]

[HEWITT et Al - 77], [HOARE - 78], [HOLT - 71], [HOWARD - 76]

[IBM], [ICHBIAH - 79 a, b]

[KARP et Al - 66]

[LADET - 77], [LEDGARD - 75]

[MENDELBAUM - 76], [MIDDLETON - 77], [MUNTEAN - 79]

[NONN - 78 a], [NONN - 78 b]

[OWICKI et Al - 76]

[REED et Al - 79], [ROBERT et Al - 77], [ROUCAIROL - 78],

[ROUCAIROL - 79]

[SCHAFF - 74], [SEMS], [SZLANKO - 77], [SYRE et Al - 78]

[THOMESSE - 74]

[YONEZAWA et Al - 78]

DEUXIEME PARTIE

Dans les quatre chapitres précédents nous avons étudié une structuration d'applications en temps réel et réparties. Nous avons proposé une description du parallélisme et des synchronisations à partir de trois concepts :

- données transmissibles
- événements
- conditions.

Associées à chacun de ces objets, des opérations de

- production
- consultation
- consommation
- duplication

permettent d'exprimer les différents cas de synchronisation, et de passer d'une expression basée sur un concept, à une expression basée sur un autre.

A partir du chapitre 5, nous abordons une autre partie essentiellement consacrée aux problèmes d'implantation, nous étudions l'implantation des modules des structures de niveau 2, des structures de niveau 3.

Au chapitre 6, nous étudions ces implantations possibles de données transmissibles, y compris les entrées sorties industrielles.

Au chapitre 7, nous abandonnons les problèmes d'implantation pour envisager une amorce de méthoce de programmation basée sur la structuration proposée dans la première partie.

CHAPITRE 5

IMPLANTATION DE L'APPLICATION

L'implantation de l'application consiste à :

- implanter les modules écrits dans un langage source
- implanter les structures de contrôle de niveau 2
- implanter les données transmissibles et les connexions avec les modules
- implanter la détection des événements, des conditions et les structures de niveau 3 associées.

Pour réaliser toutes ces implantations, il faut préciser les options qui relèvent de cette étape et donc qui n'ont pas été spécifiées au moment de la définition de l'application. Il n'y a pas que le lieu d'implantation à spécifier, il faut aussi indiquer comment certains éléments seront effectivement réalisés. Nous citerons par exemple la façon dont les éléments sont détectés, la manière de réaliser le lien événement-tâche par interprétation et par compilation puis exécution.

Dans cette partie, nous allons étudier non seulement ce qui a déjà été réalisé [COCHET-MUCHY - 78], [NONN - 78], [CAUMONT - 79] mais aussi diverses solutions aux problèmes posés. Ces diverses solutions montrent ce qui pourrait être spécifié par un langage de spécification d'implantation. Nous n'avons pas défini de tel langage mais ce serait certainement une des premières extensions de cette étude.

5.1 IMPLANTATION DES MODULES

Les modules sont écrits dans un langage de programmation ; toutefois leur spécification en termes de variable d'état, d'entrées et de sorties permet l'implantation d'un module sous forme d'un automate en logique câblée. Nous verrons un exemple dans lequel un module spécifié sous forme d'un programme peut

être implanté en logique câblée et relié aux autres modules grâce aux déclarations de connexion [AUBRY - THOMESSE - 79].

Dans ce qui suit, nous ne nous intéresserons qu'aux implantations sous forme de programmes. Cette partie fait largement appel aux travaux de [COCHET-MUCHY - 78] ,[COCHET-MUCHY et Al - 80], et [CAUMONT - 79].

Compte tenu de l'indépendance déjà largement soulignée entre conception et implantation de chacun des modules, nous avons scindé la production de programmes en deux grandes phases : une première indépendante du site d'implantation, la seconde intégrant les contraintes d'implantation.

La première phase consiste en une compilation du texte en langage source (L.S.) des modules, en un texte en langage intermédiaire (L.I.) indépendant lui aussi des matériels utilisés.

La seconde phase consiste d'abord en une génération de texte écrit en langage d'assemblage pour la machine cible. C'est donc à ce moment qu'on intégrera la paramètre "site d'implantation". Un assembleur produit ensuite le code objet.

Ces deux étapes de la seconde phase pourraient être facilement intégrées en une seule. Ce qui éviterait d'écrire un assembleur pour chaque machine. Nous n'avons pas retenu cette facilité car nous pensons que notre langage ne permet certainement pas de faire face à toutes les situations (en contraintes de temps, d'espace). Il est donc important dans certains cas de programmer les modules dans un langage très proche de la machine, en langage d'assemblage en particulier [AUBRY et Al - 80]. Il est alors possible à l'utilisateur d'écrire directement un module en langage d'assemblage de la machine cible. La première phase sera alors réduite à une compilation des déclarations d'entrée et de sortie afin de rester compatible avec le restant de l'application.

L'utilisation de langage intermédiaire n'est pas nouvelle. On peut citer [HOLLECZEK - 76] pour la compilation de PEARL et les compilations de Pascal par exemple [TISSERANT - 77]. Nous verrons que la fait d'envisager plusieurs machines cibles modifie sérieusement le problème de la détermination de ce langage.

5.12 Le langage source d'écriture des modules

Le langage source d'écriture des modules a déjà été évoqué au chapitre 2. Nous allons étudier ici un langage défini dans [COCHET-MUCHY - 78]. Un autre exemple de réalisation [CAUMONT - 79], à partir d'un autre langage source sera rapidement cité .

La langage de programmation des modules a été défini afin de permettre simplement la programmation d'algorithmes indépendamment des matériels qui les supporteraient. Il se rapproche assez du langage Pascal sans ses déclarations de type.

5.121 Les objets utilisés

Les objets manipulés dans le langage source (L.S.) sont les variables simples et les tableaux comme en Fortran. Les types prédéfinis sont les entiers, les réels, les booléens, les entiers dits courts pour tenir compte des machines 8 bits utilisées très fréquemment dans notre domaine d'application.

Nous n'avons pas introduit de déclaration de type mais seulement une déclaration d'atiribut qui permet d'indiquer ce

que représente un identificateur.

```
REEL       VIT1, VIT2
ATTRIBUT   VITESSE ;
VITESSE    VIT1, VIT2 ;
```

Il est ainsi possible à chacun de définir les attributs qu'il désire utiliser. Aucune obligation n'est faite. Toutefois, il nous a semblé important, dans le cadre de l'amélioration des qualités des programmes, de préciser d'une autre façon que uniquement par entier , réel, etc.
Les définitions de ces attributs n'ont comme portée que la portée des variables auxquelles ils se rapportent, c'est à dire le module. En revanche, on pourra vérifier à la connexion des entrées et sorties de modules que les données connectées possèdent les mêmes attributs. Ceci impose que l'attribut ait une portée globale à l'ensemble des modules concernés par les connexions. Cette portée globale est obtenue par la répétition de la déclaration d'attribut dans chacun des modules concernés. Cette déclaration d'attribut pourrait être assortie de déclaration d'unité et d'équation aux dimensions afin de vérifier l'écriture des programmes.

Déclarations

REEL	Liste d'identificateurs
ENTIER	Liste d'identificateurs
COURT	Liste d'identificateurs
BOOLEEN	Liste d'identificateurs
TABLEAU ENTIER	Liste d'identificateurs de tableau
TABLEAU REEL	Liste d'identificateurs de tableau
TABLEAU COURT	Liste d'identificateurs de tableau
TABLEAU BOOLEEN	Liste d'identificateurs de tableau.

Un identificateur de tableau est un identificateur simple suivi des dimensions entre parenthèses.

ATTRIBUT Liste d'identificateurs d'attribut

Un identificateur d'attribut est un identificateur qui peut être réutilisé comme mot clé pour déclarer l'attribut aux variables qui le possèdent.
Il manque certainement à cette liste les chaînes de caractère. Ce serait une extension à apporter dans une véritable réalisation industrielle.

5.122 Les instructions

. Les expressions arithmétiques utilisent les opérateurs : addition, soustraction, moins unaire , multiplication, division. Les règles de parenthésage et de conversion dans les cas d'expression mixte sont les mêmes qu'en Fortran.

. Les expressions logiques portent sur les variables et constantes booléennes et utilisent les opérateurs logiques habituels ET, OU, NON, OU EXCLUSIF et les parenthèses.
Les opérateurs de relation >, <, ≤, ≥, = , # sont utilisables pour obtenir des expressions logiques.
Les expressions sont utilisables dans les instructions d'affectation qui est la même qu'en Algol 60. Le signe d'affectation est :=
On ne peut pas affecter une valeur à une donnée déclarée en entrée uniquement. On ne peut affecter une valeur qu'à une variable simple ou à un élément de tableau.

. L'instruction conditionnelle

```
Si condition alors Instruction fsi
              ou
Si condition alors Instructions
           sinon Instructions fsi
```

. Les instructions de contrôle d'itérations

tant que condition faire Instructions ftq

pour variable de borne à borne pas pas faire
Instructions fp

. Comme l'ont montré divers auteurs (par exemple [LEDGARD et A1 - 75]), les structures sont plus que suffisantes pour programmer tout algorithme. Elles ne sont pas toujours agréables à utiliser et certaines instructions comme cas pourraient être facilement ajoutées.
Notre but n'était pas de définir un langage de type algorithmique nous avions seulement besoin d'un minimum de structures pour une première implantation expérimentale.

Il serait certainement intéressant de reprendre une implantation avec un langage plus évolué comme langage source de programmation. Un langage comme PASCAL défini dans [WIRTH - 71] devrait faire facilement l'affaire avec des spécifications de chaînes de caractères.
Les instructions d'entrée et de sortie sont LIRE et ECRIRE dont les paramètres sont précisés d'une manière analogue aux FORMAT de Fortran ou de Procol [RITOUT - 72].
Les périphériques sont désignés de manière globale dans l'ensemble de l'application. Chaque périphérique a donc un nom unique.
Un module commence par une déclaration de module suivie de son nom et se termine par la déclaration de fin de module suivie encore de son nom.

MODULE < nom du module >
Déclarations
Instructions
FIN DE MODULE < nom du module >

Exemple de module

```
MODULE    calcul
  ENTREE   E1 ;
  SORTIE   S ;
  ENTIER   E1, I, J ;
  REEL     S, T ;
  ATTRIBUT SOMME ;
  SOMME S ;
     LIRE I, J ;
     S := E1 + I * J ;
     T := 2 * S ;
     ECRIRE T
  FIN DE MODULE  calcul ;
```

Les deux instructions d'entrée-sortie sont implantées sous forme de modules particuliers qui sont connectés aux modules qui demandent l'exécution de ces opérations. C'est ainsi que le problème de l'implantation respective du module exécutant LIRE et ECRIRE et du périphérique concerné est résolu en les implantant sur le même site.

On aurait pu faire une exception pour les entrées et sorties, à la règle qui veut que dans une tâche, entre les modules constituants, la seule synchronisation se fait par communication de données exprimées par les connexions. Cela n'a pas été fait. C'est à dire que lorsqu'on implante une opération LIRE, ceci revient à découper le module en un module qui précède, un qui suit avec entre eux exécution du module de LECTURE qui représente en fait une tâche activée sur l'événement désir de lecture et qui communique les résultats par l'intermédiaire des connexions.
L'asynchronisme est donc ici encore exprimé au niveau de la structure de niveau 3 et des connexions.

```
MODULE      CALCUL 1
     SORTIE   COMMANDE
       :
FIN du MODULE  CALCUL 1
MODULE      LECTURE
     ENTREE   COMMANDE
     SORTIE   VALEUR 1, VALEUR 2
       :
FIN DU MODULE  LECTURE

MODULE      CALCUL 2
     ENTREE   I, J
       :
FIN DU MODULE CALCUL 2
```

Les connexions sont exprimées par :

COMMANDE de CALCUL 1 → COMMANDE de Lecture
ancien exemplaire
avec consommation

VALEUR 1 de LECTURE → I de CALCUL 2
VALEUR 2 de LECTURE → J de CALCUL 2

Les options sont ici selon la synchronisation désirée.
Nous avons laissé l'utilisateur écrire les modules concernant la gestion des entrées sorties. Certains de ces modules ont été écrits en standard et sont donc utilisables tels quels par un programme utilisateur. Toutefois, il serait nécessaire de développer considérablement ces fonctions en automatisant complètement la génération des événements et des connexions à partir des ordres LIRE, ECRIRE .

5.13 Le langage intermédiaire

Le langage intermédiaire est indépendant des matériels utilisés. Toutefois, il est voisin des langages d'assemblage. Il est du type langage à une adresse. Nous sommes ainsi voisins de la plupart des machines actuelles à registre. Il pourrait ainsi être pris comme langage d'assemblage universel comme celui de [NICOUD - 76] mais indépendant de tout mode d'adressage.

Nous allons étudier ce langage avant de comparer notre choix à quelques autres travaux dans ce domaine.

5.131 Les codes d'opérations du langage intermédiaire

Ils expriment :

a) Les transferts entre mémoire centrale et registre de l'unité centrale

RAN C (registre) → adresse en mémoire
CHA C (adresse) → registre.

b) Les opérations arithmétiques entre contenu d'un registre et contenu d'une adresse en mémoire centrale.

- addition ADD
- soustraction SOUS
- multiplication MUL
- division DIV

Ce sont les mêmes codes pour les divers types de données réel, entier ou court.
C'est au moment de la génération de code qu'on tiendra compte du type effectif des opérandes et des ressources de la machine cible pour traduire une telle instruction soit par une instruction si

elle existe, soit par une séquence d'instructions.

c) Les opérations logiques

OU logique OU
OU exclusif OUX
ET logique ET

Ces opérations logiques et arithmétiques ont toujours deux opérandes dont l'un est dans un registre, le second à une adresse en mémoire ou dans un registre. On ne fait aucune hypothèse quant à la façon dont on atteint les opérandes. Il n'y a aucune hypothèse quant aux modes d'adressages : direct, indirect, immédiat, indexé, relatif basé, etc.

Deux instructions n'ont qu'un opérande.

- moins unaire (complément à 2) NGT
- négation logique NON

Elles opèrent sur un registre uniquement.

d) Des tests et branchements

comparaison	CMP	
branchement inconditionnel	BRA	
branchement conditionnel	BE	égalité
	BNE	différent
	BI	inférieur
	BIE	inférieur ou égal
	BS	supérieur
	BSE	supérieur ou égal

e) Des instructions d'appels et retour de sous programme

BSP branchement à sous programme
RSP retour de sous programme

Bien que la notion de sous programme n'existe pas intégralement dans le langage source, nous avons introduit la possibilité dans un module d'appeler plusieurs fois une séquence d'instructions préécrite de bibliothèque.

Les sous programmes sont intégrés dans un module. Ceci est pratiqué pour au moins autoriser l'appel de fonctions. Nous ne l'avons pas implanté mais ce serait nécessaire de la faire.

5.132 Les directives du langage intermédiaire

Elles permettent :

a) La définition des constantes	CR	Constante réelle
	CE	" entière
	CC	" courte
des zones de travail	MAN	
b) La définition des variables	VLØC	Variables locales
	VCØM	Variables en entrée et/ou sortie
	TLØC	Tableaux locaux.

Nous n'avons pas introduit de tableaux en entrée et/ou sortie. Il serait possible de la faire. Ceci se traduirait par des connexions entre tableaux représentant l'ensemble des connexions entre éléments de ces tableaux.
En revanche, ce ne serait pas aussi simple d'introduire des connexions dynamiques du type : sortie → T(I) de module.
I jouerait ici le rôle de multiplexeur interne, postionné dynamiquement par une autre connexion

sortie → I de module.

Ceci n'est pas autorisé dans notre système. Cette remarque vaut pour d'autres structures que les tableaux.

c) La définition d'étiquettes EPC

5.133 L'adressage en langage intermédiaire

Compte tenu des objets manipulés dans le langage source variables simples et tableaux, nous avons défini quelques formules qui représentent l'adresse de l'opérande qui figure en

langage intermédiaire. Ainsi nous n'avons pas à faire de choix à ce niveau en ce qui concerne les modes d'adressage de la machine cible. Ces formules sont de la forme générale suivante :

$$X \left(\begin{matrix} + \\ - \end{matrix} \right) V * C \left(\begin{matrix} + \\ - \end{matrix} \right) V \left(\begin{matrix} + \\ - \end{matrix} \right) C$$

où V et C représentent respectivement une variable et une constante. Quand on n'adresse qu'une variable simple, la formule est réduite à X.

Dans le cas où l'opérande est une constante, il eut été normal de noter l'adresse de rangement de la constante. C'est vrai pour les opérandes : constantes réelles. Par contre, dans le cas des constantes entières, on les conserve telles quelles afin de choisir le plus possible l'adressage immédiat au moment de la génération de code. On est ainsi obligé de garder la valeur pour déterminer si elle est ou non accessible en adressage immédiat.

Exemple 1.

. Adressage de l'élément de tableau TABL(3,4)
L'opérande correspondant à TABL(3,4) est : TABL + 2* 't_1'+3
où 't_1' représente la valeur déclarée de la première dimension.

. Adressage de l'élément de tableau TABL(I,J)
L'opérande correspondant est : TABL + I* 't_1'+ J-'t_1' - 1

. Adressage de V(2)
L'opérande est V + 1

Exemple 2.

[COCHET-MUCHY - 78]

```
TABLEAU REEL A(6),C(3)
        ENTIER D,I
REEL B
 A(5)=B + C(I) + D
```

Il sera traduit en langage intermédiaire par :


```
CHA  B
ADD  C + I - 1
ADD  D
RAN  A + 4
```

On remarque que c'est la même instruction ADD qui est utilisée pour ajouter un opérande réel et un opérande entier.

Désignation des registres

Les registres ne sont pas désignés dans le langage intermédiaire. On suppose que les registres sont en nombre infini pour effectuer les opérations. Aucune instruction de rangement intermédiaire n'est donc programmée en langage intermédiaire.

Exemple : A:=(B+C) * (D+E)

```
CHA  A
ADD  C
CHA  D
ADD  E
MUL
RAN  A
```

L'opération MUL a lieu sur les derniers registres utilisés. On considère ainsi que les registres sont en pile.

5.134 Nous avions le choix entre plusieurs solutions pour la définition du langage intermédiaire

a) Le langage contiendrait ce nombreuses instructions afin d'être le plus près possible de toutes les machines. On pourrait l'assimiler au OU inclusif de tous les langages d'assemblage existant.

b) Le langage contient un nombre minimum d'instructions permettant en fait d'exprimer uniquement les algorithmes sans entrer dans le détail des registres et d'instructions très spécialisées. Le langage est alors proche du langage source ; il se rapproche, au moins en ce qui concerne les codes, du langage L.I.1 de [LEBARBIER - 78].

c) Le langage aurait pu être un langage à trois adresses. On était ainsi amené à introduire des variables auxiliaires au moment de la compilation en langage intermédiaire, variables qui auraient pu être supprimées à la génération de code compte tenu d'un algorithme de gestion des registres d'unité centrale.

Nous avons préféré ne les introduire qu'à la génération pour ne pas avoir à en supprimer.

d) Plusieurs langages intermédiaires auraient pu être envisagés comme [LEBARBIER - 78] ou [NEBUT - 74] pour l'implantation de systèmes. Ceci dépend du langage source choisi. Notre langage source est suffisamment simple et d'autre part, notre langage intermédiaire possède suffisamment de ressources en ce qui concerne l'adressage pour que la compilation ne nécessite pas plus d'une étape intermédiaire.

Notre langage se rapproche du cas b). Ce choix a été guidé par le fait que la plupart des matériels utilisés mini et micro ont des instructions dont les codes sont voisins. De plus, nous n'avons pas fait de différence entre les machines à registres banalisés et les machines à registres spécialisés (accumulateurs, registres d'index, ...). Les problèmes d'allocation de registre sont les mêmes dans les deux cas dès qu'il y a plusieurs registres spécialisés de même type. L'optimisation de l'allocation des registres comme dans [CARTER - 77], [BEATTY - 74] est assurée au niveau du générateur de code. Nous ne nous sommes pas posés le problème des machines à pile. Toutefois, il nous semble que la fait de considérer les registres comme étant en nombre infini et comme empilés devrait favoriser l'implantation du langage intermédiaire sur de telles machines.

5.14 Génération de code

La phase de génération de code consiste à produire un programme en langage d'assemblage de la machine cible. C'est ici qu'on doit

prendre en compte le paramètre machine cible. Le langage d'expression de l'implantation des modules devrait être analogue à :

Sur < site > implanter < module >

ou

Sur < site > implanter < tâche > dans le cas où tous les modules de la tâche sont implantés sur le même site.

On doit tenir compte des ressources de la machine et des ressources nécessaires à l'exécution du programme pour réaliser cette opération.

Ainsi, si une opération d'addition de réels est programmée dans le langage intermédiaire, elle devra être traduite par une instruction d'addition en flottant si cette opération existe sur la machine cible, sinon elle sera traduite en un appel de fonction prédéfinie.

Dans notre implantation de [COCHET-MUCHY - 78], la description de la machine cible n'est pas faite formellement. Elle est intégrée dans le jeu de macro-instructions écrites pour la machine cible. Il serait certainement intéressant d'engendrer automatiquement un générateur de code à partir d'une description formelle de la machine cible. Il faudrait alors définir un langage tel que celui de [LENZER - 79].

Nous avions le choix entre l'écriture d'autant de générateurs de code que de machines cibles ou d'un seul générateur de code paramétré. Nous avons pris la deuxième solution sous forme d'un macroprocesseur (MACP du Solar 16) avec une bibliothèque de macro-instructions pour chaque site. Dans notre optique, c'est donc les macro-instructions propres à un site que nous devons écrire à chaque adjonction de matériel nouveau ; c'est donc l'écriture de ces macro-instructions qui pourrait être automatisée à partir d'une description de la machine cible. Cette solution présente l'avantage de n'écrire qu'une fois de nombreux traitements qui sont les mêmes pour tous les sites.

Les principales fonctions remplies ici sont :

. Adapter les données (variables, tableaux, constantes) en fonction de leurs déclarations, de la représentation interne et des modes d'adressage disponibles sur la machine cible.

. Traduction des instructions compte tenu du jeu de code d'opérations disponible.

. Génération de séquences d'initialisation propres à chaque site et nécessitées par les modes d'adressage (chargement initial de registres de base, de pointeurs de piles) ou par la communication avec le réseau (transmission des données).

Sont opérationnels un jeu de macro instructions pour le Solar 16-40 et un jeu pour le microprocesseur Motorola 6800. D'autres sont en cours d'étude (INTEL 8080 et Signétics 2650).

Le compilateur a été écrit en assembleur Solar 16, les assembleurs croisés sont écrits en Fortran et le générateur de code est construit autour de MACP. L'ensemble est implanté sur Solar 16-40.

5.15 Implantation des modules sur PB6

Le PB6 est un automate programmable, matériel plus adapté à prendre en compte des événements externes, gérer des délais que faire des calculs ou exécuter des algorithmes complexes.

Une implantation originale de notre système sur ce matériel a été faite par P. CAUMONT [CAUMONT - 79]. Le langage d'écriture des modules est le langage d'assemblage du PB6. Compte tenu des particularités du matériel, certaines structures de contrôle de notre système ont été réintégrées au moment de l'implantation (cf dernier chapitre) dans le corps des modules afin de profiter des instructions de base du matériel en question.

En ce qui concerne les modules uniquement, on les considère donc comme étant écrits dans un langage d'assemblage avec les directives

d'entrée et de sortie de déclarations des données transmissibles. Les instructions sont celles du langage d'assemblage et non celles de Sygare.

L'assemblage de ce langage est fait grâce à un assembleur défini par l'outil de production de logiciel APL2M [ROHMER - 78]. Il a été implanté sur un matériel APLIXI conçu autour d'un Mitra 15.

De cette implantation on tirera d'abord une première conclusion. Le langage source d'écriture des modules n'est pas un point fondamental de notre projet. N'importe quel langage peut être utilisé, en particulier tous les langages d'assemblage si le site d'implantation du module est connu avant même son écriture. Il n'est pas nécessaire de faire abstraction du site d'implantation par plaisir. Si ce n'est pas nécessaire ni même utile, il est préférable de profiter dès le début de cette information pour écrire des programmes certainement plus adaptés et plus performants.

Si on respecte les déclarations de modules d'entrée, de sortie afin de rester compatibles, tous les langages peuvent être utilisés. Nous pouvons ainsi tirer une seconde conclusion sur laquelle on reviendra dans le dernier chapitre. Sur les automates programmables classiques, synchronisation et traitement sont couramment imbriqués dans les programmes. Il est vrai que dans les applications industrielles où on trouve des automates programmables, les traitements sont très faibles en volume. Par contre, on peut être amené à envisager des milliers d'entrée. Il est alors évident qu'à chaque entrée, on peut faire correspondre un événement, l'état des autres entrées pouvant être considéré comme contexte, comme condition de prise en compte de ces événements. Ces séquences sont donc très nombreuses. Il est impossible de les prendre toutes en compte.

Compte tenu de ces remarques, nous avons donc été amenés à restructurer une application conçue et découpée selon les éléments présentés au chapitre 2. Des modules ont été recomposés par intégration des structures de contrôle de niveau 2. Des tâches ont été refondues par intégration de structures de niveau 3. Ces intégrations peuvent être systématisées. Asini, une définition en quatre points, modules, connexions, structures de niveau 2 et de niveau 3, peut conduire à

une implantation sous forme de mono tâche dans laquelle tous les éléments ont été intégrés.

5.2 IMPLANTATIONS DES STRUCTURES DE CONTROLE DE NIVEAU 2

Chaque structure de contrôle peut être implantée de diverses façons : - soit sur un seul site, - soit sur plusieurs sites et dans chacun des cas, elle peut être compilée pour être exécutée soit interprétée. Dans le cas de compilation, on distinguera encore deux cas selon que la structure est compilée sous forme d'un programme séparé du texte des modules, soit un programme réparti dans le texte des modules. Quand les modules concernés sont implantés sur un même site et que la structure qui les relie est répartie dans le texte des modules, on retombe sur le cas où on a reconstruit un module en incluant la structure à l'intérieur.

5.21 Implantation d'une structure sur un seul site

Une déclaration de tâche se traduit par un enchaînement de modules, spécifié par diverses structures : séquentiel, parallèle, conditionnel, itératif.

Dans l'implantation de [NONN - 78], la structure de contrôle constituant une tâche à partir de modules est compilée sous forme d'une tâche.

Chaque module est une entrée dans cette table. Pour chacun d'eux, il y est noté, le ou les prédécesseur(s), le ou les successeurs. La table est une représentation du graphe d'enchaînement des modules de la tâche. La table est interprétée au moment de l'exécution de la tâche. Les structures de contrôle si alors sinon et tant que sont interprétées quand elles sont rencontrées.

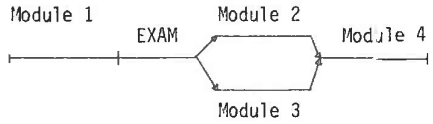
Structure de la table.

Modules	Prédécesseurs		Successeurs		Condition
	Nombre	Noms	Nombre	Noms	

Une exécution de tâche est un parcours du graphe d'enchaînement. Les structures de contrôle sont interprétées sous forme de pseudo module. La condition aussi bien dans une conditionnelle que l'itération est considérée comme une donnée transmissible de type booléen. Le pseudo module fait partie de l'interpréteur des tables d'enchaînement. Nous le noterons EXAM.
Ainsi une structure :

Module 1 - si Cond alors Module 2
sinon Module 3 fsi - Module 4

se traduit par le graphe



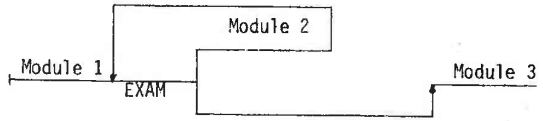
Une table des conditions d'enchaînement permet de spécifier à l'interpréteur quel est le successeur dans le cas de modules enchaînés sous forme de structures conditionnelles ou itératives. Il est à noter que si un même module apparaît plusieurs fois dans une tâche, une entrée dans la table sera associée à chaque instance de module.

Structure itérative.

Une structure

Module 1 - tant que Cond faire Module 2 ftq - Module 3

se traduit par le graphe



La table d'enchaînement associée est :

Modules	Prédécesseurs		Successeurs		Condition
	Nombre	Noms	Nombre	Noms	
Module 1	0	-	1	EXAM	de module
EXAM	1	Module 1	2	Module 2	Cond de module vrai
				Module 3	Cond de module faux
Module 2	1	EXAM	1	EXAM	
Module 3	1	EXAM	0	-	

Cette solution permet de modifier très simplement une structure de contrôle de niveau 2.

Une autre solution est la compilation de la structure de contrôle sous forme d'un programme exécutable et non plus sous forme d'une table associée à un interpréteur.

Lancer un module revient alors à activer une procédure ou un sous programme. Nous ne nous préoccupons pas pour l'instant des données transmissibles. La structure de niveau 2 définissant une tâche se traduira alors par un programme exécutable.

Comme le programme principal bâti uniquement par des appels à des sous programmes.

- Enchaînement séquentiel - Module 1 - Module 2
  call Module 1
  call Module 2
- Enchaînement conditionnel
  si cond alors call Module 3
  sinon call Module 4
- Enchaînement itératif
  tant que cond faire call Module 5
- Enchaînement parallèle
  il se ramène à un enchaînement séquentiel puisque nous avons supposé que tous les modules étaient implantés sur le même site.

Une édition de liens classique construira l'ensemble de la tâche à partir des modules et de la structure de niveau 2.

Toute modification, soit d'un module, soit de la structure de niveau 2, se traduire, dans ce cas par une nouvelle édition de liens de la tâche. De plus, cette solution facile à implanter quand la structure et les modules doivent être implantés sur le même site l'est beaucoup moins quand les modules sont répartis.

Quand les modules sont répartis sur différents sites, les appels de sous programmes sont remplacés par des ordres d'activation "Activer (Module)". Ainsi une structure de contrôle de niveau 2 est traduite par un programme composé uniquement des instructions :

- activer (Module)
- attente fin module
- si alors sinon
- tant que faire

Le parallélisme est alors obtenu par une exécution séquentielle sans attente des ordres d'activation des modules parallèles.

Exemple

La structure M1 // M2 // M3 est traduite par l'enchaînement

```
activer (M1)
activer (M2)
activer (M3)
```

La structure M1 - M2 - M3 est traduite par

```
activer (M1)
attente fin (M1)
activer (M2)
attente fin (M2)
activer (M3)
```

Note. Il est à remarquer que si un des modules parallèles est implanté sur le même site que le programme traduisant la structure, il devra être le dernier à être activé afin de permettre l'exécution de toutes les autres primitives Activer (M1).

5.22 Interprétation d'une structure de niveau 2

On fait intervenir à ce niveau les sites d'implantation des différents modules. Un désir d'activation d'un module se traduit selon sa présence ou son absence du site où est implanté l'interpréteur par :

```
Si site (Module) = ici alors activer (Module)
                   sinon .construire message
                         "activer module"
                         .émission vers site (Module)
```

A la réception de "Activer Module", le site récepteur exécutera la primitive activer (Module) pour le compte du site demandeur.
La terminaison du module se traduit toujours par un appel au système local. Le module exécute lui-même cet appel analogue à SVC EXIT

du système RTES du Solar. Le site d'exécution rendra compte de cette exécution soit à son interpréteur de structure de niveau 2, soit au site demandeur par le message "fin Module". La réception de ce message permettra alors à l'interpréteur demandeur de poursuivre l'interprétation de la table.

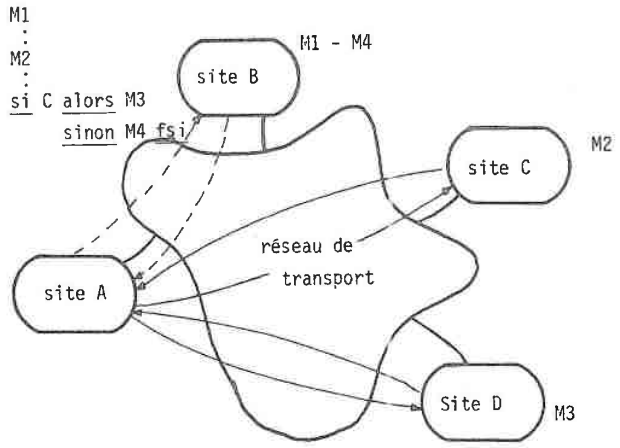
Le protocole est donc très simple, il ne comporte que les échanges des messages suivants :

+ Activer le Module < nom du Module >
+ Fin du Module< nom du Module >

Si on désire installer divers contrôles, on peut introduire de nouveaux messages comme Début d'Activation du Module < nom du Module >. Ce serait un compte rendu précédent le message "fin du module".

Dans une première réalisation, nous n'avons considéré que les deux messages correspondant à l'ordre d'activation et au compte rendu de fin d'exécution.

Exemple.



site A émet "Activer M1" vers site B

à la fin de M1

site B émet "fin du module M1" vers site A

l'interpréteur décide alors d'émettre :

"Activer M2" vers site C

à la fin de M2, site C émet "fin du Module M2" vers site A

L'interpréteur interprète alors la structure suivante

si C alors

sinon

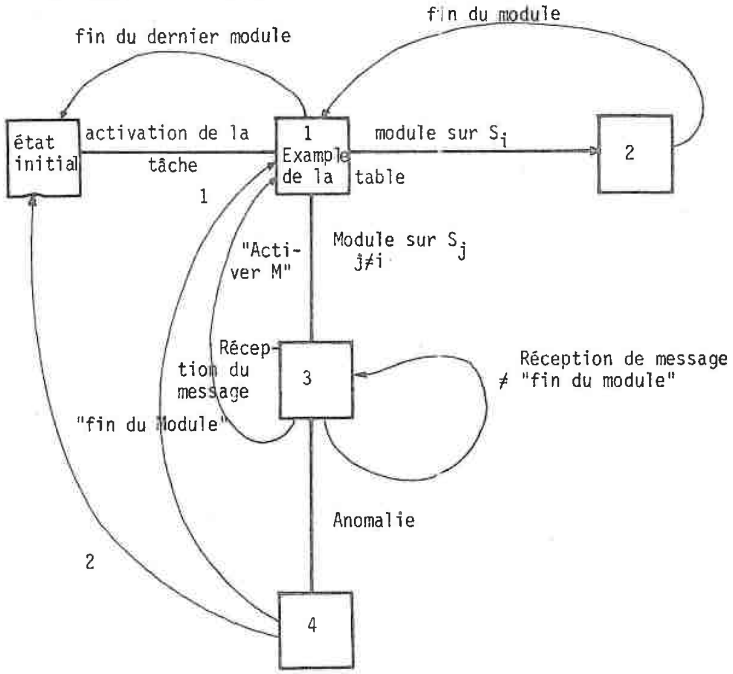
selon le résultat, il émettra :

"activer M3" vers site D

ou

"activer M4" vers site B.

Automate simplifié représentant l'interprétation d'une structure de niveau 2 sur site i.



L'anomalie sera par exemple la signalisation par un protocole de bout en bout de la non réception du message "Activer M".

L'état 4 sera alors un état de recherche d'un nouveau site, de mise à jour de la table d'enchaînement et de reprise 1 ou d'abandon pur et simple de la tâche 2 avec retour à l'état initial. De nombreuses stratégies sont applicables.

L'anomalie peut aussi être détectée par l'interpréteur car la fin du module n'est pas survenue au bout d'un certain temps alloué (déduit de la durée de vie des événements). Cette anomalie pourrait aussi être seulement détectée au niveau du traitement des structures de niveau 3.

5.23 Implantation d'une structure sur plusieurs sites

Ici nous considérons qu'une structure de niveau 2 peut être répartie sur plusieurs sites. Une répartition de la structure sur plusieurs sites peut être intéressante pour éviter de trop centraliser la gestion des enchaînements de module, créant ainsi un trafic supplémentaire. Compte tenu d'une répartition des modules, il devrait y avoir une ou plusieurs répartitions de sous structures qui minimisent les transferts.
Cette implantation n'a pas été réalisée mais seulement envisagée. Ceci revient à fractionner le graphe donc la table d'enchaînement en plusieurs sous graphes donc en plusieurs sous tables.
Chacun des interpréteurs considérera alors le sous graphe local comme une structure à part entière donc une tâche particulière.
La fin d'interprétation d'une sous structure sera un événement sur lequel sera synchronisée la sous structure suivante.

Exemple.

T : M1 - M2 - ((M3-MO) // M4 // M5) - si C alors M6 // M7 - M8
sinon M6 // M9 - M7 fsi

Cette structure pourra être fragmentée en :

T : M1 - M2
T1 : (M3 - MO) // M4 // M5
T2 : si C alors T3
sinon T4 fsi
T3 : M6 // M7 - M8
T4 : M6 // M9 - M7

On créera alors les événements : fin de T
fin de T1

et les structures de niveau 3 suivantes :

sur fin de T faire T1
sur fin de T1 faire T2.

La table d'enchaînement associée à la tâche initiale T était :

	Module	Prédécesseurs Nombre	Prédécesseurs Noms	Successeurs Nombre	Successeurs Noms	Condition
A	M1	0	-	1	M2	//
	M2	1	M1	3	M3 M4 M5	// // //
B						
	M3	1	M2	1	MO	
	M4	1	M2	1	EXAM	C de Module
	M5	1	M2	1	EXAM	C de Module
	MO	1	M3	1	EXAM	C de Module
C						
	EXAM	3	MO, M4 M5	2	GR1 GR2	vrai C de Module faux
	GR(1)	1	EXAM	2	M6(1) M7	//
	GR(2)	1	EXAM	2	M6(2) M9	//
	M6(1)					
	M6(2)					
	M7					
	M8	2	M6(1) M7			
	M9					
D						

Cette table sera fractionnée en trois tables :

de A à B
de B à C
de C à D

Chacune de ces sous tables sera implantée sur un site différent. L'enchaînement des sous tables sera alors réalisé par l'implantation des événements. La détection de l'événement "fin de tâche"

sera assurée par le site d'implantation de la structure de la tâche qui s'est terminée. L'événement sera transmis au site d'implantation de la tâche suivante pour y être traité.

Dans l'implantation de [CAUMONT - 79], les modules sont implantés sur un automate programmable PB6 de Merlin Gérin ; la structure de contrôle de niveau 2 est elle même implantée sur le PB6. Nous rappellerons qu'un automate PB6 est un matériel qui n'exécute qu'une tâche monolithique de façon cyclique, la période d'activation étant juste légèrement supérieure à la durée d'exécution.
Compte tenu de cette contrainte (une tâche monolithique intégrant tous les traitements), nous sommes amenés à "concaténer" les modules et leur enchaînement spécifié par la structure de niveau 2.
Nous sommes même appelés à concaténer les tâches entre elles.
L'implantation utilise un registre pas à pas et par l'intermédiaire de l'instruction "avance", on indique dans le registre quel module doit être activé à la prochaine itération. Une différence importante entre ces matériels et les systèmes classiques basés sur une reconnaissance aysnchrone des événements impose quelques précisions sur ce qu'est un module dans les systèmes d'automates programmables (cf dernier chapitre).

Si on devait spécifier formellement l'implantation des structures de niveau 2, on pourrait donner comme définition :

Sur < site > implanter la structure de < tâche >
à ne pas confondre avec
Sur < site > implanter< tâche >

qui spécifie l'implantation des modules de la tâche. Tous les modules de la tâche peuvent être implantés sur un site et les structures sur un autre.

Compte tenu du découpage d'une structure de niveau 2 vu au paragraphe précédent, la définition d'implantation est suffisante pour indiquer les lieux.

En revanche, nous n'avons pas spécifié si la structure devait être compilée, interprétée, etc. Il nous semble que comme pour les modules, on imposerait une seule forme par site, auquel cas, le choix est imposé par le lieu et son système associé.

5.3 IMPLANTATION DES EVENEMENTS ET DES STRUCTURES DE CONTROLE DE NIVEAU 3

5.31 Introduction

La définition de l'implantation des événements ne consiste pas seulement en la précision du lieu où il sera détecté mais aussi en la façon dont il le sera.

Il y a de nombreuses façons de définir et détecter un événement :

- interruption
- lecture d'entrée numérique et détection de changement de valeur
- lecture d'entrée analogique et comparaison à un seuil ou une référence
- comparaison de la valeur d'une donnée transmissible à une référence
- événements associés aux conditions définies (passage de vrai à faux et passage de faux à vrai)
- fin de tâche
- première production d'une donnée.

Dans tous les cas, compte tenu de la structure de Von Neuman qui régit encore la plupart des machines connectables, la seule façon de détecter l'événement est de scruter l'objet associé. Par exemple :

- interruption, la scrutation a lieu à chaque instruction
- lecture d'entrée numérique, la scrutation sera programmée périodique, de période Δt à partir d'un instant initial

jusqu'à un instant final.

Ainsi, de nombreuses définitions d'événements sont en fait des choix d'implantations.

Dans l'implantation de [NONN - 78], seuls sont pris en compte les événements associés aux structures de contrôle équivalentes aux requêtes du système RTES. Les autres événements sont pris en compte directement par l'utilisateur dans une ou plusieurs tâches propres à leur détection.

Dans l'implantation de [CAUMONT - 79], tous les événements sont également détectés dans le programme de l'utilisateur par une scrutation périodique des éléments générateurs d'événements, entrées numériques, variables d'état, etc.

On peut systématiser la détection des événements et envisager une spécification de leur implantation au même titre que pour les modules ou les structures de niveau 2.

La spécification suivante est à la fois une définition et une indication sur le mode de détection.

- Evénement : transition d'une entrée numérique scrutée tous les Δt

```
Evt_i : tous les Δt faire
        début
            lire entrée
            si entrée ≠ ∂entrée alors | ∂entrée:=entrée
                                     | evt_i
                                     fsi
        fin
```

- Evénement : transition vrai-faux d'une entrée numérique scrutée tous les Δt

```
Evt_i : tous les Δt faire
        début
            lire entrée
            si entrée = faux
                  et
               ∂entrée = vrai alors | ∂entrée=entrée
                                    | evt_i
        fin
```


- Si plusieurs événements doivent être pris en compte sur le même site, il sera certainement intéressant d'effectuer des fusions d'itération tous les Δt faire ...

- L'implantation elle-même, de cette structure tous les Δt pourra être réalisée de différentes façons

exemple : PB6 Δt est égal à la durée du programme à exécuter augmentée d'un ε ; quand le programme est terminé, on boucle au début.

exemple APILOG Δt est égal à une valeur fixée par l'opérateur. Une horloge programmable délivre une interruption qui déclenche l'exécution du programme.

exemple Solar 16

Une solution utilisant une horloge programmable est tout à fait possible. Il est à noter que c'est la solution habituelle dans les systèmes temps réel classiques : la spécification pouvant se faire par l'intermédiaire de la requête TRNON ou sous forme d'événements positionnés au bout de certains délais avec les requêtes SEVDEL et SEVENT.

Exemple : système mono événement ou mono tâche

Bien que ce cas soit trivial, nous le citerons car on rencontre de plus en plus des microprocesseurs dont la tâche est particulièrement simple. Il suffit de scruter une ou quelques entrées et effectuer un traitement selon la ou les valeurs lues. La solution la plus simple est certainement l'attente active de la (ou des) valeurs qui déclenchent les traitements. Cette technique est aussi celle qui assure le meilleur temps de réponse dans les cas critiques.

Il nous semble donc naturel de définir des événements en plusieurs étapes :

- sémantique de l'événement
- nom de l'événement
- mode de détection
- site de détection
- durée de vie (optionnelle)

Les deux points - mode et site de détection - sont effectivement les seuls à spécifier dans un langage de spécification d'implantation.

Ainsi nous pouvons constater que ces définitions sont en fait des descriptions de la sémantique des événements. En ce qui concerne le site d'implantation, les spécifications éventuellement, déjà données pour les données transmissibles sont utilisées pour désigner le site de détection des événements qui leur sont liés. Il en est de même pour les événements "fin de tâche" et "fin de module". Le site d'implantation est celui d'implantation de la structure de contrôle de niveau 2 qui permet la constatation de fin de tâche ou le site d'implantation du module.

Quant aux événements concernant des entrées numériques et/ou analogiques, selon la désignation des adresses physiques, le site sera spécifié à l'intérieur de l'adresse ou devra être précisé par ailleurs.

Dans tous les cas où la définition d'un événement ne comporte pas de spécification du site, il faudra l'indiquer dans une rubrique "sites d'implantation".

Exemple : EVT1 : EVT2 ou EVT3
EVT2 : transition VF entrée n° i site j
EVT3 : entrée n° k site j = C1

Les définitions d'EVT2 et EVT3 contiennent le nom du site ; pour EVT1, il faudra définir le site de EVT1.
Par exemple : site de EVT1 : site j, mais ce peut être un autre.

Après une déclaration des événements et de leur durée de vie si besoin est, après la définition de ces événements et celle de la structure de contrôle

Sur EVT si COND faire Tâche

il ne reste qu'à définir l'implantation proprement dite ; c'est à dire qu'il faut définir le site de prise en compte et la façon dont elle a lieu.

Cette définition se fera par :

< événement > : scrutation tous les Δt	site i
:< module >	"
: interruption	"

On remarquera que l'interruption permet non seulement la définition de la sémantique de l'événement mais aussi la façon de le prendre en compte.

5.32 Implantation des structures de contrôle de niveau 3

L'ensemble des événements déclarés sur un site constitue ce que l'on appelle une file d'événements. On prend en entrée une file d'événements à traiter et une file de conditions. Compte tenu d'une table des liens événements, conditions, tâche associée, l'interpréteur a pour charge de vérifier que, lorsque l'événement est arrivé, les conditions sont vraies et alors il peut lancer la tâche fonctionnelle associée.
Très brièvement, l'algorithme d'interprétation des structures de contrôle de niveau 3 serait :

Pour chaque événement de file d'événement faire

1 - recherche de la condition associée à l'événement dans la structure de niveau 3
2 - recherche de la condition dans la table des conditions vraies ou évaluation de la condition
3 - si condition vraie alors
- "Activer tâche"
- Retirer événement de la file si l'événement est consommable
- Mettre la condition à faux si elle est consommable

Le point 1 "recherche de la condition associée à l'événement" dépend de la façon dont est compilée la structure "sur < evt > si < cond > faire < tâche >

Le point 2 peut être implanté de deux façons :

- Les conditions sont évaluées au moment où on les teste
- Les conditions sont évaluées systématiquement soit périodiquement, soit quand l'événement associé est détecté que ce soit de façon synchrone ou asynchrone.

Cette deuxième méthode est préférable car elle maintient une certaine cohérence dans la mesure où il ne s'écoule pas trop de temps entre le moment où l'événement est détecté et celui où la condition est évaluée.
Ceci est d'ailleurs important pour les événements dont la durée de vie est nulle, c'est à dire les événements qui ne peuvent être pris en compte que juste après qu'ils ont été détectés. L'évaluation de la condition ne doit donc pas être trop retardée.

L'activation de la tâche se traduit par une demande d'interprétation de la structure de niveau 2 associée. Cet exemple d'implantation est typique des mécanismes indirects avec exclusion mutuelle sur la phase consommation de l'événement et de la condition.
C'est le genre de mécanisme adéquat dès que certains contrôles doivent être assurés avant le lancement de la tâche ; en particulier dans le cas d'un"scheduling"basé sur la date limite de fin de tâche, il n'est pas nécessaire de lancer une tâche si on est sûr qu'elle ne peut pas être terminée dans les délais.
Le non lancement est alors un événement au même titre qu'un autre. Sa détection est assurée par le "scheduler" local. Si le concepteur l'a prévu, il a pu définir une tâche particulière de traitement de ce cas. Le traitement de cette anomalie est ici complètement intégré dans le système.

D'autres implantations sont envisageables, relevant plutôt des mécanismes directs (une tâche "hardware" liée à une interruption par exemple). Ces implantations ne sont sûres que lorsque tous les contrôles ont pu être faits avant l'exécution. Aucun risque d'interblocage ne doit être possible pour qu'une telle implantation soit choisie.

Remarques

Au moment de la définition des événements, on précise le site sur lequel chacun d'entre eux se produira. Dans certains cas, le même événement peut survenir sur des sites différents. Il faudrait alors introduire la possibilité de préciser plusieurs sites.

Exemple : événement : message opérateur "< texte >"

Le message peut être introduit à partir de divers sites. On peut être amené à considérer qu'il s'agit du même événement ; on devrait donc préciser événement : site k_1, site k_2 ...
L'indication de consommation ou consultation permet de lever l'ambiguité due à deux entrées "simultanées".
En vertu de la dualité événement-condition, il en est de même pour les conditions que pour les événements. Les structures de contrôle peuvent être réparties indépendamment de la répartition des événements et des conditions. Alors au moment de l'implantation se pose le problème du temps qui n'est pas le même sur tous les sites comme l'ont déjà montré [LAMPORT - 78], [LE LANN - 77]. Nous n'avons pas étudié ce problème particulier. Compte tenu de notre conception, il s'agit bien d'un problème d'implantation des événements, des conditions et des structures de niveau 3. Si pour une structure l'événement et les conditions sont situés sur la même machine, il n'y a aucun problème de temps. En revanche, si l'événement et la condition sont situés sur des machines différentes, le problème de temps se pose comme dans le cas où un événement est défini par des relations entre d'autres événements, où un ordonnancement sera nécessaire.

De nombreux algorithmes ont déjà été proposés [LE LANN - 77], [REED et KANODIA - 79], [LAMPORT - 78], [DARGENT - 79].
Des tentatives d'évaluation de ces algorithmes sont faites. Elles sont nécessaires pour déterminer leurs coûts, leurs performances [WILMS - 78].

La plupart de ces algorithmes sont basés sur l'allocation dynamique des données dans les bases de données réparties.
Dans notre cas où toutes les ressources sont connues avant l'exécution grâce aux déclarations d'entrée et de sortie, où de plus les demandeurs des mêmes ressources sont également connus, des solutions simplifiées devraient être envisageables.
On sait en effet par les déclarations de connexion, quels sont les modules qui utilisent les mêmes ressources qu'un module donné. Une prévention de l'interblocage est donc déterminée lors des demandes d'activation des modules.

Notre implantation était du type centralisée. Tous les événements étaient pris en compte sur un seul site qui interprétait les structures de contrôle et distribuait les ordres d'activation aux modules. Cette implantation simplifie considérablement les problèmes de synchronisation. Compte tenu des travaux d'équipes travaillant sur les bases de données réparties [DARGENT - 79], [HOLLER - 79], [ELLIS - 77], il est possible d'envisager d'autres implantations.

En particulier, on peut déterminer dans une application diverses tâches indépendantes à des niveaux divers [ABIGNOLI et Al - 80] depuis des tâches complètement indépendantes au niveau des événements, conditions et données transmissibles. Elles peuvent être implantées sur des sites différents sans aucune contrainte ni problème. Des tâches liées par les conditions d'activation sans l'être par les données pourront être réparties n'importe comment mais, pour simplifier l'interprétation des structures de niveau 3, les structures pourront être implantées sur le même site.
De même, compte tenu des liens entre modules par les données transmissibles, il sera certainement intéressant d'étudier des implantations optimales afin de diminuer le temps de réponse ou le nombre de messages de service. Pour une formalisation de notre découpage d'application en termes de réseaux de Pétri temporisés [SCHNEIDER - 80], une telle tâche est en cours en utilisant un interpréteur de réseaux de Pétri temporisés écrit en APL [PLAIGNAUD - 79].

REFERENCES DU CHAPITRE 5

[ABIGNOLI et Al - 80], [AUBRY et A' - 79], [AUBRY et Al - 80]

[BEATTY - 74]

[CARTER - 77], [CAUMONT - 79], [COCHET-MUCHY - 79],
[COCHET-MUCHY et Al - 80]

[DARGENT - 79]

[ELLIS - 77]

[HOLLECZEK - 76], [HOLLER - 79]

[LAMPORT - 78], [LEBARBIER - 78], [LE LANN - 77], [LENZER - 79]

[NEBUT - 74], [NONN - 78]

[PLAIGNAUD - 79]

[REED et KANODIA - 79], [RITOUT - 72], [ROMMER - 78]

[SCHNEIDER - 80]

[TISSERANT - 77]

[WIRTH - 71]

CHAPITRE 6

IMPLANTATION DES DONNÉES TRANSMISSIBLES

6.1 INTRODUCTION

Dans ce chapitre, nous étudions comment on peut envisager l'implantation des données transmissibles.
Comme pour les événements, le problème d'implantation n'est pas qu'un problème de choix d'un site. De nombreux paramètres sont à prendre en compte.

- nombre de modules connectés
- types de connexion - consultations seulement
  - consommation seulement
- appartenance à une même tâche ou non
- taille de la file
- localisation des modules concernés sur des monoprocesseurs ou multiprocesseurs
- répartition imposée des modules concernés ou laissés au choix du système.

Notre but ici n'est pas d'envisager tous les cas qui peuvent se présenter. Toutefois, en étudiant quelques diverses solutions qui s'offrent habituellement à l'implantation, nous voudrions montrer l'universalité du concept de donnée transmissible. Ainsi, selon qu'il y a ou non risque de blocage, nous pourrons prévoir des solutions différentes. Un système d'aide à la conception et aux choix d'implantation serait une aide précieuse pour le concepteur.

Nous étudierons d'abord, l'implantation des données ne mettant en oeuvre que des connexions avec des modules d'une même tâche. Nous étudierons ensuite les problèmes d'implantation des données transmissibles entre modules de tâches différentes selon qu'elles sont consultées, consommées, etc.
Enfin, nous traiterons la façon dont les entrées sorties industrielles classiques sont réalisées. Nous verrons que nous nous ramenons simplement à un cas particulier d'implantation de données transmissibles.

Avant d'étudier toute implantation, des vérifications syntaxiques sont faites quant à l'écriture des déclarations de connexion.

Nous rappelons ici les formes possibles d'écriture des données transmissibles et des déclarations de connexion.

(1) < sortie > de < module > → < donnée transmissible >

(2) < sortie > de < module d'une même tâche > → < entrée > de < module d'une même tâche >

(3) < donnée transmissible > → < entrée > de < module > { avec / sans } consommation

{ ancien / récent } exemplaire

Certaines erreurs seront détectées si on connaît les sutrctures de contrôle de niveau 2 qui mettent en oeuvre les mêmes modules que ceux concernés par les connexions.

Exemple

Si une connexion est de la forme (2), si les modules connectés ne sont pas dans la même tâche, il y a une erreur.

On vérifie que les noms des entrées et des sorties existent bien. Si les attributs ont été déclarés dans les modules, on vérifie que les connexions les respectent. On n'acceptera pas qu'une vitesse soit connectée à une intensité.

On s'assurera, chaque fois que possible, qu'une production précède la consommation. Tous ces contrôles sont du ressort du système d'aide à la mise au point ou à la conception. La construction du schéma de contrôle, au fur et à mesure des spécifications, permet ces vérifications.

6.2 DONNEES TRANSMISSIBLES ENTRE MODULES D'UNE MEME TACHE

Nous étudions ici l'implantation d'une connexion :

sortie de M1 → entrée de M2 ;

M1 et M2 appartiennt à la même tâche.

6.21 Cas où M1 et M2 sont sur le même site

1ère solution.

La sortie est considérée comme une définition externe, l'entrée comme une référence externe. C'est une édition de liens classique , soit sur le site, soit croisée qui résoud le problème en affectant une même adresse en mémoire du site pour l'emplacement commun. Le module M1 y dispose la valeur produite. C'est cette dernière valeur qui sera utilisée en entrée par le module M2.

Ainsi dans le module M1, on devra trouver :

DEF alpha

Commentaire : alpha est le nom commun pour sortie et pour entrée dans M2.

La production de sortie se traduira par :

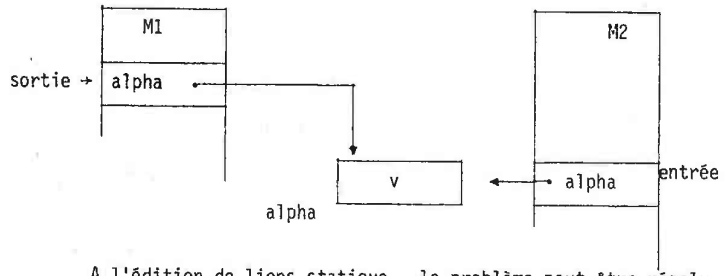
Rangement → alpha

Dans le module M2, on devra trouver : REF alpha.

L'utilisation de l'entrée sera assurée par toute instruction faisant référence à l'adresse effective alpha. Cette solution est la plus simple mais aussi la plus figée. Elle oblige à donner un même nom à deux objets qui normalement en ont des différents.

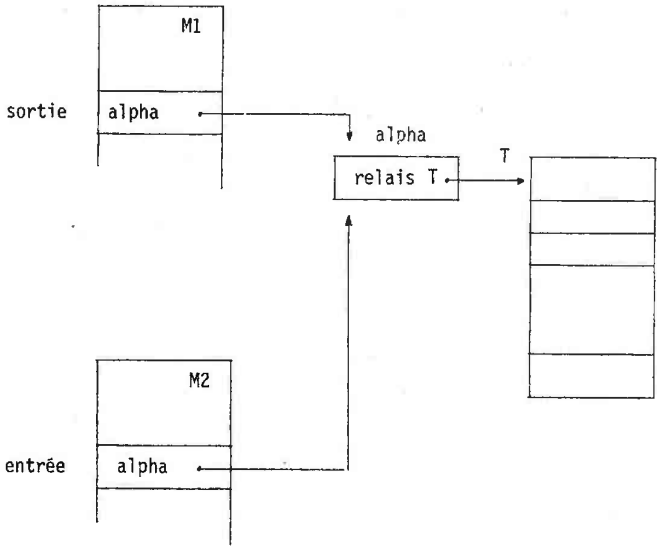
2ème solution.

L'objet local "entrée" et l'objet local "sortie" sont tous deux des pointeurs vers l'emplacement commun "alpha".



A l'édition de liens statique , le problème peut être résolu, mais aussi de façon relativement simple à l'exécution. En liaison avec l'interpréteur des structures de contrôle de niveau 2, les pointeurs sont initialisés au moment de l'activation du module. Ils peuvent d'ailleurs ne l'être qu'une fois au moment de la première exécution.

Pour traiter facilement les tableaux en données transmissibles sur Solar 16, la solution utilisée est voisine de celle-ci : alpha est l'adresse du relais. On a ainsi un double adressage indirect pour accéder à un élément.



6.22 Cas où M1 et M2 sont implantés sur deux machines sans mémoire commune

La seule solution possible est la recopie de la valeur de sortie dans l'élément d'entrée.
Deux versions peuvent toutefois être envisagées.

1ère version.

Au moment où l'interpréteur de structures de niveau 2 local au module M2 décide de lancer ce module, il s'enquiert des données d'entrée selon un algorithme voisin de celui ci-dessous.

1. Requête vers site de M1 demandant l'entrée
2. Attente réception
3. Quand toutes les entrées sont disponibles exécuter M2.

2ème version.

Dès que la sortie est disponible, elle est expédiée au site qui l'utilise en entrée.

Cette deuxième version sera plutôt utilisée quand il n'y a pas de risque de blocage puisqu'il n'y a aucune stratégie d'allocation de ressources à appliquer.

En revanche, si des blocages peuvent survenir, on préférera la première version car l'algorithme d'allocation se situera aux niveaux 1 - 2 si on fait de la prévision de blocage et au niveau 3 si on détecte des interblocages quand ils se sont produits.

Localement à chacun des sites, la première ou la deuxième solution est toujours applicable.

Nous n'avons fait aucune hypothèse sur le réseau de transport. On suppose seulement qu'un protocole ce transport est chargé d'assurer la transmission d'information entre un émetteur et un récepteur. Emetteurs et récepteurs sont les moniteurs locaux de gestion des données transmissibles.

6.23 Cas des multiprocesseurs

Par multiprocesseur, on entend un matériel capable d'exécuter plusieurs instructions à la fois. C'est à dire que nous ne faisons pas d'hypothèses sur le type de multiprocesseur - que ce soit un SIMD ou MISD ou MIMD d'après la classification usuelle Nous faisons seulement l'hypothèse qu'un multiprocesseur possède une mémoire accessible par les différents processeurs (c'est la mémoire commune). S'il n'y a pas de mémoire commune, c'est un réseau. Dans ce cas, les trois solutions (1ère et 2ème du § 6.21 et solution du § 6.22) sont applicables.

Si la donnée transmissible réside en permanence dans la mémoire commune, on appliquera l'une des deux premières solutions. Les processeurs accèderont toujours à la mémoire commune. La troisième solution est intéressante quand chaque processeur possède une mémoire privée en plus des accès à la mémoire commune comme dans l'architecture de NARCIS [LAGIER - 76],[VERNEL - 77]. Une copie des données transmissibles est assurée avant l'exécution du module entre la mémoire commune et la mémoire privée du processeur concerné. Ce dernier peut ainsi exécuter le module indépendamment des autres. De plus, on verra que dans le cas de partage de données, l'implantation centralisée des données simplifie considérablement leur allocation.

6.24 Autres connexions

Les cas des autres connexions sont :

a) Une même sortie est connectée à plusieurs entrées
   sortie de M1 → entrée de M2
   sortie de M1 → entrée de M3

b) Plusieurs sorties sont connectées à la même entrée
   S_0 de Init → entrée de M
   S de M → entrée de M

Le cas a) se rencontre dans des décompositions de définition conditionnelle ou dans des spécifications de modules parallèles.

Le cas b) se rencontre soit en sortie de conditionnelle, soit dans le cas de décomposition d'itération.

Dans la cas a), il y a duplication d'une sortie vers deux entrées (ou plus) de modules appartenant à la même tâche. Les divers cas où se rencontre cette duplication sont discriminés par les structures de contrôle de niveau 2.

Exemple 1. M1 - (M2 // M3)

Exemple 2. M1 - si Cond alors M3
sinon M2 fsi.

Selon l'implantation des modules M2 et M3 dans le premier exemple, les solutions déjà vues pour implanter les connexions seront ou non applicables. En effet, nous avons considéré que les objets en entrée et en sortie étaient locaux au module où ils étaient déclarés.

Dans l'exemple 1, ci-dessus, n'importe quelle implantation peut être utilisée pour la connexion entre M1 d'une part, M2 et M3 d'autre part, tant qu'une entrée de M2 et M3 n'est pas sortie dans un des modules.

Nous avons dit que les modules parallèles, dans une tâche, étaient sans concurrence donc ils ne pouvaient pas partager une même variable d'état. On pourrait toutefois accepter que l'un de ces modules modifie une entrée pour en évaluer une sortie (par exemple un comptage du nombre d'exécutions de la partie parallèle). Dans ce cas s'il n'y a pas de duplication, un tel module devra être exécuté en dernier afin d'assurer que tous les modules auront bien le même exemplaire en entrée.

Dans le cas b), il n'y a aucune ambiguité de par la structure de niveau 2. On sait, au moment d'exécuter M, si l'entrée est en provenance de Init ou si c'est la sortie déjà évaluée de M.

6.25 Conclusion

Nous avons donc ici la possibilité d'implanter, sans se préoccuper de synchronisation, la communication de modules non concurrents sur différents matériels. Les solutions qui ont été ici présentées ne sont pas toutes générales. Il est sûr que la seule solution vraiment générale soit la duplication ou recopie entre sortie et entrée. Ainsi, même en cas de changement de site d'un des deux modules concernés, la solution choisie sera toujours vraie. Le fait de garder toutes les possibilités pour des phases ultérieures de l'application est important dans le cadre de reconfiguration dynamique de l'implantation pour faire face aux pannes auxquelles le système devrait survivre. Il faudrait également envisager le problème de performance : la solution la plus générale n'étant pas la plus rapide.

6.3 IMPLANTATION DE DONNEES TRANSMISSIBLES ENTRE TACHES

Nous avons à résoudre ici un problème de synchronisation à partir des spécifications de consommation, consultation, production précisées dans les déclarations de connexion.

Les principales solutions sont :

- conservation d'une donnée dans un tampon d'une place
- conservation d'une donnée dans un tampon de N places
- conservation centralisée sur un site unique
- conservation répartie sur plusieurs sites
- liaison bi point avec transmission série ou parallèle.

Nous allons étudier quelques unes de ces solutions en regardant ce que signifie : production, consommation, consultation.

Nous décrirons certaines de ces implantations en termes de sémaphores de Dijkstra ou en termes de moniteurs de Hoare.

6.31 Tampon à une place

Une donnée transmissible est implantée sous forme d'un tampon à une place, c'est à dire à un exemplaire.
Une requête de production se traduit par :

si buffer libre alors début buffer:=sortie,buffer libre=faux fin
sinon attente

La condition "buffer libre" dépend des connexions avec les entrées.

- si les seules connexions sont des consommations du plus ancien exemplaire, la condition buffer libre est consommation terminée.

Dès que la donnée a été allouée en entrée d'un module consommateur, une production peut avoir lieu.

- si les seules connexions sont des consultations du plus récent exemplaire, la condition buffer libre est toujours vraie. Il ne faut pas la remettre à faux.

Toutefois certaines de ces séquences sont indivisibles, il ne peut effectivement y avoir de production et de consultation simultanée d'un même élément. Il y a exclusion mutuelle entre producteur et consultant. Un sémaphore accès libre peut être utilisé.

```
Production (sortie)
    si première production alors  consultation autorisé ;
       P (accès libre)
              buffer:=sortie
       V (accès libre)
    fin de production.

Consultation (entrée)
si consultation autorisée  alors
       P (accès libre)
              entrée:=buffer
       V (accès libre).
```

Cette formulation est très restrictive car les consultants eux-mêmes s'exécutent en exclusion mutuelle, alors que ce n'est pas nécessaire.

On autorise les productions et les consultations dans n'importe quel ordre dès qu'une production a eu lieu.

Dans le cas où il n'y a que des consommations du plus récent exemplaire, il ne peut y avoir qu'une consommation entre deux productions mais plusieurs productions entre deux consommations.

Dans le cas où il n'y a que des consommations du plus ancien exemplaire, il faut alors bloquer les productions tant que la consommation n'a pas eu lieu.

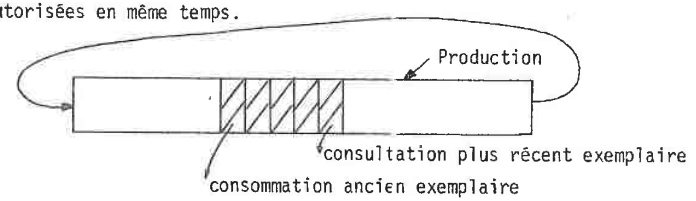
```
Production (sortie)
        P (buffer libre)
            P (mutex)
                buffer:=sortie
            V (mutex)
        V (buffer occupé)

Consommation (entrée)
        P (buffer occupé)
            P (mutex)
                entrée:=buffer
            V (mutex)
        V (buffer libre)

Initialisations  buffer libre = 1
                 mutex = 1
                 buffer occupé = 0
```

De telles solutions ne sont envisageables que lorsqu'il n'y a pas de risques d'interblocage. Il n'y a pas au moins deux modules demandant en entrée les mêmes données consommables. Si chacun des modules était libre d'exécuter lui-même les primitives, il faudrait tenir compte, à la compilation, des déclarations de connexion, afin d'engendrer les séquences dans le même ordre, ou utiliser des sémaphores généralisés [PATIL - 71].

On préférera plutôt utiliser un module de gestion des données qui effectuera les contrôles pour les modules demandeurs [NONN - 78b]. Ces algorithmes sont lourds mais permettent de prévenir l'interblocage.

6.32 Tampon à N places

Un tampon à N places peut être utilisé pour implanter une donnée transmissible. Les opérations de production, consultation, consommation peuvent être réalisées sous certaines conditions, être autorisées en même temps.



- Production

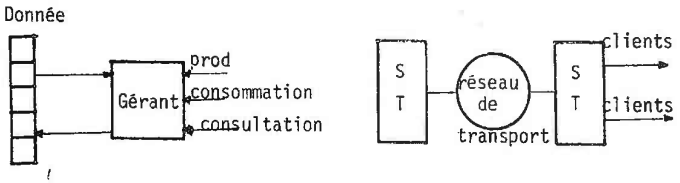
```
    P (buffer vide)
        P (mutex 1)
      buffer (i):=sortie
      i:=i+1  mod n
        V (mutex 1)
    V (buffer plein)
```

- Consommation du plus ancien exemplaire

```
    P (buffer plein)
        P (mutex 2)
                entrée:=buffer (j), j:=j+1 mod n
        V (mutex 2)
    V (buffer vide)
```

avec les initialisations buffer vide = N
mutex 1 = 1
buffer plein = 0
mutex 2 = 1

Ces deux solutions (§ 6.31 et § 6.32) sont des solutions centralisatrices dans le sens où la donnée est fixée sur un seul site. Tous les utilisateurs de cette donnée font donc référence à son gérant pour y avoir accès. Les requêtes sont ordonnées par ce gérant qui est un serveur au sens de l'expression protocole "client-serveur" dans les réseaux généraux.



Dans le cas où il y a des riques d'interblocage, le gérant se double d'un algorithme de synchronisation pour l'allocation concertée des données, sources possibles de l'interblocage.

Ainsi une description en termes de moniteur est bien adaptée . Nous obtenons alors un moniteur analogue à celui du paragraphe 4 de [HOARE - 74] avec une procédure CONSULTE en plus. L'inconvénient de cette solution est l'exclusion mutuelle entre les consultants.

```
Gérant  Monitor
          begin  buffer : array 0..N-1 of exemplaire ;
                 non vide : condition
                 non plein : condition
        procedure  produit (exemplaire) ;
          begin  if Nb=N then non plein.wait ;
                 buffer (i):=exemplaire ;
                 i:=i+1  modulo N ;
                 Nb:=Nb+1 ;
                 non vide.signal
          end
```



```
        procedure  consomme (ex) ;
          begin   if Nb=0 then non vide.wait ;
                  ex:=buffer (j) ;
                  j:=j-1 modulo N ;
                  non plein.signal ;
          end
        procedure consulte (ex) ;
          begin  if Nb=0 then non vide.wait ;
                 ex:=buffer(i-1) ;
          end
```

Cette solution impose l'exclusion mutuelle entre les consultants éventuels alors que ce n'est pas nécessaire. En revanche, il faut assurer qu'un élément consommé ne soit plus consultable. Il faut aussi assurer qu'un élément en cours de production ne soit pas consommable tant que la production n'est pas terminée, et réciproquement.

Le problème est simplifié quand il y a recopie de la valeur de l'exemplaire dans l'objet local du module consultant ou consommant. En revanche, si l'objet n'est pas dupliqué, il faut assurer à l'utilisateur que sa valeur ne change pas entre le début et la fin du module ; en particulier que la valeur n'est pas "disparue".

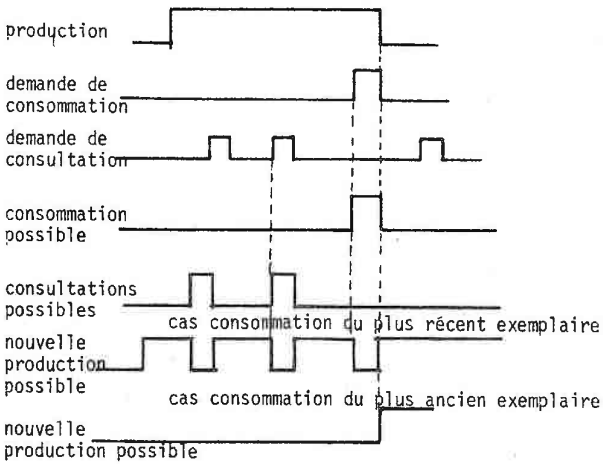
Ainsi, le système de gestion de la donnée devient-il beaucoup plus compliqué. On est alors amené à gérer les différents exemplaires d'une façon analogue à celle de [NONN - 78]. En effet, les exemplaires en consultation peuvent être en nombre égal au nombre de modules consultants. Il en est de même pour les exemplaires en consommation.

6.33 Liaison parallèle entre processeurs

Une connexion sortie de M1 → entrée de M2 est réalisée ici par une liaison parallèle entre les processeurs P1 et P2 sur lesquels les modules M1 et M2 sont respectivement implantés.

Il n'y a pas de mémoire au sens mémoire centrale de processeur mais il y a toujours un registre en sortie ou en entrée de la liaison. Supposont le en sortie.

On peut représenter les opérations de production de consommation et consultation par un diagramme de temps. On suppose que la taille en bits de la liaison parallèle est égale à la taille de l'information logique désignée par la connexion [AUBRY et Al - 79].



Les signaux de contrôle doivent être assurés par l'émetteur et le récepteur (deux signaux dans un sens pour une connexion sortie → entrée).

Une autre solution a été utilisée dans [AUBRY et Al - 80]. Les deux processeurs sont synchronisés par une même horloge physique ; les ordres de lecture, donc d'entrée ne sont exécutés qu'au N_L ème top de cette horloge et les ordres de sortie au N_S ème top de la même horloge, avec N_S N_L on est garanti qu'il n'y aura pas simultanéité des opérateurs production et consommation ou consultation.

Cette solution n'est bien sûr valable que lorsque l'unité de transfert est la même que celle de traitement. Dans le cas contraire à une production logique correspondront plusieurs productions physiques.
C'est le cas où plusieurs octets se succèdent en parallèle mais aussi celui d'une liaison série utilisée en mode synchrone ou en mode asynchrone. Ces deux techniques sont toujours des solutions au problème d'exclusion mutuelle entre production et consommation d'un même exemplaire.

6.34 Prospectives

Nous n'avons jusqu'ici envisagé que le problème d'implantation centralisée d'une seule donnée mais répartie des différentes données.
On pourrait envisager une répartition d'une même donnée transmissible N1 exemplaires sur un site S1
N2 exemplaires sur un site S2.

On peut également envisager une circulation des données entre les sites ou entre les sites intéressés.
Si tous les sites composent un anneau virtuel, on peut envisager une circulation des données transmissibles telle que les gérants de données mettent à jour les éléments au moment où ils passent dans leur site.
On peut aussi envisager une solution où les données tourneraient sur des anneaux virtuels propres. Un anneau virtuel propre à une donnée est l'anneau qui regroupe les sites possédant les modules concernés par cette donnée.

Nous ne pouvons pas dissocier le choix d'une solution d'implantation de celui d'allocation de ressources et de signalisation dans un système réparti.

Nous n'avons pas étudié spécialement ce problème. Nous avons toutefois prévu la possibilité d'une allocation en deux temps [THOMESSE et A1 - 78].
Les déclarations d'entrée et de sortie, les connexions définissent complétement les ressources nécessaires à l'exécution d'un module selon une méthode analogue aux sémaphores généralisés de Patil [1971]. Toutes les entrées sont nécessaires à la fois pour que le module puisse être activé. Si elles sont implantées sur des sites différents, il faudra synchroniser l'allocation de ressources entre ces divers sites. L'allocation en deux temps consiste en une réservation d'une ressource pour un module, réservation qui est confirmée et se transforme en allocation au cas où toutes les données sont réservées pour le même module. Dans le cas contraire (données réservées pour des modules différents) on est dans une situation d'interblocage ; il y a alors allocation d'office à un module prédéfini, c'est à dire que nous supposons que de façon statique, les modules concernés sont ordonnés a priori. Ceci est possible puisque nous connaissons la liste exhaustive, les modules connectés aux données transmissibles. L'algorithme n'a pas été complètement spécifié. Nous pensons que de nombreux travaux ont été faits et sont encore en cours sur ce sujet. L'algorithme de [DARGENT - 79] devrait pouvoir être simplifié à notre cas particulier de même que les différents algorithmes et méthodes connus et présentés dans [LE LANN - 79].

Une réalisation est en projet qui présenterait un algorithme d'allocations et ressources réparties, dans le cadre d'une spécification selon notre système et intégrant les dualités données - événements - conditions.

6.4 ENTREES-SORTIES INDUSTRIELLES

6.41 Introduction

Nous avons déjà vu que les entrées-sorties devaient être programmées en utilisant les deux instructions LIRE et ECRIRE. Or les opérations d'entrées-sorties ne sont pas synchrones avec le déroulement des modules en unité centrale. Cette programmation à l'intérieur des modules est donc contraire au fait que tout événement, toute synchronisation avec l'extérieur se traduit pas une tâche et une structure de contrôle de niveau 3. Les deux instructions LIRE et ECRIRE sont en fait des opérations réalisées par des tâches fonctionnelles différentes de celles où figurent les deux instructions. Ces tâches sont synchronisées d'une part, par les événements qu'on connaît habituellement comme les interruptions, les fins d'entrée-sortie, etc. et d'autre part, par les données concernées dans les échanges soit en entrée, soit en sortie. Les deux instructions LIRE et ECRIRE peuvent ainsi être interprétées comme des activations de tâche à l'intérieur d'un module. Or ceci n'existe pas dans notre système sous cette forme. Nous avons déjà dit (cf § 5.1) que pour garder une certaine homogénéité de description, les instructions d'entrée-sortie sont découplées desmodules où elles figurent.

Exemple

```
Module M
    Déclarations
        :
    Instructions I
    Instruction I
      LIRE        (Instruction I+1)
    Instruction I+2
        :
    Instruction N
fin de Module M
```

Le module est découpé en :

```
Module  M1
        :
        Instruction 1
            :
        Instruction I
fin du Module M1

Module  M2
        Instruction I+2
            :
        Instruction N
fin du Module M2
```

La structure de niveau 2 associée est M1 - M2.
Les synchronisations seront exprimées par les connexions de données.
L'instruction LIRE n'est plus programmée dans le corps du module.
Soit LECTURE la tâche qui réalise l'opération. On définit l'événement "Désir de LECTURE" comme étant la fin du Module M1.

- EVENEMENT : DESIR DE LECTURE : FIN DU MODULE M1

- SUR DESIR DE LECTURE FAIRE LECTURE
  avec CONSOMMATION
  ancien exemplaire

On peut aussi définir une sortie de M1 telle que LECTURE ait une entrée connectée en consommation (dualité événement - donnée transmissible).
Le module M2 a en entrée une donnée qui recevra une valeur en provenance de la tâche LECTURE.

Valeur lue de LECTURE → entrée de M2
avec consommation
récent ou ancien
exemplaire

Cette écriture correspond en fait à une lecture avec attente de la fin d'opération. En effet M2 ne pourra pas être exécuté tant que l'entrée ne sera pas pourvue. Si ce n'est pas le cas, ce n'est

pas à entrée de M2 que "valeur lue" sera connectée mais à une autre entrée d'un autre module, celui où la valeur est nécessaire.
Ainsi l'instruction classique "attente de fin d'entrée-sortie" rencontrée dans les systèmes temps réel courants est traduite par le découpage des modules et les connexions.

Nous avons ici traité le cas d'une lecture. Dans le cas d'une écriture, il en est de même. L'activation de la tâche ECRITURE sera plutôt décrite par la production par le module équivalent à M1 de la valeur à écrire plutôt que par un événement. Les deux méthodes sont équivalentes.

Il est intéressant de remarquer ici que la file d'attente des exemplaires d'une donnée transmissible permet d'exprimer simplement la synchronisation entre le production de données et la tâche ECRITURE consommatrice à son rythme, c'est à dire au rythme du périphérique associé.

Tout ce qui précède n'est pas propre aux entrées-sorties industrielles ; nous n'avons pas tenu compte des périphériques et de leurs particularités technologiques. Dans ce qui suit, nous allons étudier plus particulièrement des cas d'entrée-sortie industrielles, lecture et écriture numériques et analogiques. Les diverses synchronisations entre le périphérique et la tâche de lecture et /ou d'écriture seront étudiées également. Nous verrons que cette formulation aura de plus l'avantage d'être indépendante du fait que le périphérique est local ou non. Dans le cas des entrées-sorties industrielles ceci est très intéressant car les capteurs et actionneurs étant en général assez éloignés des organes de traitement, on peut être amené à utiliser divers moyens de transport pour assurer la gestion des Entrées-Sorties. Cet avantage est une conséquence directe de notre décomposition et de l'abstraction des outils d'implantation.

Nous supposons que l'opération de lecture ou d'écriture se situe entre deux modules M1 et M2 comme dans l'exemple ci-dessus. L'opération

de lecture a pour but de donner une valeur à l'entrée E de M2.

6.42 Lecture d'une entrée numérique

Le périphérique est supposé être constitué de n bits acquis en parallèle. Il est du type GPI (General Purpose Interface). Les signaux de synchronisation Request et ACK sont pris ou non en compte ; on obtient ainsi deux façons de réaliser les opérations d'entrée, avec ou sans synchronisation externe.

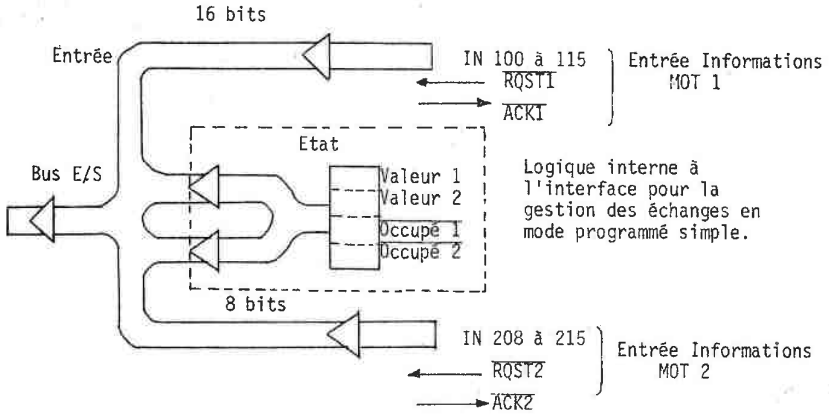


Figure tirée de [SEDS]

Il y a ainsi plusieurs cas selon qu'il s'agit de relever l'état instantané d'une entrée ou effectivement prélever un nombre déterminé de valeurs à des instants précis.

- Lecture d'une entrée sans synchronisation externe

```
MODULE  LIRE
  sortie    v
      lecture physique
    par instruction élémentaire
    propre au site d'implantation.
```

Cette lecture donne une valeur à v.

```
fin de MODULE  LIRE
```

La connexion v de LIRE → E de M2 permet de transmettre la valeur et synchroniser la tâche d'appartenance de M2.

L'instruction même d'entrée physique peut être décrite par une connexion.

Considérons le procédé industriel comme un producteur de donnée transmissible.

```
Module    procédé                  Module  LIRE
          sortie grandeur              entrée  valeur
          :                            sortie  v
                                           v:=valeur
                                   fin du module  LIRE
          grandeur de procédé ⟶    valeur de LIRE
                                   sans consommation
                                       récent exemplaire
```

- Lecture d'une entrée numérique avec synchronisation externe

Les modules sont les mêmes que dans le cas précédent. La connexion est : grandeur de procédé ⟶ valeur de LIRE
avec consommation

On peut préciser ancien exemplaire auquel cas on désire indiquer que le procédé ne peut pas afficher une nouvelle valeur tant que la précédente n'a pas été prélevée.

On peut également préciser récent exemplaire auquel cas on désire indiquer que le procédé peut modifier la valeur affichée à n'importe quel moment hormis les instants de prélèvement ; compte tenu des règles d'implantation des connexions, si le buffer n'a qu'une place, une connexion sur une entrée avec l'option récent exemplaire implique l'autorisation de perdre des exemplaires.

Les sous programmes Fortran temps réel réalisant ces fonctions sont DI et DIW (Digital Input et Digital Input avec attente) dont les séquences d'appel sont :

```
call DI (INV, JADR, KTAB, IERR)
call DIW(INV, JADR, KTAB, IERR)
```

les paramètres sont :

INV : nombre de mots à lire
JADR : tableau des adresses physiques des mots à lire
KTAB : tableau des valeurs lues
IERR : compte rendu.

On a déjà vu comment l'attente était spécifiée dans notre système (entrée d'un module).
Il nous reste à décrire comment plusieurs valeurs peuvent être lues et rangées. Plusieurs solutions s'offrent à nous. La plus logique consiste à associer une tâche à chaque périphérique d'adresse JADR (i). Nous obtenons ainsi les mêmes modules que dans l'exemple précédent auquel cas, le module M2 en attente des INV valeurs lues aura en entrée les KTAB (i) tels que :

```
grandeur 1 de procédé ⟶ KTAB(1) de M2
grandeur 2 de    "    ⟶ KTAB(2) de M2
                        :
grandeur n de procédé ⟶ KTAB(n) de M2
```

Nous faisons ici l'hypothèse que seul un registre mémorise une production. La lecture d'une entrée sans aucune synchronisation fait conduire à des lectures erronées. Si aucune information ne signale

la possibilité de lire (ou l'impossibilité), il faudra appliquer un algorithme de détection d'information stable comme celui de [LAMPORT - 77] ou deux / voire plusieurs lectures successives afin de s'assurer que la lecture n'est pas erronée. La fréquence de lecture devra être évidemment fonction de la fréquence de modification (au moins le double).

La lecture d'une entrée se fera plus souvent avec une synchronisation ; cette synchronisation est réalisée couramment avec un bit qui indique que l'information est présente et stable, bit positionné par le producteur. Ce bit correspond à l'indicateur "tampon non vide" dans une implantation logicielle. Ce bit est par exemple RQST dans la périphérie GPI [SEMS]. La consommation d'une entrée se traduit par la remise à zéro de ce bit "information disponible" sur signal du consommateur. Cette remise à zéro se fait par la producteur à la réception d'un signal "information consommée" (ACK dans le cas GPI).
La consultation d'une entrée se traduit par la non remise à zéro du signal "information disponible".

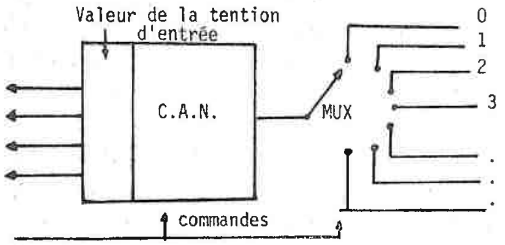
La consommation du plus récent exemplaire se traduit par l'autorisation au producteur de modifier éventuellement une information disponible, pour en afficher une autre. Une nouvelle production est autorisée alors que le tampon est plein. Il faut s'assurer que la consommation n'est pas simultanée. Le consommateur devrait donc signaler qu'il est en train de prélever l'information.

La consommation du plus ancien exemplaire se traduit par l'interdiction du producteur de modifier une entrée tant qu'elle n'a pas été consommée. On a donc une succession alternée de signaux RQST, ACK, RQST, ACK ...

6.43 Lecture d'une entrée analogique

Une chaîne analogique comprend en général un

multiplexeur qui partage le convertisseur analogique numérique entre plusieurs voies. Une lecture d'entrée analogique nécessite donc la commande préalable du multiplexeur par une sélection de la voie V concernée par la lecture. Le convertisseur lui-même peut être commandé (par exemple la résulution R).



Il est parfois possible de programmer un gain G pour spécifier en entrée du convertisseur analogique numérique, la gamme des tensions relatives à la voie concernée (0,5 V ou -5 +5 par exemple). Toute lecture analogique sera précédée, si c'est nécessaire, d'une commande (R,G,V).
Après une conversion, l'opération se ramène à une lecture numérique.

Module COMMANDE
  Sortie R,G,V
  G:= 0 → 5 volts
  R:= 12 bits + signe
  V:= 3
fin du Module COMMANDE

Module CANMUX
  Entrée COMG, COMR, NOVOIE
  Sortie Val
    mémorisation de COMG
                  COMR
    positionnement de MUX sur NOVOIE
    conversion
    Val:=
fin du Module CANMUX.

Les connexions seront :

G de COMMANDE → COMG de CANMUX sans consommation
R de COMMANDE → COMR de CANMUX sans consommation
V de COMMANDE → NOVOIE de CANMUX sans consommation
VAL de CANMUX → E de M2.

Le module M2 joue le même rôle que dans le cas "entrée numérique" (§6.42)

Une instruction de lecture de voie analogique qui contient habituellement tous les paramètres ci-dessus précisés :
LIRE (NOVOIE, G, R, Valeur)
sera donc implantée sous forme des deux modules COMMANDE et M2.
Le groupe CAN - MUX est considéré comme l'implantation matérielle du module CANMUX. La synchronisation peut être réalisée par la précision avec consommation, récent exemplaire pour la connexion VAL de CANMUX → E DE M2.
LECTURE sera en attente de production de valeur.
Si on désire exprimer la synchronisation sous forme d'événements, on peut le faire simplement en considérant l'événement "fin de conversion" qui peut être associé à la transition faux → vrai de la condition Donnée pleine. Nous retrouvons ici tout l'intérêt de la dualité événements, conditions, donnée transmissible car une programmation naturelle avec synchronisation exprimée par les connexions sera certainement implantée sous la forme sur fin de conversion faire LECTURE, avec la précision fin de CONVERSION = Interruption n°i bit j. La lecture est effectuée par exécution séquentielle du module

CANMUX. On peut vouloir synchroniser cette lecture sur un signal externe délivré par le procédé. Le module CANMUX sera décomposé en deux modules.
CANMUX 1 et CANMUX 2 qui composent chacun une tâche.

```
Module CANMUX 2
     Sortie VAL
       :
     conversion VAL:=
fin du Module CANMUX 2
```

L'exécution de CANMUX 2 sera régie par la structure de contrôle de niveau 3 suivante : sur SYNCHRO faire CANMUX 2.

On peut décrire d'une manière analogue les modules et les connexions correspondantes aux divers sous programmes AIRD, AISQ, etc. rencontrés habituellement dans les systèmes Fortran temps réel.
Les lectures successives peuvent être pilotées par une horloge interne au convertisseur ou par une horloge externe. L'expression sera toujours du type Sur Evénement faire Conversion
avec Evénement : tous les x unités de temps.
Ce n'est que l'implantation de l'événement et de la structure de niveau 3 qui change. Soit entièrement câblée après une initialisation de la période, soit ni câblée, ni programmée et commande de l'instant de conversion par l'unité centrale du site. Cette dernière solution risque de conduire à des périodes successives de durées différentes compte tenu de la charge de l'unité centrale. La solution entièrement câblée assurera toujours des périodes égales (à la précision de l'horloge). L'ensemble CAN, MUX, Horloge peut alors être considéré comme un processeur spécialisé non partagé entre plusieurs tâches.

6.44 Sorties numériques et analogiques

Nous ne considerons qu'un type de sortie. En effet, une sortie analogique n'est la plupart du temps qu'une sortie numérique dont la valeur courante sert de référence à un convertisseur

numérique analogique. Ces derniers étant habituellement de prix très peu élevés, ils ne sont pas partagés par un multiplexage quelconque. A chaque sortie analogique est associé un convertisseur numérique analogique.

Les sorties peuvent être schématisées comme suit. Un module producteur produit une valeur S, cette valeur doit être affichée sur une sortie numérique. La sortie numérique sera modélisée par un module sortie.

```
Module Sortie                    Module Producteur
     entrée v, synchro               sortie S
     affichage de v                    :
fin du Module Sortie                 S:=
                                 fin du Module Producteur
```

Les connexions sont : s de producteur -- v de sortie
L'entrée synchro de sortie est ou non utilisée selon que l'on désire effectuer la sortie dès que S est disponible ou non.

Remarque

Certains systèmes autorisent des sorties temporisées.

Exemple:sous programme DOW : la sortie reste à sa valeur pendant un certain temps, paramètre du sous programme, au bout duquel l'information peut reprendre un certain état. Ce genre de sous programme paraît très utile car agréable pour la programmation de certaines fonctions. Nous n'autorisons pas de telle programmation.

Nous sommes obligés de spécifier :

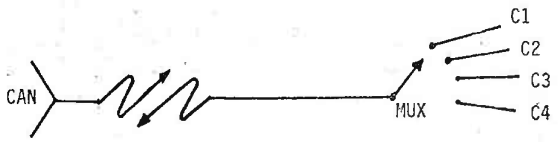
```
sur événement 1 faire sortie
sur événement 2 faire sortie
avec événement 2 :  Δt après événement 1.
```

Cette obligation bien que contraignante au premier abord est très utile pour vérifier statiquement la chronologie des événements

et l'enchaînement théorique des actions. Si toutes les synchronisations sont diffuses à l'intérieur même des sous programmes d'entrée-sortie, ces vérifications ne sont plus possibles.

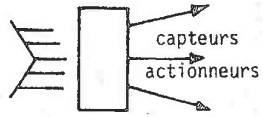
6.45 Implantation distante des entrées-sorties industrielles

Nous avons déjà signalé que dans de nombreux cas les capteurs et actionneurs sont situés loin de l'ordinateur "central". L'ordinateur est qualifié ici de "central" car c'est à lui que sont connectés les périphériques en question. Les coûts des câbles, des systèmes de protection contre les erreurs de transmission sont alors très importants. Certains systèmes classiques de multiplexage permettaient de partager une voie unique de transmission. C'était particulièrement bien adapté à la transmission de valeurs analogiques prélevées sur des capteurs géographiquement regroupés.



Nous avons supposé dans notre description que les entrées et sorties numériques ou analogiques étaient connectées directement avec un registre intermédiaire au site central. Ceci nous a permis d'illustrer les implantations câblées des opérations de consommation et de production, donc les implantations câblées des déclarations de connexion. Toutefois, la description que nous avons donnée des opérations d'entrée-sortie est indépendante de l'implantation et en particulier est bien adaptée à une implantation à distance. Tout ce que nous avons dit est seulement reporté

d'un site à un autre. Un protocole assure la gestion des messages. Un interpréteur des commandes reçues gère effectivement en local les opérations d'entrée-sortie.



Solution décentralisée

Exemple :

Les connexions

G de commande →

R de commande →

V de commande →

sont ici traduites par :

- construction du message comportant les valeurs de G, R, V et commande lecture
- émission vers le site ES
- interprétation de la commande de lecture
- lecture
- transmission des résultats vers le site "central".

6.46 Traitements associés aux entrées-sorties

Nous avons déjà dit que nous avions prévu quelques traitements pré-définis sous une forme voisine des formats de Fortran ou de Procol. Dans l'implantation de [COCHET-MUCHY - 78], des insertions de "sous programmes" ou "fonctions" sont autorisées dans les modules. C'est une façon d'implanter ces pseudo formats.

Il est certainement préférable de laisser l'utilisateur définir tous les traitements annexes, conversions, linéarisation, filtrage, détermination de nouveaux seuils, etc., libre de les spécifier où il le désire. Nous offrons en particulier la possibilité

de le faire dans la tâche contenant le module qui effectue effectivement l'opération. La synchronisation se reportera sur la connexion mettant en jeu la donnée prétraitée et non plus la donnée brute.

Quand tous les traitements possibles sont recensés, il est alors possible d'associer à chaque voie d'entrée et/ou de sortie, une table des traitements associés. La spécification des traitements choisis parmi le "menu" autorisé est alors du type "fill in the blank languages" [MUSSTOPF et Al - 79]. Cette forme de programmation est particulièrement bien adaptée à des exploitants non informaticiens [GERLL - 75] dans les cas rendus complexes parle nombre de voies aussi bien en entrée qu'en sortie.

6.5 CONCLUSION

Dans les chapitres 5 et 6, nous avons présenté quelques implantations possibles des éléments de notre langage. Les réalisations effectives sont essentiellement :

- Le compilateur des modules en langage intermédiaire [COCHET-MUCHY - 78]
- Des générateurs de code pour Solar 16 et Motorola 6800 [COCHET-MUCHY - 78]
- Un traducteur de modules écrits en Assembleur PB6 en langage machine PB6 [CAUMONT - 79]
- Un interpréteur de structure de contrôle de niveau 2 sur Solar 16 [NONN - 78]
- Un gérant de données transmissibles sous formes de gestion d'exemplaires a été décrit. Une partie a été réalisée dans une maquette [NONN - 78].

- Dans cette maquette constituée de Solar et Motorola 6800, les événements et structures de contrôle de niveau 3 étaient ceux du Solar 16 et du système RTES.
- Un mini réseau a été constitué entre Solar et Motorola 6800. Une procédure de transmission a été implantée, ainsi qu'un système multi tâches sur Motorola 6800 gérant l'enchaînement et la synchronisation de tâches et modules. Le contrôle était toutefois centralisé au niveau Solar [NONN - 78].
- L'application décrite dans [THOMESSE et Al - 79] a été spécifiée dans notre langage mais implantée "manuellement".

Il y aurait encore beaucoup à faire. Une autre réalisation est en cours, il s'agit d'implanter les événements et conditions en utilisant un algorithme de signalisation du type de celui de DARGENT [1979] qui peut être simplifié en n'envisageant pas l'allocation dynamique [CAUMONT - 79].

REFERENCES DU CHAPITRE 6

[AUBRY et Al - 79], [AUBRY et Al - 80]

[CAUMONT - 79], [COCHET-MUCHY - 78]

[DARGENT - 79]

[GERLL - 75]

[LAGIER - 76], [LE LANN - 79]

[MUSSTOPF - 79]

[NONN - 78]

[PATIL - 71]

[SEMS]

[THOMESSE et Al - 79]

[VERNEL - 77]

CHAPITRE 7

ELÉMENTS D'AIDE À LA CONCEPTION

7.1 INTRODUCTION

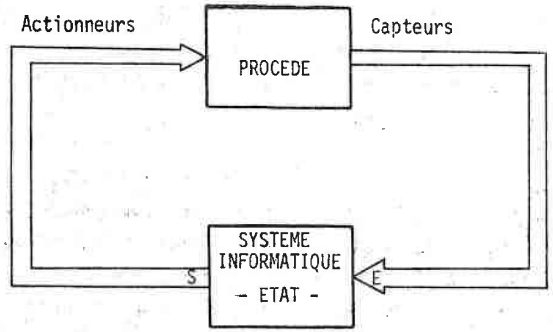
Dans les chapitres précédents, nous avons présenté comment une application pouvait être programmée selon divers éléments. Cette structuration nous a conduit à proposer plusieurs implantations possibles en fonction de divers critères. De plus nous avons introduit une équivalence entre les données transmissibles, les événements et les conditions, la synchronisation entre processus utilisant ces concepts étant spécifiée par le mode d'utilisation : consultation, consommation, ordre de l'exemplaire concerné.

Dans ce chapitre, nous voudrions donner quelques éléments qui guident le concepteur d'une application au moment de la conception. En effet, nous avions comme premier but de définir une décomposition d'applications en éléments qui permettent d'une part, la spécification de synchronisation d'une manière relativement naturelle et d'autre part, qui permettent une implantation répartie. Or, il nous est apparu qu'une programmation à partir d'éléments très précis était une programmation très structurée de telle sorte que, sans avoir la prétention de définir ici une méthode de programmation d'applications en contrôle de procédés ou plus généralement en temps réel, nous voudrions définir quelques règles qui définissent certaines décompositions et certaines compositions. Ces règles permettront la définition des divers éléments de l'application :

- Modules
- Tâches
- Evénement - condition
- Données transmissibles.

7.2 APPROCHE PAR LES EVENEMENTS - APPROCHE PAR L'ETAT

Nous pouvons schématiser très simplement l'ensemble procédé industriel, système informatique par la figure ci-dessous.



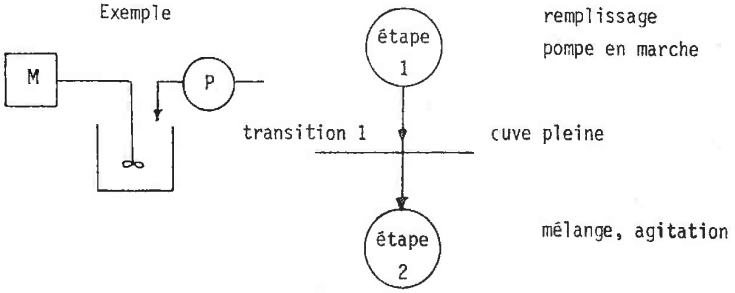
Les capteurs et les moyens de dialogue homme-machine sont des entrées du système informatique.
Nous aurons donc en entrée les valeurs des grandeurs mesurées, les changements de ces valeurs donc certains événements, les messages de l'opérateur. Le système informatique a pour tâche l'élaboration des commandes à envoyer aux actionneurs, des messages pour l'opérateur, l'édition des journaux ...
Le système informatique peut-être vu comme une tâche unique qui examine les entrées, détermine les traitements à exécuter, les exécute, envoie les commandes vers les actionneurs. Cette vue correspond à certains systèmes comme les automates programmables àù une tâche monolithique assure l'ensemble des fonctions. Les principaux inconvénients sont ceux des systèmes non modulaires :

- difficulté de mise au point
- quasi-impossibilité de prouver la moindre exactitude de la solution .

Le principal avantage est que, comme il n'y a aucune tâche, il n'y a plus de problèmes de concurrence entre tâches parallèles. Un sérieux effort de formulation pour la synthèse des systèmes séquentiels a été fait ces dernières années [BLANCHARD - 78] , [PRUNET - 78], [PLEYBER et Al - 77], [TACONET - 78]; avec les travaux sur les réseaux de Pétri, ces travaux ont débouchés sur la proposition du Grafcet comme méthode de formalisation du cahier des charges puis comme outil de conception de système [BOSSY et Al - 79]. Avec cet outil, une première forme de décomposition apparaît : les composantes connexes d'un même graphe correspondent à des "tâches différentes". Mais le graphe intégre toujours les traitements et les événements. Il faut toutefois souligner que la notion de traitement n'est pas exactement la même que dans un module selon notre définition. En effet, le grafcet indique d'une part des transitions et d'autre part des étapes. Une transition étant validée, (au sens de la validation d'une transition dans les réseaux de Pétri), les étapes qui la précèdent sont désactivées, celles qui la suivent sont alors activées.

Une transition est validée lorsque toutes les étapes immédiatement précédentes sont actives. Elle est franchie quand la condition logique associée est vraie. Ce franchissement entraîne la désactivation de toutes les étapes précédentes et l'activation de toutes les étapes qui suivent la transition, quelque soit le nombre d'étapes précédentes et suivantes.

Exemple



Une étape correspond à un état stable du procédé à commander. A une étape correspondent en fait deux traitements :

- un premier qui est l'activation de l'étape, il faut donc appliquer les commandes qui amènent le procédé dans l'état voulu. Les actions associées à l'étape sont exécutées.

- un second qui est la désactivation de l'étape.

Dans notre optique, nous définissons deux tâches, une de mise en marche, l'autre d'arrêt. Deux événements seraient générateurs des activations de ces tâches.

CUVE VIDE : Transition FV entrée numérique n° i
CUVE PLEINE : Transition FV entrée numérique n° j

sur CUVE VIDE faire mise en route pompe
sur CUVE PLEINE faire arrêt et mise en route mélange

Sur cet exemple, nous allons essayer de présenter une démarche possible. Ainsi, nous spécifions dans notre méthode toutes les actions mises en route, et arrêt alors que dans le graphe on indique les états, l'activation de l'étape 1 est la mise en route de la pompe, sa désactivation, son arrêt. Si on désire mémoriser l'état, on introduira une variable booléenne ETAT POMPE qui à 1 indiquera la marche et 0 l'arrêt. Cette variable devra être remise à zéro dans la tâche "arrêt" et remise à 1 dans la tâche "mise en route pompe". Cette variable sera donc en entrée et en sortie des deux tâches et des déclarations de connexion les relieront. L'implantation de cette donnée transmissible devrait prévoir une exclusion mutuelle si on ne tient pas compte des liens existant entre les événements. Or nous savons qu' en marche normale, les deux événements ne peuvent pas se produire en même temps. Donc l'exclusion mutuelle n'a pas à être implantée.

On pourrait penser que tout cela est bien compliqué pour implanter une variable d'état. Cela est vrai quand la variable d'état est

inconnue a priori. Alors, si on désire implanter uniquement une tâche qui gère cette variable, on écrira :

sur EVT faire gestion pompe
avec EVT : cuve pleine ou cuve vide

cuve vide et cuve pleine sont définies comme précédemment dans "gestion pompe" on définira la variable ETAT POMPE qui sera une variable d'état de la nouvelle tâche.
Notons que tous les événements sont pris avec consommation puisque nous tenons à prendre toutes leurs occurrences en compte.

Cette solution a toutefois l'inconvénient de prendre éventuellment en compte deux événements consécutifs identiques deux fois "cuve pleine" ou deux fois "cuve vide". Les tests adéquats devront être précisés dans la tâche.
Ce genre d'inconvénient n'apparaît pas dans la description avec le grafcet. En effet, une étape étant désactivée, si une occurrence d'un événement la désactivant quand elle est active, apparaît, cette occurrence n'est pas prise en compte. Ceci grâce au fait qu'on s'attache d'abord à définir les états plutôt que les événements. Si on n'est pas dans un état donné, certaines entrées ne sont pas prises en compte. Notre système de définition d'ordre des événements permet de pallier cet inconvénient.
En revanche, notre système permet toutes les combinaisons (entrées, état) puisque tous les événements peuvent être pris en compte quel que soit l'état du système. Grâce aux conditions, il est alors possible d'exprimer des tests sur l'état du procédé avant même d'activer des tâches.
Nous voyons alors tout l'intérêt de cette description car nous pouvons détecter des occurrences d'événement à n'importe quel moment et prévoir des sécurités par l'intermédiaire de tâches spécialement activées quand le système est dans un certain état.

sur cuve pleine si pompe en marche faire gestion pompe
sur cuve vide si pompe arrêtée faire gestion pompe

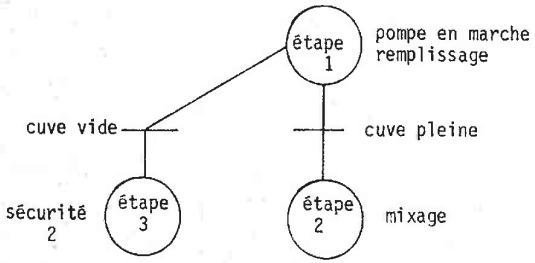
ici cuve pleine et cuve vide sont des événements dont la durée de vie

est nulle, c'est à dire qu'ils doivent être pris en compte tout de suite, si ce n'était pas le cas, l'événement pourrait être pris en compte tardivement alors que l'état n'est plus vrai.

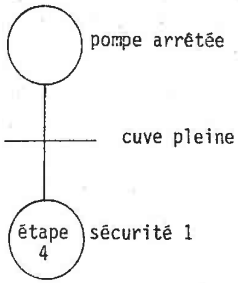
Ainsi, on peut donc prévoir d'autres tâches.

sur cuve pleine si pompe arrêtée faire sécurité 1
sur cuve vide si pompe en marche faire sécurité 2

Cette deuxième structure revient à envisager dans la construction du graphe les transitions suivantes :



La première sécurité se traduirait dans le graphe par :



où l'étape pompe arrêtée représente toutes les autres étapes que l'étape 1.

Il nous semble donc que du point de vue spécification, indépendamment de toute question d'implantation, notre système permet aussi bien l'approche à partir des entrées et des événements qu'à

partir de l'état du système. Ces descriptions complètement séparées des événements nous assurent même de ne pas présupposer un ordre quelconque d'arrivée des événements ou un état privilégié pour la prise en compte de l'événement. Les ordres sont explicitement définis dans la déclaration des événements et les états dans les conditions.
Nous pensons alors que dans un système complet, le graphe de commande pourrait être construit (au moins en partie) à partir de nos spécifications. Le graphe deviendrait alors un outil d'analyse de la solution plutôt qu'un outil de synthèse. Dans un système intéractif d'aide à la conception, cette décomposition entre synthèse et analyse aura une tendance à s'estomper. Le système d'aide à la conception aurait essentiellement comme tâche de s'assurer que l'analyse est correcte au fur et à mesure de son avancement en guidant le concepteur dans sa démarche.

Du point de vue implantation, avec notre système, nous avons vu que de nombreuses solutions s'offrent à nous ; un graphe n'est pas encore un programme, il faut le décrire dans un langage de programmation. De nombreuses propositions ont déjà été faites. La plupart sont basées sur la séparation du contrôle et des actions comme [JAY - 79] MASC 16 développé par la SEMS sur Solar [SEMS]. Dans ce langage, on exprime d'une part les actions correspondantes aux étapes du graphe ; d'autre part, chaque transition est exprimée en trois temps, la condition de franchissement, l'étape ou les étapes préalables, l'étape ou les étapes suivantes. Cette technique a déjà été utilisée pour représenter les réseaux de Pétri.

Nous terminerons cette étude en remarquant que les implantations habituelles du grafcet sont des implantations de type synchrone sur monoprocesseur. On n'accepte pas de prendre en compte des événements comme nous le permettons dans notre système.
Ainsi, le concepteur d'une application peut-il au choix fonder son analyse sur une étude des entrées et des sorties ou sur une étude des états du procédé physique ? Nous avons à ce sujet montré un critère de recomposition de tâche.

Régle 1 - Si deux ou plusieurs tâches ont les mêmes variables d'état et sont les seules à les posséder, elles peuvent être regroupées en une seule tâche sans diminuer le degré de parallélisme. L'événement qui régit l'activation de cette nouvelle tâche est alors le ou inclusif de tous les événements qui régissaient l'activation des tâches initiales. En effet, elles devraient alors s'exécuter en exclusion mutuelle. De plus, le regroupement diminue le nombre de ressources à allouer pour l'exécution. Le temps d'attente peut ainsi être diminué. Ceci correspond au cas où les diverses tâches représentent des fonctions différentes d'une même sous machine.

C'est ainsi que l'on peut englober toutes les tâches en une seule. Le système informatique, même s'il a été conçu modulairement, est construit par la fusion de toutes les tâches.

Remarque : On rapprochera le concept de moniteur [HOARE - 74] de celui d'étape du grafcet et du module de gestion d'une variable d'état. Les procédures du moniteur sont les fonctions applicables au calcul d'un nouvel état à partir des entrées et de l'état antérieur. Le concept de moniteur est donc bien adapté à une programmation basée sur la gestion des états. L'exclusion mutuelle entre procédures assure que les variables d'état sont correctement gérées.

7.3 QUELQUES REGLES DE DECOUPAGE EN TACHES

Nous voudrions ici énoncer quelques règles strictes qui permettent la définition de nouvelles tâches. Nous verrons ensuite d'autres règles moins strictes dues à l'équivalence donnée transmissible - événement - condition.

Au début de la conception du système informatique, le concepteur peut commencer à définir les états du système ou les entrées ou les sorties.

Une approche selon la méthode déductive [PAIR - 75] consiste à définir les sorties S. Elles sont a priori toutes évaluées dans une même tâche initiale TI qui au fur et à mesure de la conception va être décomposée en tâches plus fines par une méthode voisine de la méthode par raffinements successifs.

Nous noterons toutefois que le concepteur peut déjà décomposer TI en plusieurs TI1, TI2, ... TIn s'il appréhende, a priori, le problème en sous problèmes.

Exemple : Soit un moteur à réguler en vitesse et en courant.

1ère démarche - On considère que l'ensemble ne forme qu'une tâche TI.

2ème démarche - On considère d'abord le problème en un 1er sous problème TI1 : régulation de courant et en un 2ème sous problème TI2 : régulation de vitesse.

Les liens entre eux seront exprimés soit a priori s'ils sont connus, soit a postériori.

Exemple : Les régulations étant en cascade, une sortie de TI2 sera une entrée de TI1. Une connexion peut ainsi être exprimée avant le détail des tâches.

Nous nous intéressons donc ici au raffinement d'une tâche TI quelconque. Cette tâche a pour fonction d évaluer un ensemble de sorties définies au préalable. On suppose donc connus tous les actionneurs, les messages à émettre ...
Au cours de la détermination des définitions successives, on peut être amené à introduite des attentes, des délais.

Exemple : Un moteur doit tourner d'abord à petite vitesse pendant un certain temps Δt puis à grande vitesse.

Nous définissons ces deux commandes PV petite vitesse et GV grande vitesse.

PV : si ordre de démarrer alors PV=1

GV : si PV depuis Δt alors GV = 1

Cette écriture est incorrecte compte tenu du fait que toute référence au temps ne peut être dans un module. Une écriture correcte nécessite une définition de tâche donc d'événement.

Evénement : mise à 1 de PV
EVT2 : Δt après mise à 1 de PV
sur EVT2 faire Grande Vitesse.

Règle 2 - Dès qu'une définition nécessite l'introduction du temps sous forme durée, heure, cette définition doit apparaître dans une nouvelle tâche dont l'événement d'activation sera fonction de cette expression du temps.

Le même problème aurait pu être posé autrement. La commande de la petite vitesse est maintenue jusqu'à ce que la vitesse soit égale à V1.

La définition de GV devient :

3 GV : si vitesse lue = V1 alors GV=1
2 Vitesse lue : mesure
1 V1 : donnée.

1, 2, 3 sont les numéros d'ordre des définitions. Une synchronisation aura lieu au niveau de la définition 2 : quand vitesse lue sera disponible, la définition 3 pourra être évaluée. C'est un problème de découpage de tâches en modules (cf § 7.4). Nous ne nous sommes pas préoccupés des remises à zéro de PV et GV.

Règle 3 - Dans les définitions des sorties, il faut indiquer les sorties du type changement d'état de celles du type valeur.
Une sortie du type valeur donne lieu à une seule définition.
Une sortie du type chargement donne lieu à deux définitions
- une première du type "mise à un "
- une seconde du type "mise à zéro".

Ainsi, dans l'exemple précédent, on doit définir :
PV à 1 : si ordre de démarrer faire PV = 1
PV à 0 : si grande vitesse ou arrêt faire PV = 0.

En effet, ce n'est pas en général, la même condition qui permet la mise à un ou la mise à zéro. De plus, elles peuvent être réparties dans deux tâches différentes à cause du côté séquentiel des opérations (cf § 7.2).

Règle 4 - Selon la démarche déductive, les conditions d'activation de la tâche T sont obtenues à partir des définitions des entrées. Soient E_i i=1 ...n les entrées d'un algorithme, EVT_i l'événement défini par changement de E_i pour i=1...n, la structure de contrôle de niveau 3 SUR EVT faire T est définie avec $EVT = \underset{i=1...n}{OU} EVT_i$.

En l'absence de conditions d'activation définies à partir de la logique même du problème, la structure la plus générale est déduite du fait qu'on a considéré les modules comme des automates. En l'absence de changement des entrées, l'état de l'automate et ses sorties sont invariantes. On n'a donc pas à les réévaluer. C'est donc en présence d'un changement qu'il faut réexécuter l'algorithme.

Règle 4bis - Dans le cas où le temps intervient dans les calculs de l'algorithme par l'intermédiaire de la valeur d'un intervalle Δt, dans une itération, cet intervalle est aussi une entrée de l'algorithme. L'événement EVT est donc défini par (1) $EVT = \underset{i=1...n}{OU} EVT_i$ OU tous les Δt.

Exemple : Dans la programmation d'une régulation à partir d'un algorithme de type P.I. (proportionnel - intégral) l'élaboration d'une commande U_n nécessite un calcul $\int_{t_{n-1}}^{t_n} f(t)dt$. Ce calcul sera exprimé par exemple : $\frac{f(t_n)+f(t_{n-1})}{2} \times \Delta t$

Au fur et à mesure de l'écoulement du temps, l'intégration doit être assurée. La suite U des sorties U_n est définie par :

U : structure itérative définissant "calcul U_n"
où "calcul U_n" est une sous table.
La structure itérative est celle de niveau 3 avec l'événement
$EVT = \underset{i=1...n}{OU} EVT_i$ ou tous les Δt.

Règle 4 bis (suite) -

Compte tenu du fait que les événements EVT_i devront être détectés par une scrutation périodique, compte tenu du fait que certains événements peuvent être associés à des entrées dont la valeur intervient dans les calculs, la structure de niveau 3 pourra être :

sur EVT faire T
avec EVT = tous les Δt (2).

Cette forme (2) simplifie considérablement la forme 1. En effet, si on désirait prendre en compte des événements à d'autres instants que tous les Δt, il aurait fallu à chaque activation de l'algorithme prendre la valeur courante de l'intervalle de temps depuis la dernière activation.

Les données d'un algorithme (au sens de la méthode déductive) sont - soit des mesures (des données prélevées sur un périphérique)
- soit des données déjà évaluées par d'autres programmes.

Quand nous avons dit qu'à chaque entrée, on pouvait associer un événement "changement d'entrée", il s'agit aussi bien des données du premier type que du second. Nous avons vu la dualité entre événement et donnée transmissible. Naturellement, certaines données apparaissent comme événements mais d'autres apparaissent plutôt comme des données effectives, mesures ou autres. Ce n'est pas dans ce cas le changement qui est significatif mais plutôt la valeur. Ceci a déjà été vu au chapitre 4 à propos de l'exemple du carrefour.
Avec toutes les réserves d'usage, quand on parle de ce qui est "naturel" ou qui ne l'est pas, on peut énoncer une règle qu'il faut prendre

comme point de départ d'une analyse avant d'appliquer les règles de dualité.

Règle 5 - A partir des entrées physiques, on déduira certainement les événements et les conditions d'activation des tâches. A partir des données de type "logicielles", on déduira plutôt un flux de données et on exprimera la synchronisation entre tâches grâce aux déclarations de connexion.

Ceci ne doit pas être pris comme une règle stricte sachant qu'il est toujours possible d'associer un événement à une donnée et réciproquement.

Diverses raisons peuvent conduire à transformer l'un en l'autre :

- implantation (par exemple implantation câblée de la détection d'un événement par un comparateur)
- souplesse ou non d'allocation de ressources.

Quand on considère des entrées par connexion avec des données transmissibles, on a une allocation statique au niveau du module mais dynamique au niveau de la tâche :

- lecture d'une entrée physique par une tâche annexe et remplacement d'une entrée par une connexion.

Compte tenu de la concurrence entre traitement et opérations d'entrée-sortie physiques, si des définitions dans une tâche désignent des mesures (par exemple les définitions i, j, k, avec $i < j < k$), elles pourront être implantées sous forme de trois tâches T_i, T_j, T_k. Ainsi la tâche initiale TI sera décomposée en modules séquentiels :

M1, M2, M3, M4 où
M1 contient les définitions 1 à i-1
M2 " " i+1 à j-1
M3 " " j+2 à k-1
M4 " " k+1 à N.

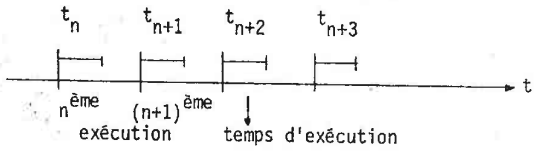
La synchronisation entre TI, T_i, T_j, T_k sera exprimée par les

connexions entre entrées et sorties de modules.

7.4 DECOUPAGE D'UNE TACHE EN MODULES

Nous venons de voir une première règle de découpage en modules pour profiter au maximum du parallélisme en affirmant les séquences parallèles à synchroniser. Nous augmentons ainsi le degré de parallélisme mais aussi les problèmes de synchronisation. Il aurait été possible de définir la synchronisation entre TI et T_i, T_j, T_k en début de TI auquel cas on avait pour T_i, T_j, T_k un parfait parallélisme possible mais aucun parallélisme avec TI.

Un module est un automate d'états finis. Les exécutions successives d'un module représentent les instants successifs d'application des équations de l'automate. Le temps est donc discrétisé selon les instants d'activation du module. On négligera ici le temps d'exécution.



Les sorties et l'état final sont considérés comme validés à l'instant t_n d'activation.

Ce qui est important à noter c'est qu'une sortie ne peut prendre qu'une valeur pour une activation donnée.

Règle 6 - Si pour une seule valeur d'entrée, un module doit parfois produire deux ou plusieurs valeurs en sortie, il sera nécessaire de décomposer ce module en plusieurs modules.

7.41 Exemple (tiré de [HOARE - 78]

Au travers de cet exemple, nous voudrions montrer

comment on peut utiliser quelques unes des règles précitées. Nous n'avons pas la prétention d'illustrer ici une méthode stricte. Toutefois, les contraintes d'écriture introduites par notre structuration impliquent un cheminement qui devrait tendre à devenir rigoureux.

Enoncé : On désire recopier, par un processus Est, des caractères émis par un processus Ouest. On remplacera deux astérisques consécutives par une flèche ↑.
Une hypothèse supplémentaire est faite : le dernier caractère d'entrée n'est pas une astérisque.

Solution

1°) Nous considérons trois tâches.

Module Ouest	Module SQUASH	Module Est
sortie caractère	entrée car 1	entrée caractère
⋮	sortie car 2	
	⋮	

2°) Les connexions sont :

caractère de Ouest → car 1 de SQUASH
avec consommation
ancien exemplaire

car 2 de SQUASH → caractère de Est
avec consommation
ancien exemplaire.

3°) Le module SQUASH a pour fonction de calculer car 2 en fonction de car 1 et de l'état courant de la chaîne de caractères d'entrée. Eventuellement, la sortie car 2 ne sera pas évaluée car le caractère d'entrée aura été reconnu comme une première astérisque. Auquel cas, il faut attendre l'exécution suivante pour connaître le traitement. Ainsi il sera introduit un retard dû à la mémorisation momentanée d'un caractère dans le module SQUASH. Ceci est dû au fait que nous avons considéré un module comme un automate d'état fini.

Une solution consiste à construire un deuxième module nommé SQUASH 2 dont la fonction est de sortir le caractère mémorisé quand on peut le sortir, c'est à dire quand il est différent de '*'

Module SQUASH 1
    entrée car 1
    sortie car 2, car memo
    état car mémo init vide

Module SQUASH 2
    entrée car mémo
    sortie car 3

Connexions
    car 2 de SQUASH 1 → carac 2
        carac 2→caractère de est
            avec consommation
            ancien exemplaire
    car 3 de SQUASH 2 → carac 2
    car mémo de SQUASH 1 → car mémo de SQUASH 2

Structure de contrôle de niveau 2

SQUASH : SQUASH 1 - si caractère mémorisé alors SQUASH 2

caractère mémorisé : CAR MEMO ≠ Astérisque.
caractère de ouest → carac 1
    carac 1 → car 1 de SQUASH 1
        avec consommation
        ancien exemplaire

Algorithme de SQUASH 1

car 2 : si car 1='*' et car mémo ='*' alors car 2='↑'
car mémo:
        car mémo = vide
    si car 1='*' et car mémo ≠ '*' alors car mémo='*'
        car 2 = vide
    si car 1≠'*' et car mémo = '*' alors car 2 = '*'
        car mémo = car 1
    si car 1≠'*' et car mémo ≠ '*' alors car 2 = car 1
        car memo = vide.

Algorithme de SQUASH 2
    car 3 = car mémo.

4°) Pour implanter cette solution, il reste à préciser comment sont réalisées les données transmissibles carac 1, carac 2 et car mémo.

Dès que nous avons plus d'une place dans le buffer associé à chaque donnée transmissible, nous avons la possibilité de parfait parallélisme entre les actions des trois tâches Ouest, SQUASH et Est.

5°) Si l'hypothèse "le dernier caractère est différent d'une astérisque" est levée, notre solution implique que cette astérisque sera mémorisée et jamais consommée par "Est". Aucune entrée supplémentaire ne sera tentée puisque nous supposons toujours que le module n'est activé que lorsque les entrées sont disponibles.

Notre système apparaît peut-être comme moins souple que celui de Hoare, mais en cas de panne, nous connaissons parfaitement l'état du système et les processus sont interrompus en des points connus dans un état connu.
Des considérations d'implantation doivent intervenir pour préciser ce qu'il advient de :

C=astérisk → west ? C quand il n'y a plus de caractères.

Pour prévoir le cas où la chaîne de caractères se termine par un nombre impair d'astérisques, il faut savoir déterminer la fin de la chaîne. Cette condition sera incluse dans la structure de contrôle de niveau 2 de la tâche SQUASH.

SQUASH : squash 1 - si caractère mémorisé alors SQUASH 2
caractère mémorisé : CAR MEMO ≠ Astérisque
ou
(CAR MEMO=Astérisque et dernier caractère)

Conclusion

Nous voyons ici un système où deux modules consécutifs ont chacun une sortie connectée à une même donnée transmissible. C'est une règle stricte dans notre structuration qui impose ces deux modules.

On ne peut en aucun cas les regrouper en une seule machine. Nous sommes ici limités par le fait qu'un module est un automate simple du type machine de Mealy à un vecteur d'entrée. Peut-être une généralisation à plusieurs vecteurs d'entrée pourrait-elle permettre plus de souplesse dans l'écriture d'un module ?

Il ne faudrait plus calculer seulement :

$Etat_{n+1} = f_1(Etat_n, Entrée_n)$

$Sortie_{n+1} = f_2(Etat_{n+1}, Entrée_n)$

mais autoriser un calcul comme :

$Etat_{n+k} = f'_1(Etat_n, Entrée_n, Entrée_{n+1} \ldots Entrée_{n+k-1})$

$Sortie_{n+1}, \ldots Sortie_{n+k} = f'_2(Etat_{n+1}, Etat_{n+2}, \ldots Entrée_{n+1 \ldots n+k-1})$

ce que nous autorisons seulement par l'expression des structures de contrôle de niveau 2.

Quel que soit le modèle de machine abstrait utilisé pour formaliser un système séquentiel [FELDMAN - 79], nous aurons ce problème puisqu'ils se ramènent toujours à définir :

- état
- entrée
- sortie
- mouvement de bandes
- un ensemble de transitions
  (état courant, entrées courantes) → action.

7.42 Découpage et composition de modules

Dans cette partie nous allons montrer comment les structures de niveau 2 peuvent être utilisées pour décomposer ou recomposer un module en ou à partir de plusieurs autres. Les raisons peuvent être : on est obligé de découper un module en deux parcequ'aucun processeur ne possède les ressources nécessaires à une exécution du module initial,

: on dispose d'un multiprocesseur, auquel cas il est intéressant de découper un module en modules parallèles pour profiter au maximum du parallélisme possible [SYRE - 76].

De nombreux algorithmes connus, dans divers domaines, ont déjà fait l'objet de modification pour être adaptés aux multiprocesseurs.

7.421 Décomposition séquentielle

Soit un module M_0 d'entrées E_0, de sorties S_0 et de variables d'état ST_0. Le début du corps du module (jusqu'à l'instruction i incluse) doit devenir un autre module M_1 et la fin un module M_2.

Module M_0

Entrée $\{E_0\}$
Sortie $\{S_0\}$
Etat $\{ST_0\}$

$\{E_0\}$, $\{S_0\}$, $\{ST_0\}$ représentent respectivement les listes d'entrée, de sortie, de variable d'état.

Instruction 1
⋮
Instruction i

devient le module M_1

Instruction i+1
⋮
Instruction n

devient le module M_2

fin du Module M_0

Les entrées de M_1 sont les entrées de M_0 utilisées dans les instructions 1 à i notées entrées (1...i). Les entrées de M_2 sont les entrées de M_0 utilisées dans les instructions i+1 à n ainsi que des variables internes évaluées entre I_1 et I_i et utilisées entre I_{i+1} et I_n. Ces variables internes de M_0 deviennent donc des sorties de M_1.

Les autres sorties de M_1 sont les sorties de M_0 évaluées entre I_1 et I_i. Les sorties de M_2 sont les sorties de M_0 qui ne sont pas évaluées entre I_1 et I_i.

Une sortie de M_0 qui serait évaluée à la fois dans la séquence de M_1 et dans la séquence de M_2 sera définie comme sortie de M_1, entrée de M_2 et sortie de M_2.

Les variables d'état de M_0 peuvent apparaître comme variable d'état de M_1 ou de M_2, mais aussi comme sortie de M_1, entrée de M_2, sorties de M_2

et entrée de M_1 d'une exécution à la suivante.
Les connexions à exprimer en plus sont celles qui relient les nouveaux modules soit dans le sens

sortie de M_1 → entrée de M_2

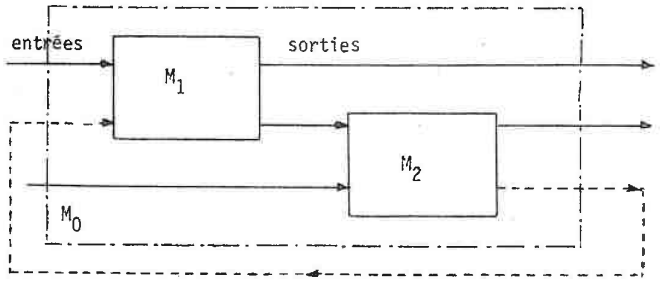
soit dans l'autre sens

sortie de M_2 → entrée de M_1

pour réaliser les variables d'état du module initial M_0.
Les connexions à modifier sont toutes celles qui faisaient intervenir un élément (entrée ou sortie) de M_0.
Selon qu'une entrée de M_0 est devenue entrée de M_1 ou entrée de M_2, une connexion à une de ces entrées de M_0, devient connexion à une entrée d'un des deux nouveaux modules. Il en est de même pour les sorties.

7.422 Composition séquentielle

Soient deux modules M_1 et M_2 enchaînés séquentiellement dans l'ordre M_1 - M_2. Ces deux modules peuvent être regroupés en un seul module M_0 qui contient l'enchaînement des instructions de M_1 puis de M_2.



Les entrées de M_0 sont : - les entrées de M_1 qui ne sont pas sorties de M_2
- les entrées de M_2 qui ne sont pas sorties de M_1.

Les variables d'état de M_0 sont celles de M_1 et de M_2 auxquelles on ajoute les variables qui correspondraient à un couple de connexions.

sortie de M_1 → entrée de M_2 ↓
évaluation d'une ↑ sortie de M_1 — évaluation d'une sortie de M_2
entrée ↑ de M_1 ← sortie de M_2 ↓

7.423 Composition parallèle

Soient deux modules M_1, M_2 liés par une structure de contrôle M_1 // M_2 au niveau 2. Ils sont à être recomposés pour former un seul module M_0.
Les entrées de M_0, les sorties de M_0, les variables de M_0 sont la réunion respectivement :

des entrées de M_1 et de M_2
des sorties de M_1 et de M_2
des variables d'état de M_1 et M_2.

7.424 Décomposition des conditionnelles et des itérations

Dans ce paragraphe, nous précisons comment les structures conditionnelles et itératives peuvent être retirées du corps des modules pour être placées au niveau 2. Les séquences d'instructions qui composent le corps de ces structures deviennent alors des modules dont nous allons préciser les entrées, sorties et variables d'état.
Ainsi par application d'une démarche déductive, une définition comme

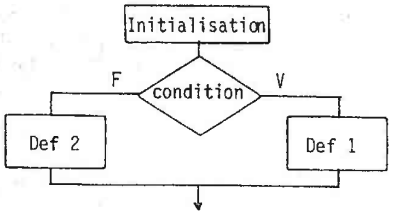
*2 def: si condition alors def 1 | def 1 { définissent def selon
sinon def 2 | def 2 { la valeur de condition
Init sous tables
1 condition=

ou *1 def: Init ; tant que condition faire traitement

peuvent être exprimées au niveau interne du module ou externe au niveau 2.
Il faudra exprimer les conditions.

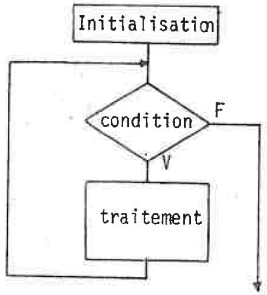
La solution est l'utilisation des données transmissibles. Les conditions peuvent être exprimées comme celles intervenant dans les structures de niveau 3.

a) Cas des conditionnelles



Module Initialisation
  sortie condition
    :
fin du Module Initialisation
connexion : condition de Initialisation→ COND
si COND alors Module Def 1
       sinon Module Def 2

b) Cas des itérations



Module Initialisation
  sortie condition
    :
fin du Module Initialisation

Module Traitement
  sortie cond
    :
fin du Module Traitement

Connexions :
condition de Initialisation → COND
cond de Traitement → COND

structure de niveau 2
Initialisation - tant que COND faire traitement ftq.

ENTREES ET SORTIES

Dans le cas des conditionnelles, les entrées des deux modules Def 1 et Def 2 sont les mêmes que celles du module initial. L'utilisation des données transmissibles permet le choix des connexions effectives à une exécution donnée.

Exemple :

Module M_0
  sortie i
si C alors i:=0
     sinon i:=1 fsi
i de M_0 → entrée de M'
i devient une sortie de M_1 et une sortie de M_2

Module M_1
  sortie i
    i:=0

Module M_2
  sortie i
    i:=1

structure de niveau 2
Si C alors M1 sinon M2

on doit connecter : i de M_1→I
i de M_2→I
I ⟶ entrée de M'
sans consommation récent
exemplaire

Des variables d'état d'un module M_0 peuvent être utilisées dans les deux modules M_1 et M_2. Elles deviennent des variables d'état de M_1 et de M_2 ; mais il s'agit en fait du même objet "global" à M_1 et M_2.

Exemple : si C alors i:=i+1
sinon i:=i-1 fsi

Cette structure exprimée au niveau 2 deviendra :

si C alors M_1
sinon M_2

avec Module M_1
entrée i
sortie i
⋮

Module M_2
entrée i
sortie i

i de M_1 ⟶ I
i de M_2 ⟶ I

I⟶ i de M_1
sans consommation récent exemplaire
I⟶ i de M_2
sans consommation récent exemplaire.

7.5 CONCLUSION

Nous avons vu dans ce chapitre quelques points qui aident le concepteur dans la construction d'une application. Nous ne pouvons pas appeler ceci une méthode de programmation. Cela est toutefois une approche vers une méthode de programmation d'applications en temps réel. Nous avons fait un parallèle avec une méthode de synthèse des systèmes séquentiels et avons montré que les deux pouvaient être complémentaires.

En ce qui concerne les découpages et réassemblages de modules,

il faudrait certainement apporter quelques précisions de façon à les rendre automatique dans un système de production de programme. Un tel système devrait maintenant être développé qui permette, relativement facilement et rapidement, des modifications de programme selon les "remarques" faites par le système par une analyse "en ligne" des spécifications données par le concepteur. C'est ici que la schématisation sous forme de réseaux de Pétri prend, à notre avis, tout son intérêt.

Nous avons présenté quelques moyens d'appliquer la méthode déductive à la programmation d'applications en temps réel. Elle peut d'ailleurs être combinée à la programmation dans le sens événement→action plutôt que dans le sens sortie→action→entrée→événement. Comme dans toute programmation, les deux démarches descendante et ascendante sont utilisables à tour de rôle. Ainsi, notre structuration permet d'assembler des modules existants à la manière dont on assemble les composants électroniques logiques sur une carte de circuit imprimé. Les spécifications sont en effet les mêmes, entrée, sortie, état. Ce qui est plus complexe est la spécification des fonctions assurées par le module ; il serait peut-être intéressant à ce niveau d'utiliser certaines spécifications développées à propos des types abstraits.

REFERENCES DU CHAPITRE 7

[BLANCHARD - 78], [BOSSY et Al - 79]

[HOARE - 74], [HOARE - 78]

[JAY - 79]

[PAIR - 75], [PLEYBER et Al - 77], [PRUNET - 74]

[TACONET - 78]

CONCLUSION

Dans ce travail nous avons essayé d'apporter une contribution à la conception et à la programmation d'applications en temps réel et réparties.

Nous avions comme objectifs

- une programmation des applications indépendante de l'implantation ;
- une programmation relativement naturelle permettant au concepteur l'expression des concepts tels qu'il peut les appréhender ;
- une programmation modulaire permettant d'assurer une certaine facilité de mise au point ;
- une spécification de synchronisation à partir de la communication.

1 - Pour atteindre ces objectifs nous avons défini une structuration, à partir de centres d'intérêt

+ modules et structure de contrôle de niveau 2 pour la programmation des algorithmes ;
+ événement et conditions avec les structures de contrôle de niveau 3 qui expriment la synchronisation entre les tâches fonctionnelles et le procédé industriel physique. Nous avons de plus défini cette possibilité entre tâches fonctionnelles ;
+ données transmissibles pour spécifier le flux de données aussi bien entre modules d'une même tâche qu'entre modules de tâches différentes, à partir duquel, une synchronisation est déduite.

Notre structuration a l'avantage de ne pas proposer d'ordre de définition des éléments. En particulier nous n'avons pas de hiérarchie comme dans les systèmes du type de JAMMEL [1977 - 1979] [SCHUTZ - 79]. Les seules relations qui existent à l'exécution sont du type producteur - consommateur ou producteur - consultant.

Nous avons ainsi proposé un sens différent au concept de modularité. La voie avait été ouverte par PARNAS [1972 a et b]. Sur le plan Nancéien, DERNIAME [1974] avait avec toute son équipe oeuvré en ce sens et a poursuivi ce travail avec l'introduction des types abstraits. Notre travail a l'avantage de présenter un autre point de vue. Il serait maintenant peut être profitable de les rapprocher.

2 - Notre structuration est indépendante de l'implantation. Nous avons esquissé un langage pour spécifier l'implantation des éléments qui composent une application. En ce sens notre projet présente un côté innovateur dans le sens où la communication est spécifiée indépendamment des autres synchronisations. Ceci rend effectivement la structuration indépendante de l'implantation. Elle présente l'avantage de régler les problèmes d'implantation les uns après les autres, un peu comme le fait le concepteur au moment de la description de sa solution.

Cette abstraction autorise toutes les démarches usuelles des automaticiens.

Une application conçue selon les règles de la commande hiérarchisée [TITLI - 75], [BINDER et Al - 78] peut être décrite dans notre système et être implantée sur n'importe quel ensemble de processeurs. Une application à contrôle centralisé pourrait être une application dans laquelle toutes les structures de contrôle, toutes les informations concernant la synchronisation seraient centralisées sur un seul site, les traitements pouvant être répartis sur des processeurs qui jouent alors le rôle de sous traitants du processeur central.

Nous n'avons pas complètement implanté les propositions contenues dans cette thèse. En particulier la synchronisation à partir d'événements et de conditions, ainsi que l'allocation de données réparties devraient compléter les premiers résultats obtenus. Comme nous ne voulions pas étudier d'algorithme de synchronisation pour un réseau général, car cela sortait de notre étude,

nous n'avons pas pu implanter ces points tant qu'un tel algorithme ne nous était pas fourni. Nous avons utilisé un cas particulier de réseau en étoile avec un contrôle centralisé pour la première maquette.

Des travaux en cours [CAUMONT] devraient étendre largement les résultats déjà obtenus. Il serait par ailleurs intéressant d'intégrer des préoccupations liées à la sûreté de fonctionnement dans nos implantations, notre opération de duplication permettant des sauvegardes de données et événements qui en cas de défaillance pourraient être utilisés pour la continuation de l'application.

3 - On distingue souvent en conduite de procédés, deux domaines d'application : la conduite de systèmes séquentiels, les régulations ou D.D.C. Des systèmes spécialisés ont été conçus pour l'un comme pour l'autre des domaines. La capacité d'exprimer des traitements algorithmiques comme celle d'exprimer des synchronisations temporelles fait que notre système est aussi bien adapté à l'un qu'à l'autre des domaines. Une lacune dans notre système est à signaler, l'absence de gestion de fichiers qui devrait être comblée pour en faire un système réel.
Par ailleurs, il manque une expérimentation en grandeur nature du type de celles décrites par KAISER [1978] ou ALTABER[1977]. Une telle expérience devrait permettre d'affiner et valider complètement les concepts que nous avons présenté.

De plus, le côté statique de la description peut apparaître comme lourd et contraignant pour des utilisateurs qui apprécient l'aptitude des systèmes à être modifiés "en ligne". Notre structuration autorise certaines modifications en ce qui concerne les événements et les structures de contrôle de niveau 3. C'est aussi un avantage car il assure une certaine sécurité en évitant les fautes de manipulation.

4 - Pour spécifier le parallélisme et la synchronisation entre les tâches et l'environnement, nous avons choisi de les décrire de façon externe aux algorithmes mais en utilisant les objets sur lesquels la synchronisation doit porter quand à la base il y a un problème de communication.

A ce sujet nous avons proposé un véritable langage de spécification des opérations à effectuer indépendamment de la procédure utilisée pour les exécuter. Notre langage apparaîtra, au moins pour ce sujet, à la fois un langage de spécification et un langage de programmation.

Nous nous situons donc à mi-chemin des deux écoles comme les CSP de HOARE [1978] à base de commandes gardées et les modules de contrôle de ROBERT et VERJUS [1977].

Notre système paraîtra lourd par l'introduction de modules et celle de structures de contrôle de niveau 2 comme dans l'exemple de HOARE traité au chapitre 7. Il présente l'avantage de spécifier tous les cas rendant ainsi relativement sûre l'expression de la solution.

Des règles simples permettent déjà de déterminer quelques cas (quand et comment) où il faut procéder à des décompositions (cf aussi le problème relatif au carrefour chapitre 4). D'autres règles restent à développer, qui formalisées, permettront a priori une vérification automatique de la correction de la solution proposée. De même, il faudrait formaliser les équivalences et les transformations événement - condition - donnée transmissible. Cette possibilité d'utiliser divers concepts pour exprimer les synchronisations que notre système offre à l'utilisateur est particulièrement importante. Le concepteur peut choisir celui qui lui apparaît le plus naturel. Ensuite, pour l'implantation, il peut transformer ses structures et/ou connexions pour adapter sa solution au matériel utilisé tel que :

logique câblée, microprocesseur, automates programmables et systèmes temps réel basés sur le concept d'événements. Il est important dans un système de disposer de moyens de description qui ne soient en aucune façon dépendants du matériel.
Les seules opérations de production, consultation, consommation et duplication permettent d'exprimer les problèmes classiques de synchronisation sur l'accès aux données et plus généralement aux ressources.

Dès qu'il s'agit plutôt de problème d'ordonnancement, cela ne suffit pas dans un premier temps, car les données ne sont pas naturelles. Nous avons introduit pour cette classe de problèmes, les conditions et les événements avec le langage de description d'événements. Il serait également possible de décrire la même langage à propos des conditions.

Ces possibilités distinguent notre système de la plupart des systèmes existants:- les systèmes type temps réel [GRIESNER - 80] où on privilégie habituellement les événements, voire les conditions sans se soucier de la communication,

les primitives SEND et RECEIVE résolvant la communication par émission - réception de messages. On remarque, à ce sujet, que le rendez-vous ou la mise en file de messages parfois distingués [CHEVANCE - 78] relèvent du même concept : donnée transmissible.

- les systèmes plutôt orientés systèmes d'exploitation dans lesquels les problèmes de communication sont plus étudiés au détriment de la spécification de synchronisation sur événements et conditions [CHEVAL et AI - 79], [CLEMENCET et AI - 79], [GUILLOT - 79] pour le système sortilège à Grenoble et BETOURNE et AI [1977] pour le système LEST développé à Toulouse.

Un point important qui n'a pas été traité ici concerne les preuves. Des preuves devraient pouvoir être faites à partir de la modélisation par les réseaux de Pétri [SCHNEIDER et AI - 80]. Ces preuves par simulation peuvent être onéreuses en temps de machine. Il serait certainement intéressant d'appliquer ou de développer un système de preuves adapté à une pure description des synchronisations [KELLER - 76], [GREIF - 77].

Le formalisme fondé sur la théorie des automates obligeant à exprimer certaines itérations au niveau 2 ou au niveau 3 devrait favoriser l'expression d'assertions de précédence et de terminaison d'exécution de modules puis de tâches. Les déclarations de connexion et l'état des données transmissibles définissent alors les conditions d'activation et de synchronisation.
Par ailleurs, il faudrait pouvoir comparer notre système à d'autres selon des critères plus quantitatifs que qualitatifs. BLOOM [1979] a proposé quelques critères qui ne sont encore quasiment que qualitatifs.

REFERENCES DE LA CONCLUSION

[ALTABER - 77]

[BETOURNE et Al - 77], [BINDER et Al - 78], [BLOOM - 79]

[CAUMONT], [CHEVAL et Al - 79], [CHEVANCE - 78],
[CLEMENCET et Al - 79]

[DERNIAME - 74]

[GREIF - 77], [GRIESNER - 80], [GUILLOT - 79]

[HOARE - 78]

[JAMMEL - 77], [JAMMEL - 79]

[KAISER - 78], [KELLER - 76]

[PARNAS - 72]

[ROBERT et Al - 77]

[SCHUTZ - 79], [SCHNEIDER et Al - 80]

[TITLI - 75]

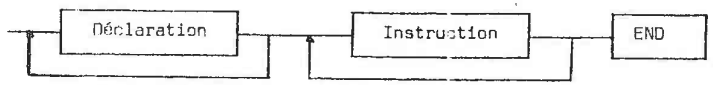
ANNEXES

ANNEXE 1

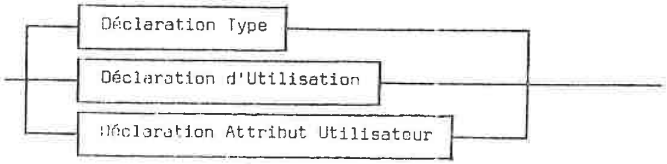
Tirée de [COCHET-MUCHY - 78].

1) La Syntaxe du langage source de Sygare

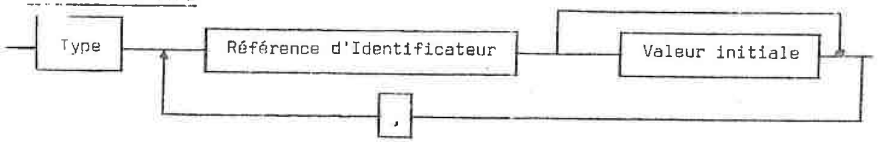
PROGRAMME



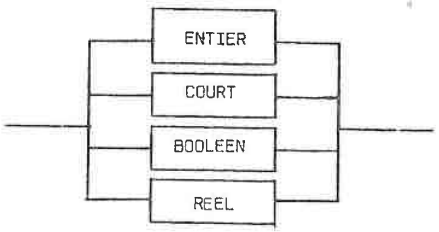
DECLARATION



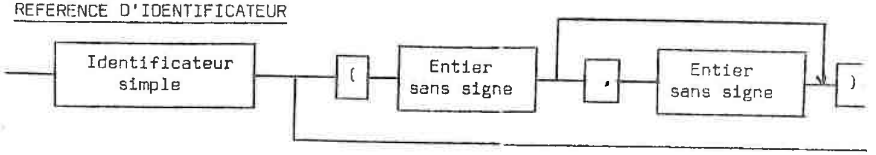
DECLARATION TYPE



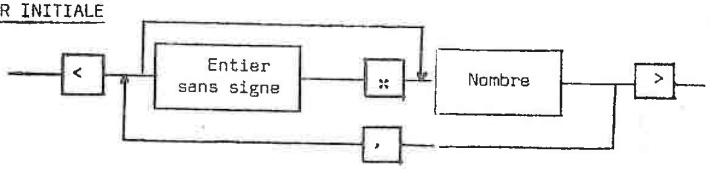
TYPE



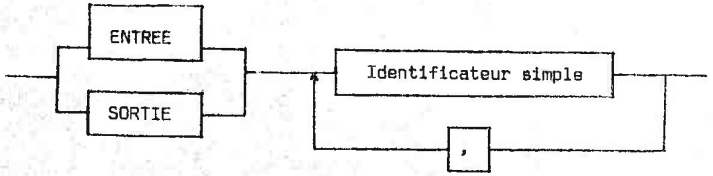
REFERENCE D'IDENTIFICATEUR



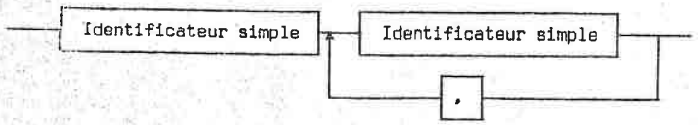
VALEUR INITIALE



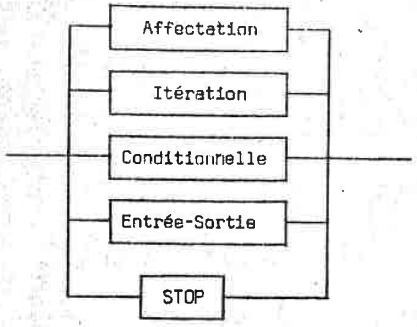
DECLARATION D'UTILISATION



DECLARATION D'ATTRIBUT UTILISATEUR

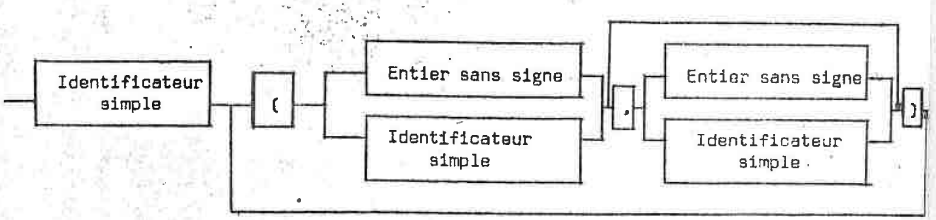


INSTRUCTION

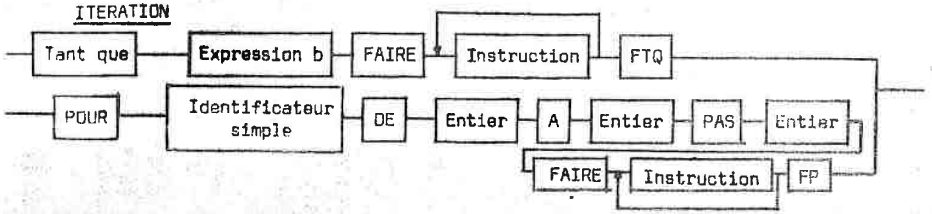


AFFECTATION

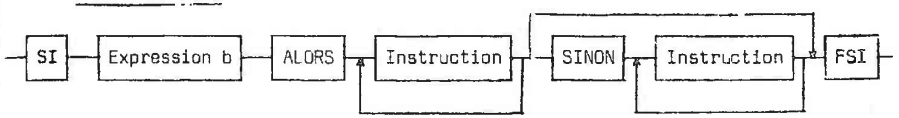
IDENTIFICATEUR



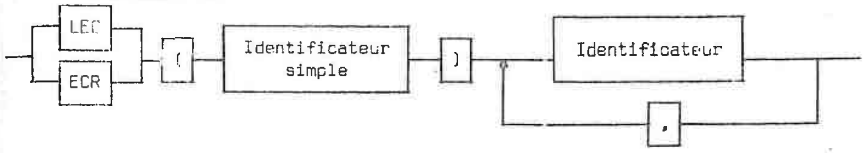
ITERATION



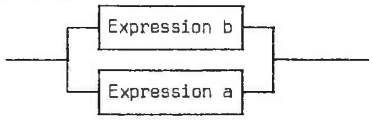
CONDITIONNELLE



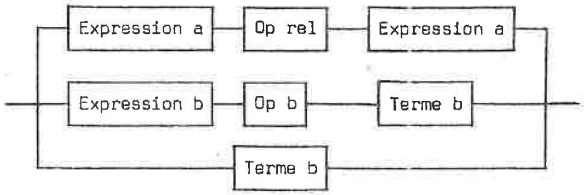
ENTREE-SORTIE



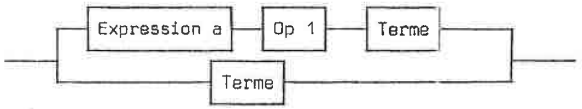
EXPRESSION



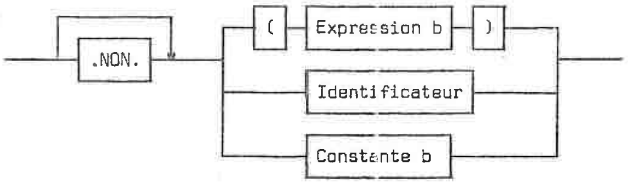
EXPRESSION B



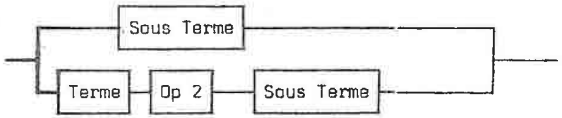
EXPRESSION A



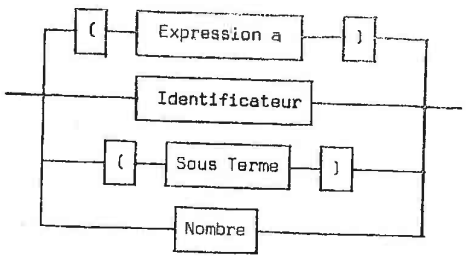
TERME B



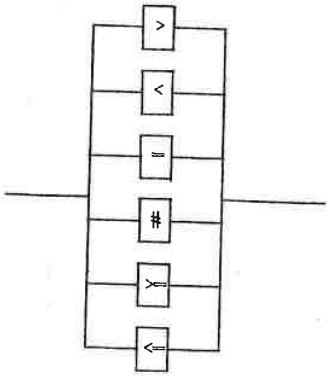
TERME



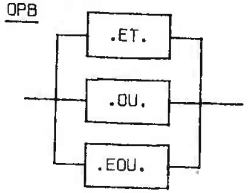
SOUS TERME



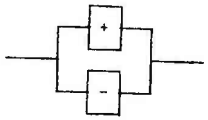
OPREL



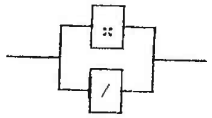
OPB



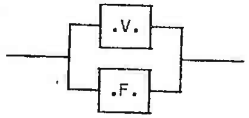
OP 1



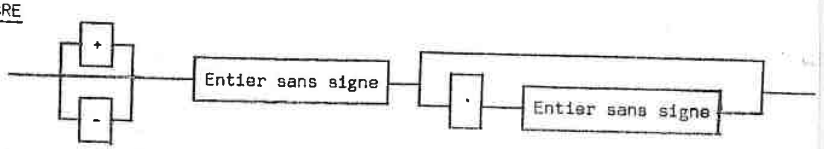
OP 2



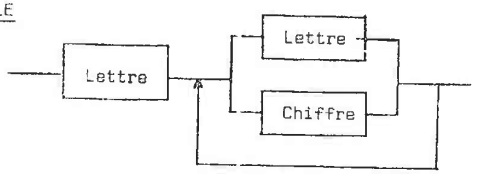
CONSTANTE B



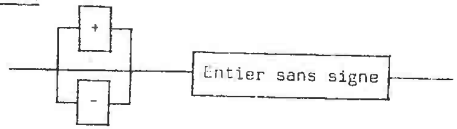
NOMBRE



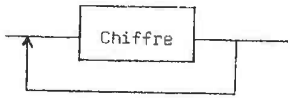
IDENTIFICATEUR SIMPLE



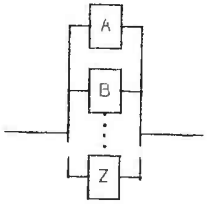
ENTIER



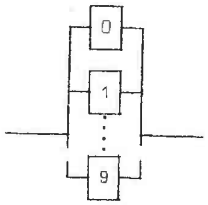
ENTIER SANS SIGNE



LETTRE



CHIFFRE



ANNEXE 2

SYNTAXE DES STRUCTURES DE CONTROLE DE NIVEAU 2

- séquentiel : < Module > - < Module >
- parallèle : < Module > // < Module >
- conditionel : si < cond > alors < Module > sinon < Module >
- itératif : tant que < cond > faire < Module >

SYNTAXE DES STRUCTURES DE CONTROLE DE NIVEAU 3

sur < événement > { récent / ancien } exemplaire

{ avec / sans } consommation

si < CONDITION > { récent / ancien } exemplaire

{ avec / sans } consommation

faire < Tâche >

SYNTAXE DES CONNEXIONS

< sortie > de < nom de module >

——▶ < nom de donnée transmissible >

< nom de donnée transmissible >

→ < entrée > de < nom de module >

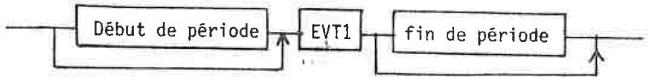
, { avec / sans } consommation

, { ancien / récent } exemplaire

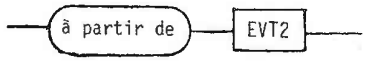
SYNTAXE DE LA DECLARATION DES EVENEMENTS

< déclaration d'événement > : < événement > , < durée de vie >

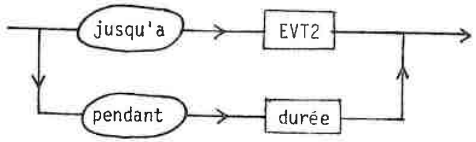
Evénement

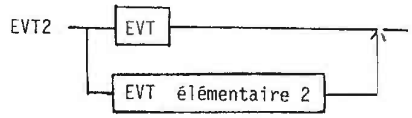


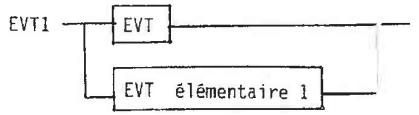
Début de période

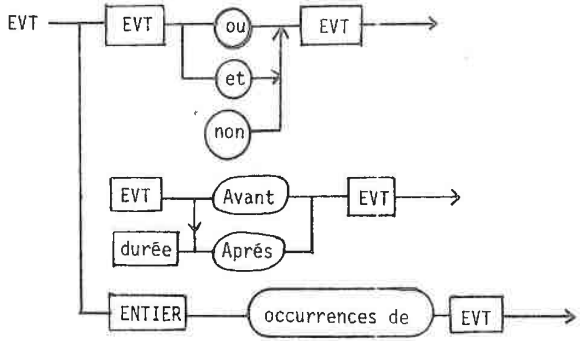


Fin de période

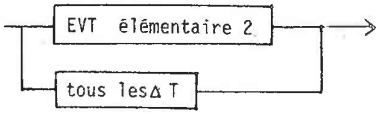






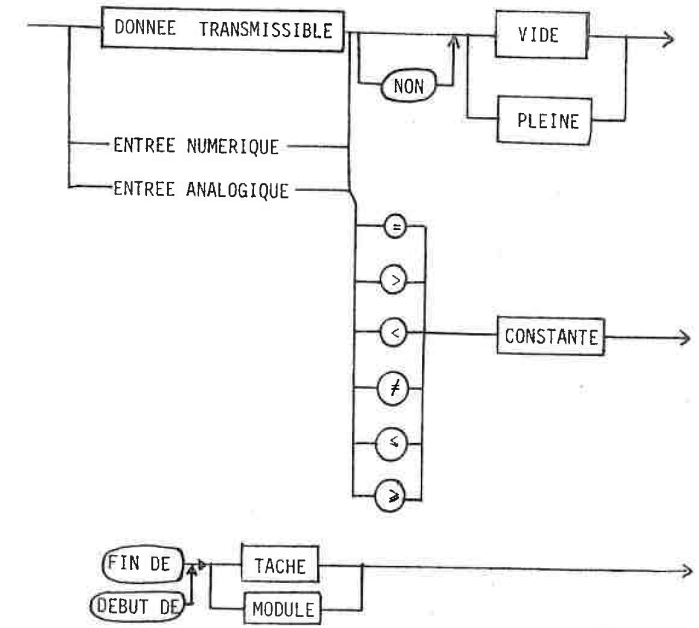


EVT élémentaire 1



EVT élémentaire 2

- HEURE →
- TRANSITION VF Entrée Numérique n° →
- TRANSITION FV Entrée Numérique n° →
- INTERRUPTION N° NIVEAU BIT →



SYNTAXE DES CONDITIONS

- A partir d'événement

< COND > : EVENEMENT < événement > VIVANT

EVENEMENT < événement > NON VIVANT

- sur les tâches

< COND > : TACHE < Tâche > ACTIVE

TACHE < Tâche > INACTIVE

- sur les données transmissibles

< COND > : < Donnée > vide

< Donnée > pleine

{ récent / ancien } exemplaire < Donnée > < opérateur de relation > référence

longueur (< Donnée >) = < entier >

Spécification d'implantation

Ce qui suit ne constitue pas à proprement parler une syntaxe mais seulement une esquisse de ce que pourrait être une telle spécification.

Sur < site > implanter < Module >

sur < site > implanter < Tâche >

sur < site > implanter la structure de < tâche >

ANNEXE 3

EXEMPLE D'APPLICATION

L'application concerne une régulation de vitesse et de courant d'un moteur à courant continu. Nous étudierons les trois tâches suivantes :

- régulation de vitesse
- régulation de courant
- sécurité sur le courant.

Ces pages sont issues de [THOMESSE et Al - 79].

TACHE REGULATION DE VITESSE

Cette tâche calcule, pour une référence de vitesse donnée, la consigne de courant pour le moteur. Elle devra aussi mesurer la vitesse réelle du moteur. La consigne sera notée ICONS. Elle sera calculée par un algorithme de type P.I. :

$ICONS_n$	: résultat	5	$ICONS_n = K_v \times ECARV_n + INTV_n$
K_v	: gain proportionnel		K_v = constante
$ECARV_n$	: différence entre la vitesse de référence et la vitesse réelle.	3	$ECARV_n = VREF_n - VRET_n$
$INTV_n$	: intégrale de l'écart	4	$INTV_n = INTV_{n-1} + \frac{1}{\tau_v} \times \Delta T_v \times ECARTV_n$
$VREF_n$	: vitesse de référence	2	$VRFF_n$ = donnée
$VRET_n$	: vitesse mesurée	1	$VRET_n$ = mesure
τ_v	: gain de l'intégrateur		τ_v = constante
ΔT_v	: période d'échantillonnage de la vitesse	2	ΔT_v = donnée

L'indice n indique le numéro de l'itération. La variable $INTV_n$ est en effet récurrente, elle devra donc être initialisée.

Structure de contrôle :

La structure de contrôle pourrait être du type :

Sur changement de VREF Ou de VRET faire régulation de vitesse. Cependant, le temps intervient dans l'algorithme pour le calcul de $INTV_n$ on préférera donc une structure de contrôle du type :

Tous les Δt_v, faire régulation de vitesse

Entrées-Sorties

La tâche sera découpée en deux modules. Le premier pour la définition 1 (il peut y avoir gestion d'un capteur). Le second, pour l'ensemble des autres, sera noté module régulation de vitesse. Le premier noté mesure de vitesse.

. Module mesure de vitesse :

Entrée VIT : vitesse (donnée capteur) Sortie VIT.M. vitesse (donnée prétraitée)

. Module régulation de vitesse

Entrées : $VREF_n$: vitesse
$VRET_n$: vitesse
ΔT_v : intervalle de temps
Sortie : $ICONS_n$

TACHE REGULATION DE COURANT

Cette tâche calcule pour une référence courant donnée, l'angle d'allumage des thyristors. Le courant réel du moteur devra être mesuré. L'angle d'allumage sera noté REFANGLE, il sera calculé par un algorithme du type PI :

REFANGLE : résultat	8	$REFANGLE_n = K_i(\frac{1}{\tau_i} \times ECARI_n \times \Delta T_i + INTI_{n-1} - IRET_n)$
K_i : gain proportionnel	5	$ECARI_n = IREF_n - IRET_n$
τ_i : gain de l'intégrateur	6	$INTI_n = \frac{1}{\tau_i} \times (ECARI_n \times \Delta T_i) \times INTI_{n-1}$
ECARI : différence entre le courant de référence et le courant réel	2	$IRET_n$ = mesure
	3	$IREF_n$ = donnée
ΔT_i : période d'échantillonnage du courant	1	ΔT_i = donnée
$INTI_n$: intégrale de ECARI		K_i = 1er cas : constante
$IRET_n$: courant réel mesuré		2° cas fonction de IREF et VRET : $K_i = G(IREF_n, VRET_n)$
$IREF_n$: courant de référence	4	$VRET_n$ = mesure
$VRET_n$: vitesse réelle mesurée	7	K_i : Si $IREF_n$ < ICRIT Alors $K_i = G_1(IREF)$ Sinon $K_i = G_2(IREF, VRET)$

ICRIT : courant au changement de régime	ICRIT = constante
G_1, G_2 : tableau de gain optimal	G_1, G_2 : tableaux constants.

La variable INTI est récurrente, l'indice n indiquant le numéro de l'itération. Cette variable devra donc être initialisée.

Structure de contrôle

Le temps intervenant explicitement dans le calcul de INTI, on préférera une structure du type :

Tous les ΔT_i, faire régulation de courant

Entrées-Sorties

Entrées : - IREF : courant
- IRET : courant
- ΔT_i : intervalle de temps
- VRET : vitesse

Sortie : REFANGLE : angle

Module Sécurité intensité

Commentaire

on ne se préoccupe que des augmentations d'intensité
un algorithme analogue permettrait de tenir compte des baisses

```
entrée  ICONS
sortie  ICONS
état  INTEN  INIT  0
si  ICONS - INTEN > SIMAX
        alors  ICONS = ITEN + SIMAX
        sinon  ICONS = ICONS
si  ICONS > IMAX  alors  ICONS  =  IMAX
INTEN = ICONS
```

fin du module

CONNEXIONS

VIT de mesure de vitesse → V → VRET de régulation de vitesse
sans consommation
récent exemplaire

ICONS de régulation de vitesse → I1 → ICONS de sécurité intensité
sans consommation
récent exemplaire

ICONS de sécurité intensité → I2 → ICONS de régulation d'intensité
sans consommation
récent exemplaire

Tâche régulation de vitesse : Mesure de vitesse - régulation de vitesse

ou

on considère deux tâches une mesure la vitesse
l'autre effectue la régulation ;

c'est dans ce cas que nous avons écrit la connexion de VIT à VRET.

Cet exemple montre l'intérêt de notre décomposition modulaire. Nous avons défini trois tâches :

régulation de vitesse
régulation de courant
sécurité sur le courant.

La tâche sécurité sur le courant pourra, au niveau de l'implantation, être intégrée soit à la première, soit à la seconde, soit complétement indépendante. La spécification en termes d'entrée et de sortie permet de faire abstraction de ce qui est en amont et en aval du module.

BIBLIOGRAPHIE

[ABIGNOLI et Al-80]

M. ABIGNOLI, J.P. THOMESSE

Contrôle - commande des processus : pourquoi et comment décentraliser le traitement des données.

Automatique et Informatique Industrielles n° 84, Février 1980, pp. 39-44.

[ABRIAL - 78_a]

J.R. ABRIAL

Présentation du parallélisme dans un langage de haut niveau.

$3^{ème}$ journées informatiques Nice, Juin 1978.

[ABRIAL - 78_b]

J.R. ABRIAL

Z, A specification language.

Actes du groupe de travail IFIP, TOKYO, 1978.

[ALTABER et Al-77]

J. ALTABER, V. FRAMMERY, C. GAREYTE, J.P. JEANNERET, P. VAN DER STOCK

Contrôle en temps réel avec architecture distribuée.

Journées Bigre nov. 1977, IRIA, pp. 1-6.

[ANCEAU - 78]

F. ANCEAU

Mécanismes primitifs de synchronisation au niveau matériel.

Colloque systèmes d'exploitation, IRIA, Octobre 1978.

[ANDLER - 78]

S. ANDLER

Synchronizations primitives and the verification of concurrent programs.

2^{nd} colloque international sur les systèmes d'exploitation, IRIA, Octobre 1978.

[ARVIND et Al-77]

ARVIND, K.P. GOSTELOW, W. PLOUFFE

Indeterminacy, Monitors and dataflow

Proc of 6th ACM Symposium on operating systems principles pp 159-169, Novembre 1977.

[AUBRY et Al-1979]

J.F. AUBRY, J.P. THOMESSE

Conception of a distributed application in process control. Example of an electromechanical converter controlled by microprocessors.

SOCOCO 79, 2nd IFAC/IFIP Symposium on Software for computer Control, Prague June 11-15, 1979, pp P.VI-1-PV1-4.

[AUBRY et Al-80]

J.F. AUBRY, G.H. PFITSCHER, C. IUNG, R. HUSSON

Commande directe à microprocesseurs d'un moteur à courant continu alimenté par des ponts à thyristors.

CONUMEL 80 - Colloque sur la commande et régulation numériques des machines électriques, LYON 28.30 Avril 1980.

[BALZER - 71]

R.M. BALZER

PORTS, a method for dynamic interprograms Communication and Job control.

AFIPS SJCC, 1971.

[BEATTY - 74]

J.C. BEATTY

Register assignment algorithm for generation of highly optimized object code.

IBM Journal of research and development.Janvier 1974.

[BELL et Al-70]

C.G. BELL, A. NEWELL

The PMS and ISP descriptive systems for computer Structures.

1970 S.J.C.C. Conf AFIPS, 1970, pp 351-374.

[BELPAIRE et Al-74]

G. BELPAIRE, J.P. WILMOTTE

A semantic approach to the theory of parallel processes in 1973 Int Comput Symp.

Günther et al Eds, North Holland, 1974, pp 159-164.

[BETOURNE et Al-77]

C. BETOURNE, L. FERAUD, J. JOULIA, A. NOURADI, J.M. RIGAUD

LEST : Un langage d'écriture de systèmes.
Journées Bigre, nov. 1977, IRIA, pp. 25-28.

[BEZIVIN et AL-77]

S. BEZIVIN, Y. JEGOU, J.L. NEBUT, R. RANNOU

A methodological approach to the design of real time operating systems.
IFAC/IFIP Workshop Real time programming, Smedema Ed. Pergamon Press pp. 153-162. 1977.

[BINDER et Al-77]

Z. BINDER, A. JANEX, B. MONNIER, D. REY

Coordinated Decentralized Control of a Complex distillation Plant.
IFAC/IFIP Conference, LA HAYE, June 1977.

[BLANCHARD - 78]

M. BLANCHARD

Rapport de la commission de normalisation de la représentation du cahier des charges d'un automatisme logique.

Automatisme tome XXIII n° 3-4, Mars-Avril 1978 pp 66-83.

[BLOOM - 79]

T. BLOOM

Evaluating Synchronization mechanisms.

ACM/IEEE 3rd conférence on operating Systems principles pp 24-32.

[BOSSY et Al-79]

J.C. BOSSY, P. BRARD, P. PAUGERE, C. MERLAUD

Le Grafcet, sa pratique et ses applications

EDUCALIVRE, Paris, 1979.

[BRANDES et Al-72]

J. BRANDES

A real time programming language and its application for measuring processes.

IFAC 5th World Congress, Paris, June 1972.

[BRINCH HANSEN - 72]

P. BRINCH HANSEN

Structured multiprogramming

Communications of the A.C.M. Vol 15, 7, pp 574-578, July 1972.

[BRINCH HANSEN - 73]

P. BRINCH HANSEN

Concurrent programming concepts

Computing surveys Vol 5 n° 4, Dec 1973, pp 222-245.

[BRINCH HANSEN - 78]

P. BRINCH HANSEN

Distributed processes : A concurrent programming concept

Communications of the A.C.M., Vol 21 n° 11, Nov 1978, pp 934-941.

[BRINCH HANSEN - 79]

P. BRINCH HANSEN

A Keynote adress on concurrent programming.

COMPUTER, May 1979, pp 50-56.

[CARTER - 77]

J.L. CARTER

A case study of a new code generation technique for compilers.

C.ACM. dec 1977, Vol 20 n° 12.

[CAUMONT - 79]

P. CAUMONT

Implantation de sygare sur PB6 Merlin Gerin par utilisation de APL 2M.

Rapport de DEA, I.N.P.L. Nancy, Septembre 1979.

[CHANDY et MISRA - 79]

K.M. CHANDY, J. MISRA

An axiomatic proof technique for Networks of Communicating processes.

IGDD, IRIA Seminar, Aix en Provence, May 1979.

[CHEVAL et Al-76]

J.L. CHEVAL, F. CRISTIAN, S. KRAKOWIAK, M. LUCAS, J. MONTUELLE, J. MOSSIERE

Un système d'aide à l'écriture des systèmes d'exploitation
Actes du Congrès AFCET 1976.
Panorama de la nouveauté informatique en France.

[CHEVAL et Al-1979]

J.L. CHEVAL, C. CLEMENCET, J.M. GUILLOT, P. LAFFORGUE, J. MOSSIERE, J. RAYMOND, S. ROUVEYROL

Un projet de système distribué : SORTILEGE

Journées Bigre 1979, Nancy, pp 223-231.

[CHEVANCE - 78]

R.J. CHEVANCE

Mécanismes de communication dans un système.

Actes du Congrès Afcet 1978, Ed Hommes et Techniques Tome 1, pp 432-440.

[CHININ - 78]

G.D. CHININ

Language specification without loss of efficiency in Constructing quality software.

Hibbard and Schuman Ed, North Holland, 1978.

[CLEMENCET et Al-79]

C. CLEMENCET, S. KRAKOWIAK

Structuration et contrôle de l'exécution dans les systèmes informatiques répartis.

Journées Bigre, Nancy 1979, pp 209-222.

[COCHET-MUCHY et Al-77]

A. COCHET-MUCHY, P. NONN, J.P. THOMESSE

Conception d'applications temps réel réparties

Présentation du projet SYGARE, Journées Bigre, IRIA nov 1977.

[COCHET MUCHY - 78]

A. COCHET-MUCHY

La production de programmes dans le projet SYGARE

Thèse de Docteur Ingénieur INPL, Nancy, nov 1978.

[COCHET MUCHY et Al-80]

A. COCHET-MUCHY, J.P. THOMESSE

Compiling Distributed Applications

A paraître dans EUROMICRO, 1980.

[COMTE et Al-78]

D. COMTE, J.C. SYRE

Parallélism , control and synchronization expression in a single assignment language

Sigplan A.C.M. Vol 13, n° 1, jan 1978, pp 25-33.

[COURTOIS - 71]

P.J. COURTOIS, F. HEYMANS, D.L. PARNAS

Concurrent control with Readers and writers

Communications of the A.C.M. Vol 14 n° 10, Oct 1971.

[CROCKER et Al-72]

S.D. CROCKER, J. HEAFNER, J. METCALFE, J. POSTEL

Function oriented Protocols for the ARPA Computer network.

AFIPS, 1972.

[CROCUS - 75]

CROCUS

Les systèmes d'exploitation des ordinateurs

Dunod Ed., 1975.

[DAHL - 66]

O.J. DAHL, K. NYGAARD

Simula - An Algol - based simulation language

Communications of the A.C.M. Vol 9 n° 9, Sep 1966.

[DARGENT et Al-79]

L. DARGENT, D. MEYER

Définition d'un système d'accès à des données réparties

Journées Bigre, Nancy, 1979, pp 164-174.

[DARGENT - 79]

L. DARGENT

Mise à jour de données réparties - contrôle de la concurrence d'accès.

Thèse de 3ème cycle, Nancy, Décembre 1979.

[DEMUYNCK et Al-78]

M. DEMUYNCK, B. MEYER

Les langages de spécification, vers une meilleure analyse des problèmes informatiques et des programmes plus fiables.

Bigre, Novembre 1978, pp 15-23.

[DENNIS - 71]

J.B. DENNIS

On the design and specification of a common base language

Proceedings of the symposium on computers and automata

Polytechnic Press of the Polytechnic Institute of Brooklyn, N.Y. 1071, pp 47-74.

[DENNIS - 75]

J.B. DENNIS

First version of a data flow procedure language

Mac Technical Memorandum 61, M.I.T., May 1975.

[DERNIAME - 74]

J.C. DERNIAME

Le projet Civa

Thèse de Doctorat es-Sciences, Université de Nancy I, Janv 1974.

[DESCHIZEAUX et Al-77]

P. DESCHIZEAUX, P. LADET

Programmation en temps réel de synchronisation par gestion des liens événement-processus.

Journées AFCET temps réel, Paris, Nov 77.

[DESCHIZEAUX et Al-79]

P. DESCHIZEAUX, R. GRIESNER, P. LADET

A real tine operating system for microcomputer network

SOCOCO'79 - 2nd IFAC/IFIP Symposium on Software for Computer Control
Prague Tchecoslovaquie, Tome 11-15, 1979, pp M-III-1-M-III-4.

[DIJKSTRA - 68]

E.W. DIJKSTRA

Cooperating Sequential processes
In Programming Languages.

F. Genuys Ed. New York, Academic, 1968.

[DIJKSTRA - 71]

E.W. DIJKSTRA

Hierarchical ordering of sequential processes.

Acta Informatica, Vol 1, pp 115-138, 1971.

[DIJKSTRA - 74]

E.W. DIJKSTRA

Self stabilizing systems in spite of distributed control

CACM, Vol 17, n° 11, nov 1974, pp 643-654.

[DIJKSTRA - 75]

E.W. DIJKSTRA

Guarded commands, nondeterminacy and formal derivation of programs

C. ACM., August 1875, Vol 18 n° 8, pp 453-457.

[DUBOIS - 77]

B. DUBOIS

Implantation du système APILOG sur Solar 16

Rapport de DEA, INPL Nancy, 1977.

[DURRIEU - 79]

G. DURRIEU

Extension of the L.A.U. system : Global specification of synchronizations in a data driven language.

1st European Conference on Parallel and Distributed processing Toulouse, Feb 1979, pp 149-155.

[ELLIS - 77]

C.A. ELLIS

A robust algorithm for updating duplicated data bases
Workshop on distributed data management systems.

Berkeley 1977, pp 146-160.

[ELZER - 75]

J.F. ELZER

PEARL ; journées AFCET langages temps réel, Paris, 1975.

[FELDMAN - 79]

M.B. FELDMAN

An application oriented programming language for sequential machine studies.

IEEE Transactions on computers, Vol C-28 n° 8, August 1979, pp 582-586.

[FORD - 78]
W.S. FORD
Implementation of a generalized Critical Region Construct
IEEE transactions on software Engineering, Vol SE.4 n° 6 Nov 1978, pp 449-455.

[FYLSTRA - 75]
D. FYLSTRA
HAL/S Programming language
Journées AFCET "Langages temps réel", Paris 1975.

[GERLL - 75]
D. GERLL
La programmation de D.D.C. par tables de paramètre.
Conférence ENSEM.

[GERNELLE - 76]
P. GERNELLE
Architecture du Micral M.
Journées IRIA, St Pierre de Chartreuse, Octobre 1976.

[GERTLER et Al-75]
J. GERTLER et J. SEDLAK
Software for process Control - A Survey
Automatica Vol 11, pp 613-625, 1975.

[GETHOEFFER et Al-79]
H. GETHOEFFER, J. LENZER, H. WALDSCHMIDT
Définition einer zwischensprache in einen mehrstufigen Übersetzungs Prossess.
Forschungbericht - Technische Hochschule Darmstadt Darmstadt F.R.G.

[GREEN - 78]
Rationale for the design of the Breen programming language.
CII-HB.

[GREIF - 77]
I. GREIF
A Language for formal problem specification.
C. ACM Vol 20 - n° 12, Dec 77, pp. 931-935.

[GRIEM - 76]
P.D. GRIEM
Approching an easy to learn method of programming real time parallel processes.
IFAC/IFIP Workshop Real time programming 2-4 juin 1976 IRIA.

[GUILLOT - 79]
J.M. GUILLOT
A synchronising mechanism for distributed applications
Bigre n° 14, Avril 1979.

[HAASE - HALLING - 77]
V. HAASE and H. HALLING
Description of real time applications using the guarded commands concept.
Journées AFCET Temps réel, Nov 1977.

[HAASE et Al-79]
V.H. HAASE et W.M. DEHNERT
High level languages structures for distributed real time programming.
SOCOCO'79 - 2nd IFAC/IFIP Symposium on Software for Computer Control, Prague jun 11-15, 1979, pp RV1,RV4.

[HABERMANN - 72]
A.N. HABERMANN
Synchronization of communicating processes
C.ACM. Vol 15 n° 3, pp 171-176, March 1972.

[HABERMANN - 75]

N. HABERMANN

Path Expression

Technical report - Carnegie Mellon University, 1979.

[HALLING - 77]

H. HALLING

Steps towards the implementation of a parallel code execution

IFAC IFIP Norkshop Real time programming 1977
C.H. Smedema Ed - Pergamon Press, pp 55-66.

[HALLING et Al-79]

H. HALLING, K. BURGER, H. HEER

Implementation and applications of PARCS

2nd IFAC/IFIP Symposium on Software for Computer Control - Prague Jun 1979, pp R.VI-1, R.VI.4.

[HEWITT et Al-77]

C. HEWITT, H. BAKER

Laws for communicating parallel processes.

Proc of the IFIP Congress 1977, Toronto, pp 987-992.

[HOARE - 72]

C.A.R. HOARE

Towards a theory of parallel programming
in "Operating systems techniques".

C.A.R. Hoare and R.H. Perott Eds. Academic Press 1972.

[HOARE - 74]

C.A.R. HOARE

Monitors : An operating systems structuring concept

Communications of the A.C.M. Vol 17 n° 10, pp 41-49,
Oct 1974.

[HOARE - 78]

C.A.R. HOARE

Communicating sequential processes

Communications of the A.C.M. Vol 21 n° 8, pp 666-677
Aug 1978.

[HOLLECZEK et AL-76]

P. HOLLECZEK, K. PELZ, F.J. PRESTER, R. RÖSSLER

The adaptation of a portable Pearl compilation system experience and futures aspects, with special emphasis on the routine package.

IFAC/Journée Internationales Real time programming
Rocquencourt, Juin 1976.

[HOLLER et Al-77]

E. HOLLER, O. DROBNIK

Implementation of decentralized coordination mechanism in distributed mini-micro computer systems

Institut für datenverarbeitung KARLSRUHE, 1977.

[HOLT - 71]

R.C. HOLT

Some deadlocks properties of computer system

3nd ACM Symposium on operating systems principles
Stanford, oct 1971.

[HOWARD - 76]

J.H. HOWARD

Proving Monitors

Communication of the ACM Vol 19 n° 5, pp 273-279
May 1976.

[HUGOT et AL-75]

P. HUGOT, M. RITOUT

PROCOL, un système de programmation temps réel

Journées AFCET, Langages temps réel, Paris, nov 1975.

[IBM1]

Functional charateristics

IBM System Reference Library GA 26-5918-8

[IBM2]

Time sharing executive system

Reference manual IBM System Ref Library.

[ICHBIAH et Al-79$_a$]

J.D. ICHBIAH, J.C. HELIARD, O. ROUBINE, J.G.P. BARNES, B. KRIEGBRÜCHNER, B.A. WICHMAN

Preliminary ADA reference Manual

Sigplan notices, Vol 14 n° 6, July 1979, Part A.

[ICHBIAH et Al-79$_b$]

J.D. ICHBIAH, J.C. HELIARD, O. ROUBINE, J.G.P. BARNES, B. KRIEGBRÜCKNER, B.A. WICHMAN

Rationale for the design of the ADA programming language

Sigplan Notices, Vol 14 n° 6, July 1979, Part B.

[JAMMEL et Al-77]

A.J. JAMMEL, H.G. STIEGLER

Managers versus Monitors

Information Processing 77 - Gilchrist Ed. North Holland 1977.

[JAMMEL et Al-79]

A.J. JAMMEL, H.G. STIEGLER

Structural decomposition and distributed systems

1st European conference on parallel and distributed processing, Toulouse, France Feb 1979, pp 16-26.

[JAY - 79]

R. JAY

Réalisation du module commande séquentielle dans le système MASC-16

Thèse CNAM, Grenoble 1979.

[KANN - MAC QUEEN - 76]

Coroutines and networks of parallel processes.

Rapport de recherche n° 202, nov 1976, IRIA - Also in Proc of Information Processing 77, Gilchrist Ed North Holland, 1977.

[KAISER et AL-78]

C. KAISER, M. KRONENTAL, J. LANGET, S. NATKIN, S. PALASSIN

Système informatique réparti pour la conduite d'un atelier mécanique.

Théorie et techniques de l'informatique, Actes du Congrès AFCET, 1978, tome 1, Editions Hommes et techniques, pp 157-169.

[KARP et Al-66]

R.M. KARP, R.E. MILLER

Properties of a model for parallel computations determinacy - termination - queuing.

SIAM J. Appl. Math. 14, Nov 1966, 1390-1411.

[KARP et Al-69]

R.M. KARP, R.E. MILLER

Parallel program schemata

J.C.S.S. Vol 3, 1969.

[KELLER - 76]

R.M. KELLER

Formal verification of parallel programs.

CACM Vol 19 n° 7, July 76, pp 371-384.

[KOSINSKI - 73]

P.R. KOSINSKI

A date flow programming language.

Report R6 4264, IBM J. Watson Research Center Yortown Heights N. York, March 1973.

[KRONENTAL - 79]
M. KRONENTAL
Towards the standardization of real time operating system kernels
IFAC/IFIP 2nd SOCOCO Conference, Prague, Juin 1979.

[LADET - 77]
P. LADET
Outils de structuration temps réel dans la commande de procédés industriels.
Thèse de 3ème cycle, INPG, 1977.

[LAGIER - 76]
Conception et réalisation d'un système temps réel pour un système multimicroprocesseurs.
Thèse de 3ème cycle, Nancy, 1976.

[LAMPORT - 76]
L. LAMPORT
The synchronization of independent processes.
Acta Informatica, Vol 7, pp 15-34, 1976.

[LAMPORT - 77]
L. LAMPORT
Concurrent reading and writing
Communications of the A.C.M. Vol 20 n° 11, pp 806-811 Nov 1977.

[LAMPORT - 78]
L. LAMPORT
Time clocks and the ordering of events in a distributed system.
Comm. A.C.M., July 1978, Vol 21 n° 7, pp 558-565.

[LEBARBIER - 78]
D. LEBARBIER
Organisation d'un compilateur multicible.
Bigre, Novembre 1978, pp 4-8.

[LE CALVEZ et Al-77]
F. LE CALVEZ, F. MADAULE, H.G. MENDELBAUM
Compiling Gaelic, a global real tume language
IFAC/IFIP Workshop Real time programming 1977
C.A. Smedema Ed. Pergamon Press, pp 37-46.

[LE CALVEZ - 77]
F. LE CALVEZ
GAELIC, un langage de description globale des synchronisations de processus.
Journées AFCET Temps réel, Nov 1977.

[LE CALVEZ - 79]
F. LE GALVEZ - LISCH
Définition d'un langage de description globale des applications en temps réel.
Thèse de 3ème cycle, Paris VI, Janvier 1979.

[LEDGARD et Al-75]
H.F. LEDGARD, M. MARCOTTY
A genealogy of control structures.
Communications of the A.C.M. Vol 18 n° 11, nov 1975.

[LE LANN - 77]
G. LE LANN
Distributed systems, towards a formal approach.
Proc IFIP Congress 1977, North Holland Ed. pp 155-160.

[LE LANN - 77]
G. LE LANN
Introduction à l'analyse des systèmes multiréférentiels.
Thèse de Doctorat d'Etat, Rennes, Mai 1977, 202 p.

[LE LANN - 79]

G. LE LANN

Les problèmes des signalisations dans les systèmes informatiques à contrôle réparti.

Note SYN-I-005, IRIA, 1979.

[LENZER - 79]

J. LENZER

An automatically generated translator system from a description of the target machine.

2nd SOCOCO, 1979, Prague.

[LISKOV - 74]

B. LISKOV, S. ZILLES

Programming with abstract data types

Sigplan notices Vol 9-4, 1974.

[LISKOV - 75]

B. LISKOV, S. ZILLES

Specification techniques for data abstraction

IEEE Trans on Software engineering Vol SE 1 n° 1, march 1975, pp 7-18.

[LORRAINS - 79]

LORRAINS

Réseaux téléinformatiques.

Hachette Ed., 1979.

[MENDELBAUM et AL-75]

H.G. MENDELBAUM, F. MADAULE

Automata as structured tools for real time programming

IFAC-IFIP Workshop on real time programming Boston, Aug 20-21 - 1975.

[MENDELBAUM - 76]

H.G. MENDELBAUM

Structuration dans la conception et la réalisation des systèmes en temps réel.

Thèse de doctorat es-Sciences, Institut de programmation, Paris VI, janv 1976.

[MIDDLETON - 77]

T. MIDDLETON

Specifiying programs structure trough sequence relation ships.

Sigplan Notices, October 1977, pp 43-47.

[MORRIS - 69]

D. MORRIS, G.D. DETLEFSEN

A virtual processor for real time operation in Software engineering,

Ton Ed. Vol 1, COINS Academic Press, 1969.

[MOSSIERE - 77]

J. MOSSIERE

Méthodes pour l'écriture des systèmes d'exploitation.

Thèse de Doctorat es-Sciences, Grenoble 1977.

[MOSSIERE - 77]

J. MOSSIERE

Sur l'exclusion mutuelle dans les réseaux informatiques

Rapport de recherche, 1977.

[MUNTEAN - 79]

T. MUNTEAN

Mutual coincidence in distributed systems of communicating sequential process

I.G.D.D. IRIA Seminar, Aix en Provence, May 1979.

[MUSSTOPF - 79]
G. MUSSTOPF, H. ORLOWSKI, B. TAMM
Program generators for process control applications.
2nd SOCOCO, 1979, Prague.

[NEBUT - 74]
J.L. NEBUT
Conception d'un système de langages de programmation.
Thèse de Docteur Ingénieur, Paris, Nov 1974.

[NICOUD - 76]
J.D. NICOUD
A common microprocessor assembly language.
2nd symposium on micro architecture, Euromicro, 1976.

[NONN et Al-78$_a$]
P. NONN, J.P. THOMESSE
Le système d'exploitation dans le projet SYGARE.
Actes du congrès AFCET Gif sur Yvette, Editions Hommes et Techniques, tome 1, nov 1978, pp 134-144.

[NONN - 78$_b$]
P. NONN
Le système d'exploitation dans le projet SYGARE.
Thèse de Docteur Ingénieur, I.N.P.L. Nancy, Nov 1978.

[OWICKI et GRIES - 76]
S. OWICKI, D. GRIES
Verifying properties of parallel programs : an axiomatic approach.
Communication of the ACM Vol 19 n° 5, pp 280-285, May 1976.

[PAIR et Al-75]
C. PAIR, J. MAROLDT
Une méthode de programmation déductive.
Rapport I.N.P.L., Nancy, 1975.

[PARAYRE et Al-75]
P. PARAYRE, M. TROCELLO
LTR - Un système de réalisation pour l'informatique temps réel.
Journées AFCET temps réel, Paris, 1975.

[PARNAS - 72$_a$]
D.L. PARNAS
A technique for software modules specification with examples.
Communications of the A.C.M., May 1972.

[PARNAS - 72$_b$]
D.L. PARNAS
On the criteria to be used in decomposing systems into modules.
Comm. of the A.C.M. Vol 15 n° 12, Dec 1972, pp 1053-1058.

[PATIL - 71]
S.S. PATIL
Limitations and capabilities of Dijkstra's semaphore primitives for coordination among precesses.
Comp Struc - Group Memo 57, Project MAC MIT, 1971.

[PETERSON - 74]
J.L. PETERSON, T.H. BREDT
A comparison of models of parallel computation
Proc IFIP, 1974, pp. 466-470.

[PETRI - 62]
C.A. PETRI
Kommunikation mit Automaten
Thèse - Darmstadt, 1962.

[PLAIGNAUD - 79]
A. PLAIGNAUD
Un simulateur de réseaux de Petri en APL.
Rapport de DEA, INPL, Nancy, 1979.

[PLEYBER et Al-77$_a$]

J. PLEYBER, M. DA SILVA

Software specification language for sequential processes

IFAC/IFIP worhshop Real time programming 1977
C.H Smedema Ed. Pergamon Press, pp 67-74.

[PLEYBER et Al-77]

J. PLEYBER, M. DA SILVA

Sage, un langage pour la programmation globale des applications de commande de procédés.

Journées AFCET Temps réel, Nov 1977.

[PRUNET et Al - 74]

F. PRUNET, J.M. DUMAS

Introduction à la modélisation naturelle des structures de commandes : l'ORGANIPHASE.

RAIRO J.2. Juillet 1974, pp 45-75.

[REED et KANODIA - 79]

D.P. REED, R.K. KANODIA

Synchronization with eventcounts and Sequencers

Communications of the A.C.M. Vol 22 n° 2, pp 115-123
Feb 79.

[RITOUT et Al-72]

M. RITOUT, P. BONNARD, P. HUGOT

Procol, a programming system adapted for process control

Proc of congrès IFAC 1972, pp 10-1 (1-5).

[ROBERT et VERJUS - 77$_a$]

P. ROBERT, J.P. VERJUS

Toward autonomous descriptions of synchronization module

Proceeding of the IFIP Congress 1977.
Totonto, August 8-12, 1977.

[ROBERT - VERJUS - 77$_b$]

P. ROBERT, J.P. VERJUS

Expressions autonomes de la synchronisation de processus concurrents.

Journées AFCET Temps réel, Nov 1977.

[ROHMER - 78]

J. ROHMER

APL, Un outil rentable de production de logiciel pour minis et micro processeurs.

Minis et Micros, dec 77, mars 78.

[ROHMER - 78]

J. ROHMER

Vers une nouvelle technologie du logiciel.

Théorie et Techniques de l'informatique, Actes du congrès AFCET 1978, Editions Hommes et techniques, tome 1, pp 193-203.

[ROUCAIROL - 78]

G.P. ROUCAIROL

Mots de synchronisation.

R.A.I.R.O. Informatique, Vol 12 n° 4, 1978, pp 277-290.

[ROUCAIROL - 79]

G.P. ROUCAIROL

Vers une caractérisation de la synchronisation de processus parallèles.

1st conférence on Parallel and distributed Computing
Toulouse, Feb 79.

[SAAL et Al-70]

H.J. SAAL, W.E. RIDDLE

Communicating semaphores

SLAC, CGTM 117, Stanford, 1970.

[SCHAFF - 74]

A. SCHAFF

Connexion et coopération de deux ordinateurs CII 10070 - IBM 1800.

Thèse de Docteur Ingénieur, Nancy, 1974.

[SCHNEIDER - 80]

R. SCHNEIDER

Les réseaux de Petri dans l'aide à la conception d'applications selon Sygare.

Thèse à paraître en 1980.

[SEMS]

MASC 16
Centralisation, surveillance d'états et de mesures commande séquentielle.

[SEMS]

RTES - Real time executive systems

Manuel de référence
Manuel d'utilisation.

[SINTZOFF et Al-75]

M. SINTZOFF, A. VAN LAMSVEERDE

Constructing correct and efficient concurrent programs.

Proc. Int. conf. on reliable Software
ACM-IEEE, 1975, pp 319-326.

[STERIA]

PROCOL T 2000
Notice Technique - Steria - Le Chesnay
Référence 1162 220/00 3900.

[SYRE - 76]

J.C. SYRE

Parallélisation de tâches séquentielles dans :

Ecole IRIA Architecture de machines, Nice, 1976, pp. 297-320.

[SYRE et Al-78]

J.C. SYRE et Al

Parallelism, Control and synchronisation expression in a single assignment language.

Sigplan Notices, Vol 13, n° 1, Jan 1978, pp 25-33.

[SZLANKO - 77]

J. SZLANKO

Petri nets for proving some correctness properties of parallel programs.

IFAC/IFIP workshop Real time programming 1977
C.H. Smedema Ed. Pergamon Press, pp 75-83.

[TACONET - 78]

S. TACONET

Contribution à la synhèse des systèmes séquentiels partition du graphe des états.

Thèse de Docteur Ingénieur, Nancy, 1978.

[THOMESSE - 74]

J.P. THOMESSE

Connexion et coopération de deux ordinateurs CII 10070 IBM 1800.

Thèse de 3ème cycle, Nancy 1974.

[THOMESSE - 77]

J.P. THOMESSE

A new set of software tools for designing and realizing distributed systems in process control.

IFAC/IFIP workshop Real time programming 1977

C.H. Smedema Ed. Pergamon Press, pp. 47-54.

[THOMESSE et Al-78]

J.P. THOMESSE, A. COCHET-MUCHY, P. NONN

Data flow analysis for the description and management of synchronization in real time applications.

IFAC/IFIP Workshop Real time Programming, June 1978 Mariehamn/Aland - Finlande.

[THOMESSE et Al-79$_a$]

J.P. THOMESSE, J.F. AUBRY

Conception et implantation - Exemple d'une application répartie.

Journées Bigre, Nancy 1979, pp 337-354.

[THOMESSE et Al - 79$_b$]

J.P. THOMESSE, J.C. DERNIAME

Flux de données et synchronisation.

IRIA/IGOD Workshop Aix en Provence, Mai 1978.

[TISSERANT - 77]

A. TISSERANT

Compilateur du langage Pascal pour miniordinateurs.

Thèse de Docteur Ingénieur, INPL, Nancy, Déc. 1977.

[TITLI - 76]

A. TITLI

La commande hiérarchisée.
DUNOD Editeur - 1976.

[VERJUS - 78]

J.P. VERJUS

Ordonnancement de processus par cheminement dans des fibres.

Bigre n° 9, Avril 1978, pp 24-26.

[VERNEL et Al-76]

P. VERNEL, M.L. LAGIER, R. HUSSON

A new Architecture for process control Computers :

NARCIS project
Euromicro newsletter, Vol 2 n° 4, oct 76, pp. 34-39.

[VERNEL - 77]

P. VERNEL

Conception et réalisation d'un multicalculateur temps réel à grande sûreté de fonctionnement.

Thèse d'état, Nancy 1977.

[VOJNOVIC - 77]

D. VOJNOVIC

Kernel real time systems

IFAC/IFIP workshop - Real time programming 1977
C.H. Smedema Ed, Pergamon Press, pp 139-144.

[WALDEN - 72]

D.C. WALDEN

A system for interprocess communication in a resource sharing computer network.

C.ACM. April 1972, Vol 15 n° 4, pp 221-229.

[WEBB - 75]

J.T. WEBB

CORAL 66

Journées AFCET temps réel, Paris, nov 1975.

[WIRTH - 71]

N. WIRTH

The programming language PASCAL.

Acta Informatica 1971, Vol 1 n° 1.

[WILMS - 78]

P. WILMS

Etude d'algorithmes de cohérence d'informations dupliquées et réparties.

Formalisation à l'aide de réseaux de NUTT
Rapport de recherche n° 160-160bis, Grenoble, Février 1979.

[WULF - 76]

W. WULF, R. LONDON, M. SHAW

An introduction to the construction and verification of Alphard programs.

IEEE Trans Softw Eng. 2.4. (1976) pp 253-264.

[YONEZAWA et HEWITT - 78]
A. YONEZAWA, C. HEWITT
Modelling distributed systems
Auto prog. 1, pp 370-376.

[ZIMMERMANN - 74]
H. ZIMMERMANN
Vers une formulation des protocoles dans un réseau d'ordinateurs.
Congrès AFCET Rennes, pp 271-291, Juin 1973.

INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE

AUTORISATION DE SOUTENANCE DE THESE DE DOCTEUR D'ETAT

DISCIPLINE : SCIENCES

Après avoir recueilli les avis de :

Monsieur le Professeur DERNIAME
Monsieur le Professeur HUSSON
Monsieur le Professeur KAISER

le Président de l'Institut National Polytechnique de Lorraine autorise :

Monsieur THOMESSE Jean-Pierre

à soutenir, devant l'I.N.P.L., une thèse de Docteur d'Etat en Sciences intitulée :

"SYGARE : UNE STRUCTURATION POUR LA CONCEPTION D'APPLICATIONS EN TEMPS REEL ET REPARTIES"

Fait à Nancy, le 02 Mai 1980

Le Président de l'I.N.P.L.

INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE NANCY

C. PAIR