Institut National Polytechnique
de Lorraine

E.N.S.E.M

THESE

présentée à

L'I.N.P.L

pour l'obtention du titre de

DOCTEUR INGENIEUR

SPECIALITE INFORMATIQUE

par

Alain COCHET-MUCHY

Ingenieur ENSEM

SUJET :

LA PRODUCTION DE PROGRAMMES DANS LE PROJET SYGARE

Soutenue publiquement le 16 novenbre 1978 devant la commission d'examen

embres du jury

Président	:	M. C PAIR
Examinateurs	:	M. C CAMOZZI
		M. J.C DERNIANE
		M. M GRIFFITHS
		M. R HUSSON
		M. J.P THOMESSE

Institut National Polytechnique
de Lorraine

E.N.S.E.M

THESE

présentée à

L'I.N.P.L

pour l'obtention du titre de

DOCTEUR INGENIEUR

SPECIALITE INFORMATIQUE

par

Alain COCHET-MUCHY

Ingenieur E N S E M

SUJET :

LA PRODUCTION DE PROGRAMMES DANS LE PROJET SYGARE

Soutenue publiquement le 16 novembre 1978 devant la commission d'examen

Membres du jury

Président	:	M. C PAIR
Examinateurs	:	M. C CAMOZZI
		M. J.C DERNIANE
		M. M GRIFFITHS
		M. R HUSSON
		M. J.P THOMESSE

AVANT-PROPOS

Nous remercions vivement Monsieur le Professeur PAIR, Président de l'Institut National Polytechnique de Lorraine, qui a bien voulu patronner notre travail, et qui nous fait l'honneur de présider ce jury.

Nous exprimons notre gratitude à Monsieur le Professeur HUSSON pour son accueil au sein de l'équipe automatique du Laboratoire d'Electronique, d'Electrotechnique et d'Automatique de l'E.N.S.E.M., et pour l'intérêt qu'il nous porte en participant à ce jury.

Nous remercions Monsieur le Professeur DERNIAME pour les nombreuses suggestions et critiques constructives qu'il nous a formulées, tout au long de notre travail, et lors de la rédaction de ce mémoire.

Que Monsieur le Professeur GRIFFITHS, Directeur de l'Institut Universitaire de Calcul Automatique de Nancy, et Monsieur CAMOZZI, Responsable du Département Logiciel de la S.E.M.S., trouvent ici l'expression de nos remerciements pour l'honneur qu'ils nous font en participant à ce jury.

Nous remercions Monsieur THOMESSE, Responsable du projet SYGARE, pour ses conseils de tous les instants et pour l'ambiance amicale qu'il n'a cessé d'entretenir.

Nous exprimons notre sincère reconnaissance à Madame DALBOURG, et au personnel du D.P.I.C. de l'Institut National Polytechnique de Lorraine, qui ont assuré avec soin et diligence la réalisation matérielle de ce mémoire.

Nous remercions enfin tous nos collègues du Laboratoire E.E.A. de l'E.N.S.E.M. et du Centre de Recherches en Informatique de Nancy pour leurs conseils et leurs encouragements amicaux.

TABLE DES MATIERES

INTRODUCTION

Concevoir et réaliser une conduite de processus industriels, répartie sur un réseau de processeurs sans se préoccuper des contraintes occasionnées par cette répartition ; être déchargé de l'implantation et de la gestion des mécanismes informatiques de protection et de synchronisation ; tels sont les souhaits de nombreux automaticiens.

L'insuffisance des moyens actuellement disponibles, pour satisfaire ces désirs, est à l'origine du projet de SYstème de Gestion d'Applications temps réel REparties dont nous préciserons les caractéristiques essentielles (chapitre I).

Nous consacrerons ensuite notre étude à la fonction production de programmes du Système Gare, objet de notre travail.
L'utilisation d'un langage de programmation unique (chapitre 2) permet d'assurer la transparence de la réalisation vis à vis de la répartition.

L'obtention du code exécutable d'une application est décomposée en deux étapes :

- une compilation du langage source en un langage intermédiaire orienté machine (chapitre 3)

- une génération de code objet, paramétrée par le site d'exécution (chapitre 4).

Nous aborderons enfin quelques problèmes liés à la production de programmes, et leur incidence dans notre contexte particulier : mise au point, entrées/sorties ... (chapitre 5).

CHAPITRE I

LE PROJET SYGARE

CHAPITRE I

LE PROJET SYGARE

I.1. EVOLUTION DES AUTOMATISMES INDUSTRIELS

Le développement des automatismes dans l'industrie s'effectue actuellement à un rythme assez rapide. Il fut marqué tout d'abord par les progrès de la technologie remplacement des techniques analogiques par les techniques numériques souvent plus fiables et enfin passage de ces dernières à la logique programmée : commande par ordinateur, automates programmables. L'avènement récent du microprocesseur a encore accéléré ce phénomène par suite de son faible coût et de sa souplesse d'utilisation.

Parallèlement à cette évolution du matériel, et grâce aux possibilités nouvelles qu'il offre, de nouveaux concepts se sont dégagés pour la conduite des processus industriels : intégration de l'automatisation au niveau d'un atelier, d'une chaîne de production ; application de diverses stratégies de commande : commande répartie, commande centralisée, commande hiérarchisée, ...

De plus en plus, il sera demandé aux automatismes industriels de ne pas tenir compte uniquement des paramètres physiques des régulations, mais aussi de facteurs économiques par exemple : commande optimale, gestion de production ...

Ces évolutions conduisent très souvent à l'utilisation de plusieurs processeurs pour effectuer une conduite de processus et ceci pour diverses raisons dont les principales sont :

- utilisation des anciens matériels disponibles
- emploi de processeurs spécifiques pour certaines fonctions ou certains traitements
- conduite trop lourde pour un seul processeur
- doublement de certains processeurs pour des raisons de sécurité.

Ces divers processeurs forment alors un réseau généralement hétérogène dont on devra assurer la gestion afin d'y répartir la commande de processus désirée.

Ce contexte physique rejaillit sur la conception et la programmation des applications industrielles réparties, nous en retiendrons trois aspects spécifiques :

*programmation d'applications industrielles

Elle impose fréquemment l'utilisation de nombreux périphériques non standards,

d'unités de traitement spécialisées (traitement du signal, unités d'entrée/sortie et d'acquisition, automates programmables, machines à commande numérique). Elle nécessite des programmes sûrs et fiables, toute erreur pouvant avoir des conséquences graves. En outre l'utilisation fréquente de calculateurs dits "industriels" et de petite taille conduit à développer des programmes d'encombrement restreint. Les structures de données manipulées sont généralement très rudimentaires.

*programmation temps réel

Les temps de réponse et de traitement doivent, dans de nombreux cas, être particulièrement courts ; cela conduit à une optimisation du code objet et à son adaptation à la machine par l'utilisation aussi complète que possible de toutes les possibilités matérielles qu'elle offre. Ce point est capital, car tout délai dans le contrôle du processus peut se traduire par une évolution dangereuse, et parfois irréversible, de celui-ci.

*programmation d'applications réparties

Nous utilisons plusieurs matériels différents, donc aussi des systèmes d'exploitation différents. Il serait cependant souhaitable de concevoir l'application de manière unique, de disposer de possibilités de reconfiguration et de s'affranchir de la structure du réseau. Cela impose en particulier de décharger l'utilisateur de l'implantation des mécanismes informatiques imposés par la répartition (échanges intersites, synchronisation et gestion des données).

Les langages utilisés pour la programmation de conduite de processus industriels furent longtemps les langages d'assemblage, car on ne disposait alors de rien d'autre ; de plus cette méthode permettait de "coller" parfaitement à la structure des machines. Bien que largement utilisé encore, le langage d'assemblage présente de nombreux inconvénients :

- programmes peu fiables et difficiles à mettre au point
- manque de lisibilité des programmes rendant leur maintenance et leur évolution difficiles, surtout par d'autres personnels que le concepteur
- intransportabilité du produit ; empêchant en particulier toute reconfiguration, en cas de défaillance d'un site, si l'on ne dispose pas de plusieurs processeurs identiques.

Ces diverses raisons, liées à des contraintes économiques, ont conduit à l'élaboration de plusieurs langages de haut niveau dits "Temps Réel" et essayant de pallier ces inconvénients. Certains sont nés de besoins précis dans un domaine donné, ce sont les langages "orientés-problème", d'autres se veulent plus généraux. Examinons maintenant les caractéristiques de quelques uns parmi les plus répandus (/15/,/16/,/17/).

I.2. QUELQUES LANGAGES TEMPS REEL

La plupart des langages décrits ci-après décomposent une application industrielle en un ensemble de tâches concurrentes.
Nous essayerons de préciser pour chaque langage, outre ses caractéristiques générales, les moyens mis en oeuvre pour assurer la gestion des tâches, implanter les protections et synchronisations et réaliser les opérations d'entrée/sortie.

I.2.1. Fortran Temps Réel

L'un des premiers langages temps réel à être apparu est certainement le Fortran temps Réel. Il est obtenu par extension du Fortran classique, non par adjonction d'ordres nouveaux, mais par insertion d'appels de sous programmes spécifiques fournis par les constructeurs. Les interfaces d'appels de ces sous-programmes sont normalisés ce qui permet de ne pas rendre le Fortran temps Réel plus intransportable que le Fortran classique (/15/, /16/, /32/).

Les principales options introduites concernent :

- gestion élémentaire des tâches
- entrées-sorties industrielles
- manipulation des chaînes de bits
- gestion (limitée) du temps.

Malgré son manque de souplesse, ce langage est largement utilisé car il est disponible sur la plupart des ordinateurs.

I.2.2. Basic Temps Réel

Le Basic temps Réel est un langage prévu pour être interprêté. C'est en fait une extension du Basic classique dont les performances sont spécialement étudiées pour une exécution dans un contexte temps Réel. Les entrées/sorties disponibles sont adaptées aux normes CAMAC (industrie nucléaire) ce qui en limite le champ d'utilisation (/16/, /40/).

I.2.3. Procol

Ce langage a été effectivement conçu pour le temps Réel. Il permet un certain nombre de facilités :

- description de la configuration utilisée
- gestion des tâches
- synchronisation
- entrées/sorties industrielles.

Cela est obtenu par l'utilisation d'instructions spéciales permettant

également la manipulation d'événements, de sémaphores, de ressources conjointement avec des instructions plus classiques inspirées de Fortran.

Une application est décomposée dans le système Procol en un ensemble de tâches bénéficiant de compilations séparées. Une section spéciale compilée à part et nommée "COMMON" permet de décrire la configuration de l'application : périphériques spéciaux, traitements de données ... Ces informations sont valables pour toutes les tâches.

Les structures de données disponibles en Procol sont également inspirées de Fortran, avec une extension en ce qui concerne les chaînes de bits.

Les entrées-sorties sont obtenues par des instructions analogues à celles de Fortran, munies de spécification de format décrivant les traitements à faire subir aux données (faisant éventuellement référence aux informations de la section COMMON).

On dispose enfin de la possibilité d'insérer du langage d'assemblage pour traiter tous les cas particuliers.

L'utilisation de Procol nécessite un système d'exploitation associé. Ce langage est implanté essentiellement sur T 2000 et Mitra (/16/, /17/, /1/).

I.2.4. RTL/2

La programmation d'une application en RTL/2 s'effectue sous forme de modules dotés d'une structure de bloc. RTL/2 permet de manipuler des données de types multiples :

- octet
- réel
- fractionnaire
- pointeurs de piles et de blocs

de manière isolée, en tableaux de variables de même type ou en enregistrements de variables et tableaux de types quelconques.

Les instructions comprennent :

- opérateurs arithmétiques et logiques
- décalages
- branchements (GO TO)
- affectations
- itérations (FOR ... WHILE ...)
- conditionnelles (IF ...)
- gestion des tâches et entrées/sorties par appel superviseur.

Le code généré est réentrant, il est obtenu après passage par un langage intermédiaire de type assembleur.
On peut éventuellement insérer du code objet directement dans RTL/2.
Le langage RTL/2 est développé en Grande-Bretagne par ICL (/6/).

I.2.5. HALS/S

Le langage temps Réel HALS/S est l'exemple même du langage "orienté problème" ; il a été développé pour l'aéronautique par la NASA et l'agence Spaciale Européenne.

HALS/S est dérivé de PL/1 et d'Algol 60 dont il reprend la partie algorithmique et la structure de bloc.
Il fournit à l'utilisateur un jeu d'instructions élaborées permettant la gestion des tâches, des événements, des données (zones de données à accès contrôlé), ainsi que des facilités d'algèbre linéaire nécessaires aux logiciels de guidage /5/.

I.2.6. Progress

Progress est un langage inspiré de PL/1 et d'Algol 68 en ce qui concerne la partie algorithmique. Il autorise l'utilisateur à définir ses propres types de données en plus des types standards (entier, fixe, flottant, bit, caractère).

Le langage fournit un jeu d'instructions spécialisées pour la gestion des tâches, des événements (synchronisation) et des ressources. Progress différencie les conditions d'exécution des tâches en distinguant les procédures synchrones des modules asynchrones.

Progress essaye d'optimiser l'occupation de la mémoire en réalisant une allocation dynamique pour les divers segments de données ou de code selon une méthode inspirée de Multics /48/, /50/.

I.2.7. CPL1

Le langage CPL1 est un sous langage de PL1. CPL1 veut être facilement portable et à faible taux d'expansion, il élude de ce fait un certain nombre de problèmes spécifiques des divers matériels, en particulier les entrées-sorties et la gestion de tâches. Il peut cependant être utilisé comme langage "temps Réel" car il est extensible et permet à l'utilisateur de définir toutes les instructions lui manquant pour manipuler les périphériques et mécanismes de synchronisation de son installation.

Le code généré par CPL1 est un code virtuel qui doit être "macro-défini" sur chaque machine. L'optimisation de ce code tient, pour une bonne part, aux restrictions imposées au programmeur : ainsi les expressions mixtes sont interdites /7/.

I.2.8. CORAL 66

La conception du langage Coral 66 a été orientéevers la génération d'un code rapide à l'exécution, le taux d'expansion du compilateur étant faible. Comme pour CPL1, ce langage ne possède pas directement d'instructions temps Réel. Il peut cependant acquérir les possibilités des autres langages en autorisant l'écriture de procédures ou de macro-instructions réalisant les fonctions désirées. Les instructions de Coral 66 sont celles d'Algol 60 avec possibilité de manipulation de bits, d'insertion de code et de macro-instructions. Les programmes Coral 66 sont munis de la structure de bloc.

Ce langage est assez largement répandu en Grande-Bretagne /6/.

I.2.9. LTR

LTR est un langage adapté à la programmation structurée. Une application est décomposée en plusieurs articles :

- les données moniteur (événements, ressources, périphérie ...)
- les données générales
- les procédures communes
- les données, procédures, initialisations, procédures d'interruptions.

Les instructions du langage permettent la programmation temps Réel grâce aux notions d'événement, de ressource, de délai, d'interruption. Elles permettent l'utilisation de structures de données assez complètes : variables de type entier, fixe, flottant, binaire, caractère, pointeur, ... sous forme de variable simple, de tableau, de structure simple, de tableau de structures, d'ensembles dynamiques.

Les entrées-sorties peuvent s'effectuer de diverses manières :

- directement par le programme
- sous contrôle du superviseur
- avec formattage des données
- entrée-sorties spécifiques sur périphérie spécialisée.

Le code est généré avec passage par un langage intermédiaire et optimisation à ce niveau /3/.

I.2.10. Pearl

La programmation d'une application en Pearl est composée de deux sections :

- une section système qui décrit le matériel utilisé : périphériques, interruptions ...

- une section programme écrite avec des instructions inspirées de Fortran et d'Algol. On dispose également d'ordres d'activation et de synchronisation de tâches.

Les instructions MOVE, GET et PUT permettent d'effectuer des entrées/sorties sur tous types de périphériques en mode formatté ou non.

La génération de code peut se faire sur plusieurs machines à l'aide d'un "macro-processeur". Les machines utilisées sont cependant de types assez voisins.
Ce langage est assez répandu en R.F.A. /2/, /8/.

I.2.11. Concurrent Pascal et Modula

Concurrent Pascal et Modula ne sont pas des langages temps Réel au sens habituel de l'expression. Ce sont deux langages inspirés de Pascal et adaptés à la programmation du parallélisme sur les multiprocesseurs.

Modula permet en outre de gérer assez facilement des entrées-sorties spéciales bien que cela ne fasse pas partie intégrante du langage /16/, /22/.

I.2.12. Divers

A ces langages plus ou moins largement répandus, il faut ajouter divers langages d'utilisation plus restreinte :

- PL/16 langage évolué conservant les possibilités de l'assembleur : accès aux registres, utilisation de toutes les instructions de la machine et à tous les niveaux d'information (bit, octet, mot). Ce langage est implanté sur les gammes T1600 et Solar. Il n'est pas transportable car bâti autour du matériel /16/, /36/.

- PL/M langage de haut niveau développé pour les microprocesseurs par Intel et Motorola /16/.

- Langages d'automates programmables (ex. APILOG /38/).

Il faut enfin noter que de nombreuses études sont en cours pour essayer de définir un langage vraiment adapté au temps réel notamment en ce qui concerne l'expression du parallélisme et des synchronisations entre tâches /23/, /41/, /46/, /47/, /54/.

I.2.13. Synthèse

Les langages que nous venons d'étudier satisfont généralement les

deux premiers points que nous avons exposés : programmation d'applications industrielles en temps réel. Ils présentent d'autre part un certain nombre de points communs. Tout d'abord les synchronisations entre tâches sont exprimées, de diverses manières, à l'intérieur même des programmes rendant ainsi délicate toute modification de l'application. La gestion des protections, mécanismes d'accès aux données, etc. sont facilités mais toujours à la charge de l'utilisateur avec tous les risques de mauvaise utilisation que cela implique.

Remarquons enfin que la plupart de ces langages sont orientés vers une gamme de machines et ne sont pas effectivement transportables (nécessité d'un système d'exploitation associé par exemple) ou s'ils le sont, c'est au détriment du service à l'utilisateur (CPL/1 ...).

Lorsque la conduite de processus doit être répartie, ces langages ne permettent plus de satisfaire le troisième point que nous avions défini : ils imposent une répartition à priori des différentes fonctions sur les divers sites et de plus ne permettent pas d'exprimer les relations entre les processeurs du réseau, car ils sont conçus dans un contexte monoprocesseur ou multiprocesseur à mémoire commune (Modula ...).

La gestion du réseau doit alors être effectuée par des procédures écrites en langage assembleur ; il convient d'ailleurs de remarquer que nous ne disposons pas encore d'outils très développés pour cela, les récents progrès dans ce domaine étant orientés vers d'autres préoccupations : télétraitement, gestion, etc. /26/, /39/, /40/.

Ces diverses lacunes ont été à l'origine du projet Sygare pour la conception et la réalisation de conduites réparties en temps réel de processus industriels.

I.3. LES CARACTERISTIQUES DE SYGARE

Le projet Sygare a été conçu pour aider l'automaticien dans la conception et la réalisation de systèmes répartis de conduite de processus industriels /43/. Dans ce but, il met à sa disposition divers outils qui permettent :

- de décrire et structurer l'application,
- de la programmer,
- de l'implanter et de la faire évoluer selon les besoins.

Ces divers outils rendent transparents, quand cela est possible, l'architecture du réseau et les mécanismes informatiques mis en oeuvre.

I.3.1. Décomposition d'une application dans Sygare

Une application industrielle peut généralement être considérée comme un ensemble de processus physiques se déroulant en simultanéité, cela se traduira dans Sygare par leur implantation sous forme de tâches logicielles parallèles. Ces tâches seront synchronisées entre elles et par rapport au milieu extérieur à l'aide de structures de contrôle appelées "structures de contrôle de niveau 3".

Une tâche peut dans la plupart des cas être scindée en divers sous-ensembles fonctionnels ou modules correspondant à des actions élémentaires. L'enchaînement des modules à l'intérieur d'une tâche est alors exprimé par des structures de contrôle spécifiques dites : "structures de contrôle de niveau 2".

Le module est l'élément de base de Sygare pour :

- la compilation,
- la répartition sur le réseau,
- l'exécution.

Le langage de programmation de ces modules constitue le niveau 1.

Ce fractionnement en tâches et modules présente le double avantage de correspondre à une décomposition naturelle de l'application industrielle, et de permettre une description et une programmation modulaires de celle-ci /34/ (fig.I-1).

Une caractéristique essentielle du projet Sygare est que toutes les liaisons entre les modules apparaissent explicitement et uniquement à l'extérieur de ceux-ci, simplifiant ainsi considérablement l'évolution future de l'application /52/.

I.3.2. Flux de données dans l'application

Chaque module de Sygare peut être considéré comme un automate. Il possède des entrées et des sorties : les variables qui lui permettent de communiquer avec les autres modules, et il est caractérisé par un algorithme, utilisant les structures de contrôle du niveau 1, qui exprime l'évolution des sorties et de l'état final, en fonction des entrées et de l'état précédent.

Nous voyons apparaître ici un autre aspect de Sygare : la description du flux de données dans l'application. En effet, la mise en correspondance des entrées et sorties des modules est faite entièrement à l'extérieur de ceux-ci à l'aide de déclarations de connexions appartenant au langage de niveau 2. Ces déclarations qui décrivent la circulation des informations dans l'application permettent, conjointement avec les structures de contrôle de niveau 2 et 3, d'en déterminer les

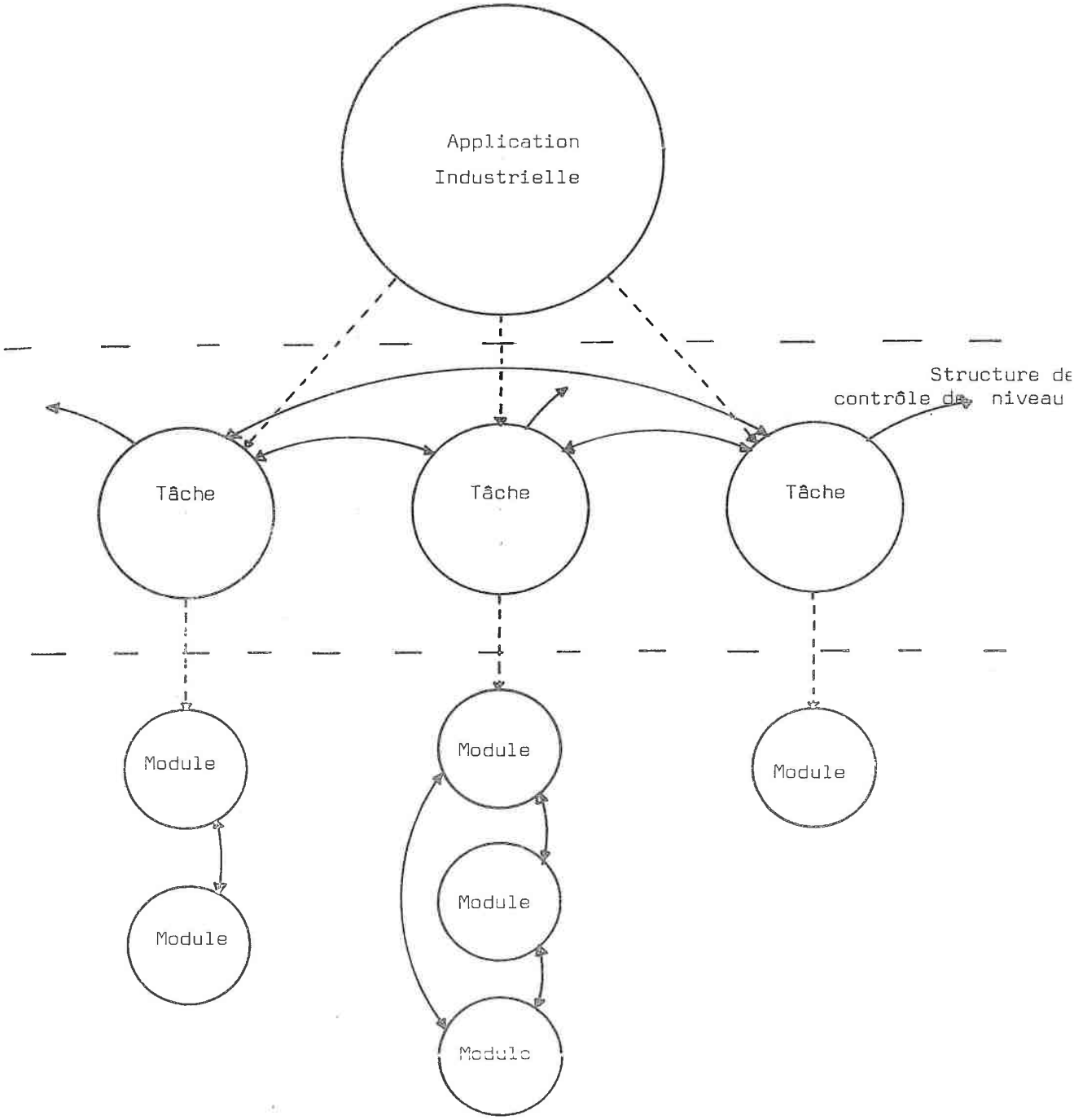


Fig. I.1. Décomposition d'une application industrielle de Sygare.

conditions d'utilisation. C'est grâce à elles que le moniteur d'exploitation pourra établir les mécanismes de protection et d'accès aux données ; ces accès sont en effet spécifiques de chaque utilisation (accès en exclusion mutuelle, duplication d'informations, mises à jour entre sites ...) /53/.

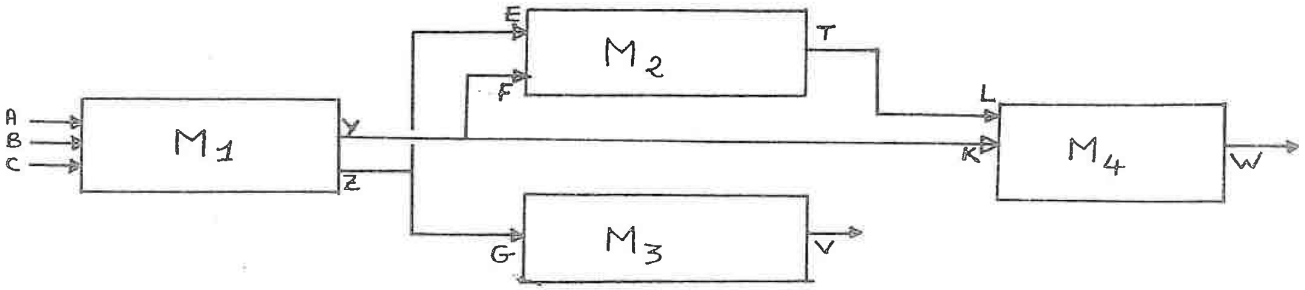
Les connexions interagissent aussi sur les conditions d'activation des modules et sont utilisées lors de la mise en place des synchronisations. L'implantation automatique de ces mécanismes de protection et de synchronisation est une autre particularité de Sygare qui contribue très largement à l'amélioration de la fiabilité de l'application /52/, /53/.

L'exemple ci-dessous nous aidera à mieux comprendre le rôle de ces déclarations de connexion.

Exemple : Considérons quatre modules.

M1	d'entrées	A, B, C	de sorties	Y, Z
M2	"	E, F	"	T
M3	"	G	"	V
M4	"	K, L	"	W

Le flux d'informations :



est décrit par les déclarations de connexions :

Y	de M1	→ F	de	M2
Z	de M1	→ E	de	M2
Y	de M1	→ K	de	M4
Z	de M1	→ G	de	M3
T	de M2	→ L	de	M4

On remarquera que la description du flux ne dépend pas directement des conditions d'activation des modules, ainsi avec les connexions ci-dessus on peut avoir divers

enchaînements :

M1 suivi de M2 // M3
M1 " de Si condition Alors M2 Sinon M3
M1 " de M2 suivi de M3

La seule condition à remplir étant bien sûr d'avoir produit la donnée avant de la consommer.

I.3.3. Répartition de l'application sur le réseau

Les choix d'implantation des divers modules et tâches sur les différents processeurs du réseau sont exprimés indépendamment de la description de l'application qui est donc totalement libérée des contraintes de répartition.

L'utilisation d'une description de l'application scindée en plusieurs niveaux de fonctionnalités différentes permet de remettre plus facilement en cause les choix initiaux et facilite l'évolution ultérieure du système (nouvelle répartition des modules sur le réseau, introduction de nouveaux modules ...).

La transparence de la répartition des modules lors de la conception de l'application est également assurée en ce qui concerne les opérations d'entrée/sortie par l'utilisation d'ordres unifiés traitant les échanges avec la périphérie conventionnelle ou industrielle (sur le site ou à distance), d'assignation de périphériques et de moniteurs d'entrée/sortie locaux interconnectés.

I.3.4. Le système Gare

Le système Gare est composé de deux parties essentielles :

*Un moniteur d'exploitation réparti qui assure la gestion du réseau, des communications entre sites, l'implantation des mécanismes de protection des données et de synchronisation, l'interprétation des structures de contrôle de niveau 2 et 3, des déclarations de connexion, des commandes opérateur d'activation et de contrôle de l'application, des déclarations de répartition des modules /42/.

*Un système de production de programmes qui fournira au moniteur d'exploitation le code binaire exécutable des divers modules de l'application sur les divers sites du réseau, et permettra l'archivage, l'évolution et la documentation de ces mêmes modules. C'est ce système que nous allons maintenant étudier plus en détail.

CHAPITRE II

LA DESCRIPTION DES MODULES DANS SYGARE

CHAPITRE II

LA DESCRIPTION DES MODULES DANS SYGARE

II.1. GENERALITES SUR LA PRODUCTION DE PROGRAMMES DANS SYGARE

La fonction production de programmes est celle qui permet la description des modules, leur traduction en code exécutable, leur archivage, leur documentation et leur évolution.

Cette fonction nécessite le choix d'un langage de programmation qui nous servira à décrire les modules. Compte-tenu des buts que nous nous sommes fixés et des remarques faites au chapitre précédent, ce langage doit être le même pour tous les sites du réseau, et de haut niveau, afin de garantir une certaine fiabilité et une programmation de l'application indépendante de sa répartition. Pour les mêmes raisons, nous n'autoriserons pas l'insertion de code objet dans le texte des modules, contrairement à la majorité des langages temps réel /16/. Le langage retenu ne possèdera pas non plus d'ordres d'activation de modules et de gestion de tâches, ceux-ci étant reportés à l'extérieur des modules dans les structures de contrôle de niveaux 2 et 3.

La traduction de ce langage source en code exécutable pourrait se faire directement ; elle nécessiterait alors autant de compilateurs que de types de sites sur le réseau (ou un compilateur à sorties multiples). Cette solution est peu intéressante car elle n'est pas souple et ne permet en particulier pas d'étendre simplement le réseau puisqu'il faut écrire de nouveaux compilateurs. Il convient d'autre part de remarquer que de nombreux traitements du langage source sont valables pour tous les sites :

- détection d'erreurs
- analyse lexicographique et gestion de la table des symboles
- analyse syntaxique.

Cela conduit à diviser la production du code exécutable en deux étapes /8/, /9/, /10/. :

- une compilation du langage source en un langage intermédiaire orienté machine, mais cependant indépendant de celle-ci

- une génération de code objet paramétrée par le site d'exécution.

L'adoption d'un langage intermédiaire présente en outre de nombreux avantages.

* Capacités de stockage réduites pour la conservation des modules
* Extension simplifiée du réseau : la première partir du traitement, la compilation, ne change pas, il reste à créer un générateur de code pour le nouveau site introduit /9/.
* Traduction rapide et aisée du langage intermédiaire en langage machine, l'analyse syntaxique et la détection des erreurs sont déjà effectués, cette possibilité est intéressante en cas de reconfiguration dynamique de l'application.
* Possibilité à partir du même langage intermédiaire et pour un site donné, de générer divers types de code objet
* Possibilité d'effectuer une simulation des modules au niveau du langage intermédiaire.
* Possibilité de disposer de bibliothèques de fonctions prédéfinies en langage intermédiaire, ce qui accélère et facilite le traitement lors de la compilation.

La production des programmes s'effectue dans Sygare sur un site unique. En effet, la plupart des calculateurs industriels disposent de configurations réduites et inadaptées à cette fonction, il suffira d'équiper une machine des moyens nécessaires. Le site choisi peut ou non participer lui-même à la conduite des processus. De plus, la fonction production de programme ne nécessite pas les mêmes impératifs de sécurité que la conduite proprement dite et sa défaillance momentanée est parfaitement tolérable dans la majorité des cas.

Le site de production est doté, en plus des traducteurs proprement dits, d'un ensemble de processeurs logiciels permettant la gestion, la conservation et la documentation des modules tant au niveau du langage intermédiaire (modification de la répartition, reconfiguration dynamique) qu'au niveau du langage source, afin de permettre une évolution aisée de l'application /34/.

II.2. LE LANGAGE SOURCE

Nous avons adopté un langage source inspiré des divers langages algorithmiques déjà cités (chapitre I). Notre but n'étant pas de définir un nouveau langage mais une méthode de production de programmes pour une application répartie, il conviendra de considérer que le langage présenté, ci-après, est une version minimum non figée. Nous suggérons d'ailleurs diverses améliorations possibles pour une plus

grande clarté de programmation et une meilleure détection des erreurs.

II.2.1. Les objets de Sygare. Les déclarations

II.2.1.1. Déclarations de type

Les structures de données adoptées sont volontairement très simples et inspirées de FORTRAN, ce sont :

- variables simples
- vecteurs de variables simples ou tableau à 1 indice
- tableaux de variables simples à 2 indices.

L'imbrication de ces structures est interdite, c'est à dire qu'un indice de tableau ne peut être lui-même un élément de tableau.

Les variables simples appartiennent à des types prédéfinis qui sont au nombre de quatre :

- entier
- réel
- booléen
- entier court (octet)

Ce dernier type ayant été introduit pour permettre une optimisation du code généré sur de nombreux mini-ordinateurs (utilisation de l'adressage immédiat) et sur les microprocesseurs dont les modèles les plus courant travaillent au niveau de l'octet.

Une plus grande souplesse pourra être apportée par l'introduction de deux types supplémentaires :

- chaînes de caractères
- chaînes de bits.

Pour faciliter la détection ultérieure d'erreurs, toutes les variables utilisées dans un module doivent être préalablement déclarées à l'aide d'un jeu de declarations spécifiques permettant de préciser également la structure retenue :

ENTIER liste
REEL "
BOOLEEN "
COURT "

où liste représente une suite d'identificateurs de variables simples ou de tableaux (il faut alors déclarer leurs dimensions). Il est en outre possible d'attribuer,

lors de leur déclaration, une valeur initiale à ces variables (cette option est analogue à l'ordre DATA de FORTRAN) :

exemple : ENTIER ITAB(5)<3*3,1,70>,K,J<7>
BOOLEEN BOOL<'V'>

Les déclarations ci-dessus définissent trois variables simples K, J, BOOL et un tableau. Seule la variable K n'est pas initialisée. Le tableau ITAB correspondra au lancement du module à l'organisation suivante :

ITAB(1)	3
	3
	3
	1
(5)	70

Nous n'avons pas offert à l'utilisateur de structures des données évolutives (tableaux à dimensions variables par exemple) car elles nécessitent des méthodes d'accès dynamiques généralement coûteuses en place et en temps et nous pensons que cela est en contradiction avec les buts à atteindre. Par ailleurs, ce genre de structure est d'une utilité assez faible pour la plupart des applications industrielles.

Nous suggérons enfin la possibilité de déclaration du domaine de variation des données ce qui permettrait un contrôle renforcé de l'exécution et éventuellement une optimisation du code généré /18/, /25/.

II.2.1.2. Utilisation des données

Ainsi que nous l'avons vu plus haut, les données utilisées par un module peuvent lui être propres, c'est à dire définies et utilisées par lui-même exclusivement. On les appelle alors variables locales et leur gestion est effectuée par le module lui-même. Elles sont aussi susceptibles d'être mises en correspondance avec les variables d'autres modules (au même titre que les paramètres de sous-programmes par exemple). On distingue alors plusieurs cas :

- variables utilisées par le module en consultation uniquement : on les déclare par la directive : ENTREE liste de variables préalablement apparues dans une déclaration de type.

- variables produites par le module : on les déclare par la directive : SORTIE liste de variables préalablement apparues dans une déclaration de type.

- variables consultées puis modifiées par le module : elles doivent être

déclarées en ENTREE et en SORTIE.

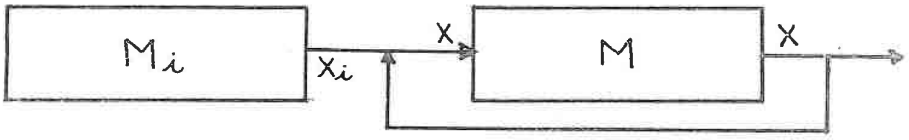
La gestion de ces variables est assurée par le moniteur réparti du réseau selon diverses méthodes en fonction du mode d'utilisation de chacune et des conditions d'activation des modules /42/, /52/.

Une utilisation particulière des variables déclarées en entrée et en sortie est celle de variable d'état d'un module itéré : la conservation de cette variable d'état est assurée par le moniteur d'une exécution à l'autre. L'initialisation d'une telle variable devra être effectuée par un module spécifique.

Exemple M_i : module d'initialisation, sortie X_i

M : module d'entrée X, de sortie X.

La connexion des modules est alors réalisée ainsi :



Remarquons que dans ce cas d'utilisation, la variable d'état est propre au module, il est donc envisageable de créer une déclaration d'emploi spécifique assurant la protection des variables de ce type vis à vis d'autres modules qui pourraient tenter de la modifier. En effet les déclarations conjointes en entrée et en sortie utilisées actuellement permettent d'assurer la cohérence des accès aux données mais ne permettent pas d'attribuer des droits privilégiés à certains modules.

Les déclarations ENTREE et SORTIE sont aussi utilisées lors de la compilation pour détecter les erreurs de cohérence du programme. C'est ainsi qu'une variable d'entrée ne devra pas être initialiée dans une déclaration de type.

Nous signalons enfin une restriction d'utilisation des variables tableaux, dans la version actuelle du projet, les déclarations ENTREE et SORTIE portent obligatoirement sur le tableau complet.

II.2.1.3. Attribut Utilisateur

Les déclarations étudiées précédemment permettent d'attribuer à chaque variable un type choisi dans une liste pré-établie. Il peut être intéressant de pouvoir donner à ces variables une qualification supplémentaire :

- amélioration de la lisibilité, et par suite de la maintenabilité des programmes,

- renforcement des contrôles de cohérence lors des interconnexions entre modules (on vérifiera alors la correspondance des qualifications)/42/,

- contrôle de la programmation des modules.

Cela est effectué par les déclarations d'attribut utilisateur. La création et l'utilisation de tels attributs sont entièrement laissées à l'initiative du concepteur /21/.

Exemple

```
(1)  ENTIER  V1
(2)  REEL V2, I1, I2, I3
(3)  ATTRIBUT VITESSE, INTENSITE
(4)  VITESSE V1, V2
(5)  INTENSITE I1, I2, I3
```

La ligne (3) correspond à la définition de deux attributs utilisateurs appelés respectivement VITESSE et INTENSITE.
Les lignes (4) et (5) permettent l'attribution de ces qualifications aux variables V1, V2, I1, I2, I3 qui ont déjà fait auparavant l'objet d'une déclaration de type (lignes (1) et (2)).

La plupart des données analogiques manipulées en contrôle de processus industriels étant des grandeurs physiques, il est envisagé de pouvoir décrire l'équation aux dimensions des attributs définis. Cela permettra ensuite à l'analyseur syntaxique de vérifier l'homogénéité des formules et de détecter ainsi les erreurs d'écriture, sources fréquentes d'erreurs.

Une autre extension possible consiste à déclarer, dans chaque module, les unités utilisées pour chaque attribut utilisateur afin de réaliser automatiquement les conversions nécessaires lors des interconnexions de modules.

II.2.2. Les instructions du langage

II.2.2.1. Expressions et affectations

Les variables et constantes peuvent faire l'objet de diverses opérations et sont combinées en expressions.
On en distingue deux types :

* les expressions arithmétiques qui utilisent les opérateurs : addition, soustraction, moins unaire, multiplication, division et les parenthésages. Ces expressions combinent les variables entières, entières courtes, réelles et les constantes de même type ; les conversions sont générées automatiquement dans les expressions

mixtes.

* les expressions logiques qui portent sur les variables booléennes et les constantes vrai (.V.) et faux (.F.) à l'aide des opérateurs : non, ou, et, ou exclusif des parenthésages ; elles peuvent être également des relations entre deux expressions à l'aide des opérateurs : >,<,>=,<=,=, #.

Les expressions sont utilisées essentiellement dans les instructions d'affectation :

<Affectation> ::=<Variable simple ou indicée>=<Expression>

L'expression utilisée doit bien sûr être de même type (arithmétique ou logique) que la variable simple ; cela est contrôlé, ainsi que toute tentative d'affectation d'une variable déclarée en entrée et en entrée uniquement (c'est à dire en consultation et non en modification).

Nous ne disposons pas d'affectations multiples (du genre A=B=C) ni d'affectations globales (affectation à un tableau).

II.2.2.2. Structures de contrôle

Les structures de contrôle disponibles pour programmer un module (niveau 1) sont :

<conditionnelle>::= Si<condition> Alors <S.I.> fsi|
Si <cond.> Alors <S.I.> Sinon <S.I.> fsi

<itération> ::= Tant que <cond.> faire <S.I.> ftq|
Pour <Var. Simple Entière> de <Val 1> à <Val 2> pas <Val 3>
faire <S.I.> fp

où <S.I.> représente une séquence d'instructions,
<cond.> une expression booléenne caractérisant la condition à tester
et <Val...> des valeurs numériques.

L'imbrication correcte de ces instructions composées est systématiquement vérifiée.

Les structures offertes sont suffisantes pour programmer tout algorithme ; elles pourraient éventuellement être étendues pour une plus grande simplicité de programmation (instruction "cas" par exemple) /24/.

II.2.2.3. Ordres d'entrée/sortie

Les périphériques conventionnels et industriels sont atteints dans Sygare par les mêmes instructions : LIRE et ECRIRE. L'utilisation de spécifications "FORMAT" permet au programmeur de préciser les traitements qu'il veut faire subir aux données (mise en page, traitement du signal ... /1/, /2/, /46/. Un module de

Sygare peut atteindre tous les périphériques du réseau, qu'ils soient ou non sur son site d'éxécution. Les problèmes posés par les entrées/sorties et la gestion des périphériques étant très importants, nous les étudierons au chapitre V.

II.2.2.4. Divers

Le programmeur peut utiliser dans le langage source des fonctions prédéfinies en bibliothèque et qui seront intégrées dans le texte en langage intermédiaire. L'utilisation de telles bibliothèques permet de s'affranchir d'une édition de liens et est donc de mise en oeuvre aisée.

L'arrêt d'un module est obtenu par l'instruction STOP.

Nous disposons enfin de deux directives de compilation :

* NOM qui permet d'attribuer un identificateur à chaque module
* END qui indique la fin du texte source.

II.2.2.5. Conclusion

Le langage que nous venons de présenter nous semble être un bon compromis entre la simplicité de la programmation et la facilité de génération d'un code objet performant.

Remarquons d'autre part, que la majeure partie des restrictions que nous avons pu signaler sont dues à la réalisation actuelle de Sygare et ne sont pas liées au langage ou à la méthode proposés.

II.3. LE LANGAGE INTERMEDIAIRE

Le compilateur produit un langage intermédiaire qui doit répondre aux exigences suivantes :

- indépendance vis à vis des sites du réseau (particulièrement en ce qui concerne les adressages de données) mais de niveau assez proche des langages machines pour faciliter la génération de code

- répertoire d'instructions complet (pour la traduction du langage source) mais limité, afin de trouver une traduction aisée sur tout processeur aussi rudimentaire soit-il. (Nous avons cependant admis une restriction : nous travaillons sur des machines à registres)

- conservation du maximum d'informations en provenance du texte source.

Plusieurs possibilités s'offraient pour le choix du langage intermédiaire. On pouvait notamment envisager un langage mono ou multi-opérandes.

On pourrait penser, a priori, qu'un langage à deux ou trois opérandes permet une génération plus performante de code objet car il garde une certaine distance vis à vis des divers processeurs et permet l'utilisation des instructions

spéciales souvent disponibles (incrémentation mémoire...). Ces cas sont cependant en nombre relativement limité.

Faute d'une étude exhaustive de toutes les possibilités de génération, les quelques configurations étudiées n'ayant pas fait ressortir d'avantage décisif pour un langage mono ou multi-opérandes, nous avons adopté un langage de type mono opérande car c'est ce type qui se rapproche le plus des langages machine actuels /29/, /30/, /31/. On le trouve d'ailleurs souvent employé lorsqu'un langage intermédiaire se révèle nécessaire /8/, /9/, /10/.

II.3.1. L'adressage en langage intermédiaire

La comparaison des diverses machines actuellement disponibles montre qu'elles possèdent généralement des jeux d'instructions assez semblables et que les principales disparités viennent des modes d'adressages, notamment avec l'apparition des microprocesseurs qui ont souvent recours à des "astuces" pour dissimuler leurs carence /27/, /28/.

Pour s'affranchir de ces difficultés, il convenait de garder un opérande indépendant des modes d'adressage physiques dans notre langage intermédiaire. Nous l'avons fait en adoptant un adressage symbolique.

L'opérande peut prendre diverses formes :

* une constante entière, logique ou réelle
* un identificateur de variable simple
* un identificateur de tableau et un déplacement éventuellement paramétré dans ce tableau (les tableaux sont supposés rangés de manière consécutive, ligne par ligne)
* une adresse de branchement (symbolique).

Les exemples ci-dessous nous montrent quelques représentations en langage intermédiaire de constantes ou variables :

Exemples :

* Adressage de l'élément de tableau TABL(I,J)

  L'opérande intermédiaire correspondant à l'élément de tableau TABL(I,J) est : TABL + I * "t_1" + J-"n"

La notation "-" désignant une valeur numérique entière

t_1 est la valeur déclarée de la première dimension

n vaut : $t_1 + 1$

* Adressage de TABL (3,4)

  L'opérande correspondant à TABL(3,4) est : TABL + "n"

avec : n = 3*"t_1" + 4 - "t_1" - 1

* Adressage de VECT (2)

On utilise alors l'opérande : VECT + 1

Pour optimiser la génération de code, nous affectuons à la compilation tous les calculs possibles, c'est à dire qui ne sont pas conditionnés par l'implantation sur les sites.

* Constantes

La référence à une constante réelle est effectuée à l'aide de l'opérande intermédiaire : C R n

où n est le rang de la constante dans la table des constantes réelles.

Une constante entière sera par contre laissée telle quelle dans le langage intermédiaire pour permettre l'utilisation de l'adressage immédiat chaque fois que cela sera possible et intéressant. Elle sera également rangée dans la table des constantes entières.

II.3.2. Les instructions du langage intermédiaire

Le langage intermédiaire contient d'une part des instructions exécutables servant à traduire celles du langage source et d'autre part un ensemble de "directives de génération" destinées au générateur de code pour la définition des données et l'organisation générale du binaire qu'il devra produire. Par abus de langage, nous regrouperons ces deux catégories sous le vocable "instructions du langage intermédiaire". Nous pouvons les répartir en plusieurs classes :

* Manipulation des registres de travail :

- rangement de registre : R A N
- chargement de registre : C H A

* Opérations arithmétiques et logiques

- addition : A D D
- soustraction : S O U
- négation (complément à 2) : N G T
- multiplication : M U L
- division : D I V
- ou inclusif : O U
- ou exclusif : E O U
- complémentation : N O N
- et : E T

* Tests et branchements

- comparaison : C M P
- branchement inconditionnel : B R A
- branchement conditionnel : B E branchement si égal
  B N E " " différent
  B I " " inférieur
  B I E " " inférieur ou égal
  B S " " supérieur
  B S E " " " ou égal.

Cet ensemble d'instructions est utilisé pour la traduction des diverses structures de contrôle.

* Gestion de sous-programmes

- branchement : B S P
- retour : R S P

Ces instructions seront employées lors de l'utilisation des fonctions de bibliothèque.

* Arrêt du programme : S T O P

* Insertion d'étiquette : E P C

Cette directive est utilisée pour insérer dans le texte intermédiaire des adresses de branchements lors de la traduction des structures de contrôle du langage source. Elle constitue uniquement un point de repère et ne donnera lieu à aucun code objet.

* Instructions d'entrée/sortie

- entrée : L I R
- sortie : E C R

Ces deux instructions correspondent aux instructions équivalentes du langage source. Leurs opérandes n'obéissent pas aux règles du langage intermédiaire ; ils sont obtenus par recopie et simplification du texte source et feront l'objet d'un traitement spécial du générateur de code.

* Directives de définition de données

- définition de constante réelle : C R
- définition de constante entière : C E
- définition de zone de travail : M A N

- définition de variables simples utilisées localement : V L O C
- définition de tableaux utilisés localement : T L O C
- définition de variables et tableaux utilisés en entrée et/ou sortie : VCOM

Ces directives seront étudiées en détail dans le chapitre consacré à la génération de code.

II.3.3. Caractéristiques du langage intermédiaire

Les instructions de notre langage intermédiaire sont indépendantes du type, et a fortiori de la représentation physique des variables manipulées.

Exemple :

Dans le contexte :

```
REEL A(6),B,C(3)
ENTIER D,I
```

L'affectation :

```
A(5) = B+C(I)+D
```

correspondra à la séquence intermédiaire :

```
CHA  B
ADD  C+I-1     C(I) variable réelle
ADD  D         D  variable entière
RAN  A+4
```

L'instruction ADD est utilisée aussi bien pour les variables entières que réelles, rangées sur un ou plusieurs mots.

Le nombre et la nature des registres disponibles sur la machine d'exécution conditionnent la stratégie d'allocation retenue, et par là-même, l'efficacité du code objet généré /20/. Ces paramètres ne sont cependant pas connus au niveau du langage intermédiaire ; c'est pourquoi nous avons choisi de travailler sur une machine virtuelle possédant un nombre illimité de registres. C'est le générateur code qui devra gérer les registres de la machine cible et effectuer les sauvegardes éventuellement nécessaires.

Exemple :

L'affectation du langage source :

```
A = (B+C)*(D+E)
```

donne en langage intermédiaire :

```
CHA  B    (B) → Reg 1
ADD  C    (Reg 1) + (C) → Reg 1
CHA  D    (D) → Reg 2
ADD  E    (Reg 2) + (E) → Reg 2
MUL       (Reg 2) * (Reg 1) → Reg 1
RAN  A    (Reg 1) → A
```

où :

*(-) représente le contenu associé à l'adresse spécifiée

* → représente un transfert

*Reg n représente le $n^{ème}$ registre de travail utilisé.

Un code opération non suivi d'opérande correspond à une opération entre deux registres de travail (ici c'est le cas de MUL).

Le langage que nous venons de présenter est facilement traduisible sur n'importe quelle machine du fait de son jeu limité d'instructions choisies pour leur adaptation aux structures courantes de processeurs. Il est cependant sémantiquement complet et permet donc de traduire toute extension éventuelle du langage source. L'utilisation de diverses directives de génération permet de conserver au niveau intermédiaire les diverses informations fournies par le langage source (déclarations ...).

CHAPITRE III

LA COMPILATION DANS SYGARE

CHAPITRE III

LA COMPILATION DANS SYGARE

III.1. INTRODUCTION

La compilation dans Sygare a pour but de transformer le langage source utilisé pour la description des modules, en langage intermédiaire assimilable par le générateur de code.

Le travail effectué à ce niveau comporte deux phases principales :

* Une analyse lexicographique permettant de créer et mettre à jour la table des symboles, ainsi que de vérifier l'emploi correct de chaque symbole.

* Une génération des instructions intermédiaires en fonction des instructions du texte source, avec intervention de l'analyseur syntaxique pour l'évaluation des expressions /10/, /11/, /12/, /13/.

Les informations recueillies lors des traitements du texte source seront, dans la mesure du possible, communiquées au générateur de code sous forme de directives de génération.

Un effort tout particulier est apporté lors de la compilation, pour la détection des erreurs de programmation.

III.2. LA COMPILATION

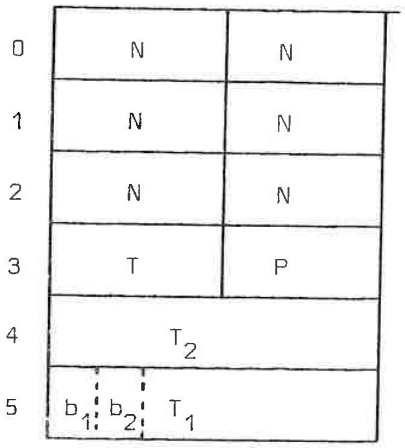
La structure du compilateur est assez simple, le texte source est traité instruction par instruction dans l'ordre de lecture et en une seule passe /11/.

III.2.1. Traitement des déclarations

Les déclarations permettent de construire et de mettre à jour la table des symboles du compilateur.

Les déclarations de type servant à définir les variables, à leur rencontre le compilateur va élaborer le descripteur des nouvelles variables et le rajouter à la suite de la table des symboles en vérifiant qu'il n'y a pas de double

définition. Le descripteur d'une variable est un pavé de six mots de seize bits ayant la structure suivante :



N : identificateur de la variable : de 1 à 6 caractères alphanumériques ASCII

T : codage du type physique de la variable (1 octet)

- T = 0 entier
- T = 1 booléen
- T = 2 réel
- T = 3 entier court

la place disponible dans l'octet peut être utilisée pour le codage de l'attribut utilisateur s'il existe.

P : conditions d'utilisation de la variable (1 octet)

- P = 0 variable locale
- P = $(80)_{16}$ variable en entrée
- P = $(40)_{16}$ variable en sortie
- P = $(C0)_{16}$ variable en entrée et sortie

T_1 et T_2 : éventuellement dimensions des tableaux selon les indications des bits b_1 et b_2 :

- b_1b_2 = 00 variable simple : $T_1 = 1$, $T_2 = 0$
- b_1b_2 = 10 tableau monoindice : T_1 = dimension, $T_2 = 0$
- b_1b_2 = 11 tableau bi-dimensionnel : T_1 = dimension 1, T_2 = dimension 2.

Ces deux derniers mots permettent de vérifier que l'emploi des variables est compatible avec leur déclaration.

Le compilateur utilise également les déclarations pour préparer les directives de génération (et éventuellement d'initialisation) de variables (directives TLOC, VLOC, VCOM).

III.2.2. Traitement des instructions composées

Le traitement des instructions composées est effectué à l'aide d'une pile /11/, /13/, permettant de conserver deux informations :

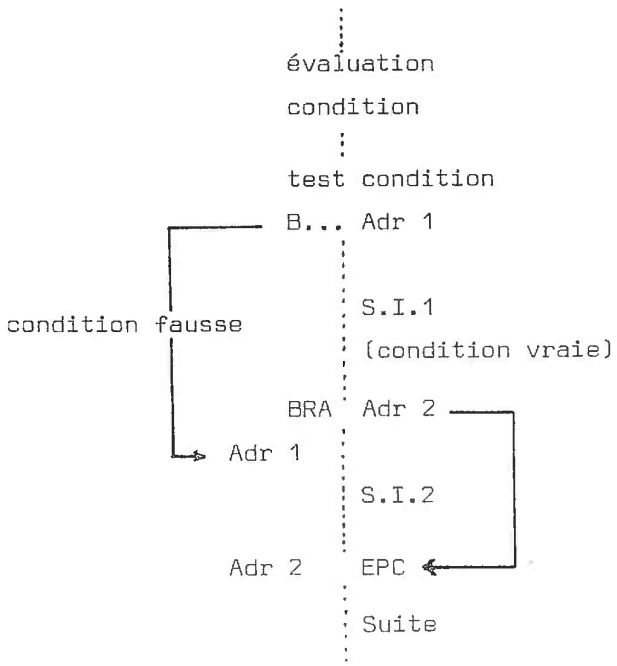
* la liste des composants élémentaires des instructions composées et leur ordre d'apparition, ce qui permet de détecter les erreurs d'imbrication

* les adresses des branchements qui devront être générées au niveau du langage intermédiaire.

Examinons à titre d'exemple le traitement du Si ... alors ... Sinon... et du tant que ... faire :

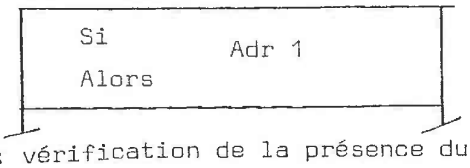
1 Si cond Alors S.I.1 Sinon S.I.2 fsi

donnera la séquence suivante en langage intermédiaire : (S.I.)= Séquence instructions)

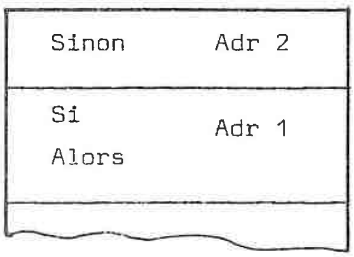


Le traitement de la pile donne alors :

Sur le "Si ... Alors" : empilement de l'adresse Adr 1 de branchement si la condition est fausse :



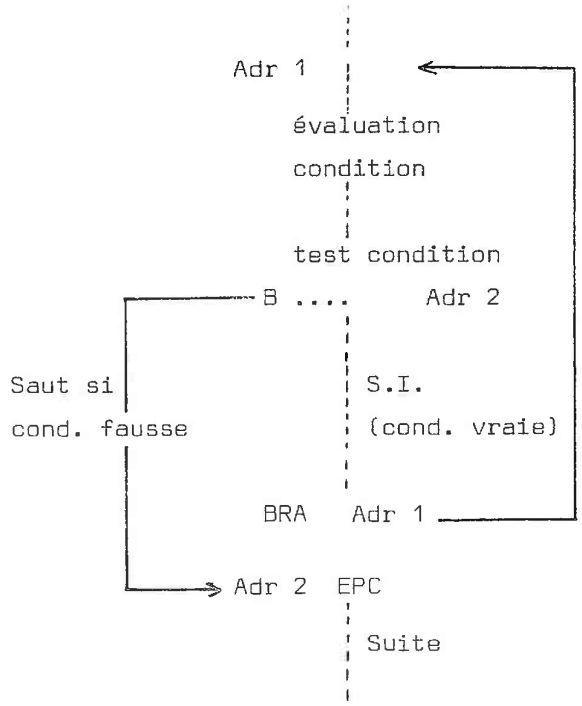
Sur le "Sinon" : vérification de la présence du "Si... Alors" et empilement de l'adresse Adr 2 de sortie :



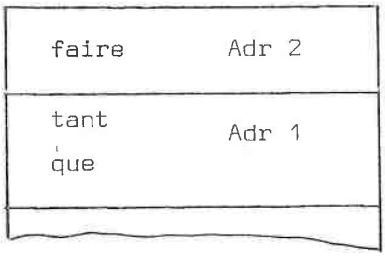
Sur "fsi" nous vérifions la présence du "Si ... Alors" et du "Sinon" (éventuellement) et nous dépilons ces mots.

2 Tant que cond faire S.I. ftq

se traduit par la séquence suivante :



Nous utiliserons cette fois deux mots de la pile à la rencontre de "tant que ... faire" où nous rangerons l'adresse Adr 1 de branchement pour l'évaluation de la condition de fin d'itération et l'adresse Adr 2 de fin d'itération.

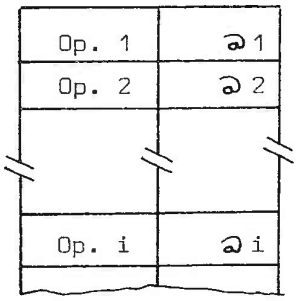


A la rencontre de "ftq" nous vérifions que la pile présente une configuration correcte et nous dépilons deux mots.

III.2.3. Traitement des expressions

Le traitement des expressions est effectué en deux temps :

*Une analyse lexicale de l'expression permettant de déterminer si toutes les variables sont bien déclarées et correctement utilisées et de détecter la plupart des erreurs d'écriture. Lors de cette analyse, le texte source est modifié pour simplifier les traitements ultérieurs ; de plus il donne lieu à la création d'une table d'analyse dont l'organisation est la suivante :



Les octets de poids fort de chaque mot représentent la suite des opérateurs mis en évidence dans l'expression, alors que les octets de poids faible indiquent l'emplacement, dans le texte source, des opérandes suivant chaque opérateur. En cas d'opérateurs consécutifs, ce pointeur est mis à zéro.

* Une analyse syntaxique qui n'est effective que si aucune erreur n'a été détectée à l'étape précédente /14/.

L'analyseur syntaxique est réalisé assez simplement. Les opérateurs sont munis de relations de priorité entre eux (représentées sous forme de matrice (fig. III.1). On explore la table d'analyse de manière séquentielle en considérant les opérateurs par couples :

$$(Op.\ 1,\ Op.\ 2),\ (Op.\ 2,\ Op.3),\ \ldots\ (Op._j,\ Op._{j+1})$$

Jusqu'à la rencontre d'un couple tel que $Op._{i+1}$ force le traitement de $Op._i$, ce qui nous est indiqué par consultation de la matrice de priorité. Nous noterons, dans la suite du texte, cette relation sous la forme :

$$Op._{i+1} \succ Op._i \quad (Op._{i+1} \text{ force le traitement de } Op_i)$$

Après le traitement de $Op._i$, on l'élimine de la table ainsi que les opérandes utilisés, et on reprend le processus de comparaison en remontant la table, avec les couples :

$$(Op._{i-1},\ Op._{i+1}),\ (Op_{i-2},\ Op._{i+1})$$

S'il n'y a plus d'opérateur à traiter avant le rang i+1, ou si aucun ne vérifie la relation : $Op._{i+1} \underset{\forall j<i}{\succ} Op._j$

b \ a	Déb	+	−	×	/	(	)	NON	OU EXCL	OU	ET	∪	<	>	#	=
Début	\	0	0	0	0	0	0	0	0	0	0	1	\	\	\	\
+	0	1	1	0	0	0	1	\	\	\	\	1	1	1	1	1
−	0	1	1	0	0	0	1	\	\	\	\	1	1	1	1	1
×	0	1	1	1	1	0	1	\	\	\	\	1	1	1	1	1
/	0	1	1	1	1	0	1	\	\	\	\	1	1	1	1	1
(	0	0	0	0	0	0	1	0	0	0	0	1	1	1	1	1
)	0	1	1	1	1	0	1	0	1	1	1	1	1	1	1	1
NON	0	\	\	\	\	0	1	1	1	1	1	1	\	\	1	1
OU EXCL.	0	\	\	\	\	0	1	0	1	1	1	1	\	\	1	1
OU		\	\	\	\	0	1	0	1	1	1	1	\	\	1	1
ET		\	\	\	\	0	1	0	1	1	1	1	\	\	1	1
∪																
<																
>																
#																
=																

Fig.III.1. Matrice de relation des opérateurs

Un "1" signifie que l'on a la relation : $Op._a \gtrdot Op._b$

le processus reprend séquentiellement :

$(Op._{i+1}, Op._{i+2}), (Op._{i+2}, Op._{i+3})$ ....

Lorsque la condition de fin est rencontrée, on vérifie qu'aucun opérateur n'a été oublié (ce qui serait dû à une erreur de syntaxe).

Le traitement des opérateurs est effectué de la façon suivante :

* Parenthèses : gestion d'un compteur permettant de vérifier l'imbrication correcte de sous-expressions parenthésées et positionnement des pointeurs pour la reprise de l'analyse.

* Début d'expression : le traitement de ce pseudo opérateur correspond en fait à la fin d'analyse de l'expression et provoque selon les cas l'affectation (instruction d'affectation) ou l'évaluation de la relation (expressions logiques).

* Opérateurs arithmétiques et logiques : le traitement est commun pour tous les opérateurs arithmétiques ou logiques binaires ; il est décrit par l'algorithme suivant :

```
Si  il existe un opérande précédent et un opérande suivant
Alors  générer
                CHA    O.P.  (opérande précédent)
                CODE   O.S.  (opérande suivant)

Sinon  si il existe un opérande suivant
       Alors générer
                CODE   O.S.

       Sinon  si il existe un opérande précédent
              alors  si opérateur commutatif
                     alors générer
                     CODE   O.P.

                     Sinon générer
                     CHA    O.P.
                     CODE
                     fsi

              Sinon (il n'existe ni précédent, ni suivant)
                     générer :
                     CODE

              fsi
       fsi
fsi
```

On désigne par opérande précédent, s'il existe, un opérande non encore traité situé dans la table d'analyse entre l'opérateur en cours de traitement et le premier opérateur non encore traité le précédant. La définition est symétrique pour l'opérateur suivant. CODE représente l'instruction du langage intermédiaire exprimant la fonction demandée par l'opérateur.

Les opérateurs unaires sont toujours générés sous la forme CODE précédée si nécessaire d'une instruction de chargement.

Lors du traitement des expressions, le compilateur gère les tables des constantes entières et des constantes réelles en y inscrivant, si elles n'y figurent pas déjà, les constantes rencontrées en cours d'analyse. On inscrit de même dans la table des constantes entières, toutes les constantes élaborées lors des calculs d'adresse.

III.2.4. Traitement des entrées/sorties

Les instructions d'entrée-sortie disponibles au niveau du langage intermédiaire constituent une forme simplifiée de celles disponibles dans le langage source. Le traitement de ces instructions va donc simplement consister en la transformation de l'ordre source en une série d'opérations élémentaires d'entrée/sortie avec réarrangement des paramètres concernant chaque opération.

Il n'est pas possible de traiter plus avant ces instructions, car le mode d'appel du moniteur d'entrée-sortie, le nombre, la nature et le mode de passage des paramètres nécessaires pour ces opérations, sont largement influencés par la machine cible /35/.

III.2.5. Divers

Lors de la rencontre de la directive END, le compilateur effectue un réarrangement de ses informations en vue du traitement ultérieur par le générateur de code et par le moniteur d'exécution (ce travail n'est effectué que dans le cas où la compilation s'est passée sans erreurs). La table des symboles est condensée par élimination des variables locales et forme alors le bloc de traitement qui sera utilisé par le moniteur d'exécution lors des interconnexions entre modules (déclarations de connexion des variables). Les tables des constantes sont scrutées pour créer les directives de génération de constantes entières et réelles utilisées par le générateur de code.

Le compilateur crée enfin un fichier portant le nom du module et dans lequel il va ranger les trois catégories d'informations qu'il a préparées, à savoir :

* directives de génération de variables et de constantes
* programme en langage intermédiaire
* table des symboles réduites.

III.2.6. Conclusion

Lors de la compilation, nous avons effectué un contrôle constant des erreurs de programmation : erreurs d'écriture, erreurs de syntaxe, mauvaise utilisation des variables indicées, mélange de types interdit, erreurs de cohérence, mauvaise imbrication d'instructions composées. Nous disposons alors à la fin de celle-ci de tous les éléments nécessaires à la génération de code objet sur les divers sites du réseau.

Les textes intermédiaires des modules sont archivés sur le site de production et seront utilisés lorsque l'opérateur aura fixé la répartition de son application. Ils seront également employés lors d'une éventuelle reconfiguration de l'application.

Nous n'avons pas rencontré de difficulté notable dans la traduction du langage source en langage intermédiaire, ce qui semble indiquer que ce dernier est bien adapté à nos besoins.

CHAPITRE IV

LA PRODUCTION DE CODE EXECUTABLE

CHAPITRE IV

LA PRODUCTION DE CODE EXECUTABLE

IV.1. PRINCIPE RETENU

Le réseau, pour lequel nous désirons fournir du code objet, peut être composé de matériels très divers : mini-ordinateurs, microprocesseurs, processeurs spécialisés (entrées/sorties, traitement du signal ...) et la génération est de ce fait plus compliquée que dans le cas où les machines choisies sont de caractéristiques voisines /8/, /9/, /10/, /25/.

Un générateur de code devra réaliser le travail suivant :

* Structuration et adaptation des données (variables simples, tableaux, constantes) en fonction de leur format, de leur type et de la représentation interne prévue pour le traitement dans la machine cible. Il faudra également tenir compte à ce niveau des moyens disponibles pour atteindre ces données (prévoir par exemple les relais d'adressage indirect).

* Traduction des instructions du langage intermédiaire en langage machine en fonction du répertoire d'instructions disponible sur la machine cible (il faudra éventuellement simuler les instructions n'existant pas, ce qui est le cas par exemple pour MUL et DIV sur de nombreux microprocesseurs. Cette traduction devra tenir compte du type de l'opérande manipulé, de sa représentation interne dans la machine considérée, des registres de travail disponibles et du contexte (génération éventuelle de sauvegardes de registres).

* Adaptation des fonctions d'accès aux opérandes en fonction de leur type, de leur structure, de leur représentation interne, des adressages disponibles sur le site considéré et enfin du répertoire d'instructions.

* Génération des séquences d'initialisation propres à chaque site : chargement initial de registres, bases, pointeurs, définition de pile ..., des séquences de communication avec le moniteur du réseau et avec le moniteur d'entrées/sorties (méthode de passage des paramètres).

* Gestion des conversions automatiques lors du calcul d'expressions mixtes.

* Mise à jour de la table des variables partagées en fonction des conditions d'implantation sur le site.

* Optimisation éventuelle du code, et en particulier allocation des registres /19/, /20/.

Pour atteindre ces objectifs, deux possibilités s'offraient à nous ;

* Utiliser un générateur autonome par type de site du réseau. Cette solution est peu intéressante car elle manque de souplesse en cas de modification du réseau : il faut écrire à chaque fois un nouveau générateur ce qui n'est pas toujours facile. De plus, une partie des traitements est commune aux divers sites, il y a donc redondance.

* Utiliser un générateur unique dont le fonctionnement est paramétré par les caractéristiques de la machine cible.

Nous avons opté pour cette dernière solution, et plus précisément pour sa réalisation sous forme d'un macro-processeur /25/.

Nous rappelons qu'un macro-processeur est un processeur capable d'effectuer de la substitution paramétrée de texte. Dans notre cas, il devra en outre effectuer cette substitution de manière conditionnelle, et il aura à gérer et mémoriser divers événements destinés à permettre la gestion du contexte de génération /44/.

Le macro-processeur peut nous donner directement du binaire objet, cependant nous avons préféré passer par une étape symbolique supplémentaire : le langage d'assemblage de chaque site. Cette méthode facilite la mise au point de la phase de génération et permet également de bénéficier de logiciel déjà existant.

La génération de code est alors constituée des deux étapes suivantes :

* Transposition du langage intermédiaire en langage d'assemblage du site considéré par le macro-processeur

* Assemblage (et éventuellement translation) qui nous donne le binaire exécutable.

Les macro instructions du macro-processeur sont constituées par les directives de génération, les instructions du langage intermédiaire et les structures des

opérandes intermédiaires (pour la génération de la fonction d'accès associée).

Les macro-définitions ou définition du texte et des paramètres associés aux macro-instructions constituent une bibliothèque propre à chaque site et à chaque type de code généré. En effet, les macro-définitions sont d'écriture relativement simple, on peut donc en prévoir répondant à divers critères de code généré : mode mise au point, mode rapide, mode peu volumineux ...

Il convient de remarquer cependant que ces jeux de macro-définitions doivent être écrits avec soin car c'est d'eux que dépendent en grande partie les performances du code généré, ainsi que nous le verrons sur quelques exemples.

On peut envisager, dans un but de simplification, l'automatisation de l'écriture de ces macro-définitions par un pré-traitement partant des caractéristiques de la machine cible données dans un langage de description approprié (cette simplification risque de se faire au détriment des performances) /30/, /33/.

L'algorithme de génération découle directement des choix effectués lors de l'analyse syntaxique, il est valable pour tous les sites et se combine avec l'algorithme d'allocation de registres qui est propre à chaque machine :

Si code instruction n'est pas CHA ou RAN
Alors Si instruction possède un opérande
    Alors traitement de l'opérande, génération de la méthode d'accès
        (avec si nécessaire application de l'algorithme d'allocation lors des calculs d'adresse)
        Choix du code machine et génération de l'instruction
    Sinon Si instruction précédente est CHA
        Alors utiliser comme opérande la donnée manipulée avant l'instruction CHA (on a affaire à une instruction non commutative, il faut respecter l'ordre des opérandes)
        Sinon Si instruction commutative
            Alors effectuer opération entre les deux dernières données dans l'ordre le plus commode
            Sinon effectuer opération entre les deux dernières données en respectant l'ordre d'évaluation
            fsi
        fsi
    fsi
Sinon Si instruction est CHA
    Alors effectuer le chargement de registre en fonction de l'algorithme d'allocation
    Sinon (instruction est RAN) effectuer le rangement et libérer les registres
    fsi
fsi

Cet algorithme n'est pas appliqué aux instructions spéciales : entrées/sorties, branchements, instructions implicites (n'ayant jamais d'opérande) qui sont générées directement en fonction des possibilités du site.

Le résultat de la génération de code est un fichier directement exploitable par le moniteur d'exécution et comportant les trois zones suivantes :

* le code binaire exécutable du site concerné, tel qu'il vien d'être élaboré par le générateur

* un descripteur du code (taille, point d'entrée ...)

* la table des symboles réduite provenant de la compilation et mise à jour pour permettre l'accès aux données gérées par le moniteur. Celui-ci y trouvera l'adresse, dans le module, d'un relais local qu'il initialisera avant chaque exécution avec l'adresse effective de la donnée (Fig.IV.1) /42/.

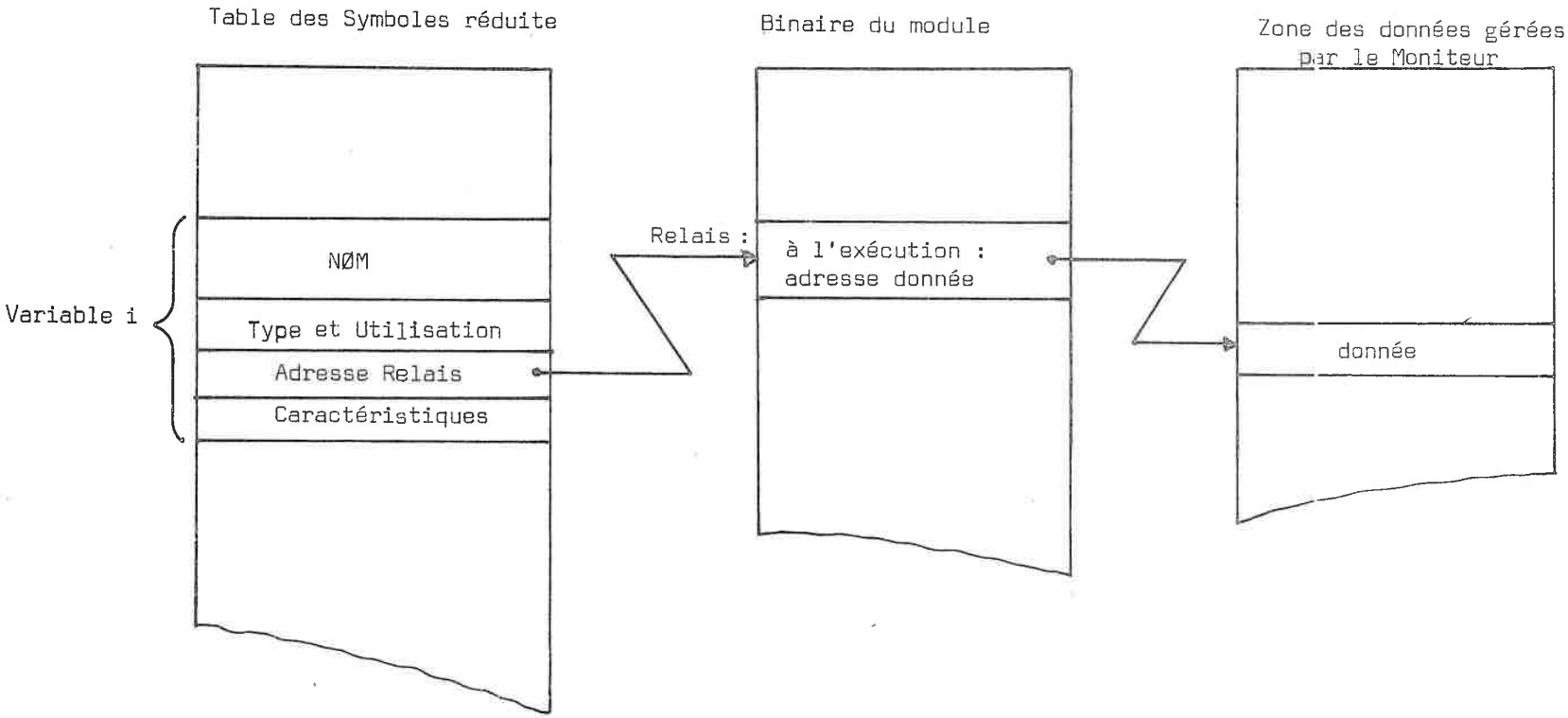


Fig.IV.1. Accès aux données gérées par le moniteur.

IV.2. ETUDE DE QUELQUES CAS DE GENERATION

Nous allons maintenant préciser sur quelques cas concrets les points esquissés dans le paragraphe précédent, en mettant en valeur les problèmes auxquels nous sommes confrontés et les compromis adoptés. Les machines étudiées ont été choisies pour leur disponibilité, permettant de tester les solutions retenues ; et pour leur diversité dans les caractéristiques que nous rappellerons brièvement.

IV.2.1. Cas du Solar 16/40 S.E.M.S.

IV.2.1.1. Présentation du matériel

Le Solar 16/40 est un minicalculateur dit "industriel" à mots de seize bits. Il dispose d'un jeu très complet d'instructions permettant des manipulations en mémoire, dans les registres, entre registres, entre registres et mémoire. Il est également capable de travailler au niveau de l'octet et du bit de manière plus restreinte /37/.

Nous y trouvons plusieurs mode d'adressage :

- relatif à une parmi trois bases
- immédiat
- indirect
- indirect post indexé
- relatif au compteur ordinal pour les branchements.

Nous disposons de quatre registres de travail :

- A : accumulateur
- B : extension d'accumulateur
- X : registre d'index ou de travail
- Y : registre de travail.

Ce site est aussi caractérisé par un processeur de virgule flottante câblé.

IV.2.1.2. Représentation des données et fonctions d'accès

Les quatre types de données sont représentés ainsi :

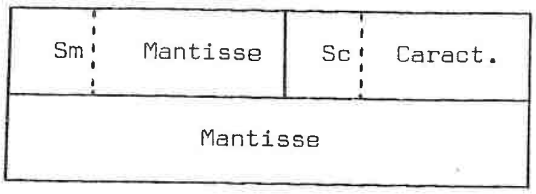
* entiers : sur un mot (16 bits), c'est la représentation naturelle sur le Solar, il n'y a donc pas de problème.

* courts : ils sont représentés sur les 8 bits de poids faible d'un mot, leur utilisation sera cependant identique à celle des entiers car les instructions sur

octet sont en nombre limité et leur utilisation serait pénalisante en temps et en occupation mémoire.

* booléens : ils seront aussi représentés sur 16 bits, les opérations logiques disponibles sur la machine travaillant sur les 16 bits en parallèle.

* réels : représentés sur 32 bits selon la convention de l'opérateur flottant :



Mantisse : 23 bits + signe (Sm)
Caractéristique : 8 bits+signe(Sc)

L'accès aux diverses variables est fonction de leur utilisation et de leur structure :

* variables simples locales : on les atteint par adressage direct (basé)

* variables simples gérées par le moniteur : on emploie cette fois l'adressage indirect via un relais positionné par le moniteur lors de l'interconnexion des modules

* tableaux locaux ou non : l'adressage indirect post indexé est utilisé. Le relais d'indirection est, selon le cas, défini localement ou par le moniteur. Le registre d'index permet de préciser le déplacement dans le tableau qui a évolué à chaque fois.

L'adressage des tableaux fait ressortir l'importance d'un choix judicieux des instructions lors de l'écriture des macro-définitions. Le contexte est également important : le calcul des déplacements dans les tableaux fait intervenir des constantes numériques ce qui peut conduire à l'utilisation d'adressage immédiat dans certain cas ; certains calculs font intervenir les registres A et B qu'il faudra sauvegarder s'ils sont utilisés, puis restaurer.

Le tableau, ci-dessous, nous donne les performances respectives de douze modes de calcul de l'adresse d'un élément de tableau : TAB(I,J). Ce calcul utilise deux constantes numériques ce qui conduit à distinguer quatre cas principaux selon que l'on peut ou non utiliser l'adressage immédiat.

Cas	1	2	3	4	5	6	7	8	9	10	11	12
Nb instr.	6	6	5	6	6	6	6	6	5	6	6	6
t(µs)	21,75	20,25	17,75	21,25	19,75	20,75	21,62	20,12	17,62	21,22	19,62	20,62
C1	< 256						> 256					
C2	< 256			> 256			< 256			> 256		

Les différences constatées peuvent paraître mineures, mais elles sont importantes compte-tenu du fait que ces calculs sont répétés de nombreuses fois et que certaines applications ont des temps de réponse critique.

Les constantes C1 et C2 correspondent respectivement à t_1 et n dans l'exemple suivant. On constate sur ce tableau que l'utilisation de l'adressage immédiat n'est pas toujours intéressante puisque les cas 7 à 12 (qui correspondent respectivement aux cas 1 à 6) donnent des temps d'exécution plus courts.

Exemple

Description de quelques séquences possibles pour le calcul de l'adresse de TAB(I,J).

Si le tableau a été déclaré de dimension t_1 et t_2, l'élément TAB(I,J) correspond à un déplacement à partir de l'adresse TAB de :

I * "t_1" + J-"n" avec n = t_1 + 1

Nous noterons :

"-" une valeur numérique

@x l'adresse de la constante x

(-) une instruction dont la présence est optionnelle (fonction du contexte) *t_1 < 256 (cas 3 et 5)

(PSR A,B) Sauvegarde éventuelle des registres A et B utilisés pour le calcul.

LAI	"t_1"	LAI	"t_1"	
MP	I	MP	I	calcul de I*t_1
LX	J	LR	B,A	
ADR	B,X	AD	J	I*t_1 + J
ADRI	"-n", X	SB	@n	
		LR	A,X	I*t_1 + J - n

(ADR X,X) Si le tableau est réel, il faut doubler le déplacement calculé car chaque élément occupe deux mots.

(PLR A,B) Restauration éventuelle des registres A et B.

* t_1 > 256 (cas 9, 11 et 12)

(PSR A,B)

LA	t_1	LA	t_1	LA	t_1	
MP	I	MP	I	MP	I	$I*t_1$
LX	J	LR	B,A	LX	J	
ADR	B,X	AD	J	ADR	B,X	$I*t_1 + J$
ADRI	"-n",X	SB	@n	LB	@n	
		LR	A,X	SBR	B,X	$I*t_1 + J-n$

(ADR X,X)
(PLR A,B)

IV.2.1.3. Stratégie d'allocation de registres

Nous avons essayé de voir l'effet d'un algorithme d'allocation de registres sur le Solar dans le but d'éviter des sauvegardes systématiques en mémoire /20/.

L'algorithme employé est le suivant :

Si instruction est CHA

Alors Si il existe un registre libre dans la liste
Alors mettre l'opérande dans ce registre
Sinon libérer un registre (sauvegarde) et y charger l'opérande
fsi

Sinon (on suppose que le premier opérande est déjà dans un registre)
Si on peut faire l'opération entre registre et mémoire
Alors on la fait
Sinon Si on peut faire l'opération entre registres
Alors Si un registre est libre
Alors on y charge le deuxième opérande et on effectue
Sinon on libère un registre (sauvegarde), on y charge le deuxième opérande et on effectue
fsi
Sinon on libère l'accumulateur (sauvegarde)
Si instruction précédente est CHA
Alors on la modifie pour avoir l'opérande dans l'accumulateur
Sinon on transfère l'opérande dans l'accumulateur
fsi
on effectue l'opération
fsi
fsi
fsi

Cet algorithme a été expérimenté sur de nombreuses évaluations d'expressions en utiiisant comme liste de registres soit A et B, soit A, B, X et Y. Les résultats furent décevants comme le montrent les deux résultats ci-dessous :

Cas	Sans allocation	Alloc. sur A, B	Alloc. sur A,B,X,Y
1	33 instructions	33	39
2	21 instructions	22	22

Ces résultats peuvent d'ailleurs s'expliquer facilement : les registres ne sont pas banalisés donc il faut effectuer de nombreux transferts entre eux. De plus, à part pour l'accumulateur, les instructions entre mémoire et registres sont limitées, il faut donc utiliser des instructions entre registres ce qui nécessite des chargements supplémentaires ; le tout est plus pénalisant que la sauvegarde systématique. Dans l'exemple 1 les deux premiers résultats sont identiques car l'expression comportait des multiplications utilisant simultanément A et B. Dans l'exemple 2 , le faible niveau d'imbrication de l'expression ne nécessitait jamais plus de deux registres, ce qui explique l'identité des deux derniers résultats.

Nous avons donc choisi pour le Solar une politique de sauvegarde systématique, d'autant plus que le registre d'index X est aussi souvent utilisé (adressage des tableaux), et que l'utilisation des nombres réels se fait obligatoirement dans A et B.

IV.2.1.4. Traduction des instructions intermédiaires

La traduction des instructions du langage intermédiaire ne pose aucun problème car chacune a un équivalent dans le langage machine Solar, quel que soit le type des données manipulées :

* utilisation des instructions sur registres, opérations d'arithmétique entière et de logique pour les entiers, courts, booléens.

* le processeur flottant dispose aussi d'un jeu complet d'instructions

* les tests et branchements existent tels quels

* les appels et retour de sous programmes peuvent se faire à l'aide des instructions BSR et RSR ; l'appel du moniteur d'exécution ou du moniteur d'entrée/sortie par l'instruction SVC. Le passage des paramètres s'effectue très simplement par la pile de travail.

* directives de génération :

  - VCOM provoque la réservation d'un relais dans la section de données COMMON

- VLOC provoque la réservation de place (1 ou 2 mots) dans la section de données LOCAL
- TLOC provoque la réservation d'un relais dans la section LOCAL et de place dans la section TABLE
- CE et CR : définition de constantes dans la section de données
- MAN : réservation de la zone de manoeuvre, de la pile de travail.

La gestion des mémoires de manoeuvre utilisées dans les sauvegardes est effectuée par le macro-processeur.

Exemple :

L'affectation R = (A+B)*(C+D)
qui donne en langage intermédiaire :

```
CHA   A
ADD   B
CHA   C
ADD   D
MUL
RAN   A
```

se traduit sur Solar par :

```
LA    A
AD    B       Calcul de A+B
STA   MV00    Sauvegarde du résultat précédent
LA    C
AD    D       Calcul de C+D
MP    MV00    (A+B)*(C+D)
LR    B,A     Récupération du résultat dans A
STA   R
```

Si l'opérateur n'est pas commutatif, il faut en outre tenir compte de l'ordre d'évaluation : avec R = (A+B)/(C+D)
la séquence en langage intermédiaire est la même en remplaçant MUL par DIV mais la traduction Solar devient :

```
LA    A
AD    B       Calcul de A+B
STA   MV00    Sauvegarde du résultat précédent
LA    C
AD    D       Calcul de C+D
XM    MV00    Permutation des opérandes
SARD  16      Cadrage du dividende
DV    MV00    (A+B)/(C+D)
STA   R
```

IV.2.1.5. Gestion des conversions de données

Le générateur assure la gestion automatique des conversions et la préparation des données :

Exemple : ENTIER I,OPER 1, OPER 2,RES(50)
RES(I)=OPER 1/OPER 2

```
L.I. :      CHA  OPER 1
            DIV  OPER 2
            RAN  RES+I-1
```

sur Solar :

```
            LA   OPER 1
            SARD 16          Préparation opérande
            DV   OPER 2
            LX   I
            ADRI -1,X        Calcul déplacement
            STA  &RELRES
```

La même affectation devient avec la déclaration : REEL OPER 2, RES(50)

```
            LA   OPER 1      Opérande entier
            FLT              Conversion
            FDV  OPER 2
            LX   I
            ADRI -1,X
            ADR  X,X         Les réels tiennent sur 2 mots
            FST  &REL RES
```

Il convient de remarquer que la génération automatique de conversions peut conduire à du code non optimal, du fait du traitement séquentiel du langage intermédiaire ; ce serait le cas dans l'exemple précédent avec OPER 1 réel et OPER 2 entier. Pour éviter cela, on dispose de deux solutions :

- le macro-processeur utilisé est capable de revenir en arrière pour annuler les opérations précédentes et générer une nouvelle séquence :

Exemple : séquence non optimale (traitement purement séquentiel)

```
            FLD  OPER 1
            FST  MV00        Sauvegarde de OPER 1
            LA   OPER 2
            FLT              Conversion de OPER 2
            FST  MV01
            FLD  MV00        Restauration de OPER 1
            FDV  MV01        Division
```

Séquence améliorée

```
LA   OPER 2
FLT              Conversion de OPER 2
FST  MV00
FLD  OPER 1
FDV  MV00        Division
```

- on ajoute une phase d'optimisation après la génération de code (c'est la seule méthode possible pour nous, avec le macro-processeur actuellement utilisé)/44/

IV.2.2. Cas du Motorola M6800

IV.2.2.1. Caractéristique du matériel

Le Motorola M6800 est un microprocesseur 8 bits. Il possède un jeu assez complet d'instructions travaillant au niveau de l'octet et quelques unes travaillant sur 16 bits (index, pile, ...) /27/.

Les adressages disponibles sont :

- direct dans la première page mémoire (256 octets)
- étendu dans toute la mémoire (64 K octets)
- immédiat
- indexé, le décalage par rapport à l'index étant toujours positif et inférieur à 256
- relatif au compteur ordinal pour les branchements.

Les registres utilisables sont :

- A et B deux accumulateurs entièrement banalisés de 8 bits chacun
- X registre d'index de 16 bits

Le M6800 ne dispose pas des opérations multiplication et division ni du flottant.

IV.2.2.2. Représentation des données

Les différents types de données seront représentés ainsi :

* entiers : sur 16 bits c'est à dire 2 octets consécutifs
* courts : sur un octet
* booléens : sur un octet puisque les instructions logiques travaillent sur 8 bits en parallèle
* réels : pour avoir une précision suffisante, il faudrait les représenter sur 4 octets, on obtiendrait alors des traitements particulièrement longs et compliqués ce qui nous a conduits à interdire l'emploi des réels sur le 6800. La répartition des modules devra être faite en tenant compte du besoin éventuel de cette ressource.

Les variables simples locales sont atteintes par adressage étendu (éventuellement direct) et les variables simples gérées par le moniteur par adressage indexé en chargeant l'index à l'aide du relais positionné par le moniteur.

Pour les tableaux locaux ou non on emploie également l'adressage indexé, mais compte tenu des restrictions de ce mode d'adressage, ce n'est pas le déplacement dans le tableau qui figure dans l'index, mais l'adresse réelle de l'élément de tableau qui doit être calculée à chaque fois. Ces calculs d'adresse se font sur seize bits et doivent utiliser les accumulateurs nécessitant généralement une sauvegarde.

Comme dans le cas du Solar, l'adressage immédiat sera utilisé pour les constantes chaque fois qu'il sera possible et rentable, on voit par contre apparaître l'importance de la déclaration "court" puisqu'un tel élément est manipulé sans problème par le 6800 ce qui n'est pas le cas d'un entier de seize bits. Reprenons l'exemple déjà étudié du calcul d'adresse de l'élément de tableau TAB(I,J) par la formule

$$TAB + I^{\times} \text{ "}t_1\text{" } + J - \text{"}n\text{"}$$

selon diverses valeurs des constantes et diverses déclarations des indices

```
( PSH  A )
( PSH  B )      Sauvegardes éventuelles

  LDX  #t1      (sur 16 bits)
```

LDA	A	I	(poids forts)	I entier	CLR	A	I court
LDA	B	I+1	(poids faibles)		LDA	B	I

```
  BSR  SPMUL    (multiplication)
```

ADD	B	J+1	J entier		ADD	B	J	J court
ADC	A	J			ADC	A	#0	
SUB	B	n+1	n ⩾ 256		SUB	B	#n	n < 256
SBC	A	n			SBC	A	#0	

```
( ASL  B )    Si tableau d'entiers, il faut
( ROL  A )    doubler le déplacement

  ADD  B   TAB+1
  ADC  A   TAB
  STA  B   INDEX+1   |
  STA  A   INDEX     | changement de l'index
  LDX      INDEX     |

( PUL  B )
( PUL  A )    Restauration si nécessaire
```

nous obtenons alors les performances suivantes : (sans les séquences parenthésées)

Nbre cyles	50	46	48	44	48	44	46	42
Nbre octets	36	34	35	33	34	32	33	31
n	≥256	<256	≥256	<256	≥256	<256	≥256	<256
J	entier		court		entier		court	
I	entier				court			

compte tenu des sauvegardes et du doublement éventuels, l'écart entre le cas le plus favorable et le cas le plus défavorable représente 20 cycles et 9 octets, ce qui est loin d'être négligeable.

IV.2.2.3. L'allocation de registres

Le problème de l'allocation des registres est très simple sur le Motorola : on ne dispose pratiquement pas d'instructions sur le registre X, il est donc inutilisable comme registre de travail, de plus il est monopolisé par l'adressage des variables partagées et des tableaux.

En ce qui concerne A et B, si on travaille sur des entiers les deux sont utilisés comme un seul accumulateur de 16 bits, on est donc conduit à effectuer des sauvegardes systématiques de la paire A, B. Lorsqu'on utilise des variables courtes ou booléennes, A et B étant banalisés, on cherchera par contre à les occuper au maximum.

C'est ainsi que le calcul de l'expression faisant l'objet du cas n° 1 sur le Solar est traduite avec 27 instructions sur le 6800 avec allocation de A et B et 30 si on utilise un seul accumulateur (en supposant que toutes les variables sont courtes). Le temps, que nous cherchons à optimiser en priorité, varie dans les mêmes proportions.

IV.2.2.4. Traduction des instructions intermédiaires

Nous rencontrons cette fois quelques problèmes :

* certaines instructions du langage intermédiaire n'existent pas sur le Motorola
* certaines instructions n'existent que pour un nombre limité de types de données.

Dans le premier cas, nous trouvons les opérations MUL et DIV, on est alors obligé de simuler ces opérations par l'intermédiaire de sous-programmes. Le passage des paramètres s'effectue de façon simple : l'un dans le registre d'index (16 bits),

l'autre dans A et B formant un registre double longueur. Le sous programme n'est prévu que pour des nombres de 16 bits, pour des nombres courts, on forcera à 0 les poids forts.

Toutes les instructions portant sur des variables de type entier se trouvent dans le second cas ; elles seront simulées par une association d'instructions sur octet assurant la même fonction :

Exemple : chargement de variable entière

```
LDA  A  Var
LDA  B  Var + 1
```

addition d'une variable entière à A, B

```
ADD  B  Var + 1
ADC  A  Var (propagation de la retenue)
```

Les instructions de test, branchement, appel/retour de sous programme existent telles quelles ; pour les appels moniteur, on pourra utiliser l'instruction SWI ou des branchements à des adresses fixes. Le passage des paramètres de 8 ou 16 bits est possible par la pile de travail.

Le traitement des directives de génération est effectué ainsi :

- VCOM : réservation d'un pointeur de 2 octets : adresse de la variable
- VLOC : réservation de 1 ou 2 octets selon le type de la variable
- TLOC : réservation de 2 octets qui contiennent l'adresse du tableau, suivis des octets de ce tableau
- MAN : réservation de la mémoire de manoeuvre et de la pile de travail
- CE : réservation d'une constante entière sur 2 octets
- CR n'est pas autorisée.

La gestion des mémoires de manoeuvre est effectuée par le macro-processeur, de la même manière que pour le Solar, compte tenu de la disponibilité des registres.

Le problème des conversions est grandement simplifié du fait de l'absence des nombres réels ; il ne reste plus que la conversion du format court en format entier qui s'effectue très facilement par concaténation d'un octet nul.

Exemple : RES = OPER 1 + OPER 2
Si on a déclaré OPER 1 entier et OPER 2 court, nous obtenons :

```
LDA  A  OPER 1          |
LDA  B  OPER 1 + 1      |Chargement double longueur

ADD  B  OPER 2          |Addition et conversion double longueur
ADC  A  #0              |de OPER 2

STA  B  RES+1           |
STA  A  RES             |Rangement double longueur
```

Avec les déclarations inverses, nous aurions :

```
LDA  B  OPER 1          Chargement simple
CLR  A                  Conversion double longueur

ADD  B  OPER 2+1        |
ADC  A  OPER 2          |Addition double longueur

STA  B  RES+1           |
STA  A  RES             |Rangement double longueur
```

IV.2.3. Cas du Intel 8080

IV.2.3.1. Caractéristiques de la machine

Le 8080 de Intel est aussi un microprocesseur 8 bits. Il dispose d'un accumulateur unique A de 8 bits et d'un jeu de 8 registres 8 bits groupables par paire/28/.

Les adressages possibles sont :

- immédiat
- direct en mémoire
- "indirect par registres" avec utilisation de deux paires de registres (adresses de 16 bits)

Les instructions permettent d'effectuer diverses opérations sur l'accumulateur, ainsi que des transferts entre registres et entre registres et mémoire. Quelques instructions double longueur sont disponibles au niveau des paires de registres de travail ; elles sont cependant très limitées (ainsi on dispose d'une addition mais pas d'une soustraction).

IV.2.3.2. Représentation des données

La représentation des données sur le 8080 est identique à celle du 6800 :

- entiers : sur 2 octets consécutifs
- courts : sur 1 octet
- booléens : sur 1 octet
- réels : non implantés

l'accès aux diverses variables est également similaire :

- variables simples locales : adressage direct en mémoire
- variables partagées : adressage "indirect par registres"
- tableaux locaux : adressage "indirect par registres"

dans le cas de l'adressage d'éléments de tableaux, nous sommes encore obligés de calculer à chaque fois l'adresse réelle de l'élément de tableau avant de la charger dans les registres d'adressage indirect. Ce calcul pose souvent des problèmes, car l'adresse est sur 16 bits ; or nous ne disposons que d'un seul accumulateur 8 bits et les instructions double longueur disponibles sur les registres de travail sont insuffisantes pour de tels calculs. Nous serons donc obligés d'effectuer de nombreux transferts qui nous pénaliseront.

Reprenons, à titre de comparaison, l'exemple déjà traité sur les autres machines, à savoir le calcul de l'adresse de l'élément TAB(I,J).

Nous aurons, comme dans le cas des autres sites, une influence de la nature des variables et des constantes utilisées, cependant cette influence sera plus limitée du fait d'un certain nombre de traitements effectués de manière systématique en 16 bits. Nous ne traiterons l'exemple que dans un cas pour un tableau de courts puis pour un tableau d'entiers (doublement du déplacement).

Exemple : TAB + I* "t_1"+J-"n"

Tableau de courts

```
LXI  D,"t1"      |
LHLD I           | Calcul de I*t1
CALL SPMUL       |

LHLD J           |
DAD  D           | Calcul de I*t1+J

XCHG               Positionnement des opérandes

LHLD TAB         |
DAD  D           | Calcul de TAB + J*t1+J

MOV  L,A         |
SUB  @n+1        |
STA  RELAIS+1    | Calcul de TAB+I*t1+J-n à l'aide d'opérations
MOV  H,A         | 8 bits (on ne dispose pas de soustraction 16 bits)
SBB  @n          |
STA  RELAIS      |
LHLD RELAIS      |
XCHG             | Préparation du relais d'indirection
```

Tableau d'entiers

LXI	D,"t_1"	
LHLD	I	Calcul de I*t_1
CALL	SPMUL	
LHLD	J	
DAD	D	Calcul de I*t_1+J
DAD	H	Doublement de I*t_1+J
XCHG		
LHLD	TAB	
DAD	D	Calcul de TAB + 2(I*t_1+J)
LDA	@ n	
CMA		
MOV	A,D	
LDA	@ n+1	Calcul de -n (par complémentation sur 8 bits)
CMA		
MOV	A,E	
INX	D	
DAD	D	2 fois
DAD	D	Addition de (-n) à TAB+2(I*t_1+J)

IV.2.3.3. Traduction du langage intermédiaire

Le 8080 n'ayant qu'un accumulateur, nous serons obligé d'effectuer des sauvegardes systématiques lors de la traduction des instructions intermédiaires. Cette traduction présente quelques problèmes :

- instructions arithmétiques et logiques : elles se traduisent toutes directement pour des opérandes 8 bits sauf MUL et DIV pour lesquelles on effectue une simulation par sous-programme. L'instruction de complémentation à 2 (négation) n'existe pas non plus, on la réalise en utilisant la définition :

Complément à 2 = Complément à 1 (instruction existante) + 1

En ce qui concerne les opérandes 16 bits, le problème est plus délicat, en effet il n'est pas rentable d'utiliser les instructions double longueur vu leur faible nombre et leurs conditions restreintes d'application. Comme on ne dispose que d'un seul accumulateur, nous serons obligés d'effectuer les traitements en deux passes avec utilisation éventuelle d'une mémoire tampon.
Ainsi une addition en 16 bits donnera :

```
LDA  OPER 1+1 (poids faibles)
ADD  OPER 2+1 (poids faibles)
STA  --------
LDA  OPER 1 (poids forts)
ADC  OPER 2 (poids forts)
STA  ------
```

En ce qui concerne les tests et branchements, il n'y a pas correspondance systématique entre les instructions intermédiaires et les instructions machine, il faudra donc dans certains cas utiliser plusieurs instructions du 8080 pour obtenir l'effet désiré :

Exemple

On dispose sur le 8080 des instructions :

JP	Branchement si positif ou nul
JZ	Branchement si nul
JNZ	Branchement si non nul
JM	Branchement si négatif

L'instruction intermédiaire branchement si positif pourra se traduire par :

```
      JZ   APRES
      JP   -----
APRES
```

D'autre part, les instructions de tests et de branchement ne sont pas indépendantes du contexte, il faudra donc effectuer généralement des tests supplémentaires pour obtenir les résultats attendus.

Pour les sous-programmes, il n'y a pas de problèmes : nous disposons de toutes les instructions nécessaires. Le passage des paramètres peut se faire par la pile ou dans certains cas (multiplication par exemple) par les registres de travail.

Les directives de génération sont traitées comme sur le 6800, nous n'y reviendront donc pas ; il en va de même pour les conversions automatiques entre types.

IV.2.4. Cas du Signétics 2650

Le Signétics 2650 est encore un microprocesseur 8 bits. Il ne dispose que d'instructions sur un octet en un jeu assez complet (avec cependant l'absence de multiplication et division). Il travaille sur un accumulateur principal et, de manière plus restreinte, sur deux jeux de trois registres auxiliaires utilisables

en bascule. Ces registres auxiliaires peuvent également être utilisés comme registres d'index /54/.

Les adressages disponibles sont :

- immédiat
- relatif
- absolu dans une page mémoire de 32Koctets (simple et indexé)
- absolu dans 64 K octets pour les branchements
- indirect simple ou post indexé.

La génération de code sur un tel microprocesseur peut se faire assez facilement compte tenu des règles présentées pour le 6800 et surtout pour le 8080. Il est à noter cependant que ce matériel ne dispose que d'une pile à 8 niveaux limitant fortement l'imbrication des sous-programmes (il faut garder des niveaux pour les interruptions), cette limitation risque de poser des problèmes lors de l'implantation du moniteur réseau et du moniteur d'entrée/sortie.

IV.3. ASSEMBLAGE ET TRAITEMENTS ANNEXES

La phase d'expansion des macro-instructions que nous venons d'examiner dans le cadre de divers processeurs nous donne un module écrit dans le langage d'assemblage de chaque site. Il nous reste alors à traduire ce dernier en code binaire exécutable.

Nous pouvons effectuer l'assemblage sur le site de production ou sur le site d'exécution, chaque méthode ayant ses avantages et ses inconvénients.

Dans le cas d'un assemblage sur le site d'exécution, nous pouvons utiliser l'assembleur fourni par le constructeur pour sa machine ainsi que les divers processeurs logiciels dont la mise en oeuvre est généralement nécessaire après assemblage. Par contre cette méthode oblige à transférer, sur les liaisons intersites, du texte symbolique, ce qui se traduira nécessairement par une forte occupation de celles-ci au détriment du reste de l'application.

L'assemblage sur le site de production évite l'inconvénient de l'encombrement des lignes de transmission, mais nécessite de disposer d'assembleurs croisés pour les divers processeurs.

Les raisons qui nous ont fait opter pour une production de programmes sur un site unique peuvent cependant s'appliquer encore à ce niveau et imposer l'assemblage sur le même site. Ainsi parmi les machines précédemment étudiées, seul le Solar est capable, de par sa configuration, d'effectuer cette phase d'assemblage ; nous l'avons donc muni de divers assembleurs croisés pour toute une gamme de microprocesseurs (exemple : /45/).

Après l'assemblage, le binaire obtenu est généralement translatable, il conviendra donc de réaliser une translation d'adresses avant de pouvoir implanter le code objet sur le site d'exécution, c'est le rôle du chargeur translateur qui peut de même s'exécuter sur le site d'assemblage ou sur le site d'exécution (si ce n'est pas le même).

Le dernier traitement à effectuer est la mise à jour de la table des symboles réduite afin de compléter, pour chaque variable gérée par le moniteur, l'adresse effective (après assemblage et translation) du relais utilisé par le module.

Le module est maintenant prêt à être pris en compte par le moniteur d'exploitation.

IV.4. SYNTHESE DES DIVERS CAS DE GENERATION

L'étude de la génération de code objet effectuée précédemment nous permet de tirer des indications utiles tant sur le plan du choix des matériels que sur celui de la méthode mise en oeuvre.

Nous constatons d'abord l'importance de l'adaptation des données manipulées aux caractéristiques du site. Ainsi le traitement des nombres réels doit être réservé à des machines suffisamment puissantes pour effectuer ce genre de calcul (ou à des machines munies d'un processeur spécialisé).

Le traitement des tableaux joue aussi un rôle important dans les performances du code obtenu, nous constatons ainsi qu'un miniordinateur l'exécute dans de bien meilleures conditions qu'un microprocesseur à cause de ses possibilités d'adressage plus riches.

La taille des opérandes traités influe également : sur microprocesseurs, l'emploi de variables entières nécessite l'utilisation d'instructions double longueur. On remarquera à ce niveau que les choix effectués par les constructeurs rendent certaines machines mieux adaptées à notre cas ; ainsi le Motorola 6800 permet de travailler simplement sur 16 bits grâce à ses deux accumulateurs banalisés.

Les points que nous venons de souligner sont importants par leur influence sur les performances globales des modules, car ils interviennent généralement avec un fort taux de répétition, ils ne sont cependant pas les seuls à prendre en compte et il faudra examiner le contexte particulier de chaque application.

Nous conclurons sur ce sujet en remarquant que le projet Sygare permet de concevoir une application indépendamment de sa répartition mais que celle-ci influe largement sur les caractéristiques d'exécution. Cela n'est pas contradictoire puisque l'implantation des modules s'effectue après la description et la programmation de l'ensemble. C'est à ce niveau que l'on devra tenir compte des performances et des critères indiqués, chaque fois que les sites ne seront pas imposés par des

contraintes physiques extérieures (temps de réponse, périphériques spécifiques, ..)

En ce qui concerne la méthode employée, nous avons pu constater sur de nombreux exemples que le code produit par le macro-processeur est bien optimisé et répond aux conditions que nous nous étions fixées, sauf dans quelques rares cas (comme les conversions de types dans l'évaluation des expressions) qui peuvent éventuellement être améliorés par une phase ultérieure d'optimisation.

Nous pouvons donc estimer que les résultats obtenus sont satisfaisants et que le choix d'une génération paramétrée de code objet n'est pratiquement pas pénalisant , à ce point de vue, par rapport à l'utilisation de compilateurs séparés pour chaque site. Ces résultats sont liés pour une grande part au soin apporté à l'écriture des macro-définitions.

Remarquons enfin que le langage intermédiaire, qui s'était déjà révélé bien adapté pour la traduction du langage source, ne pose pas de problèmes particuliers pour son adaptation aux différents sites envisagés.

IV.5. ECRITURE DES MACRO-DEFINITIONS

Le jeu de macro-définitions nécessaires pour chaque site comprend une définition par instruction ou directive du langage intermédiaire, qui permettra de générer les instructions machine correspondantes ; et une définition par structure d'opérande afin de mettre en oeuvre le calcul d'adresse correspondant.

Nous allons mettre en évidence sur un exemple leur écriture et leur utilisation :

Exemple :

Génération sur le Solar de l'instruction intermédiaire :

```
RAN     RES+I-1
```

Nous allons utiliser ici deux macro-définitions traitant l'instruction RAN et la structure ...+ ...-1

```
*DEF   RAN    P1          Macro-définition de l'instruction RAN avec opérande
                          formel P1
*"P1"                     Traitement de l'opérande effectif par appel de la
                          macro-définition correspondant à sa structure
* Si "P1" est réel
  Alors code objet est FST   Choix du code objet adapté au type de l'opérande
  Sinon code objet est STA
  fsi
```

* Génération de l'instruction
  "code objet" "opérande machine"

* FINDEF

* DEF P1+P2-1 — Macro-définition liée à la structure ... + ... -1 avec les paramètres formels P1 et P2

* "P1" — Traitement de la variable P1 (acquisition des informations la concernant : type, adressage, relais ...)

* Génération de : — On effectue le calcul d'adresse

```
LX      "P2"
ADRI    -1,X
```

* Si "P1" est réel
  Alors générer

```
ADR     X,X
```

pour doubler le déplacement

  fsi

* Composition de l'opérande machine

"Op.machine" → &Relais d'adressage
(& est le symbole de l'indirection)

* FINDEF

L'exécution de ces macro-instructions va se faire de manière imbriquée et donnera le texte final :

```
LX      I
ADRI    -1,X
ADR     X,X
FST     &RELRES
```

Si RES est déclaré en réel
ou :

```
LX      I
ADRI    -1,X
STA     &RELRES
```

Si RES est d'un autre type.

L'exemple que nous venons d'étudier est en fait un cas très simplifié. L'écriture réelle des macro-définitions est beaucoup plus complexe car il faut alors gérer toute une série de variables internes servant à mémoriser le contexte et gérer les zones de travail.

Nous devons entre autres savoir à tout moment l'état d'occupation des registres afin de générer si nécessaire les sauvegardes en mémoire de manoeuvre, savoir quelles sont les mémoires de manoeuvre utilisées et ce qu'elles contiennent.
Nous devons également connaître le type de l'expression en cours d'évaluation afin de générer les conversions nécessaires si le nouvel opérande utilisé n'est plus directement compatible.

Un problème important, posé par l'utilisation du macro-processeur, est son incapacité à gérer une table des symboles, ce qui est pourtant nécessaire pour acquérir les caractéristiques de chaque variable (type, organisation,...).

Nous avons tourné cette difficulté en ajoutant, à la famille des macro-définitions présentées ci-dessus, un jeu de macro-définitions temporaires dont la durée de vie est la phase de génération de code d'un module, et qui correspondent aux différentes variables utilisées par ce module ; créant ainsi une pseudo table des symboles exploitables par le macro-processeur. Les caractéristiques de chaque variable sont alors communiquées par l'intermédiaire de variables internes.

La création de ces macro-définitions supplémentaires est effectuée par le macro-processeur grâce à un traitement préalable des diverses directives de génération de variables que nous avons étudiées (VLOC, TLOC, VCOM).

Nous signalerons enfin un problème rencontré en ce qui concerne la gestion des noms. En effet, le macro-processeur fournissant du langage d'assemblage, il faut respecter certaines règles en ce qui concerne les identificateurs de variables ; l'introduction d'identificateurs supplémentaires lors de la phase de génération (relais d'adressage indirect par exemple) risque de provoquer des doubles définitions ou des incohérences, on peut être conduit à effectuer alors des renominations systématiques (cas de la génération sur Solar).

Une grande partie des contraintes exposées ici est due à l'utilisation d'un macro-processeur pré-existant qui n'a pu être conçu pour cette utilisation. La création d'un macro-processeur spécifique fournissant du code symbolique ou même directement du code binaire simplifierait grandement l'écriture des macro-définitions et améliorerait encore le code obtenu en permettant l'optimisation directe des séquences générées.

CHAPITRE V

QUELQUES PARTICULARITES DE SYGARE

CHAPITRE V

QUELQUES PARTICULARITES DE SYGARE

V.1. LES OPERATIONS D'ENTREE/SORTIE DANS SYGARE

Nous avons décrit, lors de la présentation du langage source de Sygare (& II.2), les ordres d'entrées-sorties disponibles. Ces ordres sont uniques, quel que soit le type du périphérique concerné.

Les échanges avec les périphériques industriels sont généralement très délicats. En effet, ceux-ci sont bien souvent très spécialisés et n'obéissent à aucun standard aussi bien en ce qui concerne la forme des échanges (format des données ...) que le mode de fonctionnement de ces échanges (programmation spécifique du périphérique, synchronisation de l'échange, gestion de signaux de contrôle, ...). De ce fait nous devrons communiquer au gérant du périphérique, lors d'une demande d'échange, un certain nombre d'informations sur le mode de fonctionnement et la nature du travail désirés.

Pour cette raison, nous avons adjoint aux ordres d'entrée/sortie proprement dits, des spécifications de type "FORMAT" permettant à l'utilisateur de préciser les paramètres de l'opération d'entrée/sortie demandée /1/.

Ces spécifications de FORMAT peuvent être utilisées avec la périphérie traditionnelle. Elles concerneront alors généralement la mise en page des données (elles jouent alors un rôle analogue aux spécifications FORMAT de FORTRAN).

Lors de l'utilisation de périphériques spécialisés les paramètres du FORMAT serviront essentiellement à préciser les points suivants :

* Mode de fonctionnement physique du périphérique

  Exemple : - résolution d'un convertisseur analogique-numérique programmable
  - cadence d'échantillonnage d'une entrée
  - mode de synchronisation de l'échange

* Conversion des données

  - changement d'unités, application d'un facteur d'échelle, adaptation de la représentation interne du site à la représentation acceptée par le périphérique.

* Traitements spéciaux

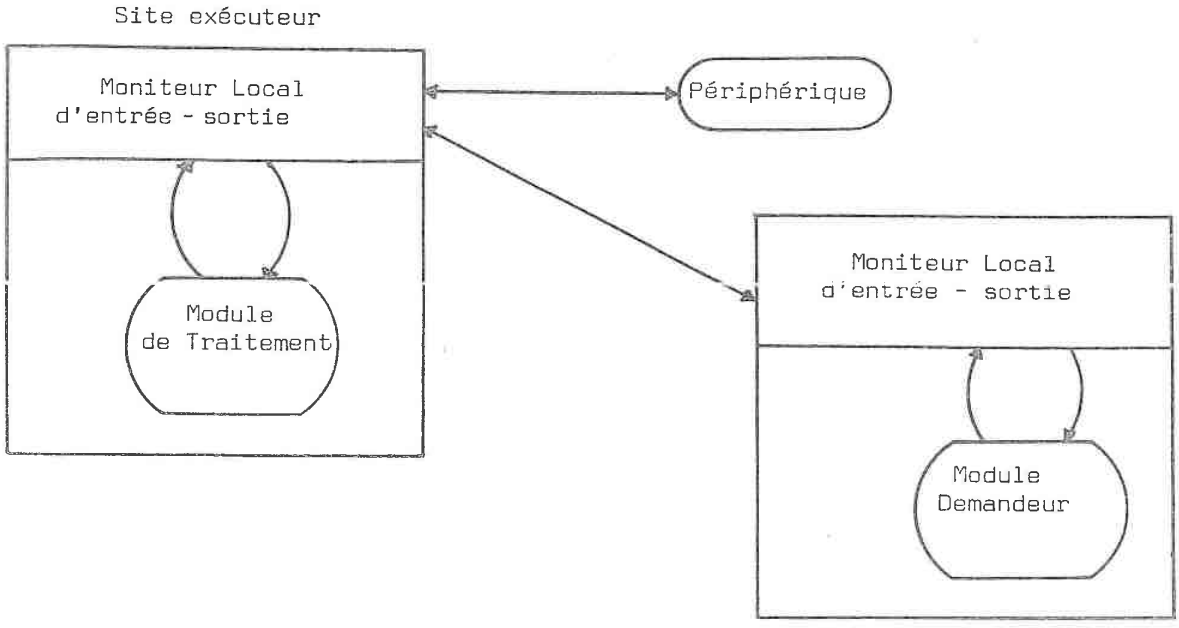
Exemple : - linéarisation
- filtrage
- comparaison à des seuils

Les fonctions et traitements demandés pourront être réalisés par le moniteur d'entrée-sortie lui-même, s'ils appartiennent à ce qui est disponible en standard /1/. Ils peuvent également être effectués par l'intermédiaire d'un module de l'utilisateur écrit à cet effet et pouvant s'exécuter sur le site demandeur de l'échange ou sur un autre. L'activation de ce module est assurée par le moniteur d'entrée-sortie /1/, /46/.

Il est intéressant de remarquer que certains efforts sont actuellement en cours pour essayer de normaliser les interfaces avec la périphérie spécialisée et les traitements associés; on notera entre autres les travaux de l'Instrument Society of America /32/ et les normes CAMAC utilisées dans l'industrie nucléaire.

Les instructions d'entrée-sortie permettent d'atteindre tous les périphériques disponibles sur le réseau. Ils sont traités par le moniteur d'entrée-sortie local du site d'exécution du module, qui établira, si nécessaire, les liaisons avec ses homologues des sites concernés par les échanges. Les moniteurs locaux gèreront également les traitements standards des formats et les passages de paramètres aux modules de traitement de l'utilisateur.

Exemple : mise en oeuvre d'une opération d'entrée-sortie.



Les périphériques du réseau sont référencés, dans les instructions d'entrée-sortie, par l'intermédiaire "d'unités symboliques" réassignables par l'utilisateur /35/, cela permet une reconfiguration de l'environnement d'entrées-sorties de l'application, en cas d'évolution de celle-ci, ou de défaillance de matériel.

Lorsque nous effectuons des demandes d'entrée-sortie dans un contexte mono-processeur, les situations conflictuelles sont généralement évitées grâce au moniteur d'entrée-sortie unique gérant l'ensemble de la périphérie. Dans le cadre d'une application répartie, nous pouvons cependant aboutir à un interblocage global sans pour autant avoir de conflit local sur un quelconque des sites mis en cause.

Diverses méthodes existent pour résoudre ces problèmes. L'une d'elles consiste à déclarer, à un allocateur global, toutes les ressources dont on aura besoin ; cette méthode, valable pour le traitement par lot, n'est guère envisageable en contrôle de processus industriels.

On emploiera alors plutôt l'allocation par "réquisition", chaque module s'appropriant un périphérique lorsqu'il désire l'utiliser, avec interposition d'une file d'attente pour l'accès à ce périphérique. Dans cette méthode, l'interblocage est résolu de manière autoritaire par le système d'exploitation /26/.

Nous ne considérons, pour l'instant, que des périphériques entièrement locaux à un site. C'est ainsi que dans le cas des fichiers (qui sont considérés comme des périphériques), nous n'envisageons pas la gestion de fichiers répartis. Ce choix devrait diminuer fortement les risques d'interblocage qui pourront alors de toute façon être résolus au niveau de chaque site.

La synchronisation et la communication entre les divers sites seront gérées par le moniteur d'entrée-sortie à l'aide des outils fournis par le moniteur d'exploitation /42/.

Lorsque le site demandeur et le site exécuteur sont différents, les délais nécessités par la transmission des informations entre les deux processeurs peuvent ne pas être négligeables, il conviendra de s'assurer qu'ils ne sont pas prohibitifs vis à vis des constantes de temps de l'application. Ce problème sera à examiner lors de la répartition des modules sur le réseau.

V.2. REENTRANCE DES MODULES

Le code produit à l'heure actuelle par le générateur de code de Sygare n'est pas réentrant. Lorsque plusieurs tâches parallèles demandent l'exécution d'un même module, plusieurs cas peuvent se produire :

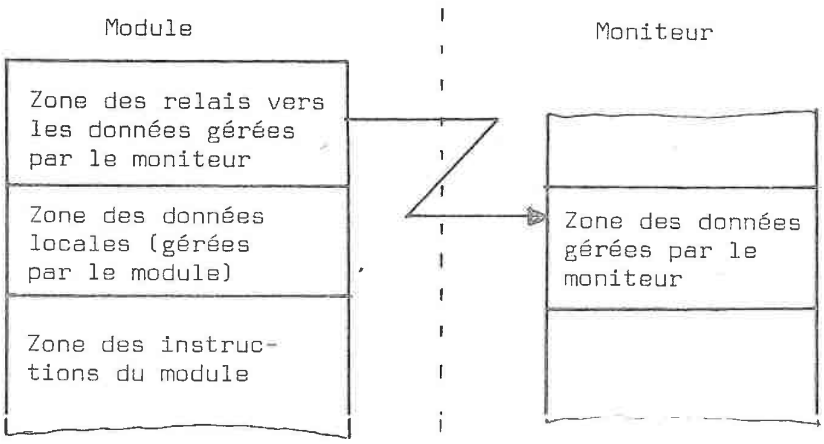
- on réalise la duplication du module sur le même site ou sur des sites différents
- on utilise une seule fois le module qui devient alors une section critique, et on met en attente les tâches qui en demandent l'activation alors qu'il est déjà en exécution.

Cette dernière méthode est à utiliser avec précaution car elle peut conduire éventuellement à des temps de réponse trop longs si l'on est obligé de bloquer une tâche de forte priorité.

Rendre un module réentrant consiste à le faire travailler sur différents jeux de données, chacun correspondant à une activation particulière. Dans Sygare, ce problème est compliqué du fait des deux catégories de données manipulées :

- les données gérées par le module
- les données gérées par le moniteur

Le schéma ci-dessous rappelle l'organisation des diverses zones utilisées :



L'attribution d'un nouvel espace de travail à un module correspond donc à lui donner trois nouvelles zones :

- une zone de relais
- une zone de données propres
- une zone de données moniteur

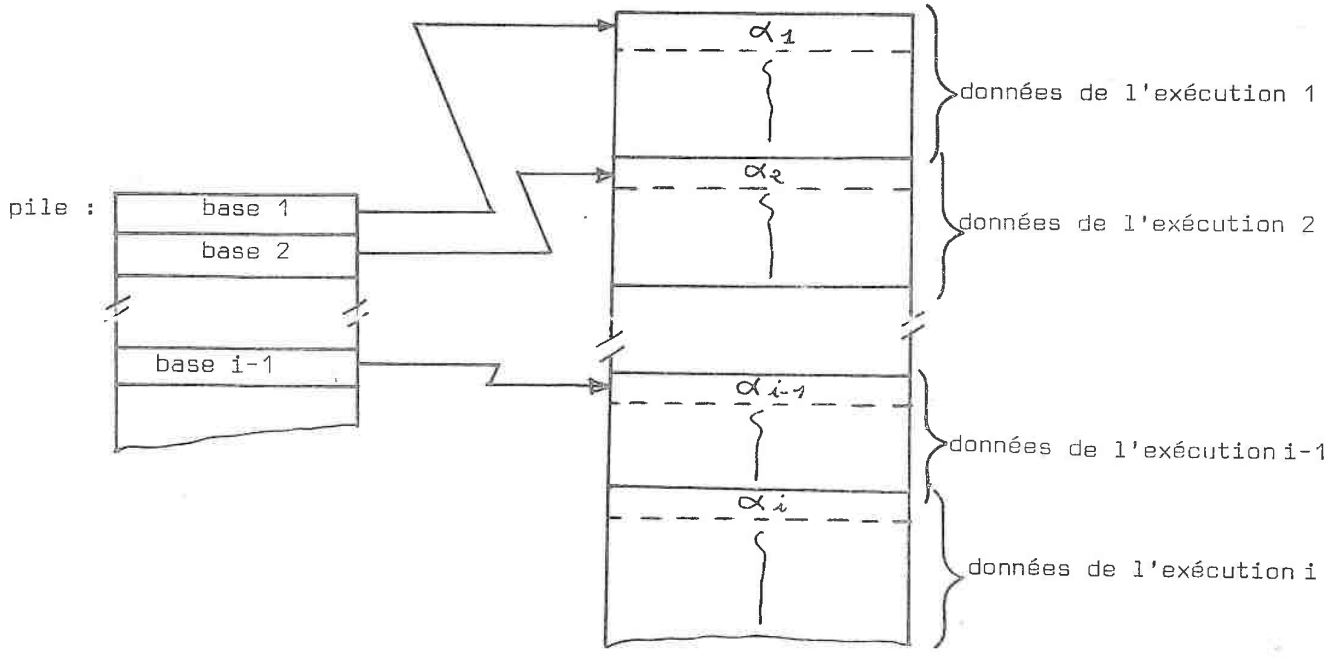
La gestion de chacune de ces zones est effectuée de manière spécifique. Les mécanismes mis en oeuvre vont être les suivants :

* Zone de données propres : cette zone est réservée à l'intérieur du module qui y accède directement, la réentrance sera assurée par la méthode habituelle d'empilement à n niveaux, n étant le degré de réentrance.

* Zone de relais : cette zone est également réservée à l'intérieur du module qui y accède directement ; nous employons de même un empilement à n niveaux. Un problème particulier se pose pour l'initialisation des relais qui est effectuée par le moniteur lors de l'activation du module ; celui-ci doit en effet connaître à tout instant le nombre d'activations en cours pour savoir quelle famille de relais il doit mettre à jour, il devra conserver ce paramètre avec le descripteur de module.

* Zone des données moniteur : à chaque activation d'un module correspond un jeu particulier de données gérées par le moniteur : l'accès à celles-ci est automatiquement assuré par l'initialisation des relais. Il est de la responsabilité du moniteur de contrôler l'emploi cohérent de ces données lors des interconnexions entre modules.

Lors de la production de programmes, nous devrons nous préoccuper uniquement de la gestion des zones de relais et de données propres. On remarquera que certaines machines se prêtent mieux que d'autres à la génération de modules réentrants. Ainsi le Solar, qui dispose d'un adressage basé, permet d'effectuer l'empilement nécessaire par simple modification du registre de base dont les précédentes valeurs sont sauvegardées.



Cette méthode permet d'accéder de manière directe aux diverses informations, lors d'une exécution, ce qui accroît les performances par rapport à d'autres machines, telles les microprocesseurs, qui nécessitent une gestion effective, dans le programme, de l'empilement des données.

Le choix d'un code réentrant ou non intervient au niveau de la phase de génération, cela permet d'écrire, au moins pour certains sites, une famille de macro-définitions pour chaque type de code. L'utilisateur peut alors produire chaque module en fonction de ses besoins par l'intermédiaire des options du générateur.

Notons enfin que l'utilisateur a toujours la ressource de programmer lui-même la réentrance de ses modules, lorsque cela n'est pas assuré automatiquement. Il peut ainsi rajouter un indice à ses diverses variables, cet indice correspondant aux différents niveaux d'activation du module. La conservation de cet indice doit alors être assurée par le moniteur et il faudra pour cela le déclarer en ENTREE et SORTIE.

V.3. EXTENSION DU RESEAU - PORTABILITE DU COMPILATEUR

Lors de la mise en oeuvre d'une application répartie se pose parfois le problème d'une modification du réseau (remplacement de matériel, extension de la configuration, ...). L'utilisateur rencontre alors deux difficultés :

* La connexion du nouveau processeur au réseau
* La modification de la répartition de son application pour prendre en compte les nouveaux sites.

En ce qui concerne ce dernier point, il est résolu très facilement dans Sygare. Nous avons vu en effet que l'intérêt du générateur de code sous forme de macro-processeur est précisément la simplicité d'écriture de nouveaux jeux de macro-définitions ; il suffit donc de complétér la famille de celles-ci en fonction des caractéristiques du nouveau site.

Le premier point est plus délicat à résoudre et le système Gare n'apporte pas de solution miracle. Il peut par contre simplifier cette phase de raccordement au réseau en permettant au concepteur d'utiliser les mêmes outils que pour toute autre application.

Un sujet actuellement à l'ordre du jour est la portabilité des compilateurs. Ce problème, très important lorsque chacun des divers sites doit produire son propre binaire exécutable à partir du même langage source, est relégué au second plan

dans Sygare puisque la production de programmes est assurée sur un site unique.

La portabilité d'un compilateur est souvent assurée par les techniques dites de "bootstraping" /3/.

Dans Sygare, nous pouvons l'obtenir en disposant du texte du compilateur écrit en langage intermédiaire. Le générateur de code permet alors de disposer d'une image exécutable de celui-ci pour n'importe quel site du réseau. Ce texte en langage intermédiaire peut lui-même être obtenu à partir d'un texte en langage source, par compilation.

Le compilateur n'est pas cependant écrit à l'heure actuelle dans son langage source (ni même intermédiaire). Il est à craindre que ces langages, développés pour la gestion d'applications industrielles, soient mal adaptés aux manipulations de texte nécessités par un compilateur. Il conviendrait donc de les compléter pour pouvoir utiliser la méthode décrite.

Notons enfin une autre méthode largement utilisée depuis quelques temps pour assurer la portabilité des logiciels croisés pour microprocesseur. Il s'agit d'écrire ces logiciels à l'aide d'un langage universellement répandu (généralement le Fortran) et qui pourra être disponible sur tout site susceptible d'assurer la production de programme. Nous ne retiendrons pas cette solution qui présente de nombreux inconvénients et essentiellement :

* Il existe autant de version de Fortran que de compilateur ce qui nuit à la portabilité pratique du produit et même éventuellement à sa fiabilité.

* Le Fortran est un langage lui aussi très mal adapté aux traitements de texte et un compilateur écrit en Fortran risque de ne pas être performant.

V.4. MISE AU POINT DES MODULES

Après avoir écrit les modules, et avant de les intégrer dans l'ensemble de l'application, il faut s'assurer de leur exécution correcte. Tout mauvais fonctionnement peut en effet entraîner des conséquences graves pour le processus contrôlé.

Le projet Sygare autorisant une définition bien structurée de l'application et la programmation des modules en langage de haut niveau, la phase de mise au point devrait de ce fait être considérablement réduite et simplifiée.

Pour réaliser le test des modules, plusieurs solutions sont possibles et souvent complémentaires.

Une première méthode de mise au point utilisable est la simulation dont nous allons rappeler brièvement le principe. Le simulateur est un logiciel qui réalise le décodage des instructions du programme à mettre au point et les exécute sur une machine virtuelle dont les registres, indicateurs, etc. sont remplacés par la mémoire de la machine-hôte. Il permet de réaliser une exécution fractionnée du programme afin d'en suivre régulièrement les effets /14/, /27/.

Dans le système Gare, il est possible d'effectuer une simulation des modules directement au niveau du langage intermédiaire, ce qui présente l'avantage de pouvoir tester d'un coup plusieurs modules sur le site de production.

Cette méthode peut se révéler insuffisante entre autre parce qu'un simulateur croisé ne peut jamais reproduire toutes les particularités matérielles de chaque site ; on peut alors la remplacer ou la compléter par une simulation de chaque module, après génération de code, sur le site final d'exécution que l'on a choisi (il faut alors prendre des précautions en cas de reconfiguration).

Une autre approche de la mise au point des programmes consiste à les exécuter réellement en contrôlant systématiquement un certain nombre de sources d'erreurs dont les principales sont :

- indices de tableaux en dehors des bornes
- débordements de nombres lors de calculs
- divisions par zéro
- conversions impossibles dans les calculs d'expressions mixtes
- mauvaise utilisation des indices d'itération
- tentative d'altération des zones de code.

Cela peut être réalisé facilement par la génération d'un code objet en mode test à l'aide d'un jeu adéquat de macro-définitions. Un tel code traduit les instructions du langage intermédiaire en rajoutant en divers points-clés du programme les tests nécessaires (tests des indicateurs "carry" et "overflow", comparaison des indices aux bornes déclarées ...).

Les contrôles que nous venons de décrire ne sont pas générés lors de la production courante de code objet car ils conduisent à un ralentissement notable de l'exécution du module et sont pénalisants lors de l'utilisation effective de celui-ci dans l'application. Dans certains cas, ils peuvent même conduire à une incompatibilité avec les temps de réponse demandés.

Lors de la mise au point de programmes industriels, les entrées-sorties sur la

périphérie spécialisée constituent souvent un problème délicat. En effet, il est rarement possible d'effectuer les tests "en ligne" sur le processus ; il faut alors disposer d'une instrumentation importante si l'on veut pouvoir le simuler.

L'utilisation dans Sygare d'unités symboliques pour atteindre la périphérie permet une autre méthode de test de ces opérations d'entrée-sortie. Il est possible en effet par réassignation au niveau du moniteur d'entrée-sortie, de dérouter l'échange vers un autre périphérique de type conventionnel permettant de simuler l'échange réel (par exemple par dialogue opérateur sur machine à écrire ou lecture de cartes ...). Ce procédé nécessite cependant un traitement spécial des informations de façon à les présenter dans les mêmes conditions que pour un échange effectif sur le périphérique industriel.

Ces opérations de conversions et de formattage peuvent être effectuées soit directement au niveau du moniteur d'entrée-sortie, soit par substitution d'un module spécial au module utilisateur éventuellement précisé dans une spécification FORMAT.

Les solutions que nous venons de proposer pour la mise au point des modules constituent une adaptation au système Gare des quelques méthodes habituellement utilisées et ne constituent pas une liste exhaustive. La souplesse du système devrait permettre à tout utilisateur de trouver la meilleure méthode pour son application particulière.

CONCLUSION

Le projet que nous venons de présenter a fait l'objet d'un début de réalisation sur un réseau maquette constitué d'un mini-ordinateur de type industriel (Solar 16-40 SEMS) et d'un microprocesseur 8 bits (M 6800 Motorola).

Le Solar s'est vu confier la fonction production de programmes en raison de sa configuration matérielle bien adaptée et à cause également des nombreux utilitaires logiciels dont il dispose (système de gestion de fichiers, éditeur de texte ...). La première phase de la génération de code objet a pu ainsi être réalisée à partir d'un macro-processeur appartenant au logiciel de base Solar.
La deuxième phase (assemblage) est également réalisée à partir du logiciel du constructeur pour les modules destinés au Solar, et à l'aide d'un assembleur croisé écrit pas nos soins pour les modules devant s'exécuter sur le microprocesseur M 6800.

En ce qui concerne la compilation, le problème du choix d'un langage de programmation s'est posé pour la réalisation du traducteur. Ce choix est d'ailleurs limité puisque nous ne disposons sur Solar que du Fortran et de l'assembleur. Fortran est très mal adapté aux traitements de textes que nous devons effectuer et conduit à des programmes volumineux et peu performants, nous l'avons donc éliminé au profit de l'assembleur. Celui-ci, d'emploi parfois un peu lourd, présente l'avantage d'utiliser au mieux les possibilités du calculateur, il rend par contre le traducteur intransportable (cf paragraphe V.3).

Les tests effectués pour l'obtention de modules exécutables ont fait ressortir un fonctionnement satisfaisant de l'ensemble, malgré la forte hétérogénéité du réseau expérimental. Il faut cependant remarquer que la réalisation actuelle présente encore quelques restrictions au niveau du langage source effectivement assimilé.
Les exigences que nous nous étions fixées à l'origine du projet Sygare semblent largement respectées.
La transparence vis à vis de la répartition est bien assurée par le langage de programmation unique, et demeure au niveau du langage intermédiaire et de l'archivage des modules.
La sécurité est obtenue par une programmation modulaire permettant la mise au point séparée des divers constituants de l'application. La description du flux de données autorise, de plus, un maximum de contrôle statique et dynamique, ainsi que l'implantation automatique des mécanismes de protection, éliminant ainsi un nombre important de sources d'erreurs.

Le code objet issu du générateur de code donne des performances analogues (sinon meilleures) à celles qui auraient été obtenues par une programmation directe en assembleur. Les quelques redondances subsistant actuellement devraient être assez facilement éliminées par l'utilisation d'un optimiseur. Le temps d'exécution d'un module est grandement influencé par les caractéristiques du site d'implantation, il est donc important de tenir compte des possibilités de chaque machine lorsque la répartition n'est pas imposée par ailleurs.
La simplicité d'utilisation est assurée par la décomposition modulaire de l'application, permettant une évolution aisée. Elle provient également de la souplesse de description, notamment de la facilité de transposition des structures de contrôle d'un niveau à un autre.

Les perspectives encourageantes du projet Sygare permettent d'envisager son emploi pour décrire également des applications non réparties, pour lesquelles les problèmes de synchronisation et de protection subsistent, problèmes qui doivent toujours être résolus par l'utilisateur. De même notre projet, conçu dans le cadre des automatismes industriels, devrait pouvoir s'étendre à d'autres domaines de l'informatique, moyennant quelques adaptations.

Dans un avenir plus immédiat, il serait souhaitable de pouvoir valider Sygare par une réalisation à plus grande échelle permettant de juger sur une application réelle les résultats obtenus. Une phase ultérieure de développement pourra être la reconfiguration automatique de l'application en cas de défaillance d'un site.

Nous suggérons enfin une solution intéressante pour améliorer la production de programmes et faciliter les reconfigurations : il s'agit de microprogrammer le langage intermédiaire sur les divers processeurs du réseau. Cette méthode devrait pouvoir être aisément mise en oeuvre compte tenu de la structure de la majorité des mini-ordinateurs actuels.

BIBLIOGRAPHIE

/1/ PROCOL T 2000 - Notice technique Stéria Le Chesnay France - référence 1162 220/00 39 00.

/2/ ELZER : PEARL Journées AFCET "Langages Temps Réel", Paris 1975.

/3/ P. PARAYRE, M. TROCELLO
LTR un système de réalisation pour l'informatique temps Réel - Journées AFCET Temps Réel", Paris 1975.

/4/ J.T. WEBB : Coral 66
Journées AFCET "Langages Temps Réel", Paris 1975.

/5/ D. FYLSTRA : Hals/S Programming language
Journées AFCET "Langages Temps Réel", Paris 1975.

/6/ J.G.P. BARNES : The design and use of RTL/2
Journées AFCET "Langages Temps Réel", Paris 1975.

/7/ J.M. TULLI : CPL1 un compilateur transportable et extensible
Journées AFCET "Langages Temps Réel", Paris 1975.

/8/ P. HOLLECZEK, K. PELZ, F.J. PRESTER, R. RÖSSLER
The adaptation of a portable Pearl compilation system experience and future aspects, with special emphasis on the routine package. Séminaire international sur la programmation temps réel.
IFAC/IFIP Rocquencourt, 2-4 juin 1976.

/9/ M. MAZAUD : Système d'aide à la production de traducteurs
Thèse de 3ème cycle, I.N.P.L. Nancy, 27 septembre 1976.

/10/ A. TISSERANT : Compilateur du langage Pascal pour miniordinateurs (Réalisation sur Solar 16). Thèse de Docteur-Ingénieur, I.N.P.L., Nancy 15 décembre 1977.

/11/ Cl. MAUGER : Techniques de compilation sur miniordinateurs
Thèse de Docteur-Ingénieur, Université Cl. Bernard, Lyon, 21 octobre 1976.

/12/ Collectif : Compiler Construction, Springer-Verlag, 1976.

/13/ C. PAIR : Cours de compilation, Nancy.

/14/ IBM 1800 : Program Logic Manual (1800-36 n° GY26-3702-2).

/15/ P. WEGNER : Programming languages - The first 25 years
I.E.E.E. trans. on computers, vol. 25, n° 12, december 1976.

/16/ F. MICHOT, A. DORIS
Etude comparative des principaux langages en temps réel pour minicalculateurs
A.I.I. n° 59, août-septembre 1977.

/17/ J. GERTLER, J. SEDLAK
Software for Process Control. A survey. Automatica, vol. 11, p 613-625, 1975.

/18/ W.H. HARRISON : Compiler Analysis of the value ranges for variables
I.E.E.E. trans. on Software Engineering, vol. SE3, n° 3, may 1977.

/19/ J.L. CARTER : A case study of a new code generation technique for compilers
Communication of ACM, december 1977, vol. 20, n° 12.

/20/ J.C. BEATTY : Register assignment algorithm for generation of highly optimized object code. IBM Journal of Research and Development, january 1974.

/21/ M. BANATRE, A. COUVERT, D. HERMAN, M. RAYNAL
Conservation d'objets typés, Journées Bigre IRIA 24-25 novembre 1977.

/22/ J. HOLDEN, I.C. WAND : Experience with the programming language Modula
IFAC-IFIP Workshop on Real Time Programming, Eindhoven, 1977.

/23/ F. LE CALVEZ, F. MADAULE, H.G. MENDELBAUM
Compiling Gealic, a global Real time language - IFAC/IFIP Workshop on Real Time Programming, Eindhoven, 1977.

/24/ H.F. LEDGARD, M. MARCOTTY : A genealogy of control structures
Communication of the ACM, vol. 18, n° 11, november 1975.

/25/ J. DUCLOY : Compilation dans le projet CIA
Thèse de Docteur-Ingénieur, Université de Nancy I, mars 1973.

/26/ J.C. CHUPIN : Répartition d'applications et de bases de données sur un réseau général d'ordinateurs. Thèse d'Etat, I.N.P.G., 21 octobre 1977.

/27/ MOTOROLA : M 6800 Microprocessor Programming Manual.

/28/ INTEL : MCS 80/85 Programming Manual.

/29/ J.P. MEINADIER : Structure et fonctionnement des ordinateurs, Larousse 1971.

/30/ C.G. BELL, J. GRASON, A. NEWELL
Designing computers and digital systems, Digital Press.

/31/ J.D. NICOUD : A common microprocessor assembly language.
2nd Symposium on micro-architecture, Euromicro, 1976.

/32/ Instrument society of America : Industrial computer system Fortran procedures for executive functions and process input-output - I.S.A. Pittsburgh.

/33/ M.C. DENDIEN : Simulation de machines pour l'enseignement de la programmation
Thèse 3ème cycle, Université de Nancy I, 26 juin 1971.

/34/ J.C. DERNIAME : Le projet CIVA : un système de programmation modulaire
Thèse d'Etat, Université de Nancy I, 25 janvier 1974.

/35/ S.E.M.S. : IOCS Manuel de Référence - Notice Réf. 1 164 151 00 36.

/36/ S.E.M.S. : PL 16 Manuel de Référence - Notice Réf. 1 164 001 00 36.

/37/ S.E.M.S. : Handbook Solar 16 - Réf. MP 8873 F.

/38/ S.G.N. : Apilog, Notice technique.

/39/ D.L. RAIMONDI, H.M. GLADNEY, G. HOCHWELLER, R.W. MARTIN, L.L. SPENCER
LABS/7 - A distributed real time operating system, IBM System Journal, vol. 15 n°1, 1976.

/40/ Colloque IRIA : Réseaux d'ordinateurs - Workshop IRIA/ACM, 23-24 mars 1972 IRIA.

/41/ J. BEZIVIN, Y. JEGON, J.L. NEBUT, R. RANNON
Production de systèmes temps Réel fiables et efficaces, Journées "Bigre" IRIA 24-25 novembre 1977.

/42/ P. NONN : Le Système d'Exploitation dans le projet Sygare.
Thèse de Docteur-Ingénieur, I.N.P.L., à paraître.

/43/ J.P. THOMESSE : A new set of software tools for designing and realizing distributed systems in process control.
IFAC/IFIP Real Time programming Workshop, Eindhoven 1977.

/44/ S.E.M.S. : MACP Manuel de Référence
Notice Réf. 1 164 079 00 36.

/45/ A. COCHET-MUCHY : "BABAR" Assembleur croisé pour M 6800 sur Solar 16
L.E.E.A. - E.N.S.E.M. : Note interne.

/46/ P. LADET, P. DESCHIZEAUX
Programmation en temps Réel des synchronisations par gestion des liens événements-processus. Journées AFCET "Temps Réel", 3-4 novembre 1977.

/47/ P. ROBERT, J.P. VERJUS : Toward autonomous description of synchronisation modules. Proceeding of IFIP Congress, Toronto, august 1977.

/48/ CROCUS : Systèmes d'exploitation des ordinateurs - Série Informatique
Dunod, 1975.

/49/ S.E.M.S. : Basic Manuel Référence
Notice Réf. 1 164 005 00 36.

/50/ ZIMA : Progress
Journées AFCET "Langages Temps Réel", Paris 1975.

/51/ C. BETOURNE, L. FERAUD, J. JOULIA, J.M. RIGAUD
LEST : un langage d'écriture de systèmes, Journées "Bigre", IRIA 24-25 novembre 1977.

/52/ J.P. THOMESSE, A. COCHET-MUCHY, P. NONN
Conception et réalisation de systèmes répartis en conduite de processus industriels - Présentation du Projet Sygare.
Journées "Bigre", IRIA 24-25 novembre 1977.

/53/ J.P. THOMESSE, A. COCHET-MUCHY, P. NONN
Data flow analysis for the description and the management of mutual exclusion and synchronization in real Time applications - IFAC/IFIP Workshop, Helsinki 78.

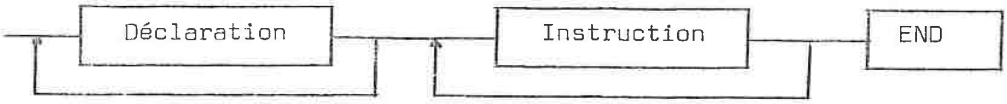
/54/ Signétics : Microprocesseur 2650.

ANNEXES

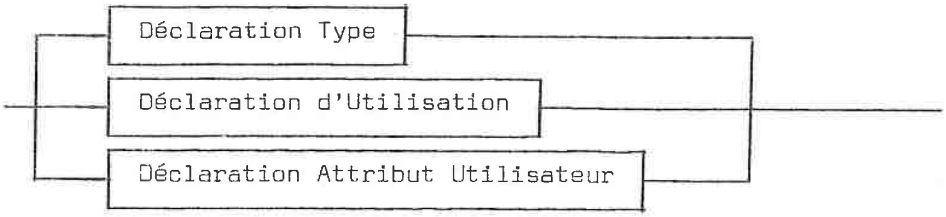
ANNEXE 1

1) La Syntaxe du langage source de Sygare

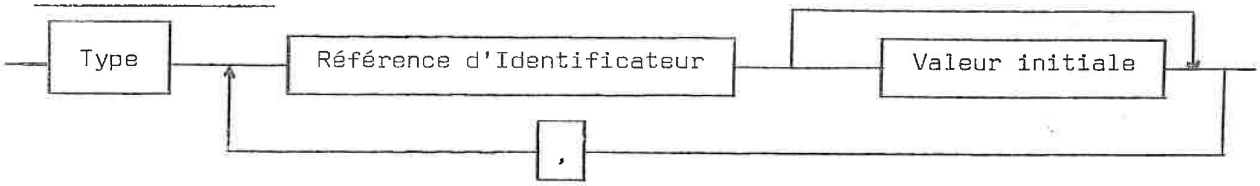
PROGRAMME



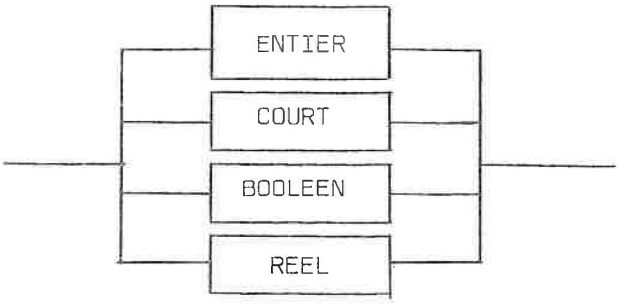
DECLARATION



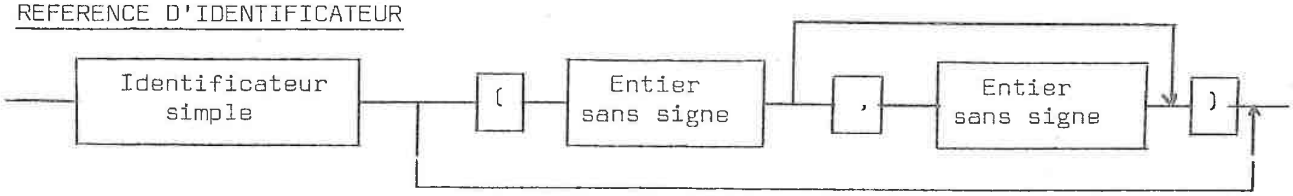
DECLARATION TYPE



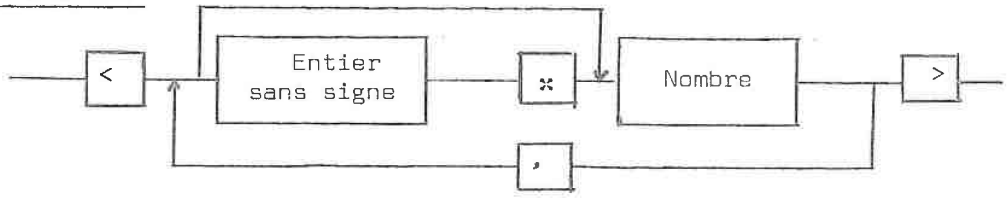
TYPE



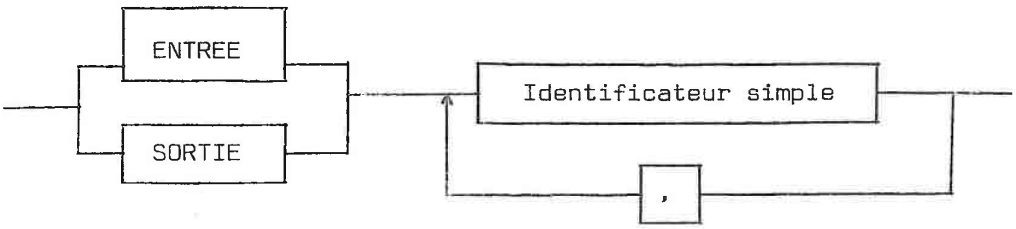
REFERENCE D'IDENTIFICATEUR



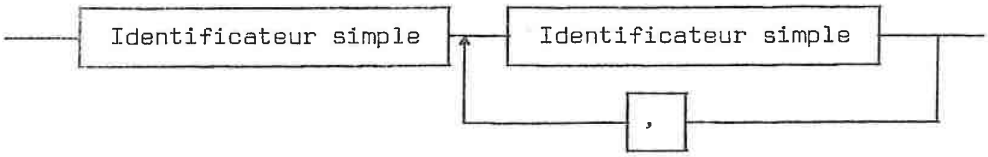
VALEUR INITIALE



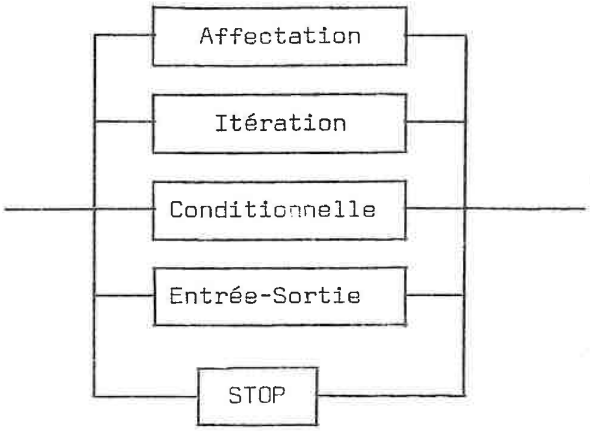
DECLARATION D'UTILISATION



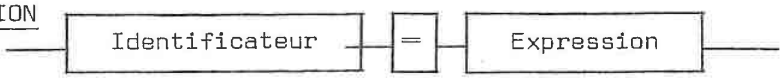
DECLARATION D'ATTRIBUT UTILISATEUR



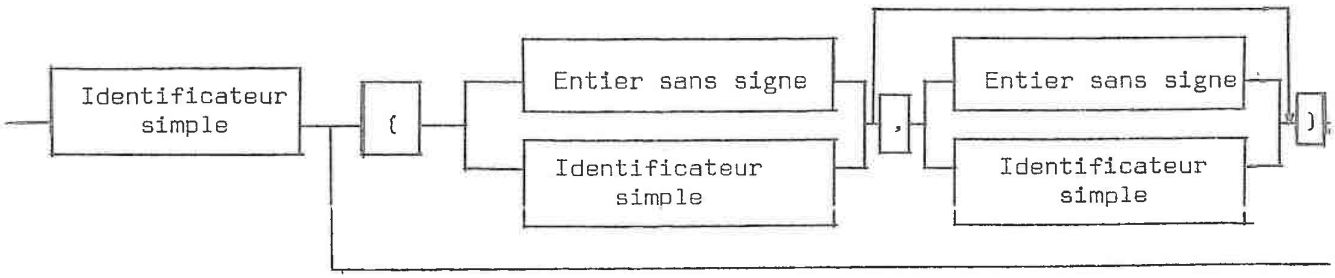
INSTRUCTION



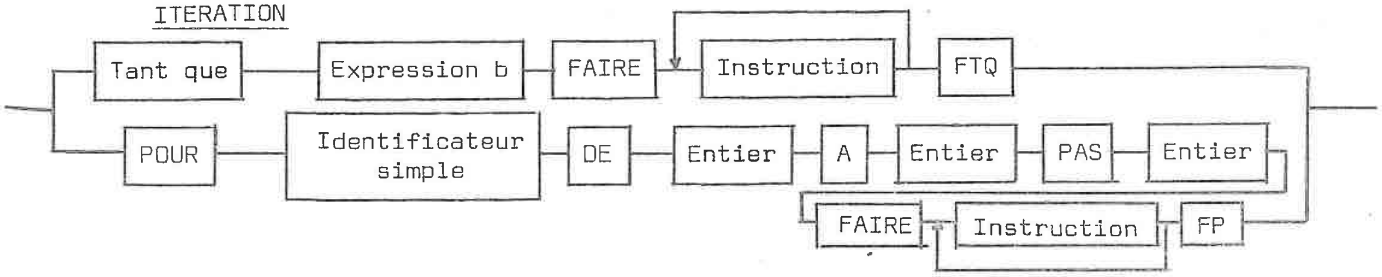
AFFECTATION



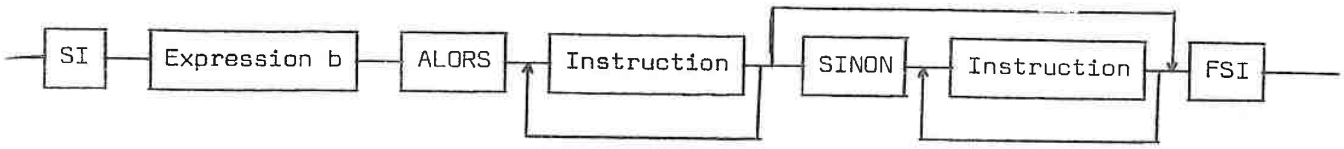
IDENTIFICATEUR



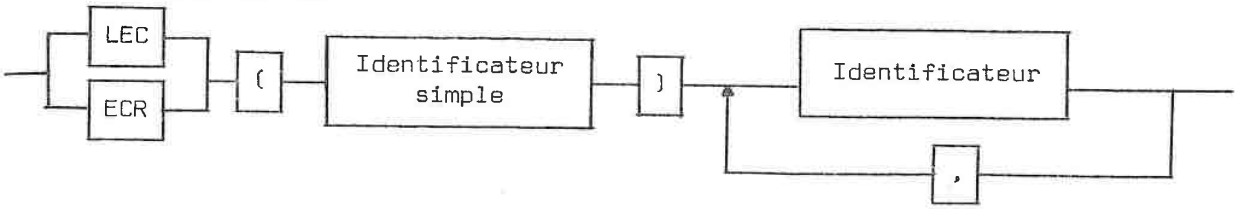
ITERATION



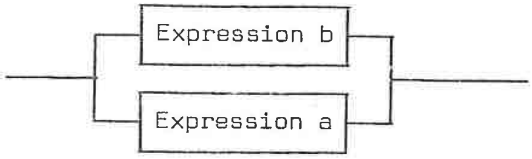
CONDITIONNELLE



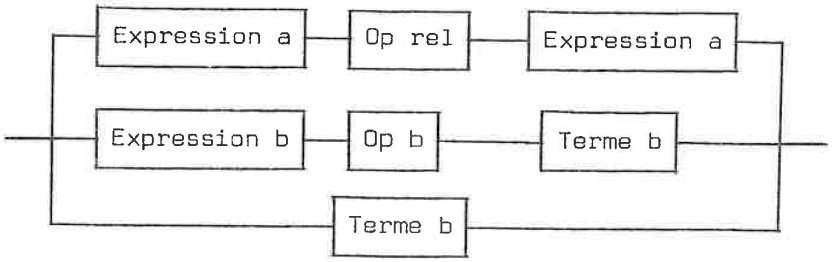
ENTREE-SORTIE



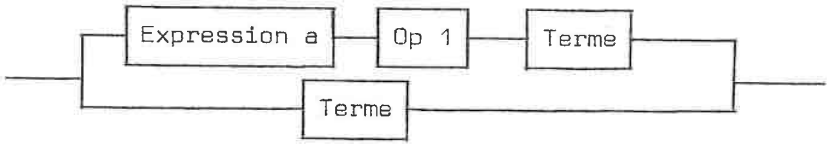
EXPRESSION



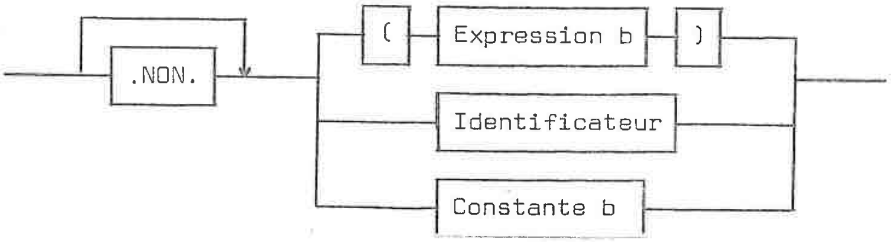
EXPRESSION B



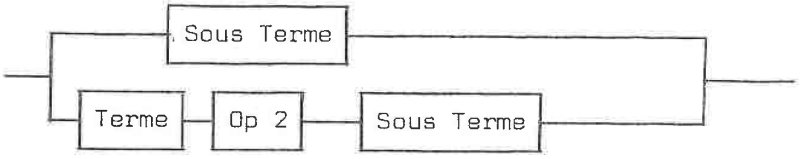
EXPRESSION A



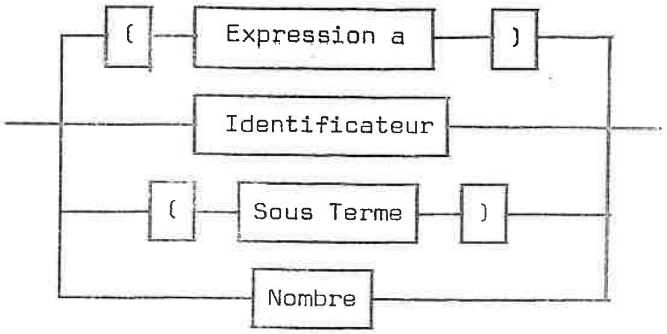
TERME B



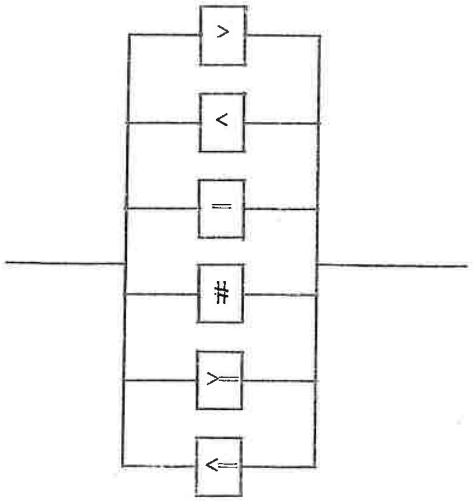
TERME



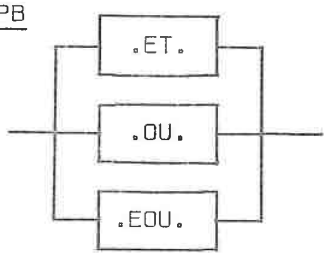
SOUS TERME



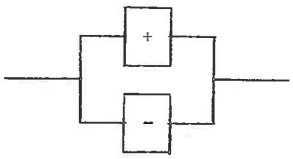
OPREL



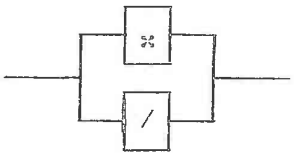
OPB



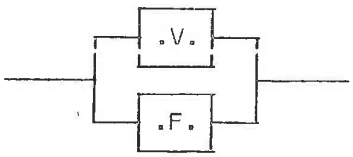
OP 1



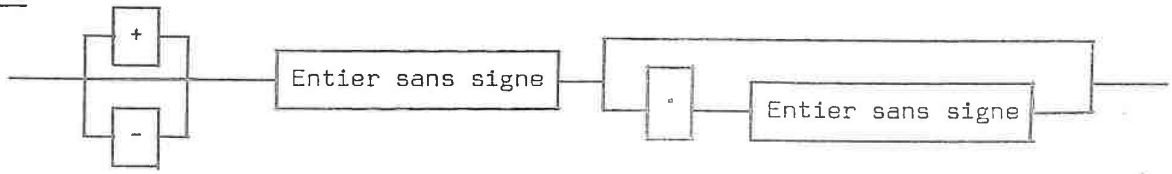
OP 2



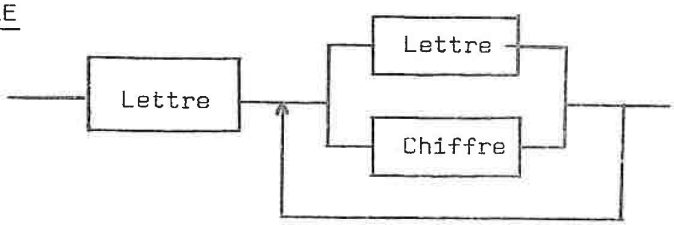
CONSTANTE B



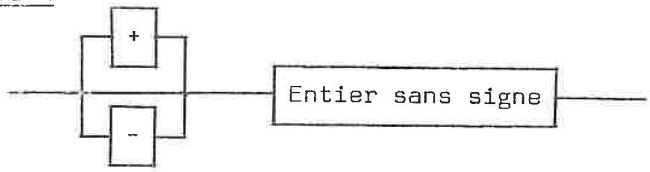
NOMBRE



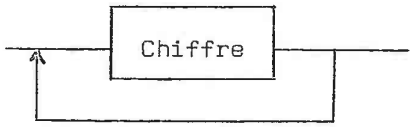
IDENTIFICATEUR SIMPLE



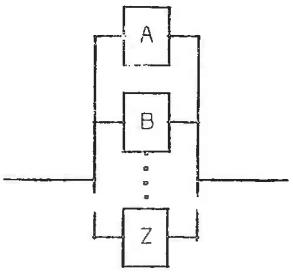
ENTIER



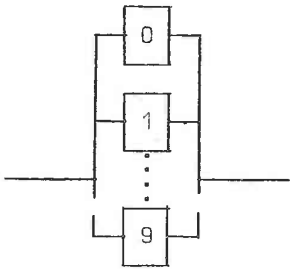
ENTIER SANS SIGNE



LETTRE



CHIFFRE



ANNEXE II

II. LES MACRO-DEFINITIONS DU GENERATEUR DE CODE

a) Macro-définitions correspondant aux instructions du langage intermédiaire

Dans la description qui suit, le point d'interrogation représente un paramètre formel attendu par la macro-définition.

* CHA ? : chargement de registre
*?CHA ? : chargement de registre avec étiquette
* RAN ? : rangement de registre
* ADD ? : addition
* SOU ? : soustraction
* MUL ? : multiplication
* DIV ? : division
* ET ? : intersection logique
* OU ? : union logique
* EOU ? : disjonction
* CMP ? : comparaison

Dans les macro-définitions précédentes, le paramètre effectif est optionnel (sauf pour les trois premières). On utilisera pour son traitement deux macro-définitions utilitaires :

AVECOP ? : traitement du paramètre effectif
SANSOP : macro-instruction sans paramètre

Les autres macro-définitions de cette première classe n'ont pas de paramètre ou ont un paramètre de type adresse ; ce sont :

* NON : complémentation logique (complément à 1)
* NGT : complémentation arithmétique (complément à 2)
* STOP : arrêt du programme
*?ETQ : définition d'étiquette
* BI /? : branchement si inférieur
* BIE/? : " " " ou égal
* BE /? : " " égal
* BS /? : " " supérieur
* BSE/? : " " " ou égal
* BNE/? : " " différent
* BRA/? : " inconditionnel

b) Macro-définitions correspondant aux directives de génération

*VLOC ?,? : définition d'une variable simple gérée par le module (type, nom)
*TLOC ?,?? : définition d'un tableau géré par le module (type, taille, nom)
*VCOM ?,? : définition de variables gérées par le moniteur (type, nom)

Le codage de paramètres est :

Type : 0 = entier
1 = logique
2 = réel
3 = court

Nom : nom de la variable sur 6 caractères ASCII
Taille : taille du tableau (nombre total d'éléments)

*CE ? : définition d'une constante entière
*CR ? : définition d'une constante réelle
*MAN ? : réservation des zones de travail et insertion des séquences d'initialisation.

c) Macro-définitions correspondant aux structures d'opérande

Dans cette classe, nous trouvons d'abord une série de macro-définitions correspondant aux adressages d'éléments de tableau :

*?+? : adressage du type : TAB + n (n désigne une constante)
*?+?+? : " " " : TAB+I+n
*?+?-1 : adressage du type : TAB+I-1
?+?? : " " " : TAB+I*n
?+??+? : " " " : TAB+I*n+m (m désigne une constante)
?+??-? : " " " : TAB+I*n-m
?+??+?-? : " " " : TAB+I*n+J-m

Les dernières macro-définitions concernent le traitement des constantes :

*!? : traitement des constantes logiques !V et !F (vrai et faux)
*R'? : " " réelles

ANNEXE III

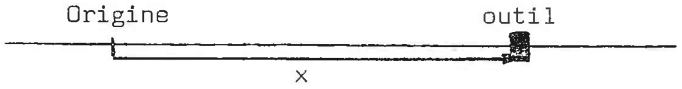
EXEMPLE D'APPLICATION DECRITE A L'AIDE DE SYGARE :

COMMANDE D'UN MOTEUR PAS A PAS

Le problème étudié est celui du positionnement d'un outil sur un axe à l'aide d'un moteur pas à pas. Nous pouvons déjà distinguer deux phases dans ce travail :

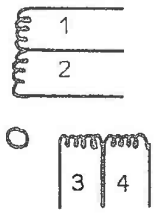
1) Initialisation, c'est à dire recalage à l'origine de l'outil qui a pu être laissé n'importe où. Ce recalage s'effectue à l'aide d'une butée fin de course.

2) On effectue les divers positionnements de l'outil aux endroits indiqués.



Cet exemple nous permettra de mettre en évidence la souplesse d'expression de Sygare ainsi que la possibilité de transposition des structures de contrôle d'un niveau à un autre.

Avant d'aller plus loin, examinons brièvement le fonctionnement d'un moteur pas à pas. Celui-ci peut être schématisé par un ensemble de quatre enroulements :



Ces enroulements sont alimentés deux par deux, la séquence de commutation étant : 1 et 3 - 2 et 3 - 2 et 4 - 1 et 4
pour un déplacement dans un sens et : 1 et 4 - 2 et 4 - 2 et 3 - 1 et 3 pour un déplacement dans l'autre sens.

Nous pouvons représenter les quatre enroulements par 4 bits, un bit à 1 indiquant un enroulement alimenté. Nous obtiendrons alors les séquences de commande :

1010 - 0110 - 0101 - 1001 et 1001 - 0101 - 0110 - 1010

Ces séquences doivent être espacées d'un temps minimum Δt pour tenir compte des constantes de temps du moteur.

Examinons maintenant la phase de recalage à l'origine (nous supposerons celle-ci située à gauche de l'axe de déplacement), elle pourra être activée ainsi :

Faire init.recal
Tant que fin course = faux faire dépl. gauche

ou d'une manière équivalente :

Faire init. recal
Faire dépl. gauche
Sur fin course arrêter dépl. gauche

La tâche "dépl. gauche" (déplacement à gauche) peut elle-même s'écrire :

Faire dépl.g 1 pas
Répéter tous les Δt

Le module dépl.g 1 pas (déplacement à gauche d'un pas) étant écrit ainsi :

```
ENTIER T(10)<0,0,0,0,9,5,0,0,10,6>,U
ENTREE U
SORTIE U
U = T(U)
ECR(---)U
END
```

T est le tableau contenant la séquence des commandes correspondant à un déplacement à gauche.

U est la commande appliquée au moteur, c'est une variable d'état dont la rémanence est assurée par le moniteur.

L'ordre de sortie ECR correspond à l'affichage de la commande U sur le périphérique de contrôle du moniteur pas à pas.

Le rôle du module "init,recal" (initialisation du recalage) est uniquement destiné à l'initialisation de la variable U (à une valeur arbitraire de la séquence de commande) et de quelques autres variables que nous étudierons plus loin (cf II.2.1.2).

Nous allons maintenant pouvoir effectuer les positionnements successifs de l'outil sur l'axe. Il faut pour cela déterminer le sens du déplacement et le nombre de pas à effectuer ; c'est le rôle du module "calc.dépl" :

```
ENTIER POSIT,U,X,SENS,NPAS,COEF<—>
ENTREE POSIT,X
SORTIE POSIT,NPAS,SENS
SI X-POSIT>0
ALORS
SENS=1
SINON
SENS=0
FSI
NPAS=(X-POSIT)*COEF
POSIT=X
END
```

POSIT représente la position initiale de l'outil et X la position à atteindre (POSIT est une variable rémanente dont la valeur initiale 0 est fournie par "Init. recal").

La valeur de X peut provenir d'un module la calculant à chaque fois ou d'une file d'attente du moniteur.

COEF est le coefficient permettant de déterminer le nombre de pas NPAS nécessaire pour effectuer le positionnement.

La fonction positionnement peut se déduire ainsi :

Faire calc.dépl
Si SENS=0 Alors faire NPAS fois dépl g : 1 pas - Δt
Sinon faire NPAS fois dépl d. 1 pas - Δt

Le déplacement à gauche d'un pas est effectué par le module déjà décrit. Pour le déplacement à droite, on utilise un module identique avec la séquence de commande appropriée, on peut aussi inclure la structure conditionnelle à l'intérieur du module qui effectue le déplacement de m pas ; il devient alors :

```
ENTIER TD(10)<——— >,TG(10)<——— >,U
ENTREE U,SENS
SORTIE U
SI SENS=1
ALORS
U=TG(U)
SINON
U=TD(U)
FSI
ECR(— )U
END
```

La séquence d'activation est alors :

Faire calc.dépl.
Faire NPAS fois dépl.1 pas - Δt

La tâche de déplacement d'un outil que nous venons de décrire peut être utilisée par exemple pour piloter une table traçante X-Y. On la dupliquera alors, chaque exemplaire ayant la gestion d'une coordonnée.

Si on désire effectuer un tracé par point, on peut effectuer directement la duplication des tâches. Par contre si on désire un tracé continu on peut réaliser une interpolation du tracé en modifiant la vitesse de l'un des axes en augmentant sa période d'attente Δt entre deux pas. (Il est aussi possible de fragmenter le tracé en déplacements suffisamment petits pour que les écarts ne soient pas sensibles, les deux moteurs tournant à la même vitesse).

Nous utiliserons un module supplémentaire, pour gérer la table traçante, qui fournira les valeurs des déplacements X et Y à effectuer à partir des tableaux de coor données TABX et TABY et des facteurs d'échelles COEFX et COEFY :

```
ENTIER : TABX(-),TABY(-),COEFX,COEFY,I,X,Y
ENTREE : I,TABX,TABY,COEFX,COEFY
SORTIE : I,X,Y
X=TABX(I)*COEFX
Y=TABY(I)*COEFY
I=I+1
END
```

La variable I conservée par le moniteur doit être initialisée par le module d'initialisation.

Nous allons terminer la description de cette application en examinant les diverses phases de traduction du module "calcul déplacement" précédemment décrit.

Le module en langage intermédiaire comporte trois articles :

* Un article des déclarations de données :

```
VCOM 0,POSIT        définition des variables
VCOM 0,X
VCOM 0,SENS
VCOM 0,NPAS
VLOC 0,U
VLOC 0,COEF,n       (n : valeur initiale)
VLOC 0,I
VLOC 0,J
CE   0              définition des constantes
CE   1
MAN  1
*EOT
```

* Un article d'instruction

```
     CHA X                  RAN SENS
     SOU POSIT        0001  EPC
     CMP 0                  CHA X
     BIE /0000              SOU POSIT
     CHA 1                  MUL COEF
     RAN SENS               RAN NPAS
     BRA /0001              CHA X
0000 CHA 0                  RAN POSIT
                            *EOT
```

* Une table des symboles réduite ne comportant plus que POSIT,X,SENS et NPAS (seules variables utilisées en entrée et/ou sortie).

La première phase de génération de code nous donne du langage assembleur, sur le Solar nous obtenons :

```
      COMMON    COM
COMMON : EQU    $
VC1 : WORD      <POSIT
VC2 : WORD      <X
VC3 : WORD      <SENS
VC4 : WORD      <NPAS
```

Relais d'adressage des variables gérées par le moniteur.

```
      LOCAL     LOC
LOCAL : EQU     $
VL1  :  WORD     <U
VL2  :  WORD    n<COEF
VL3  :  WORD     <I
VL4  :  WORD     <J
 C0  :  WORD     0
 C1  :  WORD     1
MAN  :  DZS      1*2
VRAI :  WORD     -1
FAUX :  WORD     0
```

Définition des variables gérées par le module

Définition des constantes

```
      TABLE     TAB
TABLES : DZS    0
KSTORE : DZS    70
```

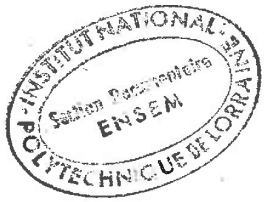
Définition des tableaux et de la pile de travail

```
      PROG      PROGR
DEB:  LRM       C,L,K
      WORD      COMMON+128
      WORD      LOCAL+128
      WORD      KSTORE-1
```

Séquence d'initialisation

```
        LA    &VC2
        SB    &VC1
        CPI   0
        JLE   E0000
        LAI   1
        STA   &VC3
        JMP   E0001
E0000:  LAI   0
        STA   &VC3
E0001:  EQU   $
        LA    &VC2
        SB    &VC1
        MP    VL2
        LR    B,A
        STA   &VC4
        LA    &VC2
        STA   &VC1
        SVC   12
        END   DEB
```

Traduction des instructions du Langage intermédiaire



Séquence de fin