ISSN 0291 - 5413

Informatique fondamentale et Applications

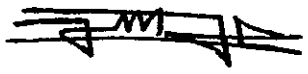
Editeur
L.I.T.L.

Depositaire

G. Ambard

Sommaire

J.-M. KNIPPEL



Juin 1982

Faculté des Sciences de Luminy
Département de Mathématique-Informatique
70, route Léon-Lachamp - Case 901
13288 MARSEILLE CEDEX 9

EDITORIAL

Edmond BIANCO

INFORMATIQUE ET VACANCES.

L'année a été dure... voilà un constat qui n'est nouveau que parce qu'un an nous sépare de Juin dernier. Nos fatigues récentes ne nous paraissent si grandes que parce que les autres sont déjà bien lointaines. Et puis peut-être notre endurance grandit-elle ?

J'ai l'air d'exprimer une plainte, mais dans le fond quel ennui serait le mien si je n'avais pas à constituer de belles séries de dossiers dont tous ne sont pas forcément inutiles. Avec ce léger piment qui rend si délicat ce plaisir de créer : la commission se réunit dans trois jours, vous savez c'est hier que les dossiers devaient parvenir au greffe ; de toute manière, ce soir minuit vous êtes forclos. Quelle importance, il va me falloir quinze jours pour collecter l'information nécessaire. Ah quels délicieux frissons de plaisir vont accompagner l'incomparable attente de la ligne pour avoir Paris : un petit délai s'il vous plaît ? Et quelquefois même la ligne se coupe toute seule. Le plaisir alors touche à l'orgasme.

Il va falloir abandonner cela pour un temps bien long. Le temps de constituer des files bien rangées, le long de ces bus-autoroutes qui doivent nous guider à bon port. Et puis alors, livrés à nous-mêmes, nous errerons comme des âmes en peine, cherchant les pires fatigues nous imposant les pires contraintes pour essayer d'oublier ce lancinant leit-motif : que faire, je n'ai vraiment rien à faire...

Et là les longues attentes, la peau cuisant à petit feu sous le soleil brûlant, les marches épuisantes dans des chemins impossibles, les grands voyages expiatoires sur des navires où les tâches harassantes nuit et jour vous laissent les muscles brisés, les longues prostrations du farniente. Bref, tout pour essayer d'oublier.

Bien heureux si le délire ne s'empare pas de nous comme le personnage de Ron Cob, nous promenant l'air hagard dans un paysage de désolation, un micro-ordinateur l'écran brisé à la main, cherchant désespérément une prise électrique.

Je m'en voudrais de rester sur une note aussi démoralisante. Ainsi je m'empresse d'affirmer que l'EDF, contre vents et marées, continuera à fournir coûte que coûte (oui) ce fluide précieux sans lequel il n'est nulle informatique. Nous pourrons continuer à calculer, bâtir des projets audacieux dont le but sera de toute évidence d'accroître encore notre puissance de calcul.

Que la nature sera belle avec ses arides déserts numériques, ses fleuves de langages, ses océans de mots aux tons variables, ses montagnes d'information. Quel plaisir de mettre de l'ordre dans tout cela, d'aligner tous ces chiffres comme des petits soldats, d'ordonner ces mots comme dans un discours ministériel, et de donner à cette information un aspect saisissant. Enfin l'Ordre va régner, le vrai, cette fois-ci. Et non pas cet Ordre qui ne cesse de se renouveler. Gare à qui dérangera ces beaux alignements.

Mais voilà, les vacances sont là, plus que trois ou quatre dossiers, et à nous les petites Anglaises.

= : = : = : = : =

FORMALISME DE REPRESENTATION DE LA CINEMATIQUE DES SYSTEMES DE GESTION AUTOMATISES ET QUASI AUTOMATISES

A. PANARIELLO
Groupe de Recherche en Analyse de Systemes
et Calcul Economique (E.R.A n° 640)
AIX-EN-PROVENCE

C.R Subject Classifications Informatics 2.11 3.31 3.57 4.20 5.23

Résumé. Depuis les années soixante, automatiser la direction générale des entreprises a été un rêve pour bien d'informaticiens de gestion. Mais, si certaines décisions sont automatisables, les décisions stratégiques ne le sont que partiellement ou pas du tout en raison de la multiplicité des paramètres émanant de l'environnement.

Il est cependant possible de réaliser un formalisme de représentation commun à ces deux types de décision. Diverses extensions sont possibles pour permettre la simulation de décisions automatisées. Des enseignements en matière de politique économique et de management peuvent être également déduits de ce "langage" commun à tout problème de gestion.

= : = : = : = : = : =

FORMALISME DE REPRESENTATION DE LA CINEMATIQUE DES SYSTEMES DE GESTION AUTOMATISES ET QUASI AUTOMATISES

par Alain PANARIELLO

INTRODUCTION : GESTION INFORMATIQUE OU INFORMATISEE ?

Parallèlement à l'échec des macro-modèles d'entreprises comportant des milliers de variables et d'équations, l'évolution rapide des connaissances en matière de conception des Systèmes Interactifs d'Aide à la Décision (SIAD) laisse penser que la tendance est plutôt en faveur de la décision informatisée. En effet, si la décision "informatique" (automatique) est réalisable et souhaitable pour certains problèmes de gestion dits structurés, la décision "informatisée" (non automatique, intervention partielle de l'informatique au cours du processus de décision) parait à l'heure actuelle plus adaptée aux problèmes faiblement structurés (la plupart des problèmes de management).
Cependant, ces deux types de décision possèdent certaines caractéristiques communes, ce qui permet de proposer un formalisme de représentation valable dans ces deux cas.

1. DECISIONS AUTOMATISEES ET QUASI AUTOMATISEES.

1.1. Décisions structurées (automatisées).

On parle de décisions programmées ou structurées lorsque l'ensemble des stratégies qui définissent la séquence des réponses du système considéré (l'entreprise) face à un certain problème peut être formulé. C'est le cas de décisions de routine, répétitives pour lesquelles on dispose de procédures parfaitement définies.

On peut aussi programmer sur ordinateur les mécanismes de résolution du problème posé. Parmi ces décisions automatisées, on trouve les décisions de gestion de stocks, de facturation, etc...

1.2. Les décisions non structurées (quasi automatisées).

Ce sont des décisions intuitives, où interviennent souvent jugement et expérience, parfois hasard, telles que les décisions d'investissements, de politique salariale des entreprises, les décisions de grands projets publics, ou autres décisions stratégiques dépendant des circonstances qui les nécessitent. Il s'agit donc de décisions uniques, nouvelles, imprévisibles, irréversibles et complexes au point que le nombre de critères à prendre en compte décourage toute tentative de formalisation.

On utilise dans ces cas, et ce de manière interactive, des algorithmes d'aide à la décision et des raisonnements empiriques, intuitifs, heuristiques (analogie, récurrence, absurde, approximation, images symboliques, ...). Ces raisonnements du décideur sont influencés par l'environnement qui a un impact considérable dans ce type de décision.

1.3. Bases pour un formalisme commun.

Ces deux types de décision ont à l'évidence une caractéristique commune qui est leur fonction : transformer de l'information par étapes successives pour progresser vers l'information-action (la décision).

Cette évolution étape par étape des processus de décision est ce que nous nommerons cinématique.

Il existe de nombreux formalismes pour spécifier la cinématique des systèmes. Les réseaux de Pétri ont apporté une contribution significative pour la représentation de la cinématique en automatisme et constitueront à ce titre les bases théoriques du nouveau formalisme proposé pour l'étude de la cinématique des systèmes de décision.

2. LE FORMALISME.

2.1. Les concepts.

2.1.1 : Aspect statique. Le concept de signal.

Les principales variables intervenant dans les processus de décision sont des projets, des rapports, des ordres d'intervention, des pressions, etc... Ces variables interviennent au cours d'un processus décisionnel non par leur forme mais par leur contenu informationnel.
En effet, l'information est bien le point commun de ces différentes variables : rapports, pressions diverses, ordre constituent des éléments d'information pour les participants à la décision.
Ces variables, véhiculant de l'information, sont fabriquées, puis émises par certains acteurs en direction d'autres participants à la décision dans le but de les informer et aussi d'influer sur le cours de la décision. Il est permis de penser que ces informations sont émises chacune selon un certain code, et doivent, pour être utilisables et interprétables, pouvoir être décodées. Ainsi, un rapport d'ingénieur rédigé en des termes mathématiques a peu de chances d'être interprété correctement par des responsables administratifs mais sera parfaitement compris par le bureau d'études.

Ayant donc remarqué les similitudes du contenu des différentes variables intervenant au cours des processus de décision, on peut alors mettre en évidence cette analogie en regroupant ces variables en un même concept : le concept de signal endogène.
Nous définirons ce signal endogène comme le message codé transmis par les supports informationnels précédents "fabriqués" par les participants à la décision.
Ces signaux sont supposés être identifiables et décodables par des acteurs récepteurs.
Exemples de signaux endogènes : une décision de principe, un projet en cours, etc...
Ces signaux vont être émis en direction de récepteurs particuliers. En effet, le mode de transmission des signaux dépend du réseau de communications dans l'organisation concerné par la décision.

Mais il existe d'autres variables intervenant au cours des phénomènes décisionnels. Il s'agit de variables véhiculant également de l'information mais cette information est élaborée en dehors de l'organisation considérée et émane donc de l'environnement. Ce peut être une demande sociale forte, une pression politique, une découverte scientifique ...

Ces variables ont les mêmes caractéristiques informationnelles que les précédentes. Nous pourrons les définir comme des signaux exogènes, c'est-à-dire des messages codés émanant de l'environnement de l'organisation. Au delà de cette différenciation des signaux, explicitée par besoin de clarté, un concept unique se dégage, rendant bien compte de l'idée d'information, le concept de signal.
En un instant, la liste des signaux émis par des décideurs au cours d'un processus de décision dans une organisation constituera ce que nous proposons d'appeler un état de la décision.
L'étude descriptive d'un phénomène décisionnel va donc consister à envisager la suite des états de la décision de l'instant initial (détection du problème) à l'instant final (action correctrice déclenchée).

2.1.2 Aspect cinématique.

a) Le concept de synergie. 2 + 2 = 5.

Ce concept traduit le fait que l'évolution d'un processus de décision ne se produit qu'à la condition qu'existent simultanément des évènements ou signaux. Par exemple, le service marketing ne commencera une étude de marché qu'après la demande de la direction générale et la dotation budgétaire nécessaire à cette étude.
C'est la combinaison de deux ou plusieurs signaux qui va provoquer l'évolution du processus de décision en éveillant l'attention d'un décideur de l'organisation. Cette combinaison est plus que la "somme" de plusieurs signaux.

Le résultat de la synergie évoque en quelque sorte la molécule que l'on obtient en fusionnant des atomes. La molécule est à la fois plus et moins que la conjonction des atomes qui la constituent ; le résultat de la synergie est à la fois plus et moins que la conjonction des signaux qui le constituent. La synergie provoque l'interaction des signaux, et c'est de cette interaction que va naître le résultat de la synergie. Les interactions entre signaux à l'occasion de la synergie sont représentables par le ET logique.

Le résultat de la synergie possède les mêmes caractéristiques informationnelles que les signaux à son origine.

D'une manière plus générale, on peut définir la synergie comme une combinaison de signaux en un instant donné produisant une information nouvelle susceptible de faire évoluer le processus de décision en cours.

b) Le concept de décodage.

La synergie des signaux doit désormais être interprétée par un ou des participants à la décision de façon à faire évoluer le processus ; la synergie n'était que l'amorce de cette évolution, le décodage du résultat informationnel de la synergie en constituera la seconde étape. Les signaux, une fois entrés en synergie ne sont plus interprétés, décodés individuellement, mais en teant compte des autres signaux de façon à intégrer l'interdépendance de ces différentes variables informationnelles intervenant dans la décision.
Le décodage du résultat de la synergie a modifié la compréhension du problème soulevé. Ce décodage est à la fois objectif et subjectif : chaque acteur décode la synergie d'une manière qui lui est propre, mais en tenant compte de certains critères.
Il s'agit donc d'une phase de reflexion pour le décideur concerné, phase au cours de laquelle celui-ci va analyser l'état de la décision. C'est essentiellement une étape de recherche et de hiérarchisation des buts opérationnels (critères) du décideur, lui permettant ainsi d'évaluer le besoin d'une intervention sur l'état de la décision.
Remarquons qu'il n'est pas évident que le décideur concerné parvienne à interpréter la synergie convenablement : le décodage dépend des capacités cognitives et en particulier des capacités de reconnaissance des codes par lesquels sont émis les signaux.

c) Le concept de traitement.

L'information véhiculée au cours des étapes précédentes va alors être transformée par un ou des acteurs, par celui ou ceux qui ont précédemment décodé le résultat de la synergie.

Grâce aux prévisions, aux capacités cognitives de ces participants à la décision et dans le cadre de procédures détaillées tels les règlements administratifs par exemple, l'information extraite du résultat de la synergie une fois décodée va être transformée.

Ces informations sont traitées selon des programmes reproductibles ou bien par des programmes heuristiques, purs produits de l'imagination et non reproductibles, contraints par certaines procédures qui vont réduire l'éventail des transformations possibles de l'information.

Il y a ainsi production d'information nouvelle. Peuvent alors se présenter deux types de productions :

- une information nouvelle complémentaire (à partir de plans et de calculs on produit une maquette) ,
- un ordre d'intervention ou action.

Ces informations, résultant des trois étapes successives nécessaires à l'évolution élémentaire des processus de décision, se trouvent être par conséquent des signaux nouveaux émis en direction des récepteurs particuliers (décideur suivant de la chaine de décision) susceptibles de les comprendre pour à nouveau les transformer ou les exécuter.

2.2. Modélisation des concepts.

a) Les signaux :

Nous avons précédemment défini l'état (sous-entendu stable) de la décision : la liste des signaux émis à un moment donné d'un processus de décision.

Cette idée de repérage de la décision à un instant donné nous fait penser immédiatement à la notion de place d'un réseau de Pétri qui matérialise précisément une situation dans laquelle l'état du système considéré est donné.

Nous pouvons donc convenir de représenter les signaux comme les places d'un réseau de Pétri.

b) Synergie :

Soit S l'ensemble des signaux possibles :

$$S = \{ s_1 \ / \ 1 \in N \}$$

alors une synergie peut se définir par la fonction

$$C : \mathcal{P}(S) \longrightarrow M$$

$$((s_1, s_2, \ldots, s_k) \xrightarrow{C} m$$

avec :

$\mathcal{P}(S)$ l'ensemble des parties de S

$(s_1, s_2, \ldots, s_k)$ un k - uplet de signaux

m le résultat de la synergie C

M l'ensemble des résultats possibles des synergies.

c) Décodage :

Cette opération consiste à interpréter m .
Soit E l'ensemble des interprétations possibles de m alors le décodage est une fonction :

$$d : M \longrightarrow E$$

$$m \xrightarrow{d} d(m) = e$$

d) Traitement :

Il s'agit, à partir de l'interprétation des faits précédents, de modifier les faits, l'information initiale reçue par l'acteur concerné.
Cette modification sera fonction de l'interprétation et redonnera des signaux nouveaux, un état de la décision nouveau.
Le traitement est une fonction de l'ensemble E vers l'ensemble $\mathcal{P}(S)$ des signaux.

$$t : E \longrightarrow \mathcal{P}(S)$$

$$e \xrightarrow{t} t(e) = (s'_1, s'_2, \ldots, s'_\ell)$$

L'ensemble de ces trois fonctions a permis le changement d'état de la décision.
L'évolution des processus de décision est par conséquent due à la composition de ces fonctions.

C'est la séquence synergie - décodage - traitement qui a modifié l'état de la décision.
Cette séquence peut ainsi se traduire :

$$(s_1,\ldots,s_k) \xrightarrow{C} C\,(s_1,\ldots,s_k) \xrightarrow{d} d\,[C\,(s_1,\ldots,s_k)] \xrightarrow{t} t\,[1[C\,(s_1,\ldots,s_k)]]$$

On a alors :

$$t \circ d \circ C\,(s_1,\ldots,s_k) = (s'_1,\ldots,s'_\ell)$$

qui est équivalent à :

$$T\,(s_1,\ldots,s_k) = (s'_1,\ldots,s'_\ell)$$

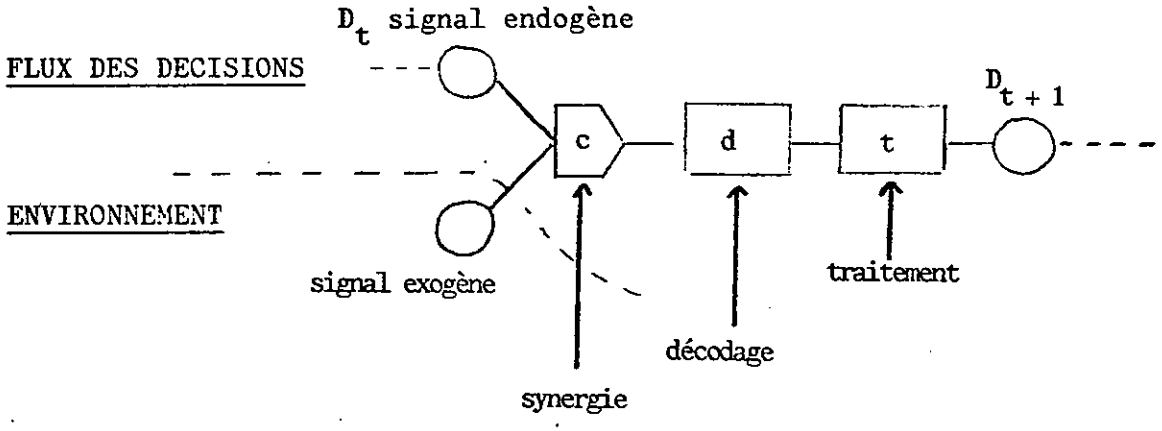
C'est à dire que l'état de la décision à l'instant t' est fonction de l'état de la décision à l'instant t

$$(s_1^{t'},\ldots,s_\ell^{t'}) = T\,(s_1^{t},\ldots,s_k^{t}) \quad .$$

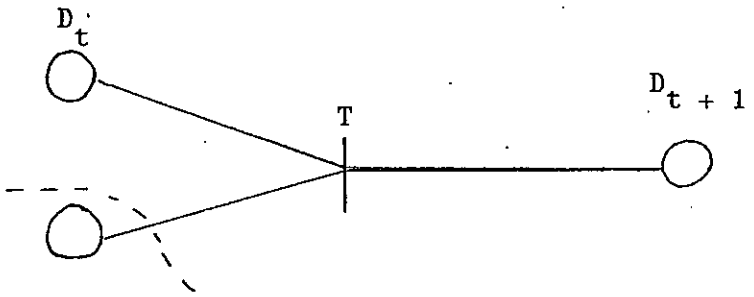
2.3. Représentation graphique.

2.3.1 Représentation des concepts :

Le processement élémentaire de l'information est réalisé par la séquence synergie - décodage - traitement. Cette séquence peut être représentée ainsi :



ou :

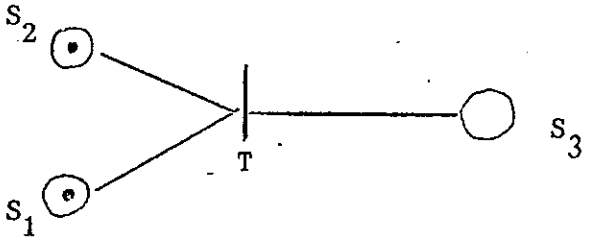


Le processus de décision complet est formé de la jonction d'une série de graphes élémentaires de ce type.

2.3.2 Marquage du graphe :

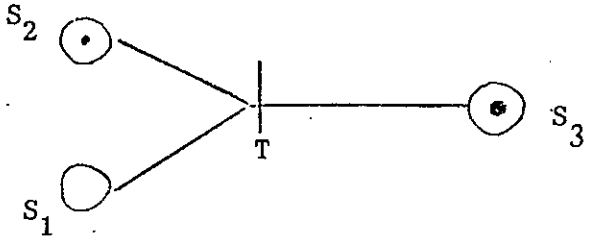
Le processus de décision évolue suivant une succession d'états stables matérialisés au moyen de marqueurs disposés sur les signaux.
Lorsqu'un signal est marqué, cela signifie que l'évènement est réalisé. L'évolution du marquage du réseau traduit la cinématique du système de décision.
Cette évolution se produit selon un mécanisme identique à celui qui régit la cinématique des réseaux de Pétri.

Soit le graphe élémentaire suivant :



S_1 et S_2 sont réalisés

Pour franchir la fonction T , on enlève un marqueur dans chaque signal précédent T et on en rajoute un dans le signal suivant :



S_3 est réalisé

ne peut être franchie que lorsque les évènements S_1 et S_2 sont
e franchissement de T prend un temps variable selon le type de
nsidéré (instantané pour les procédures automatiques, et plusieurs
mois pour les projets complexes).

2.4. Correspondances Réseaux de Petri - Formalisme.

Nous avons étudié les processus de décision en conceptualisant leurs caractéristiques essentielles selon des représentations rappelant celles utilisées dans les réseaux de Pétri.

Le concept de signal informationnel est voisin de la notion de place des réseaux de Pétri. L'un et l'autre traduisent la stabilité de l'état du système considéré.

La fonction T évoque le passage d'un état stable à un autre et présente donc la même aptitude que la transition des réseaux de Pétri.

Le marquage adopté pour suivre l'évolution du processus de décision est celui des réseaux de Pétri simples.

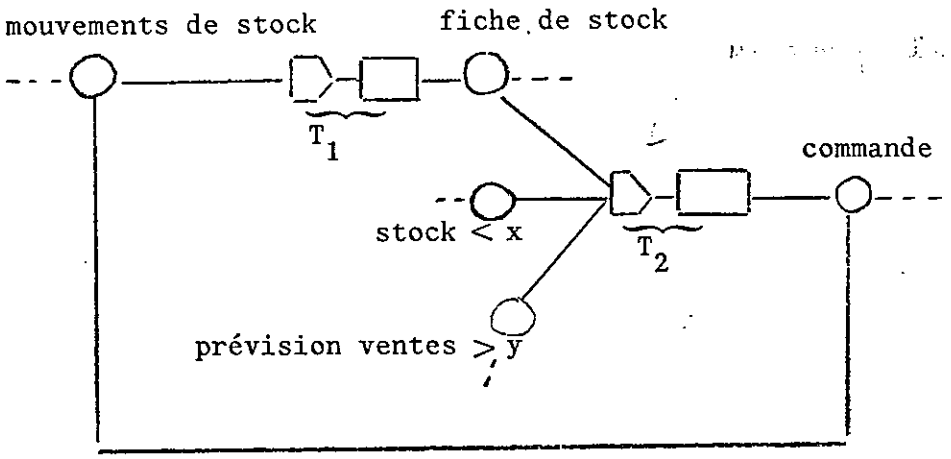
Réseaux de Pétri	Formalisme
Place	Signaux
Transition	synergie décodage traitement

En dépit des similitudes conceptuelles remarquées ci-dessus, le formalisme d'étude des processus de décision se révèle être plus riche que les réseaux de Pétri car il représente une réalité complexe, la réalité des comportements humains et non un phénomène mécanique automatique.

3. APPLICATIONS.

3.1. La décision automatique.

Le formalisme précédemment décrit permet de représenter à posteriori tous les processus de décision, des plus élémentaires aux plus complexes.
Un problème de décision bien structuré tel que la gestion des stocks peut ainsi être analysé à l'aide d'un graphe.



Sur ce réseau, on remarque d'abord l'absence de fonction de décodage. Cela est dû au fait que dans le cas de procédures de décision automatisées il n'y a pas de phase de reflexion, et les critères de décisions sont intégrés directement dans la fonction de traitement.
Il est alors possible de caractériser la décision "informatique".
Soit T l'ensemble des transformations d'information d'un processus de décision,
N l'ensemble des interventions au cours de ce processus,
$T = \{ T_i \ ; \ i \in N \}$.

On appellera décision informatique ou automatique tout processus tel que :

$$\forall \ i \quad , \ d_i : M \rightarrow E$$

$$m_i \rightarrow d_i (m_i) = m_i$$

3.2. La décision "informatisée" ou quasi automatique.

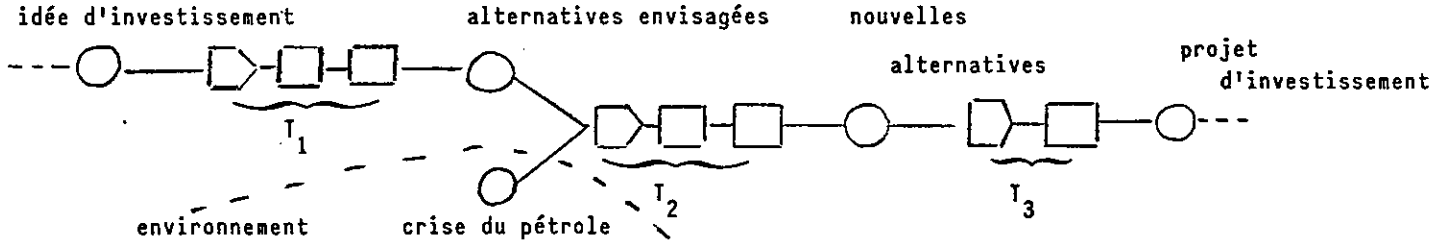
Ce type de décision est le plus fréquent en gestion. Investissements, politique salariale, négociations sont des problèmes de management où sont couramment utilisés des formalismes de traitements de l'information parallèlement à des intuitions, des jugements, et autres raisonnements humains.

De tels processus de décision ne se reproduisent jamais pour deux raisons essentielles :

- l'intervention de l'environnement au cours du processus
- l'aléa des jugements humains.

Ainsi, le réseau construit à l'aide du formalisme ne peut qu'avoir un but d'analyse.

Imaginons une décision d'investissement, nous pouvons la représenter sous une forme de graphe.



Au cours de ce processus alternent deux sortes de transformations élémentaires d'information :

T_1, T_2 se présentent selon la séquence synergie - décodage - traitement

T_3 ne comporte pas de phase décodage, car il peut s'agir d'un programme d'aide à la décision classique.

Ceci caractérise les décisions quasi automatiques. On dira qu'un processus de décision est quasi automatique si :

$$\exists\, i \in N \;,\quad d_i : M \rightarrow E$$

$$m_i \rightarrow d_i\,(m_i) = m_i$$

et :

$$\exists\ j \in N \quad , \quad d_j : M \rightarrow E$$

$$m_j \rightarrow d_j\ (m_j) = m'_j \quad \text{avec} \quad m'_j \neq m_j$$

4. CONCLUSION : LES ENSEIGNEMENTS.

Le formalisme présenté n'a qu'un but explicatif de la cinématique des processus de décision. Il est cependant possible de simuler les processus automatisés à l'aide d'un formalisme élargi. On doit alors définir un prédicat associé à la synergie et des règles d'émission des résultats du traitement sous forme de proposition logique.

Un tel schéma peut alors être utilisé dans la conception et la mise en oeuvre des systèmes d'Information Automatisés. Il peut également servir de cadre à la communication entre intervenants pour la conception et la réalisation de projets informatiques, informatique de gestion en particulier.

Ce formalisme peut aussi produire un échange nouveau de certains phénomènes économiques, les problèmes de coordination des activités des agents économiques plus particulièrement. On en déduit une série d'actions possibles de politique économique ou de management susceptibles d'améliorer la cohérence des plans des acteurs de l'économie, ou de l'entreprise.

= : = : = : = : = : =

BIBLIOGRAPHIE

E. DACLIN, M. BLANCHARD — "Synthèse des systèmes logiques" Supaero - Cepadues - 1976.

H. HECKENROTH, H. TARDIEU, B. ESPINASSE — "Modèle et outils pour la conception de la cinématique d'un système d'Information" - Rapport IRIA - 1980.

J.L. LE MOIGNE — "Les systèmes de décision dans les organisations" - P.U.F - 1974.

A. PANARIELLO — "Vers une théorie générale des processus de décision" - Mémoire de D.E.A. - Faculté d'Economie Appliquée - G.R.A.S.C.E. - 1981.

R. PAPIN — "Automatiser la direction des entreprises : mythe ou réalité ?" - Banque - Juin 1979.

= : = : = : = : = : =

ADA PLUS

ou

LES STRUCTURES ADA

présenté et commenté par

Patrick ISOARDI

1.52

C.R. Subject classifications Informatics 4.20 4.34

7.4

Résumé. "ADA langage de haut niveau, capable d'aborder avec succès les applications temps réel et la programmation système".
Qu'en est-il exactement ?
Cet article est une présentation très large de ce langage. La sémantique de l'ensemble et plus particulièrement certains points saillants sont ici développés et commentés en vue de répondre à la question précédente. La syntaxe n'y est que partiellement définie à l'occasion de quelques exemples ; ce n'est pas un manuel de programmation.

= : = : = : = : = : =

ADA PLUS

ou

LES STRUCTURES ADA

PRESENTATION.

Si l'on veut parler de ADA dans son ensemble, il est bon d'en préciser son origine. Le Département de la Défense des Etats-Unis qui semble être un des plus grands consommateurs mondial de logiciel estimait, en 1974, trop élevé le coût du développement et de la maintenance des programmes, intégrés au sein de son Ministère. Une large partie des coûts était attribuée au manque de standardisation.
A la suite de cette étude, ce département mit en concours la conception d'un langage unique qui pourrait être universel : en quelque sorte un langage A Tout Faire.
ADA est le fruit de ce concours.

Avant d'entreprendre une revue des aspects les plus significatifs de ce langage, nous allons essayer d'en résumer les caractéristiques générales.
Ce nouveau langage de programmation est fortement inspiré de PASCAL auquel il emprunte quelques uns de ses aspects les plus simples : il a à peu près les mêmes constructions de programmation structurée, en particulier la notion de type.

TYPES.

Les objets simples qui peuvent être manipulés en ADA sont les variables et les constantes. Dans un programme, le type de ces objets doit être déclaré.

Un type est une association d'un ensemble de valeurs et, très souvent sous-entendu, d'un certain nombre d'opérations définies sur ces valeurs. Il y a deux sortes de types : les types énumératifs et les types numériques dont certains peuvent être prédéfinis.

. Les types énumératifs.

Pour pouvoir spécifier aisément toutes les propriétés d'un type, celui-ci peut être déclaré.

TYPE nom IS (énumération des valeurs) ;

L'ensemble est ordonné : l'énumération définit l'ordre des valeurs. En l'absence de toute autre déclaration, les seules opérations applicables sont l'affectation et toutes les opérations de comparaison. Dans l'exemple :

TYPE FEUX IS (VERT, ORANGE, ROUGE) ;

la relation VERT < ROUGE est vraie.

. Les types numériques.

Ce sont les entiers, les réels et plus généralement des types dont l'ensemble des valeurs serait trop long à énumérer : l'ensemble étant de toutes façons dénombrable. Ils sont introduits par des déclarations telles que :

TYPE INTEGER IS RANGE I .. J ; introduit un type entier où I et J sont des bornes paramétrables.

TYPE FLOAT IS DIGITS 10 [RANGE I .. J] ; définit un type réel à virgule flottante avec 10 chiffres décimaux significatifs.

Enfin, on peut en ADA définir des types numériques en donnant le pas de variation désiré.

TYPE FIXE IS DELTA 0.01 RANGE 1.0..10.0 ;

. Les types prédéfinis.

La possibilité de définir ses propres ensembles de valeurs et surtout la précision des nombres est un important facteur de portabilité du langage. Mais ADA possède aussi des types standards prédéfinis par le compilateur. Ce sont en général les types INTEGER REAL, BOOLEAN, CHARACTER, ...

. Les opérations sur les types.

Quelque soit le type, deux opérations peuvent toujours être réalisées entre éléments d'un même type : c'est l'affectation et le test d'égalité. On peut empêcher cela en rajoutant la mention LIMITED PRIVATE : on retrouve là le langage militaire.

Les autres opérations peuvent être prédéfinies, ou entièrement définies par des fonctions et s'appliquent ou non selon la nature des objets décrits par le type.

☐ Tout ceci décrit les types de base du langage c'est-à-dire les types qui ne dépendent d'aucun autre, par opposition aux types composés qui sont construits à partir d'autres types et qui, en fait, définissent des structures.

STRUCTURES DES DONNEES.

. Le record.

Ce n'est rien d'autre que le statut correspondant à une collection d'éléments de types différents regroupés pour des raisons de programmation et qui est défini par :

```
TYPE   nom   IS
   RECORD
   - -  liste de (nom : type)
   END RECORD
```

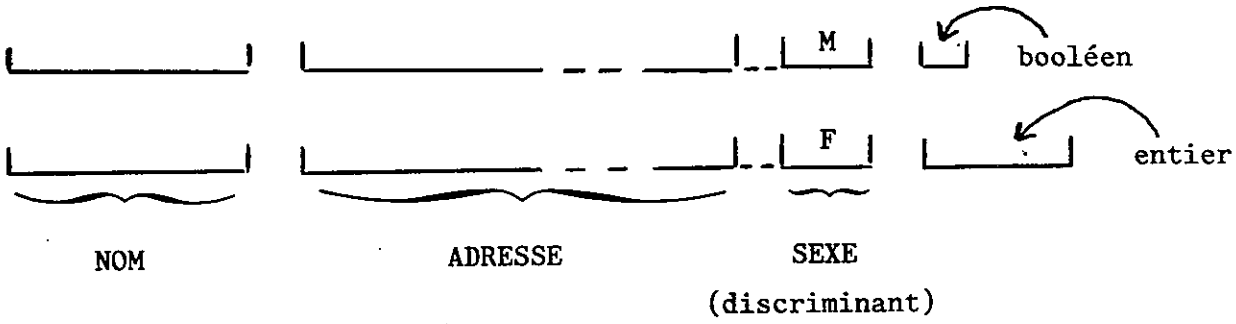
De telles données hétérogènes sont courantes dans les enregistrements sur les fichiers.

L'apport principal de ADA en ce qui concerne les records est la notion de type paramétré dans lequel les différents enregistrements d'une même série peuvent ne pas avoir la même structure. La suite de la structure dépend de la valeur d'un élément alors appelé discriminant.
Par exemple, dans un fichier client on aura pour chaque personne :

```
TYPE   GENRE   IS  (M,F)  ;
-----------------------
TYPE   PERSONNE  (SEXE : GENRE)  IS
   RECORD
   NOM : STRING (1..10) ;
   ADRESSE : STRING (1..30)  ;
   -------------------------
   CASE  SEXE  IS
```

```
        WHEN   M ⇒ SPORTIF : BOOLEAN  ;
        WHEN   F ⇒ NBR. ENFANT : INTEGER  ;
    END CASE
END RECORD
```

ce qui produit les deux structures suivantes :



On remarquera que dans la syntaxe le nom et le type du discriminant doivent être placés entre parenthèses, à la suite du nom du type record lors de sa déclaration. D'un point de vue sémantique, le discriminant ne peut être assigné que lors de l'affectation globale de tout le record : le cas d'un changement de sexe, qui modifierait la structure de l'enregistrement précédent n'est pas envisagé.
Le discriminant peut également être utilisé pour spécifier la longueur d'un tableau intervenant dans l'enregistrement.

. Les tableaux.

Ils permettent de définir une collection homogène d'éléments d'un même type que l'on peut atteindre individuellement par indexation.
La déclaration d'un tableau définit le type de ses éléments, ses index et sa dimension.
Un tableau peut avoir des dimensions déterminées

```
TYPE MATRICE IS ARRAY (1..8,1..5) OF REAL ;
```

ou non

```
TYPE TABLEAU IS ARRAY (INTEGER RANGE < > ) OF FLOAT ;
```

auquel cas, la véritable dimension sera définie par l'argument effectif de l'appel :

```
T : TABLEAU (1..10) ;
```

Ceci est un tableau de dimensions variables. Mieux que PASCAL.

. Les accès.

Ce sont des pointeurs qui font références à des valeurs d'un type spécifié. Ils permettent de gérer dynamiquement la mémoire.

TYPE REFERENCE IS ACCESS ARRAY (1..5) OF FLOAT ;

déclare un type dont les objets seront des pointeurs capables d'accéder à des vecteurs de cinq réels.

R : REFERENCE ; indique que R est un pointeur du type défini ci-dessus.

La valeur globale du vecteur se désigne par R.ALL.
R.ALL (2) ou R (2) désigne la valeur de la deuxième composante.

□ Maintenant que la présentation des différentes structures a été abordée, il est bon de revenir sur leurs utilisations pour la création de listes ou de files d'objets.

Un tableau est un support vide à la déclaration destiné à être rempli et utilisé au cours du programme. La déclaration n'est là que pour définir son architecture et réserver la place mémoire qui lui sera allouée.

Un record est essentiellement une description de structure.
Des structures plus complexes, c'est là un point fondamental, peuvent être obtenues par la déclaration de tableau de tableaux, de record avec tableaux ou de tableau de records.
Cette dernière est de loin la plus intéressante pour la description de fichier : elle permet une déclaration globale du fichier en définissant son statut (le record) et sa dimension (le tableau). On est alors en présence d'une structure aussi puissante que la file périodique du langage du MIL. Mais, même lorsque l'enregistrement est à variante, la file apériodique n'est pas égalée. Un pas reste à franchir.

Pour information, le MIL a été défini par Edmond BIANCO dans sa thèse : Etude et Formalisation d'une classe de Systèmes, 1969.

Un compilateur de ce langage a été implanté par Roland STUTZMANN sur l'IBM 360-44 de Strasbourg I en 1970-1971.

Plus de dix ans entre le MIL et ADA. C'est beaucoup !

STRUCTURES ALGORITHMIQUES.

C'est là que se trouvent les aspects les plus saillants de ADA. En dehors des différentes formes d'instructions pour la plupart classiques et bien connues : on y trouve les instructions traditionnelles que sont l'affectation, les opérations arithmétiques et logiques, les instructions conditionnelles IF et CASE, les boucles WHILE et FOR ainsi que le branchement inconditionnel GOTO , nous allons nous pencher sur quelques originalités que sont entre autres les notions de package et de tache.

Auparavant, nous rappellerons les formes traditionnelles de sous-programme. Tout programme se présente comme un ensemble d'unités de compilation ; ces unités sont emboitables. Chacun de ces modules est formé de deux parties bien distinctes :

- la partie déclarative introduite par l'en-tête PROCEDURE, FUNCTION, PACKAGE, TASK ou par DECLARE si c'est un simple bloc. Elle définit les données qui seront manipulées dans le module.

- la partie exécutive qui commence par BEGIN et se termine par END. Elle contient les instructions exécutables qui représentent le traitement proprement dit.

. Les fonctions.

Elles sont appelées à l'intérieur d'une expression arithmétique, calculent une valeur et renvoient un résultat.

Une fonction est spécifiée par :

```
FUNCTION  nom (liste des paramètres) RETURN spécification de type IS
   --     déclarations
        ┌ BEGIN
        │ - -  instructions
        └ END
```

La suite des instructions de la fonction doit se terminer par une instruction "RETURN expression" où l'expression est du type spécifié dans la déclaration.

. Les procédures.

Ce sont des sous programmes qui sont appelés à la simple citation de leur nom et de la liste des paramètres. Alors qu'une fonction ne peut avoir que des paramètres d'entrée, les procédures peuvent accepter des paramètres pour lesquels il existe trois modes possibles :

- un paramètre d'entrée IN correspond à un argument fourni à la procédure. Il se comporte comme une constante donc ne peut pas être modifié par celle-ci;

- un paramètre de sortie OUT permet de modifier la valeur d'une variable par suite de l'exécution du sous-programme ;

- un paramètre IN OUT correspond à un argument fourni au sous-programme puis renvoyé après modification.

Pour exemple : l'échange de deux tableaux nécessite un troisième tableau du même type que les deux précédents.

La procédure ci-après permet l'échange.

```
PROCEDURE  ECHANGE (A,B : IN OUT TABLEAU)  IS
  C : TABLEAU ;
BEGIN
  C : = A ;
  A : = B ;
  B : = C ;
END  ECHANGE ;
```

où C est une variable locale à cette procédure. L'échange de deux tableaux T1 et T2 est alors appelé par

```
ECHANGE (T1,T2) ;
```

Comme dans tout langage de haut niveau, une procédure ou une fonction peut être récursive.

. Les packages.

L'une des plus grandes innovations. Un package est une unité qui met à la disposition du programme ou du programmeur non seulement un algorithme mais aussi les données associées, leurs types et leurs structures, à la différence des sous-programmes.

Le plus souvent, l'intérêt est là, est d'utiliser un package tout fait dont on connait l'existence. N'est-ce-pas un enrichissement de la notion de bibliothèque de programmes ?

Un package comprend deux parties :

- la spécification qui contient toutes les informations nécessaires au programmeur ou au compilateur.

- le corps qui permet la réalisation de ces propriétés.

Ceci est très intéressant en programmation : lorsqu'un besoin particulier se présente, il suffit de le déclarer dans la partie spécification. On a alors un package dont on connait les propriétés sans se préoccuper (tout au moins momentanément) de la façon dont elles seront réalisées.

Pour prendre exemple, la nécessité d'utiliser les complexes peut se présenter dans un programme. On définit la spécification par :

```
PACKAGE  NBR COMPLEXES  IS
   TYPE  COMPLEXE  IS
      RECORD
      REEL : FLOAT ;
      IMAGINAIRE : FLOAT ;
      END  RECORD
   I : CONSTANT  COMPLEXE : = (0.0,1.0) ;
   FUNCTION "+" (A,B : COMPLEXE)  RETURN  COMPLEXE ;
   ------------------------------------------------
   FUNCTION PARTIE REELLE (A : COMPLEXE)  RETURN  FLOAT  ;
   ------------------------------------------------
END  NBR COMPLEXES  ;
```

Le corps du package sera la description de toutes les fonctions

```
PACKAGE  BODY  NBR COMPLEXES  IS
   FUNCTION "+" (A,B : COMPLEXE)  RETURN  COMPLEXE IS
      BEGIN
        RETURN (A. REEL + B. REEL, A. IMAGINAIRE + B. IMAGINAIRE)  ;
      END "+"  ;
   ------------------------------------------------------------
   FUNCTION PARTIE REELLE (A : COMPLEXE)  RETURN  FLOAT  IS
      BEGIN
        RETURN  A. REEL  ;
      END  PARTIE REELLE  ;
   ------------------------------------------------------------
END  NBR COMPLEXES  ;
```

La spécification et le corps du package peuvent être compilés séparément.

Utilisation des packages.

Afin d'imaginer dans son ensemble l'emploi que l'on peut faire d'un package, il convient de le placer dans un contexte plus large. Un package est accessible dans tout bloc entouré par le bloc où le package est déclaré.

La clause USE "nom du package" peut apparaitre n'importe où dans une partie déclarative. A ce moment là, on a accès aux objets du package sans rappeler son nom.

En compilation séparée, on suppose là que des packages sont regroupés dans des bibliothèques d'opérations standards. Ces packages ont été compilés une fois pour toutes séparément du programme que l'on est en train d'écrire. En tête de notre programme, on doit préciser la liste des packages que nous allons utiliser. C'est l'objet de l'instruction WITH "nom du package".

Tout programme ADA se déroule dans un environnement. Les définitions des données de cet environnement sont regroupées dans un package appelé STANDARD qui serait le bloc le plus extérieur. C'est un grand pas dans la programmation système.
En plus de ce package, les éléments d'entrées-sorties peuvent être prédéfinis dans le langage. Jusqu'à présent deux solutions étaient proposées :

- rien n'est défini dans le langage et chaque concepteur de compilateur réalise ses propres entrées-sorties. La perte de portabilité est certaine : c'est le cas d'ALGOL ;
- on définit quelques jeux d'instructions (cas de FORTRAN) ou des procédures standards (PASCAL) et c'est trop restrictif.

ADA a suffisamment de souplesse et de méthodes de paramétrisation pour permettre de définir des packages d'entrées-sorties : la clause GENERIC offre la possibilité de paramétriser un package par un type ou même un sous-programme.

☐ Le formalisme du langage permet un contrôle complet sur le système, les périphériques et les bibliothèques de ressources,...
C'est une approche dans la description des systèmes, mais pas une formalisation de cette notion.

. Les tâches.

ADA offre la possibilité de traitement en parallèle. A ce sujet, nous devons émettre quelque réserve sur ce parallélisme, car il est uniquement fonction de la façon dont est écrit le compilateur : les conditions et les hypothèses sur la réalisation effective du traitement n'étant pas contenues dans le langage.

L'outil principal de parallélisme est la tâche. Une tâche est une unité de traitement capable de s'exécuter en parallèle (au sens du compilateur) avec d'autres. Lorsque plusieurs processus se déroulent, ils peuvent s'exécuter indépendamment ou bien communiquer entre eux et avoir des conditions de synchronisation. ADA offre les outils pour exprimer ces conditions.
Cela semble recouvrir le champ des systèmes temps réel.

Une tâche est introduite de façon semblable à un package avec une spécification qui déclare les entrées et un corps qui contient les instructions exécutables.
Toutes les tâches déclarées dans un bloc ou un sous-programme sont démarrées "simultanément" lorsqu'on exécute le BEGIN du bloc.

Mécanisme du rendez-vous

Un point de synchronisation dans une tâche s'appelle une entrée. Elle est annoncée dans la spécification par :

```
ENTRY  nom (arguments)  ;
```

Elle est matérialisée dans le corps de la tâche par :

```
ACCEPT  nom (arguments)  DO
   - -  suite d'instructions
END
```

Les arguments sont les noms et types des variables qui doivent être fournis à la tâche. De ce fait, il va y avoir une instruction d'appel dans une autre tâche.

On distingue donc la tâche appelée de la tâche appelante.

tâche appelée

```
TASK A IS
  ENTRY M (X : type) ;
END A ;

TASK BODY A IS
  BEGIN
  - - suite d'instructions 1
  ACCEPT M (X : type) DO
  - - suite d'instructions
  END
  - - suite d'instructions 2
END A ;
```

tâche appelante

```
TASK B IS
  - - déclarations d'entrées
END B ;

TASK BODY B IS
  BEGIN
  - - suite d'instructions 1
  A.M (X) ; c'est l'appel
  - - suite d'instructions 2
  END B ;
```

Le mécanisme est le suivant :

Lors du départ, les deux tâches sont démarrées. Le rendez-vous consiste à ce que la première tâche arrivée à l'ACCEPT ou à l'appel de l'entrée considérée attende l'autre.

Lorsque les deux tâches sont arrivées au point de rendez-vous, c'est la synchronisation qui se déroule de la façon suivante :

- la tâche appelante est suspendue ;
- les paramètres sont transmis comme pour un sous-programme ;
- le corps de l'ACCEPT est exécuté ;
- lorsque le rendez-vous est terminé, chacune des tâches continue son activité.

Ainsi, une même donnée ne sera pas "simultanément" manipulée par plusieurs tâches. Si plusieurs tâches attendent à la suite d'un appel à une même entrée, les appels sont mis en file d'attente et seront acceptés par ordre d'arrivée.

Ce mécanisme peut être étendu par l'instruction SELECT qui réalise une attente sélective.

Dans une tâche appelée, elle est de la forme :

```
SELECT
     WHEN  condition 1 ⇒ alternative 1 ;
OR   WHEN  condition 2 ⇒ alternative 2 ;
     ----------------------------------
     ELSE
     - - suite d'instructions
END  SELECT
```

Si aucune des conditions indiquées n'est satisfaite, la tâche est suspendue. Dans le cas contraire, l'une des alternatives est choisie : le choix dépend de l'alternative dont les différentes formes sont :

- ACCEPT ... DO ... END ;
- TERMINATE ;
- DELAY temps en seconde ;

La notion de temps est introduite grâce à cette dernière instruction qui suspendra la tâche pour la durée de temps indiquée. Il faut pourtant prendre garde à ces attentes qui sont là encore fonctions de la façon dont le compilateur traite le parallélisme.

Une autre forme de l'instruction SELECT placée cette fois dans une tâche appelante permet d'exécuter des appels d'entrées conditionnels ou sous un certain délai.

```
SELECT
- - appel d'une entrée
ELSE
- - suite d'instructions
END  SELECT
```

```
SELECT
- - appel d'une entrée
OR  DELAY  temps ;
END  SELECT
```

L'entrée considérée n'est appelée que si l'appel peut être accepté immédiatement ou dans le délai indiqué (2ème cas). Sinon on passe à la suite des instructions.

☐ L'instruction SELECT est extrememement puissante. Elle semble permettre d'exprimer facilement tout algorithme de simultanéité. Pourtant, c'est le programmeur qui doit éviter les interblocages où toutes les tâches sont suspendues et s'attendent entre elles. Cela ne serait-il qu'un problème de "bonne" programmation ?

ADA capable d'aborder avec succès les applications temps réel ?
Il faudrait peut-être vérifier que ce langage permet aussi de décrire des systèmes logiques présentés sous forme de Réseaux de Pétri ou de Grafcet.

DIRIGISME.

ADA possède quelques outils pour orienter le travail du compilateur.

. Dirigisme en compilation : les pragmas.

Ce sont des suggestions faites au compilateur. Les pragmas ne peuvent pas être définis dans le langage. Ils sont tous prédéfinis par le compilateur.
Voici les principaux pragmas :

PRAGMA INTERFACE (langage, nom de sous-programme) ;
informe le compilateur que le corps du sous-programme est écrit dans le langage indiqué.

PRAGMA PACK (type RECORD ou ARRAY) ;
la représentation du type indiqué doit être la plus compacte possible.

PRAGMA MEMORY SIZE (n) ;
fixe à n la taille mémoire totale demandée par l'utilisateur.

PRAGMA OPTIMIZE (TIME ou SPACE) ;
préconise un gain de temps ou un gain de place mémoire.

. Les spécifications de représentation.

Une idée originale est très intéressante.

Les spécifications imposent au compilateur le mode de représentation de certaines données ou types. Ceci permet dans certaines applications de tirer parti au maximum des particularités de la machine utilisée.

Les spécifications de représentation peuvent apparaitre en fin de partie déclarative d'un bloc et s'appliquent aux objets déclarés dans cette partie. En l'absence de ces spécifications, la représentation est choisie par le compilateur.
Elles permettent de spécifier :

- la longueur d'un type ;

  FOR INTEGER'SIZE USE 16

- le codage d'un type énumératif ; A chacun des symboles d'un type énumératif, on associe un nombre entier. Ces nombres doivent être ordonnés mais non forcément consécutifs. Si je considère les instructions du SCAMP II on a :

  TYPE INSTRUCTION IS (HALT, XAE, CCL,...,CAD,CAI) ;

  FOR INSTRUCTION USE (HALT ⇒ 00, XAE ⇒ 01,...,CAI ⇒ FC) ;

L'assembleur et le désassembleur ne sont pas loin !

- la structure d'un type RECORD en précisant pour chaque composante l'adresse relative au début de l'enregistrement et sa longueur sous forme d'intervalle de bits ;

- l'adresse d'un programme par :

  FOR PRO USE AT 16 # 00FF # qui est une routine implantée en FF hexadécimal.

☐ Cet ensemble de spécifications permet des manipulations extremement faciles au niveau machine. On peut définir un package lié à une machine particulière en spécifiant les différentes instructions avec leur codage et leur format : par exemple le PACKAGE SCAMP II ;

CONCLUSION.

ADA est une évolution en matière de conception de langage. L'importance est placée dans la paramétrisation et sur les outils permettant à l'utilisateur de définir ses propres abstractions.

Cet exposé qui n'est qu'une approche du langage manque très certainement de rigueur dans la description syntaxique. De plus larges informations dans ce domaine pourront être obtenues en consultant les différents ouvrages énumérés dans la bibliographie. Pourtant, pour bien connaitre une langue, il est nécessaire de la pratiquer. Nous n'attendons plus qu'un compilateur ADA pour réaliser ce souhait. L'apprentissage se fera alors sur la machine : le compilateur jouera le rôle du maître qui guide la bonne écriture des programmes en donnant des informations suffisantes pour la correction des erreurs.

BIBLIOGRAPHIE

. Les livres :

. P. WEGNER : Programming with ADA - Prentice Hall 1980.

. DANIEL - JEAN DAVID : Le langage ADA - Editions du P.S.I. 1981.

. Les revues :

. Introduction au langage ADA par OLIVIER ROUBINE
"minis et micros" n° 128 pages 41-44
n° 130 pages 35-39

. "Software Practice and Experience".

. "ACM SIGPLAN Notices".

= : = : = : = : = : =

VOUZZAVEDIBISAR ?

Ce n'est pas parce que vous avez des lunettes sur le nez qu'il vous faut ouvrir les yeux.

Ce n'est pas parce que vous avez les yeux ouverts qu'il vous faut regarder.

Ce n'est pas parce que vous regardez qu'il vous faut voir.

Ce n'est pas parce que vous voyez qu'il vous faut comprendre.

Ce n'est pas parce que vous avez compris qu'il faut vous souvenir.

De toute manière se souvenir n'oblige pas à ouvrir les yeux.

signé : Le Masque.

Oui mais ce n'est pas parce que vous avez les yeux fermés qu'on ne vous voit pas, c'est peut être simplement parce qu'on a également les yeux fermés.

Sémantique.

Le mot "sémantique" renvoit tout naturellement à "langage" et à "mot". Ne se laissant pas enfermer dans un faux piège, on dit que "sémantique" est un mot qui décrit certaines propriétés des mots, convenablement placés dans des langages eux-mêmes encastrés dans des civilisations. La sémantique attachée au mot "sémantique" risque d'être perçue plus facilement que définie. Mais alors comment "mesurer" la "quantité" perçue. J'emploie le mot "quantité" dans le sens quantité d'information, ou moyen de replacer un concept dans un ensemble de concepts déjà organisés entre eux et que cet apport peut compléter, modifier, voire bouleverser. Par "mesurer",j'entends :l'information étant issue d'une source, moyen de communication d'une pensée, où elle est supposée intégrée, elle est perçue par un récepteur qui la communique à une autre pensée qui l'intègre à son tour. C'est une comparaison de gré à gré entre les deux "pensées" qui est censée contrôler et corriger les "dérives". Cela "présuppose" entre les deux sortes de pensées une "équivalence" des moyens mis en jeu.

Après avoir énoncé tout ceci, je n'ai pas forcément la conviction d'avoir avancé dans l'appréhension d'un processus censé converger vers un certain équilibre. Je vais donc changer de moyens et user d'une métaphore.

Je me place dans le champ formel des langages de programmation. Je parle alors de la "sémantique" d'un programme ou d'un élément de programme, lesquels s'insèrent dans un langage de programmation, lui-même faisant partie intégrante d'une civilisation informatique.

Peut-on imaginer qu'il y a des gens qui ne parlent que le "basic" ? c'est pourtant la réalité. Et les équivalences entre le basic et un certain Français (non le moindre sans doute) sont d'ores et déjà bien établies. N'est-ce point là le prologue à toute une nouvelle civilisation ?

Le programmeur qui confie son programme à l'ordinateur le fait dans un certain but, il veut obtenir un résultat qui a un sens pour lui. Mais il n'aura "son résultat" que s'il se plie aux nécessités de la démarche suivante : il doit expliquer pas à pas toutes les diverses étapes du calcul à son ordinateur, qui n'en improvisera aucune. Cette "explication" présente un "sens" pour lui, programmeur.
Il ne reste alors plus qu'à prier que ce sens soit parfaitement identique à celui qu'elle aura pour l'ordinateur. Quelquefois les voeux sont exaucés. Le sens de l'"explication" est une chose, celui du résultat en est une autre. Il est tout naturel de penser que l'ordinateur, ce cerveau électronique, est au fait de la sémantique du programme qu'on lui soumet. C'est vrai, on peut aller un peu plus loin : il est au fait de la sémantique du langage qui a servi à écrire ce programme. Puisqu'il "parle" ce langage, pourquoi ne point songer à l'intégrer, personne physique à part entière, dans la civilisation informatique ?

Grave question en vérité je vous le dis. A propos de questions je me sens fondé alors à me poser celle du sens des résultats. L'ordinateur a réfléchi, il a étalé le résultat sur une feuille blanche. Mais que pense-t-il de ce résultat, qui le lui a demandé ?

Retournons la question pour être honnête. J'imagine quelqu'un explicant à quelqu'un d'autre comment s'y prendre dans un acte à accomplir ; plantons le décor : un contremaître d'industrie automobile décrit à l'OS comment obtenir la pièce d'automobile, ou bien encore, l'inspecteur d'Académie explique à l'institutrice

débutante l'art d'enseigner la table de multiplication. Que pensent l'OS et l'institutrice du résultat obtenu ? Qui le leur a demandé ?

Voilà donc deux personnes qui me semblent parfaitement intégrables dans notre civilisation, je veux dire dans la civilisation informatique.

Mais alors, seul le programmeur parait apte à maitriser à la fois le langage et le résultat obtenu par application de celui-ci. N'avons nous pas devant les yeux l'Etre supérieur, Maitre incontesté de la civilisation future ?

Je poursuis mon raisonnement. Désormais peu de gens programment dans les langages de machine. L'élite pour la plupart utilise des langages plus nobles. Mais qui sont, à la base, incompréhensibles pour un ordinateur normalement constitué. Le Concepteur Suprême entre en jeu qui va lui apprendre. Une explication longue et complexe est soigneusement mise au point qui fournit à l'ordinateur le moyen de répondre à tout.

Essayons de suivre un peu ce que devient la sémantique dans ce dédale de transformations. D'abord le programmeur matérialise et condense son idée dans une phrase symbolique qui est soumise telle quelle à l'ordinateur. Cette phrase recouvre bien sûr programme et données. Il semble qu'elle contienne toute la sémantique que voulait exprimer l'auteur.Or, cette sémantique dépend exclusivement de ce qui a été mis dans la phrase symbolique. Comment procède l'ordinateur pour travailler, et bien il lit une explication qui est rédigée dans son langage à lui, et qui le renvoit aux éléments de la phrase symbolique, aux lettres qui les composent. C'est selon ce qui est rencontré dans la phrase, que le programme qu'il lit lui enjoint de faire telle ou telle chose. En somme il travaille un peu comme un élève appliqué qui, à chaque caractère chinois rencontré se réfèrerait à sa grammaire qui lui indiquerait d'écrire quelque chose en regard. Peut-on dire pour autant que cet élève comprend le chinois ?

Revenons à notre ordinateur, dans les bons cas on sait qu'il va bien faire ce qu'en attend le programmeur, après avoir épluché toute la phrase le résultat est là, sans que l'ordinateur ait rien "compris". Travail parfaitement aveugle en suivant les indications du compilateur. Mais alors où est passée la "sémantique" ?

Et ce résultat obtenu sous forme de listes plus ou moins longues de nombres, n'ayant d'ailleurs pas toujours signification de nombres. Que peut en dire notre ordinateur ?

Vraiment rien, sauf, bien sûr, s'il dispose d'un autre compilateur lui permettant d'appliquer dessus un autre calcul tout aussi arbitraire pour lui.

J'essaye de résumer ce qui vient de se passer :

Quelqu'un s'est exprimé dans un langage qui n'est pas celui de l'ordinateur. Il est apparu une certaine quantité de sémantique, décrite sous une certaine formulation syntaxique. L'ordinateur lisant le compilateur associé au langage, transforme la phrase symbolique, soit en une nouvelle phrase symbolique à soumettre à un compilateur de déroulement, soit en une phrase de son propre langage qui lui permettra ensuite d'effectuer le calcul, soit il donne les résultats au fur et à mesure.

L'ordinateur a opéré ainsi une transformation purement syntaxique, et ceci dans les trois cas.

Le "résultat" lui-même, si l'on y regarde d'un peu près, n'a vraiment de sens que pour l'auteur du programme. C'est donc lui qui fait réapparaître à l'autre bout de la chaîne la signification.

En d'autres termes, imaginons qu'un discours soit prononcé, dont le but est d'obtenir une certaine réalisation, des actions vont être entreprises, qui font appel à des "routines" d'interprétation des divers points du discours ; la réalisation se bâtit ainsi jusqu'à l'obtention du résultat final. Alors, seul le promoteur, auteur du discours, peut mesurer l'intérêt du résultat. Il est à peu près évident que la plupart des acteurs n'auront rien compris à rien. Si le discours était bon, c'est de la syntaxe qui se sera transférée tout au long de la chaîne, la sémantique y sera demeurée impénétrable.
Allons un peu plus loin. Imaginons une déclaration qui définit une intention d'action. Mais supposons alors qu'un empêchement apparaisse qui se mette en travers de la réalisation, par exemple un blocage dû à des frustrations ou à toute autre cause. La non atteinte du résultat fait qu'il apparaît un point d'accumulation de la sémantique. En effet, si l'intention persiste et le blocage également, un processus complexe s'élabore qui pousse à rechercher à tout prix un moyen d'aboutir.
S'il n'apparaît pas à un moment donné un "défoulement" quelconque, qui va décharger cette quantité de sémantique, on risque d'atteindre un processus paroxystique dangereux.

Des esprits non avertis pourraient ainsi déduire de tout cela que la folie est finalement l'expression la plus pure de la sémantique.
Pour ma part, j'aimerais insister sur le fait que le phénomène sémantique n'est pas aussi évident qu'il n'y parait pas. Et pour cela je reprends l'exemple de l'ordinateur qui lit son compilateur.

En fait, ce compilateur est un programme comme les autres, et il a bien fallu l'écrire un jour ou l'autre. Or, cet acte a pu être commis en langage d'ordinateur, ou en langage autre. Et on peut se livrer à ce jeu : il existe forcément une version en langage machine du compilateur, l'ordinateur lit ce compilateur et on lui soumet la version du même compilateur écrit dans le langage qui lui est associé.
Question : que se passe-t-il ?

Edmond BIANCO

= : = : = : = : = : = : =

AUTOMATE PROGRAMMABLE

Interpréteur de réseaux de Petri

J.C. FUMANAL

Troisième partie : Utilisation - Exemples.

Après avoir défini les spécifications de l'automate basées sur le formalisme des réseaux de Petri (RDP), nous allons dans cette troisième partie aborder son utilisation. Deux exemples permettront d'en apprécier les applications :

Le premier concerne la conduite d'un processus dont nous avons réalisé la synthèse du RDP associé dans la première partie de cet article (système de réenclenchement de disjoncteur). Il s'agit en l'occurrence d'un RDP sauf (une seule marque par place).

Le deuxième, à partir d'un RDP plus général, montre la commodité des opérations à effectuer pour le simuler et le vérifier.

Ces deux exemples non exhaustifs doivent être intégrés dans l'ensemble des procédures séquentielles susceptibles d'être gérées par cet automate.

X. UTILISATIONS : LANGAGE ET CARACTERISTIQUES DE L'AUTOMATE.

Celles-ci sont précisées en annexe. (Il s'agit en fait d'un mode d'emploi distribué aux étudiants). Elles résument :

- Les caractéristiques syntaxiques et l'organisation des phrases.
- L'utilisation des temporisateurs.
- Les messages d'erreurs.
- Les commandes de l'éditeur.

A l'aide de l'éditeur, l'utilisateur introduit les caractéristiques de son réseau et ensuite en demande la simulation.

L'exécution peut être arrêtée par une action sur la touche "retour chariot" du terminal. A ce moment là, des modifications peuvent être apportées sur la structure et le marquage du réseau.

La reprise de l'exécution est obtenue par une action sur les touches E ou C :

- touche E : le réseau est considéré dans un état initial (pas de "passé") ;
- touche C : le contexte du réseau précédant l'arrêt de la simulation est restitué (état des temporisateurs, transitions déjà franchies...).

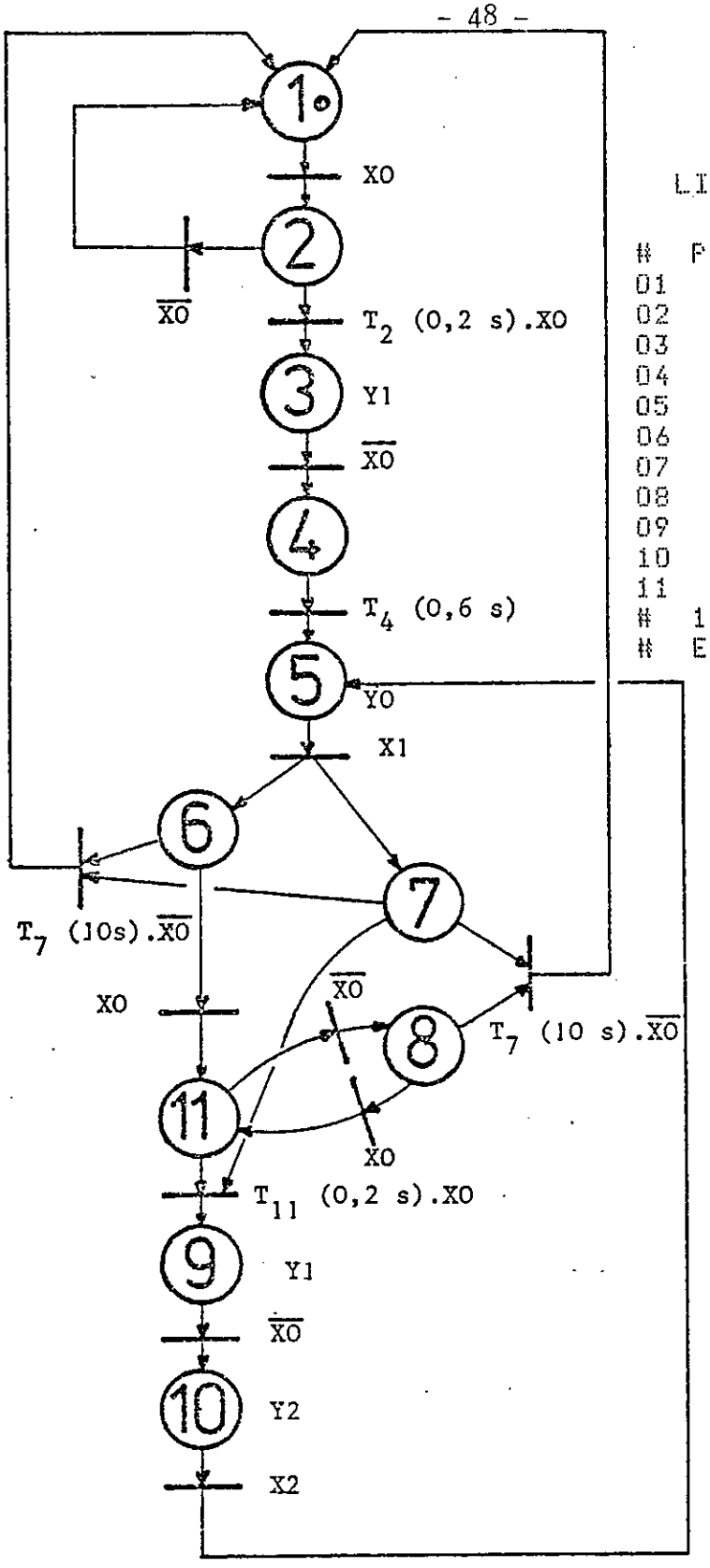
Dans le cas d'un départ de simulation, les deux touches ont le même rôle.
L'automate étant sensible à des niveaux logiques (durée indéfinie) et non à des fronts (actions ponctuelles) ne doit considérer que les éléments temporels liés au réseau pour les évolutions futures. La touche C permet, lors d'une reprise d'exécution, de restituer ces conditions.

XI. EXEMPLES.

A. Système de réenclenchement de disjoncteur.

La figure 10 reproduit le RDP associé à ce système, dont nous avons étudié la synthèse précédemment.
Les variables électriques étant définies, le programme peut être rédigé.

Pour cela, à chaque place nous associons une "phrase" qui contient dans une première partie les équations décrivant les transitions de sortie de la place. Après un " ; " figurent les équations décrivant l'activation des variables de sorties sur la place.
Il est toujours conseillé lors du franchissement d'une transition de désactiver les sorties de la place à démarquer.



LISTING DU PROGRAMME

```
#  P1:11
01  X0=P2 ; =/Y0./Y1./Y2
02  /X0=P1 , T1.X0=P3.Y1
03  /X0=P4./Y1
04  T3=P5
05  X1=P6.P7./Y0 ; =Y0
06  X0=P11
07  T50./X0(P6+P8)=P1
08  X0=P11
09  /X0=P10./Y1 ; =Y1
10  X2=P5./Y2 ; =Y2
11  T1.X0.P7=P9 , /X0=P8
#  1M1
#  E
```

Fig.10 ; Système de réenclenchement. Représentation fonctionnelle à partir des variables électriques.

Selon les règles syntaxiques décrites en annexe et d'après le réseau de la figure 10, sur la place 1 il n'y a qu'une seule transition de sortie validée par X2 et amenant à P2. Sur cette place initiale, il est bon de mettre toutes les sorties à 0 . La phase associée à la place 1 s'écrit :

1 : X0 = P2 ; = /Y0./Y1./Y2 .

Selon les mêmes règles, la phase associée à la place 2 qui comporte deux transitions de sorties s'écrit :

2 : /X0 = P1 ; T1 . X0 = P3 .
(T1 = temporisateur de durée 1 x 0,2 secondes).

Sur la place 3, Y1 est mis à 1 et on désactive cette sortie lors du franchissement de la transition de sortie de cette place, soit :

3 : /X0 = P4./Y1 ; = Y1 .

L'absence de déclaration avant le signe ' = ' signifie qu'il n'y a pas de conditions liées à l'activation des variables du 2ème membre.
Dans le cas d'une machine de Moore (état des sorties liées aux places) on peut activer les sorties d'une place sur la transition amenant à cette place. Dans le cas de Y1 sur la place 3 on peut adopter la rédaction suivante :

2 : /X0 = P1 , T1 . X0 = P3 . Y1
3 : /X0 = P4./Y1

Une des versions possible du programme représentant le réseau peut s'exprimer ainsi :

1 : X0 = P2 ; = /Y0./Y1./Y2
2 : /X0 = P1 , T1.X0 = P3.Y1
3 : /X0 = P4./Y1
4 : T3 = P5
5 : X1 = P6.P7./Y0 ; = Y0
6 : X0 = P11
7 : T50./X0 (P6 + P8) = P1
8 : X0 = P11
9 : /X0 = P10./Y1 ; = Y1
10 : X2 = P5./Y2 ; = Y2
11 : T1.X0.P7 = P9 , /X0 = P8

Sur la place 7, les deux transitions de sorties qui amènent sur la place P1 sont réunies dans une seule équation (on aurait pu écrire :
7 : T50./X0.P6 = P1, T50./X0.P8 = P1).

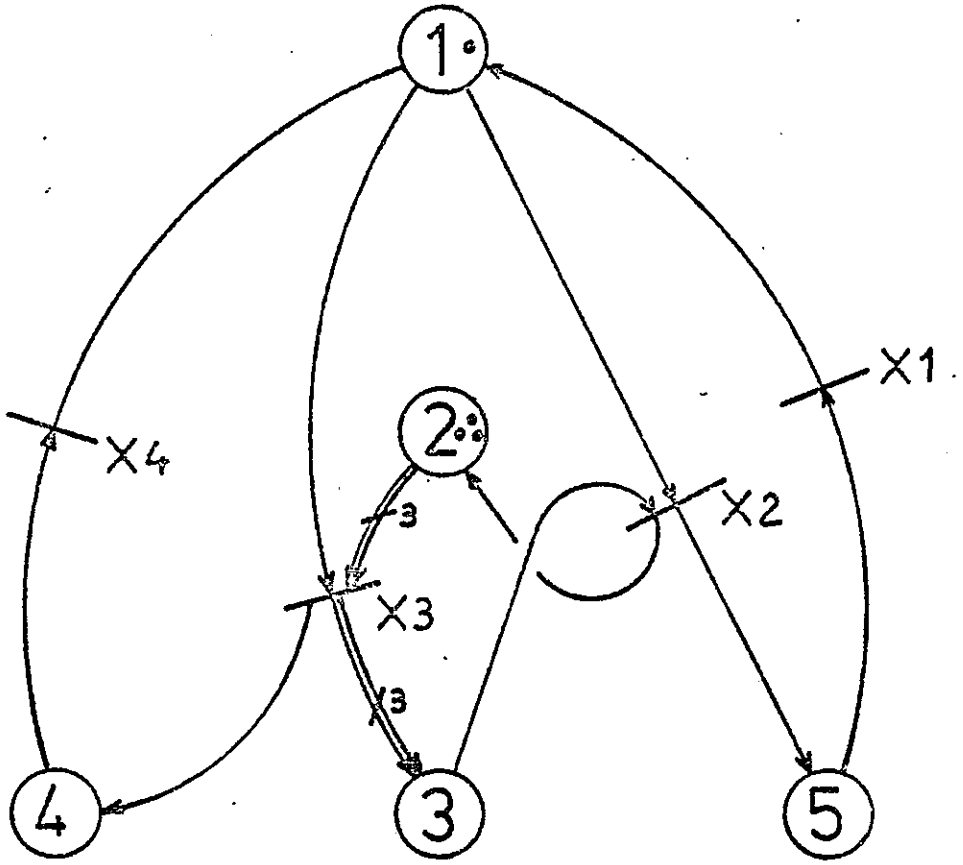
Le listing du programme, obtenu par la commande 'P' de l'éditeur, est donné à côté du réseau de la figure 10.
'1 M 1' permet de marquer initialement le réseau (1 marque sur la place 1).
Les temporisateurs sont associés aux places où ils sont activés.
Une fois la simulation lancée (touche 'E'), l'utilisateur peut vérifier sur les E/S le comportement de l'automatisme.

B. Vérification des évolutions dans un RDP.

Dans ce cas, nous allons nous intéresser directement au réseau, la synthèse de celui-ci étant supposée résolue.

La figure 11 en donne la représentation. De prime abord, on peut constater que le parcours empruntant les places 1 et 4 et les transitions X3 et X4 est effectué une fois tous les 3 cycles empruntant le chemin X2, P5, X1, P1. Les places 2 et 3 n'ont qu'un rôle de validation de ces parcours.
Le programme du réseau est donné sur la figure 11. Il y figure également les arrêts de la simulation et le listing du marquage après franchissement des différentes transitions.
La relance de l'exécution est ici demandée par la commande C .

Il est impératif qu'une transition ne soit décrite qu'une fois sur une des places sources. Dans le cas où un temporisateur figure dans l'évènement, la description est obligatoirement effectuée sur la place source de référence du temporisateur.




```
#  P1:5
01   X2.P3=P5.P2                    } programme
02   3(X3.P1)=3.P3.P4
03
04   X4=P1
05   X1=P1
#  1M1                              } Marquage initial
#  2M3                              }
#  L                                }
01   1                              } listage du marquage.
02   3                              }
#  E          Execution
#  L                                }
01   1                              } Reprise du contrôle : sans action sur
02   3                              } les variables X, on retrouve le
                                    } marquage initial
#  C          Execution  et action sur X3
#  L                                }
03   3                              } Arrêt et listage : les marques
04   1                              } ont evolué conformément au
                                    } franchissement de X3
#  C          Execution et action sur X4
#  L                                }
01   1                              } Arrêt : la transition X4 a
03   3                              }       été franchie
```

```
#  C          Execution et action sur X2
#  L                                }
02   1                              } ici : la transition X2 a été
03   2                              }       franchie
05   1                              }
#  C          Execution et action sur X1
#  L                                }
01   1                              }
02   1                              } X1 a été franchie
03   2                              }
#  C          Execution et action sur X2 puis sur X1
#  L                                }
01   1                              }
02   2                              } nouvel état du marquage.
03   1                              }
#  C          Execution et action sur X2 puis sur X1
#  L                                }
01   1                              } nouvel état du marquage
02   3                              }
                                    etc ...
#
```

FIGURE 11 : Exemple de vérification du déplacement des marques dans un réseau de Pétri.

XII. CHAMPS D'APPLICATIONS ET LIMITATIONS.

Cet automate a été initialement réalisé pour les besoins de l'enseignement de l'automatisme. Il possède donc les caractères suivants :

* Il est pratique à mettre en oeuvre :
  - réalisation matérielle simple et faible prix de revient,
  - langage peu contraignant et très proche des définitions fonctionnelles des RDP,
  - 12 messages d'erreur assurent une grande sécurité d'utilisation.

* Il permet de traiter les réseaux de Petri :
  - sensibilité à des niveaux logiques,
  - receptivité possible à toutes les variables définies,
  - traitement accepté jusqu'à 9 marques par place ou par capacité de transfert des arcs.

* Il est toutefois limité par les points suivants :
  - necessité d'utiliser un terminal,
  - 8 entrées, 8 sorties et 50 places définissables (pouvant éventuellement être portées très facilement à 100),
  - la conservation des programmes n'est pas permise,
  - restriction sur l'utilisation des temporisateurs. Ceux-ci sont toujours activés à partir d'une place source de la transition où ils sont affectés L'activation peut être cependant conditionnée par des états de marquages d'autres places.
  - le temps de traitement est de l'ordre de 15 mS par place marquée. D'une part l'interprétation du langage entraine des pertes de temps provoquées par les répétitions systématiques d'analyses, et d'autre part le μP 9980 est un 'faux' 16 bits où le multiplexage des informations sur le bus de 8 bits ralentit considérablement les opérations.
  - le déplacement simultané des marques franchissant plusieurs transitions n'est possible qu'à partir d'une place.. Ce défaut de synchronisme, qui peut apparaître pour ces cas complexes de RDP, necessite alors une rédaction judicieuse du programme pour l'éliminer.

XIII. CONCLUSIONS.

Utilisé depuis 2 ans par les étudiants, cet automate se caractérise par l'efficience de son langage et sa simplicité d'utilisation.

Il constitue ainsi un outil de vérification de procédures séquentielles avant leur réalisation matérielle et un dispositif de synthèse des automatismes.

Une version en cours de réalisation supprimera certains inconvénients, notamment :

- la sauvegarde des programmes par adjonction d'un programme d'écriture lecture sur cassettes magnétiques et sur EPROM,
- le travail à partir du clavier de la carte.

A l'aide d'interfaces adéquates, il permettra la conduite de processus industriels.

BIBLIOGRAPHIE.

Synthèse des systèmes logiques - E. DACLIN, R. BLANCHARD.
Pratique séquentielle des réseaux de Pétri - S. THELLIEZ
Modélisation et analyse des systèmes utilisant des réseaux de Pétri
(Document de travail INP Grenoble).
Rapports des journées d'études de l'AFCET sur les automates programmables.
Revue "L'ingénieur et le technicien de l'enseignement technique" :
articles sur les RDP et le Grafcet parus dans les
numéros 214 - 215 - 219 - 221 à 225.

Système d'évaluation et de mise au point utilisé pour la conception de l'automate : Texas Instrument :

- carte 990/100 M unité centrale.
- carte 990/302 logiciel de développement - assembleur 9900.
- carte 990/206 mémoire 16 K.

= : = : = : = : = : =

ANNEXE

AUTOMATE PROGRAMMABLE

Interprêteur de réseaux de Pétri : Utilisation

I. CONSTITUTION.

La configuration matérielle comprend une carte 990/189 (Texas Instrument) et un terminal permettant à l'utilisateur de dialoguer avec l'automate (liaison RS 232 C).

Huit entrées (X0 à X7) et 8 sorties (Y0 à Y7) sont utilisables pour les liaisons extérieures (logique TTL).

II. INTRODUCTION DES DONNEES (EDITION) - COMMANDE DE L'EDITEUR.

Le niveau des données à fournir correspond aux spécifications fonctionnelles des réseaux de Pétri et du modèle GRAFCET. Cependant, dans ce dernier cas, les marques restent des objets indivisibles.

La synthèse de l'automatisme est obtenue en introduisant sur le terminal, selon les règles conventionnelles de l'écriture, les "phrases" relatives aux différentes places du modèle. Chaque "phrase", associée à une place, contient, sous la forme d'équations logiques, les conditions d'évolution ainsi que les affectations des sorties sur la place.

La connaissance de l'état du réseau (marquage), la modification et l'insertion de données au niveau des phrases et du marquage, la reprise de l'exécution, permettent de suivre et de contrôler l'évolution du système logique simulé.

Les communications se font à l'aide de l'éditeur. Les différentes commandes de celui-ci sont explicitées en annexe.

L'utilisateur peut définir 50 "places" (étapes) numérotées de façon quelconque entre 0 et 99.

Une "phrase" peut contenir jusqu'à 99 caractères ASCII et la mémoire de phrases permet le stockage de 1204 caractères dont 1 délimiteur pour chaque phrase.

III. CARACTERISTIQUES SYNTAXIQUES.

A) Assignation

- Variables :

P : place (étape)
X : entrée
Y : sortie
T : temporisateur

- Opérateurs :

. : ET logique
+ : OU logique
( ) : mise en facteur d'expressions logiques
= : séparateur situé entre l'évènement et l'action qui en découle
, : séparateur d'équation
; : séparateur situé entre les relations d'incidences et les équations de sortie
/ : complémentation d'une variable, déclaration d'un temporisateur réarmable
< CR > : touche retour chariot, indique une fin de "phrase"

- Chiffres :

Après une variable ou entre opérateurs, ils permettent d'exprimer : le marquage et le démarquage à réaliser, le numéro d'une variable, la durée d'une temporisation par pas de 0,2 secondes. Ils sont obligatoirement suivis par un opérateur et seuls les deux derniers chiffres introduits sont pris en compte.

- Espaces (blancs) :

Utilisés pour la mise en page à l'édition, ils sont ignorés lors du traitement.

B) Organisation des phrases

(La notation <F> signifie : introduction sur le terminal de l'expression correspondant à F).

A chaque place est associée une "phrase" qui comprend 2 parties séparées par un " ; " et est terminée par <CR> .

Format : <E1> ; <E2> <CR>

E1 : ensemble des relations explicitant les conditions d'évolution à partir de la place.

E2 : équations combinatoires précisant l'activation des sorties sur la place lorsque le marquage est suffisant.

C) Les relations d'évolutions E1

Séparées par une virgule, elles définissent chacune une transition de sortie de la place et ont le format suivant :

<m> (<V>) = <W> <Y>

m : nombre de marques à enlever à la place si la transition est franchie.

V : combinaison logique (évènement) liée à la transition de sortie. Ce test peut être effectué sur les variables précédemment définies.

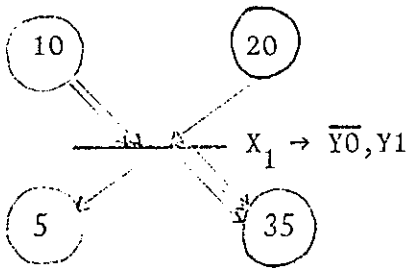
W : explicite le nombre de marques à ajouter à chaque place d'arrivée si le premier membre de l'équation est vérifié. Exemple : 3.P35.1.P5 signifie 3 marques à ajouter sur la place 35 et 1 sur la place 5 si la transition est franchie.

Y : ensemble des sorties à activer lors du franchissement de la transition.

La parenthèse n'est pas nécessaire si M = 1 ou peut être remplacée par "." si V ne comporte pas de réunion logique.

Lorsque le nombre de marques n'est pas précisé, la valeur 1 est prise par défaut (réseaux saufs).

Exemple :



La phrase associée à la place 10 décrit la transition validée par X1 et qui est franchie si P10 possède 2 marques et P20 une marque. Le franchissement amène une marque dans P5 et 3 dans P35, et provoque la mise à 0 de Y0 et à 1 de Y1.

10 : 2 (X1.1.P20) = 1.P5.3.P35. /Y0.Y1 ou expression équivalente :
10 : 2.X1.P20 = P5.3.P35. /Y0.Y1 .

Si l'évènement nécessite le test du marquage d'une place sans démarquage, il suffit de faire précéder la déclaration de la place par un "/" .

D) Equations combinatoires de sorties E2.

Format : <C> = <d>

C : combinaison des variables à tester. Mettre directement " = " si il n'y a pas de test à réaliser.

d : variables de sorties à activer. Il est à noter que chaque sortie déclarée prend l'état logique de C (pour complémenter l'état d'une sortie, faire précéder "Y" par "/").
Chaque équation de sortie est séparée par une "," de la suivante.

IV. TEMPORISATEURS.

Des temporisateurs peuvent être utilisés dans les relations d'évolutions (partie <V> des équations) et dans les équations de sorties (partie <C>).

Un temporisateur est activé ou maintenu actif seulement si la place est marquée et si l'élèvement qui le précède est réalisé. Ainsi, ils permettent pendant le temps précisé, de rendre indisponible les marques arrivant sur une place ou d'effectuer une temporisation à partir de la réalisation d'un évènement activant une transition ou des sorties.

- temporisateurs simples :

Format : T <M> temporisation : M x 0,2 secondes

Ils n'autorisent qu'une seule fois le franchissement au bout du temps écoulé, si la transition reste encore franchissable.

- temporisateurs réarmables :

Format : /T <M>

Ces temporisateurs autorisent le franchissement, s'ils restent activés, tous les intervalles de temps précisés.

V. MESSAGES D'ERREURS.

Ils délivrent l'utilisateur des vérifications qui seraient nécessaires pour éviter un dépassement de capacité de traitement de l'automate.

Les "erreur 9" signalent quelques anomalies dans la conception du réseau.

Les erreurs identifiées à l'exécution sont précédées du numéro de la place où la phrase est incorrecte. Le contrôle est rendu à l'utilisateur.

** ERREUR 0 : erreur de syntaxe - variable P déclarée avant une parenthèse.

1 : plus de 98 caractères ASCII dans une phrase.

2 : plus de 50 places définies.

3 : dépassement de capacité de la mémoire de phrases (1204 caractères).

4 : déclaration d'une place inexistante.

5 : plus de 10 temporisateurs en activité (la désactivation à lieu lorsque les marques sont stabilisées) - un temporisateur est situé après un "+" dans une parenthèse. - plus d'un temporisateur réarmable déclaré dans une relation d'évolution.

6 : plus de 3 transitions simultanément franchies sur une place ou plus de 10 transitions encore franchissables dans le réseau.

7 : détection d'un marquage supérieur à 9. Le dernier marquage de la place trop marquée est restituée à celle-ci.

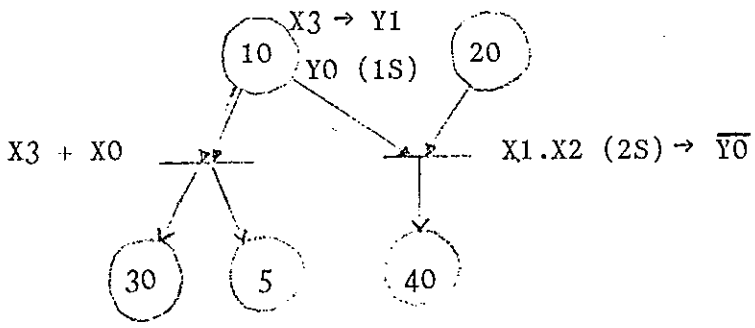
8 : plus de 5 places sources sont à démarquer lors d'un franchissement.

9 : * 1 : système instable (plus de 50 évolutions successives).

* 2 : réseau non marqué.

M : conflit de transition sur la place M (réseau sauf).

VI. EXEMPLE DE REDACTION DE PHRASE.



Sur la place 10, Y0 est activé 1 seconde après le marquage. La transition X1X2 est franchissable 2 secondes après la réalisation de l'évènement. Le franchissement provoque la mise à 0 de Y0.

La rédaction de la phrase associée à la place 10 est la suivante :

10 : 2 (X3 + X0) = 1.P30.1.P5, 1 (X1.X2.T10.1.P20) = 1.P40./Y0 ; X3 = Y1, T5 = Y0.

ou plus simplement :

10 : 2 (X3 + X0) = P30.P5, X1.X2.T10.P20 = P40./Y0 ; X3 = Y1, T5 = Y0.

Le deuxième membre d'une équation ne peut comporter comme opérateur que le ET logique.

Si le réseau n'est pas sauf (pas plus d'une marque par place), une transition ne doit être décrite qu'une seule fois.

EDITEUR : COMMANDES

Format		Description de la commande
<N> : <phrase> < CR>	:	Introduction des données (N = numéro de place) <CR> : touche Retour Chariot.
<CNTRL H>	:	Correction des données. A chaque action, cette commande, utilisée lors de l'introduction des données, élimine de la mémoire de phrase le dernier caractère (erreur de frappe par exemple).
P <N1> : <N2> <CR>	:	Impression sur le terminal des phrases associées aux places numérotées entre N1 et N2. Sont listés les numéros de places et les phrases correspondantes.
P <N> <CR>	:	Impression sur le terminal de la phrase associée à la place N. Ex : # P10 10 X1 = P2 , X0 = Y0
<N> M <Nm> <CR>	:	Affectation de Nm marques à la place de numéro N. Nm compris entre 0 et 9. Ex : # 10M1
L	:	Impression sur le terminal du marquage du réseau. Sont listés les numéros des places marquées et le nombre de marques contenues. Ex : # L 10 1 #
R	:	Remise à zéro de la zone mémoire de phrase. L'automate émet un message d'initialisation.
E	:	Exécution, reprise de l'exécution du programme. Le marquage du réseau est considéré comme état initial.
C	:	Reprise de l'exécution du programme dans le contexte précédant cette demande. Cette commande peut être utilisée s'il n'y a pas eu modification des phrases par l'utilisateur.
<CR>	:	Arrêt de l'exécution. Le contrôle est rendu à l'utilisateur.
<F>	:	signifie action ou introduction de l'expression correspondant à F . Par exemple <CR> signifie action sur la touche retour chariot.
<CNTRL H>	:	est la 2ème fonction de la touche H. Pour cela, maintenir appuyée la touche contrôle.

AUTOJECTION ET COMPILATEURS

Un bon travail de compilation ne s'improvise pas. Il faut se donner un ensemble d'outils et de méthodes cohérents. Compiler signifiant,en gros, prendre en compte les propriétés conventionnellement contenues dans une phrase symbolique censée appartenir à un langage symbolique connu, au moyen d'un programme décrit dans un deuxième langage, qui, lui, est généralement un langage machine. Ou proche d'un langage machine.

Dans la compilation on fait quelque chose en plus : on prépare le programme qui est construit à l'utilisation qui en sera faite.

Bien que toutes ces choses soient inextricablement mêlées dans une réalisation, elles procèdent, comme je l'ai déjà dit, de concepts différents. Dans la conception du compilateur je distinguerai donc soigneusement les divers niveaux d'approche.

Il me faut donc deux langages de programmation. L'un, un langage machine, choisi cependant suffisamment symbolique, ou abstrait, pour être assez proche de n'importe quel langage machine couramment utilisé, sans comporter pour autant les défauts inévitables de l'un quelconque d'entre eux. L'autre sera un langage symbolique, non trop compliqué, mais comportant les propriétés essentielles qui interviennent dans les grands langages de programmation. Le compilateur de ce second langage sera décrit au moyen du premier. On peut imaginer tout de suite que si ce premier langage est "bien" conçu, l'écriture du compilateur, objet complexe, en sera facilitée. Je vais donc munir ce premier langage d'un ensemble de propriétés qui se justifient par leur utilisation. Je vais en quelque sorte me munir d'une systématique qui est destinée à me faciliter l'adaptation d'un langage à l'autre.

LE PREMIER LANGAGE, OU LANGAGE DE LA MACHINE.

J'ai donc besoin d'une machine et de son langage. Je choisis la machine et la procédure formelles, et je renvoie pour l'ensemble de leurs propriétés à la référence [1] . Je ne vais, ici, reprendre que l'essentiel. En particulier la structure des instructions qui doit demeurer simple pour que le langage puisse être saisi rapidement. Je m'étendrai davantage sur la structure de la configuration. En effet, celle-ci est fondamentale pour ce qui va suivre. Pratiquement rien dans la partie algorithmique du langage ne forcera à la respecter, afin de conserver un maximum de simplicité de forme à celle-ci.

Je renvoie également à la référence [1] , pour l'exposé des notions de paramètre formel et de variable locale. Je n'insiste, ici, que sur l'universalité de ces notions. Constituer un programme en rassemblant un nombre quelconque de programmes qui existent déjà est pratique courante. On arrive même ainsi à constituer des édifices considérables. Il arrive également que les diverses pièces de l'assemblage aient été écrites en utilisant des langages différents. Le problème le plus aigu qui se pose à ce moment-là est précisément celui de l'assemblage. C'est le problème qui correspond à la notion de procédure et d'insertion de procédure. C'est l'application de cette notion qui matérialise en quelque sorte l'interface enchaînant deux programmes dont l'un fait appel à l'autre.

La notion de procédure se subdivise elle-même en plusieurs notions dont on pourrait dire qu'elles sont plus élémentaires. Il s'agit précisément des paramètres formels et effectifs, des variables locales, des statuts, des types, etc... Ce sont les deux premiers cas qui vont nous préoccuper dans les lignes qui suivent. Nous nous préoccuperons,également, de ce qui concerne l'insertion au sens algorithmique.

LA CONFIGURATION.

Pour une procédure déterminée, il existe une configuration sur laquelle celle-ci s'applique. Je dirai que cette configuration se compose de trois parties : Pi, Pf, Pv.

[1] Informatique Fondamentale.E. Bianco, Birkhauser Verlag, 1979.

Ces parties symbolisent les rôles joués par les variables qui les composent. Dans Pi on trouve un jeu de variables destinées à réaliser l'insertion. Dans Pf , ce sont des variables qui représentent les paramètres formels, et permettent d'atteindre aux paramètres effectifs. Et dans Pv il y a quelque chose qui correspond aux variables locales.

A ce point du choix des structures, je suis amené à faire une remarque. On constate, en terme de procédure formelle, que les variables locales sont représentables de deux façons différentes : soit par la valeur, et alors dans l'algorithme on la désigne par l'écriture [a , si "a" est le numéro de case (relatif) qui contient la valeur, soit par le nom, et alors dans l'algorithme on la désigne par [a,0 , si "a" est la case qui contient l'adresse de la variable.

Ces deux possibilités peuvent être, bien sûr, mélangées, mais ce qu'il faut voir ce sont les différences d'implications des deux cas. Dans le second cas il faut créer, en plus, une configuration des valeurs locales, ceci peut paraître dispendieux en place occupée. Mais l'insertion de la procédure s'en trouve simplifiée, car paramètres formels et variables locales sont référencées dans l'algorithme par la même forme d'opérande. Ce choix a une implication dans la construction du compilateur.

Introduisant alors une notion de gestion de la mémoire, je pars de la propriété suivante :

Propriété 1.

La configuration attachée à un algorithme n'existe qu'à partir du moment où on commence à dérouler cet algorithme, elle n'existe plus - n'a plus de sens - à partir du moment où on sort définitivement de l'algorithme.

N'oublions pas que tout algorithme ici est une procédure, donc possède une seule entrée et une seule sortie. Sortir définitivement signifie passer par cette sortie unique. Un nombre quelconque de déroulements ultérieurs verront à chaque passage par l'entrée : création de la configuration, et suppression de cette configuration au passage par la sortie. Il s'agit là d'une sorte de pagination, mais bien adaptée à la notion d'algorithme. Et non pas rigide et indépendante de l'algorithme, comme le découpage de la mémoire en tranches égales.

Première conséquence.

Le paramètre, c'est-à-dire la donnée ou le résultat doivent exister à l'extérieur de la configuration, au moins en tant que valeur. Ce qui signifie que dans la configuration il n'apparaît que sous forme d'une adresse - d'un nom -.

Deuxième conséquence.

L'algorithme doit pouvoir s'appliquer successivement sur tout un ensemble de configurations différentes pourvu qu'elles soient conformes. L'écriture de l'algorithme étant figée, il est donc nécessaire que l'opérande soit déterminé par rapport à l'origine de la configuration. (Choix réalisé en procédure formelle).

C'est d'ailleurs là une des principales raisons de l'invention de l'index. Index qui joue ici le rôle de "base", pour employer une terminologie non universelle mais courante.

Troisième conséquence.

Pour réaliser l'insertion d'un algorithme on a besoin de deux sortes d'information : d'une part l'accès à l'algorithme inséré et le retour à l'algorithme de départ, d'autre part un accès à la nouvelle configuration et un retour à l'ancienne.

En procédure formelle la clef d'accès au nouvel algorithme se trouve dans l'ancien, plus exactement dans l'instruction d'insertion. La valeur d'accès, elle, est dans la table des procédures ; et ça c'est déjà un choix de système. Une seule clef d'accès suffit car on s'arrangera pour que l'algorithme existe en exemplaire unique en mémoire centrale même s'il sert à plusieurs déroulements différents.

Propriété 2.

Le code de l'algorithme existe en exemplaire unique dans la mémoire. Même s'il fait l'objet de plusieurs déroulements distincts dans un même intervalle de temps.

En conséquence directe des propriétés 1 et 2, s'il y a unicité dans la constitution de la configuration, il devient tout-à-fait naturel que celle-ci comporte toutes les caractéristiques (dynamiques) de l'insertion. Entendons par là l'insertion côté algorithme et côté configuration.

Quatrième conséquence.

La configuration doit contenir obligatoirement une indication de retour à la configuration précédente. Il faut également qu'elle contienne son volume actuel ; qu'on peut utiliser directement comme en procédure formelle, ou bien indirectement pour gérer la place disponible (problème de système).

Des propriétés et de leurs conséquences je déduis alors un ensemble de règles à respecter pour construire une configuration. Ces règles sont intéressantes pour la construction d'un compilateur, dont le but sera, précisément, de produire des codes de programmes dont la structure aura intérêt à être systématique.

Je résume ces règles sous la forme suivante :

configuration = Vc, Vp, Rs, liste de paramètres, liste de variables locales, valeurs des variables locales ;

Vc volume de la configuration courante.
Vp accès à la configuration précédente.
Rs indication de retour dans l'algorithme précédent.

L'origine de la configuration est placée sur la première case de celle-ci. En procédure formelle cela signifie que le [I a pour valeur l'adresse de cette case.

Cinquième conséquence.

Il est facile de voir qu'une telle structure de configuration est tout à fait adaptée à la construction de procédures récursives. La démonstration en est simple : je me donne une procédure P dont la configuration est Cp .

```
P : début
    T1
    si (condition) vers  e1
    T2
  insérer  P
    T3
  e1 : T4
    fin
```

T1, T2, T3, T4 sont des suites d'instructions dont ne sort aucune commutation.

Je représente dans la colonne de gauche un déroulement possible de l'algorithme, et dans celle de droite l'étalement correspondant des configurations :

début	Cp
T1	
si (cond) vers e1	
T2	
insérer P	
début	Cp, Cp
T1	
si (cond) vers e1	
T2	
insérer P	
début	Cp, Cp, Cp
T1	
si (cond) vers e1	--------- Supposons que la condition est
e1 : T4	maintenant vérifiée.
fin	Cp, Cp
T3	
T4	
fin	Cp
T3	
T4	
fin	

J'ai supposé que la condition s'est trouvée vérifiée après la deuxième insertion. Cela a permis de traverser le premier fin donc de supprimer la configuration en cours. Et on remonte dans l'algorithme à la suite de l'insertion, d'où le déroulement de T3, T4 puis fin et suppression de la configuration en cours.

On constate ainsi que, disposant de toute l'information nécessaire dans chaque configuration pour remonter à la fois dans l'algorithme et dans la configuration précédente, peu importe que la procédure que l'on insère soit la même ou une autre.

Sixième conséquence.

Si l'on n'interrompt pas le déroulement d'un algorithme entre son point d'entrée et son point de sortie, il est très facile de l'utiliser pour plusieurs tâches différentes dans un même intervalle de temps.

Il suffit de lui fournir, au moment de l'insertion, la bonne adresse pour créer sa configuration. Je dirai qu'il s'agit là encore une fois d'une question de système. Le système intervient au niveau de l'insertion : j'appelle ça de la commutation globale.

Nous verrons plus loin, lors de la description du compilateur, comment le respect des règles édictées ci-dessus permet pratiquement d'obtenir ces propriétés.

On verra également plus loin comment utiliser cette sixième conséquence, en rajoutant une propriété supplémentaire : l'autojectivité.

Remarque :

Dans la situation où je me trouve ainsi placé, il devient inutile de gérer une "pile" séparée pour les "retours systématiques", puisque la configuration dans laquelle je place l'indication de retour joue ce rôle. La configuration est donc unique qui rassemble toute l'information de l'insertion et du retour.

LE SECOND LANGAGE OU LANGAGE SYMBOLIQUE LAC.

Ce Langage-A-Compiler est conçu pour isoler plus facilement les difficultés fondamentales de la compilation des difficultés corollaires qui peuvent être prises en charge en un second temps. Le but est de définir clairement les propriétés fondamentales qui vont permettre de résoudre les difficultés fondamentales.
On va trouver dans le LAC les propriétés essentielles que l'on rencontrera dans tout bon langage de programmation symbolique, sous des formes qui peuvent être éventuellement différentes.

Je vais décrire rapidement l'essentiel de ce langage, renvoyant pour les détails à la référence [1] .

On ne construit que des procédures. Un programme est une procédure.

Une procédure, donc, se compose de trois parties : une en-tête, une déclaration, une partie instruction.

Avant d'illustrer ceci avec un petit exemple qui est bien plus parlant, je précise encore que les déclarations se composent de trois listes : les paramètres formels, les variables locales, les index. Les index sont des variables locales par leur statut, on ne les distingue que par leur rôle : elles servent à indicer les variables-tableaux.

La partie instruction se compose d'instructions dont les types sont à prendrè dans les suivants : début, fin, l'affectation à trois opérandes maximum, la condition, l'insertion, l'aller-à.

A titre d'exemple, je construis deux procédures sans grande signification, l'une recherche la lettre "A" dans un tableau de lettres, et l'autre fournit plusieurs tableaux de lettres.

```
procédure RL
par T, N, R ;
Vl   P ;
Index I ;
     début
   P : = 0 ;
   I : = 0 ;
e1 : si I = N vers fin ;
     P : = P + 1 ;
     I : = I + 1 ;
si T  (I)  ≠ "A" vers e1 ;
     R : = P ;
  fin : fin
```

```
procédure FTL ;
  Vl L (3), M, R ;
index Z ;
   début
  M : = 3 ;
  L (1) : = "Q" ;
  L (2) : = "A" ;
  L (3) : = "S" ;
insérer RL (L, M, R) ;
  - - -
   fin
```

Le contenu de la procédure FTL montre bien ce qui manque ici : un langage qui permette d'approvisionner des tableaux tels que L . De fait, pour être complet en termes de programmation, il faudrait également un troisième langage qui permettrait, lui, de procéder aux déroulements, d'appliquer des programmes sur des configurations déjà constituées. La présentation de ces deux langages supplémentaires n'est pas indispensable pour soulever les problèmes fondamentaux que j'expose ici, aussi les laisserai-je pour l'instant de côté.

LE COMPILATEUR DU LAC.

Ce compilateur sera compact, je renvoie donc à la définition du compilateur compact. Je voudrais essayer de montrer la généralité du procédé en l'appliquant à un cas particulier. D'abord la vérification syntaxique. Prenons le cas limite du monoïde libre, vérifier qu'un mot lui appartient n'exige que de vérifier si les lettres qui le constituent appartiennent bien à l'alphabet. A un moment de l'analyse, chaque lettre peut être suivie par n'importe laquelle des lettres de l'alphabet.

Si j'observe alors un mot qui appartient à un langage, lequel est évidemment un sous-ensemble monoïde libre qui lui correspond, je m'aperçois que les lettres n'apparaissent pas dans un ordre tout à fait quelconque.
Ce que je viens de dire implique évidemment un sens déterminé pour la lecture. Par exemple le mot sera construit, ou lu, de gauche à droite.

Si j'imagine connues les règles de construction du mot, je peux donc prévoir la structure de celui-ci. Une étude un peu soutenue de ces règles, ou plutôt de leur impact sur la forme du mot va permettre de dégager une technique d'analyse. J'imagine connues les grammaires génératives. J'imagine également le lecteur capable d'en construire une qui représenterait le LAC.

Je vais faire quelques remarques :

A. Un programme se compose de deux parties, les déclarations et les instructions. C'est la lettre début qui marque rédhibitoirement la séparation entre les unes et les autres.

B. Dans les déclarations, j'observe quatre parties, l'en-tête, la liste des paramètres, la liste des variables locales, la liste des index. Ces objets arrivent dans l'ordre, un ou plusieurs des trois derniers peuvent être absents.

C. Les instructions arrivent en nombre quelconque, et dans l'ordre que l'on veut, pourvu qu'elles respectent les types imposés.

D. C'est toujours Procédure et fin qui commencent et qui achèvent respectivement un programme.

E. Il existe des objets élémentaires; des identificateurs, des entiers, des variables indicées, qui servent à composer des objets plus complexes : listes déclaratives, instructions.

Je vais donc me donner le moyen de repérer la construction de ces diverses étapes. Pour cela j'utilise un ensemble de variables d'état. Le nombre des états dépend du langage, mais le nombre des variables d'état, lui, sera choisi pour la commodité du traitement. Ainsi, par exemple, je prends la liste suivante :

PRO qui me sert à contrôler la présence dans l'en-tête, les déclarations, les instructions.

DECL qui sert à se localiser dans les déclarations, en paramètres, en variables locales, en index.

AFF dont les valeurs indiquent si l'on est dans une instruction d'affectation, et en quel point.

COND même but que le cas ci-dessus, mais pour une instruction conditionnelle.

INS également le même jeu pour une instruction d'insertion.

VERS même chose pour le "aller à".

PAR sert à contrôler la présence d'un indice.

IDENT pour les identifications.

NB pour les entiers.

Je donne à chacune de ces variables autant de valeurs que j'en ai besoin pour repérer une position, ainsi il me faut quatre valeurs pour PRO, par exemple :

1 lorsqu'on est dans l'en-tête,
2 dans les déclarations,
3 dans les instructions.

Je réserve une quatrième valeur, le zéro, pour l'extérieur du programme. Comment alors faire franchir leurs valeurs à ces variables.
Dans l'alphabet du langage, je distingue certaines lettres par le rôle qu'elles jouent. Je dirai qu'il y a les "ouvrants" et les "fermants". La lettre procédure est un ouvrant, la lettre ? est un fermant. J'utiliserai indistinctement le ";" ou le "?" .
On a également comme ouvrants les "(" , début, " : = ", "+", "-", etc...
Et comme fermants les ")", fin, et aussi les " : = ", "+", "-", etc...
C'est à la rencontre de procédure que PRO passe de 0 à 1. De 1 à 2 sur la rencontre de ";" , et de 2 à 3 sur début. C'est fin qui la ramène à 0.

Il en est de même pour DECL qui passe de 0 à 1 sur paramètre, de 1 à 2 sur variable locale, de 2 à 3 sur index et revient à 0 sur début. On constate déjà que deux possibilités se présentent ici qui dépendent du choix du langage. Soit les trois listes sont ordonnées, et la gradation sur les valeurs de DECL permet de vérifier l'apparition dans le bon ordre, soit les listes peuvent apparaître dans un ordre quelconque, voire même être chacune, multiple. Deux valeurs suffiraient alors pour la variable d'état.

Pour les identificateurs et les entiers, qui sont les objets les plus élémentaires de cette sorte de langage, je dispose d'une variable chacun, dont les valeurs évoluent en fonction de la première lettre rencontrée et de la lettre fermante correspondante. Par exemple, IDENT passe de 0 à 1 à la rencontre de la première lettre de l'alphabet latin. Et c'est l'un des fermants possibles "," ou ":" ou "(" ou les opérateurs arithmétiques ainsi que les opérateurs de relation, etc qui ramènent IDENT de 1 à 0 .

Une remarque s'impose : un identificateur doit commencer par une lettre de l'alphabet latin, mais peut contenir des chiffres. Exemple :

A1 : = BH9 - 227 ;

Le premier opérande ne peut être un entier, c'est une lettre qui doit apparaître en premier. Mais pour les deux autres il en est autrement, si une lettre apparaît IDENT passe à 1, si un chiffre apparaît, c'est NB qui passe à 1. Ces deux situations étant incompatibles l'une par rapport à l'autre, c'est cela précisément qui assure le contrôle syntaxique.

J'insiste sur ces quelques points de détail, pour essayer de mettre en évidence le rapport qui existe entre les structures de programmes et les variables d'état. Quand le langage est connu, on peut déterminer un ensemble d'états susceptibles de marquer sans ambiguité l'évolution syntaxique, mais le découpage en un ensemble de variables d'états est arbitraire et dépend de, ou conditionne l'approche algorithmique de l'analyse.

Nous avons affaire ici à un langage de Kleene, mais on pourrait envisager tout aussi bien un langage de Chomsky. Il suffirait de rajouter des variables-compteurs pour contrôler les occurrences du langage de Dyck restreint. J'étudierai plus loin et plus complètement les relations qui existent entre les langages et ces jeux de variables d'état.

Sémantique immédiate.

Il existe plusieurs degrés d'approche de la sémantique, ceci étant variable selon la richesse d'expression du langage étudié. Le premier degré concerne la signification attribuée aux objets élémentaires, en l'occurrence, les identificateurs censés représenter les variables. Un même objet apparaît dans des rubriques différentes, par exemple dans une déclaration puis ensuite dans des instructions. En Pascal, et nombre de langages à la mode, une variable peut même apparaître dans une définition.

Dans tous les cas, s'il en est ainsi c'est que l'on veut fixer le cadre d'utilisation de la variable dont les caractéristiques auront intérêt à être notées. C'est là précisément l'objet de la sémantique immédiate.

Au fur et à mesure que les déclarations sont analysées, le compilateur construit le tableau des variables. C'est ce tableau qui va servir à établir une correspondance entre, donc, les variables de notre second langage et la configuration de la procédure formelle, cf. plus haut la 4ème conséquence.

A chaque variable déclarée on attribue un Numéro, et on va s'arranger pour que ce numéro coïncide avec celui d'une case de la configuration-support. Ainsi les trois premières cases de cette configuration jouant le rôle que l'on sait quant-à l'insertion, la case N°3 sera la première disponible pour la première variable. On s'arrange également pour que ces premières cases disponibles soient attribuées aux paramètres formels du second langage, pour respecter la convention exprimée lors de la quatrième conséquence.

Remarque :

Toutes les variables sont prises en compte de cette façon, même ces variables particulières que sont les étiquettes ; qui, comme on le sait, sont déclarées quand elles sont suivies d'un ":" dans les langages "algol-like", et affectées quand elles apparaissent dans un "aller à" conditionnel ou non, ou dans toute liste d'aiguillage. Je renvoie à l'ouvrage cité en référence pour la description du traitement des étiquettes, qui est d'ailleurs dépendant pour le détail, du choix des structures du programme généré.

L'intérêt fondamental de la sémantique immédiate est de servir de base de travail à la construction de la sémantique définitive.

Remarque :

Cela peut servir à des époques diverses, soit directement pendant la construction du programme généré, au cours de la compilation, soit à des étapes différées au cours d'une correction éventuelle du programme déjà construit. On sent qu'on fait intervenir là quelque chose qui s'insère dans la gestion du déroulement des opérations, donc qui fait partie du Système.

J'entends par là que le fait de conserver la sémantique immédiate après la fin de la compilation d'un programme, dans le but de pouvoir reprendre ultérieurement tout ou partie de la compilation, est un fait d'organisation et qui, par là-même, s'insère dans une structure de système.

En quoi la sémantique immédiate est-elle indispensable ? On ne peut répondre à cette question qu'en tenant compte des faiblesses du programmeur. Même si le langage est non déclaratif, il sera toujours nécessaire de vérifier que les divers emplois d'une même variable dans les instructions demeurent cohérents au sens de leur signification ou de leur type.

Remarque :

La non-spécification des types de paramètres formels, possible en Algol , est de nature un peu différente. Le paramètre effectif imposera toujours à l'insertion un type précis au paramètre formel. A charge pour le compilateur de déroulement de vérifier alors l'adéquation au moment de l'emploi. Ainsi un programmeur audacieux peut, au cours de son programme, utiliser un même paramètre avec des types différents, il lui suffit de s'arranger pour qu'au moment de l'emploi l'insertion fournisse un paramètre effectif du bon type. Il y a là de toute évidence un risque.

Le traitement des étiquettes exige également la mise en réserve de sémantique immédiate. En effet, une étiquette déclarée, (suivie d'un ":" , dans les langages Algol-like), peut apparaître après que une ou plusieurs affectations en soient faites.

Pour éviter de se compliquer la vie il est plus simple de noter l'information à compléter et sa localisation, afin de réaliser toutes les affectations sur la fin du programme (rencontre de la lettre fin).

On crée donc un tableau dont les éléments vont contenir les groupes d'objets :

(identificateur), (type), (numéro), (dimension).
" , " , (localisation), (x).

La première ligne décrit une variable ordinaire, la deuxième une variable étiquette.

Ceci marche pour un choix particulier de sémantique définitive.

Sémantique définitive.

Je laisse de côté le cas d'un compilateur interprétatif, qui découlera simplement de ce que nous allons dire. Je définis ainsi le produit du compilateur :

C'est une phrase-code, image de la phrase symbolique, constituée d'une suite d'éléments images des éléments de la phrase symbolique qui ont une structure autonome, par exemple les instructions. La phrase-code est écrite dans un langage choisi pour les propriétés de la sémantique.

Remarque :

Les niveaux de découpage qui définissent une structure autonome, sont choisis avec un certain degré arbitraire. A la limite on pourra séparer l'identificateur, il n'est pas possible d'aller plus loin. Mais alors il faudrait traiter les ":=", "+", "-", etc... comme des entités séparées, ce qui n'est pas forcément le plus commode. On pourrait prendre quelque chose de beaucoup plus gros, le bloc Algol par exemple. Mais je me l'interdis pour une raison fondamentale : un bloc risque de contenir de la sémantique de commutation, et nous verrons plus loin qu'une certaine prise en compte de la commutation va nous amener, précisément, à concevoir des structures autojectives. Il reste donc à prendre en compte en bloc une structure raisonnablement moyenne, l'instruction. A condition, toutefois qu'elle soit définie comme étant finie et bornée.

Je vais envisager la structure d'un programme généré. Pour cela je fais l'hypothèse que paramètres et variables locales auront même représentation. Il s'agit d'un choix discutable, mais nous sommes ici justement pour discuter. Partant de là, si je tiens compte également des propriétés des paramètres formels et de la sémantique des instructions début et fin de la machine formelle, le programme généré se construit ainsi :

début

Mise en place de Vc, volume de la configuration actuelle, dans la configuration actuelle.

Mise en place des adresses des variables locales dans la configuration actuelle.

Suite des codes-images des instructions de la phrase symbolique.

fin

On se rappelle qu'au moment de l'insertion Vp et Rs sont également mis en place dans la configuration ; mais cela est systématique, on peut donc concevoir que ce soit réalisé sur l'instruction début. C'est une considération qui a ainsi des implications sur la structure de la machine support.

Edmond BIANCO

à suivre...

= : = : = : = : = : =