Signal : un langage pour le traitement du signal

Paul LE GUERNIC

IRISA/INRIA, Campus de Beaulieu, 35042 RENNES CEDEX, France

Paul LE GUERNIC obtient le diplôme d'ingénieur de l'INSA de Rennes, option informatique, en 1974. Sa thèse de troisième cycle porte sur les langages de programmation; au cours de cette étude, il a réalisé un interpréteur pour le langage Algol 68. Depuis 1978, il est à l'IRISA et s'intéresse aux schémas d'exécution de processus parallèles et au génie logiciel.

Albert BENVENISTE

IRISA/INRIA, Campus de Beaulieu, 35042 RENNES CEDEX, France

Albert BENVENISTE est ingénieur des Mines de Paris. Après des premiers travaux en probabilité (thèse d'État à Paris-VI en 1975), il s'intéresse au traitement du signal. Il est conseiller au CNET Lannion B, entre 1978 et 1982. Depuis, son principal domaine d'intérêt comme chercheur à l'INRIA (IRISA, Rennes) est le développement d'outils d'aide à l'implantation d'algorithmes de traitement du signal.

Thierry GAUTIER

IRISA/INRIA, Campus de Beaulieu, 35042 RENNES CEDEX, France

Thierry GAUTIER obtient le diplôme d'ingénieur de l'INSA de Rennes option informatique en 1980. Il entre ensuite à l'IRISA où il travaille dans le domaine de l'analyse syntaxique et de la compilation. Il prépare une thèse de docteur-ingénieur consacrée à la définition du langage SIGNAL.

RÉSUMÉ

SIGNAL est un langage en cours de définition à l'IRISA dans le cadre d'un projet d'aide à l'implantation d'algorithmes en traitement temps réel du signal. Les caractéristiques générales du langage sont inspirées des principes flots de données : un programme SIGNAL décrit un réseau de processus (opérateurs arithmétiques ou temporels) interconnectés à travers des ports orientés, sur lesquels circulent des signaux. Ces réseaux sont construits progressivement à l'aide d'opérations permettant d'établir des liaisons entre les ports. SIGNAL permet la description et l'exécution de tout algorithme temps réel synchrone mettant en œuvre éventuellement plusieurs horloges pour rythmer différents signaux. La présentation est illustrée par la description d'un algorithme de prédiction linéaire en treillis.

MOTS CLÉS

Traitement du signal, implantation d'algorithmes, langage de description, flot de données, temps réel.

SUMMARY

SIGNAL is a language (which definition is in progress at IRISA), intended to be the algorithms description language of a CAD system for real time signal processing applications. Main characteristics of SIGNAL are issued from data flow principles: a program describes an oriented graph the nodes of which are processes (i. e. arithmetical and temporal primitive functions) connected through their input and output ports. SIGNAL is able to describe and run any real time algorithm. The presentation is illustrated by the way of an adaptive gradient lattice algorithm, treated as an example.

KEY WORDS

Signal processing, algorithms implementation, description language, data flow, real time.

TABLE DES MATIÈRES

Préambule

Cet article présente une première version d'un langage de description d'algorithmes constituant l'un des éléments d'un projet de développement d'outils d'aide à la conception de machines pour le traitement du signal. Ce projet, mené conjointement à l'IRISA et à l'INRIA-Rocquencourt, a pour motivation d'une part les besoins croissants en algorithmique traitement du signal et d'autre part l'apparition de processeurs rapides permettant de mettre en œuvre des fonctions complexes :

– Le traitement du signal a des débits moyens, élevés, ou très élevés, est appelé à se développer fortement dans les domaines suivants des Télécommunications : traitement de la parole (codage, analyse, synthèse), annulation d'échos (électriques et acoustiques), modems, et, ultérieurement codage d'images bas débit/basse qualité; on peut également prévoir son développement dans les domaines de la régulation et de la commande de système dynamique (avionique, robotique).

– L'apparition de processeurs programmables intégrés orientés traitement du signal permet d'envisager pour l'avenir une implantation dans de bonnes conditions de fonctions « Traitement du Signal », et ce d'autant plus qu'une chaîne de CAO bien adaptée sera disponible. Dans cette optique, les délais actuels d'implantation sur microprocesseurs, ou de définition d'architectures prototypes, qui sont de plusieurs hommes × années, devraient pouvoir se ramener raisonnablement à plusieurs mois, voire moins, pour les applications simples.

Les processeurs considérés possèdent en général une structure interne et un jeu d'instructions permettant d'optimiser le calcul du produit de convolution. Cependant la taille de leurs mémoires de données et de programmes est faible, les possibilités de couplages forts entre boîtiers du même type restent très limitées; on rencontre là, les principaux obstacles à la programmation (au mieux en assembleur) d'applications nécessitant plusieurs de ces composants.

Un système de CAO permettrait d'implanter dans de bonnes conditions, des fonctions complexes de traitement du signal pour que les modules qui le composent permettent au pire de guider l'opérateur dans ses choix de répartition (au mieux réalisent automatiquement cette répartition) et restent indépendants des processeurs cibles. L'un des moyens de cette indépendance est l'existence d'un langage de haut niveau pour la description des algorithmes; ce langage permet l'étude des propriétés de l'algorithme hors contraintes de réalisation et l'étude de différentes contraintes de mise en œuvre générales (par exemple arithmétiques), travail partiellement réalisé à l'aide de Fortran par les concepteurs dans l'état actuel. Le passage de l'étude à la mise en œuvre doit assurer la consistance sémantique du produit obtenu, consistance actuellement hypothétique dans la mesure où elle résulte uniquement de l'attention portée par le programmeur à la transcription en assembleur des algorithmes. D'une part pour alléger les difficultés syntaxiques dans l'utilisation de Fortran (gestion des indices temporels, des indices de vecteur, étude d'arithmétique...) et d'autre part pour permettre de garantir une continuité entre étude et mise en œuvre d'un algorithme, nous avons décidé de définir un langage prenant en compte les spécifités des applications en traitement du signal.

1. Introduction

1.1. La classe d'algorithmes visée

Cette classe d'algorithmes, que nous allons maintenant décrire, n'est pas à considérer comme exclusive d'autres applications possibles de la CAO, mais elle servira de guide pour effectuer les choix à chaque fois que cela sera nécessaire.

Nous distinguerons *grosso modo* trois grandes classes d'algorithmes relevant du traitement du signal ou de l'automatique temps-réel.

a. Les algorithmes de nature purement séquentielle

Chacune des fonctions composant ces algorithmes consomme et produit du signal à la volée à une cadence constante : la cadence « échantillon ». Dans cette classe figurent les algorithmes séquentiels d'identification ou de modélisation, et les algorithmes de commande. La structure de ces algorithmes peut être assez irrégulière; de plus une tâche donnée peut être effectuée de manière également satisfaisante par des algorithmes non formellement équivalents.

Dans cette classe, des applications-type sont les systèmes de transmission (modems, annuleurs d'écho...) et de codage différentiel (parole, données, images...) en télécommunications, et les procédés de commande temps-réel de processus.

b. Les algorithmes de transformation traitant le signal par blocs

Les exemples classiques sont la transformée de Fourier, la transformée cosinus, la transformée de Walsh-Hadamard, mais aussi les méthodes « de prédiction linéaire » intervenant en traitement de la parole.

Ici encore, le signal est consommé à la volée. Par contre, il est traité par blocs à l'aide d'algorithmes parfaitement définis et figés, et dont la structure est en général assez régulière. Le résultat peut être la production d'un bloc de signal (algorithmes de transformation proprement dits) ou bien la production d'une information condensée (coefficients dans la méthode de prédiction linéaire).

Les applications les plus courantes sont dans le domaine de l'analyse spectrale (parole, sonar, radar...).

c. Les algorithmes à structure irrégulière

Cette dénomination volontairement floue est simplement là pour rappeler que les deux catégories précédentes ne recouvrent pas l'ensemble des besoins. Citons l'exemple de la plupart des procédés de codage par transformation [23, 7] dans le domaine du traitement de la parole : le signal y est consommé à la volée, puis traité par blocs; mais la structure du traitement effectué sur chaque bloc est difficile à caractériser.

Nous allons maintenant nous attacher à décrire les caractéristiques essentielles de la classe d'algorithmes qui servira de référence pour la définition du langage. De la description de ces caractéristiques, il ressortira que sont visées par ordre de préférence les classes *a*, *b* et *c* décrites précédemment.

Caractéristiques des algorithmes devant pouvoir être décrits par le langage

Nous les indiquons par ordre d'importance décroissante.

(*C* 1) *Ce sont des algorithmes séquentiels*

Du point de vue algorithmique, le point essentiel est la propriété suivante : *il existe une unique boucle de longueur non bornée, et toutes les autres boucles de longueur non bornée lui sont assujetties* (sous-multiples de cette boucle, ou extraction d'événements « aléatoires » dont l'occurrence est rythmée par cette boucle); nous interpréterons cette boucle comme *l'horloge-temps*. Toutes les autres boucles possèdent une longueur bornée.

Jouissent de cette propriété, évidemment les algorithmes de la classe *a* (en incluant les détections d'événements tels que : changements brusques de caractéristiques du signal, ou pannes en commandes), mais aussi les algorithmes de la classe *b* puisque les boucles de calcul y ont une longueur caractérisée par l'algorithme, et ne s'arrêtent pas sur tests (encore qu'il soit toujours possible de borner *a priori* la longueur d'une boucle d'itération en calcul numérique, mais ce n'est guère satisfaisant).

(*C* 2) *Les fonctions et processus qui coopèrent pour réaliser l'algorithme sont connus* a priori, *et peuvent être spécifiés de manière statique*

Ainsi les automaticiens et traiteurs de signal utilisent volontiers les blocs-diagrammes pour décrire le réseau statique dans lequel circulera le signal.

L'intérêt est ici d'arriver à ce que les nœuds de ce réseau statique soient assez simples pour que la description de ce réseau constitue une part importante de la description de l'algorithme.

Ici encore, la classe *a* d'algorithmes mentionnée plus haut jouit de cette caractéristique. Mais, pour les algorithmes de la classe *b*, on peut remarquer que, à l'intérieur des boucles de calcul numérique, le flot des calculs est en grande partie décrit par un réseau statique (« perfect shuffle » et papillons dans le cas de Fourier, et « cascade en treillis » dans le cas de la méthode de prédiction linéaire).

(*C* 3) *Ces algorithmes offrent une large place au calcul numérique, et ne mettent en cause que rarement de l'analyse numérique matricielle ou vectorielle de grande dimension*

Il s'agit là d'une propriété bien reconnue en traitement du signal : la plupart des processeurs spécialisés sont justement bâtis autour de l'opération $\sum xy$, opération essentielle dans le domaine du filtrage numérique classique. Il faut néanmoins signaler que les algorithmes adaptatifs nécessitent une variété d'opérations qui peut être bien plus grande (divisions, rotations dans le plan complexe...).

(*C* 4) *Ces algorithmes présentent fréquemment des rebouclages de signal*

C'est par essence le cas pour les algorithmes de commande, qui travaillent en boucle fermée, mais c'est aussi le cas de tous les algorithmes adaptatifs [11]. L'existence de ces rebouclages limite les possibilités de pipe-line lors de l'implantation : ainsi, on peut dire qu'un algorithme reste invariant sous l'effet d'un pipe-line dans le cas où l'introduction de ce pipe-line laisse globalement invariant chacun des flots de signaux qui circulent dans l'algorithme. Les rebouclages de signaux sont donc l'obstacle essentiel à l'introduction de pipe-line lors de l'implantation.

(*C* 5) *L'effet de l'implantation en précision numérique limitée doit pouvoir être étudié sur ces algorithmes*

Il s'agit en fait, non pas d'une caractéristique des algorithmes, mais d'une conséquence de l'implantation sur « petits processeurs », qui correspond à la situation standard dans la période actuelle; ce point perdra peu à peu de son importance avec la disponibilité de processeurs plus puissants. Le point à noter ici est que, dans bien des cas parmi lesquels les algorithmes adaptatifs, il est difficile, voire impossible, d'analyser *a priori* l'effet des erreurs d'arrondi et des mauvais cadrages, ce qui conduit alors à reporter une telle étude au niveau de la simulation.

1.2. Le type d'implantation visé

Pour les raisons technologiques mentionnées dans le préambule, mais aussi en raison des propriétés des algorithmes qui nous intéressent, notre choix se porte vers le type d'implantation suivant :

— un réseau *statique* de *« petits » processeurs* interconnectés;

— avec prise en compte de la contrainte prioritaire de satisfaction du débit de calcul (contrainte « *temps-réel* »).

Chacun de ces processeurs peut, soit être réalisé par un circuit physiquement autonome (cas des réseaux de

processeurs orientés traitement du signal, mentionnés dans le préambule), soit être une partie d'un circuit plus important (cas des circuits VLSI – circuits intégrés à grande échelle – de type systolique [15]).

Bien entendu, un des problèmes clés de l'implantation (le problème de « *l'allocation des ressources* ») sera de plonger le réseau statique associé à l'algorithme dans le réseau statique des processeurs, de façon que chaque nœud de ce second réseau soit un sous-réseau du premier.

Un des outils essentiels pour la satisfaction de la contrainte débit sera l'usage de pipe-lines (plutôt que l'utilisation du parallélisme vectoriel, ceci en raison de la structure des algorithmes), d'où l'importance de la propriété (C 4) et des commentaires qui l'accompagnent : le langage décrivant ces algorithmes devra permettre d'exprimer aussi clairement que possible les possibilités de pipe-line.

2. Présentation générale du langage « SIGNAL »

L'objet de cet article est de présenter l'état actuel (donc préliminaire) du langage de description des algorithmes. On en présentera d'abord les principes généraux, avec quelques indications sur les types envisagés, et les opérateurs correspondants. L'objet principal de l'article est l'étude détaillée des réseaux statiques associés à chacun des algorithmes : il s'agit d'être capable *d'effectuer formellement la construction, la description d'un tel réseau, et d'en explorer les transformations possibles.* Ce dernier point en particulier nécessite l'introduction d'un formalisme abstrait de manière à isoler les difficultés clés, sans s'embarrasser d'emblée de l'interprétation de ce formalisme.

Nous attirons l'attention du lecteur sur le fait que le langage SIGNAL est le langage de haut niveau qui devra, d'une part servir de point de départ à l'implantation, et d'autre part permettre le transport aisé d'un algorithme. Ce langage n'est pas à confondre avec le langage de commande de la CAO (qui sera lui seul vu de l'utilisateur lors de la mise au point d'un programme ou d'une implantation), dont la structure (langage question-réponse, éventuellement représentations graphiques, par exemple) et la syntaxe pourront être quelque peu différentes.

Le présent chapitre contient tout d'abord une description de caractéristiques générales du langage, résultant des propriétés de la classe d'applications visées; elle est suivie de la représentation graphique adoptée pour décrire les filtres; on trouve enfin les caractéristiques externes de ces filtres.

2.1. Caractéristiques générales

Les principes généraux que nous allons énoncer découlent naturellement des caractéristiques (C 1 à C 5) relevées pour la classe d'algorithmes que nous avons définie.

Le point essentiel est la caractéristique synchrone des algorithmes considérés. Les réseaux de Petri [21] sont un outil permettant de décrire des systèmes parallèles et asynchrones. Leur utilisation ne semble pas naturelle ici car une part importante du réseau serait consacrée à resynchroniser des signaux. Les langages flots de données, basés sur une synchronisation implicite des calculs, sont par contre beaucoup plus adaptés à notre objectif.

2.1.1. *SIGNAL s'inspire des principes flots de données*

La recherche de l'efficacité dans une exploitation du parallélisme interne à un programme a conduit au développement, en marge des multiprocesseurs et des processeurs vectoriels, d'architectures flots de données pour lesquelles le séquencement des instructions n'est plus dirigé par un compteur ordinal mais par le flot de données (*voir* en particulier [22] pour une étude générale). Parallèlement sont apparus les langages applicatifs, pour lesquels un programme n'est pas une suite d'instructions, mais un ensemble de définitions (LISP [17], LUCID [2], FP [3]). A cette classe de langages appartiennent les « langages flots de données » [1] conçus pour le calcul parallèle, adaptés à des exécutions sur architectures flots de données ([9, 14], VAL [18], CAJOLE [12]); les langages à assignation unique (SAMPLE [5], LAU [6]) reposent également sur les principes qui nous intéressent ici : un calcul peut être représenté par un graphe flot de données dans lequel les arcs représentent les chemins des données et les nœuds des opérations. Pour ces langages, différents modèles ont été proposés [8]. Notre objectif n'étant pas ici une étude de ces formalismes, nous nous contentons d'illustrer les principes flots de données à l'aide du graphisme introduit dans [9].

Un calcul est représenté par un graphe bi-parti orienté; dans ce graphe un nœud peut être :

– une action (dotée de une ou plusieurs entrées et d'une sortie);

– une connexion (dotée d'une entrée et de une ou plusieurs sorties).

Les arcs du graphe décrivent les chemins de données entre les différents nœuds.

Chaque nœud du graphe est activé selon un principe illustré par les schémas de la figure 2.1.

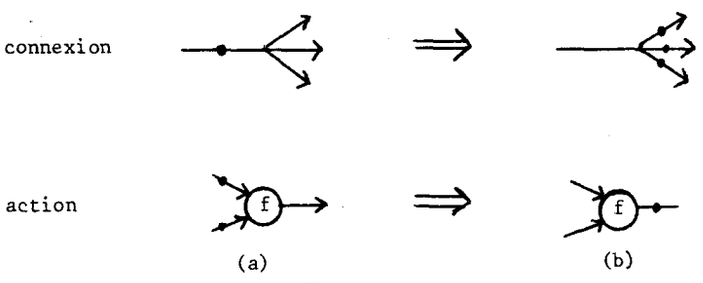


Fig. 2.1.

Un nœud peut être activé (*a*) si toutes ses entrées sont disponibles (présence d'un jeton sur chaque entrée), et si ses sorties précédentes ont été consommées (absence de jeton sur chaque sortie).

Le nœud est alors automatiquement activé : il consomme les valeurs disponibles à ses entrées et produit de nouvelles valeurs sur ses sorties (*b*).

Dans un langage flot de données, la notion de variable est proscrite, au profit de la manipulation de *valeurs* désignées par un *nom*. La contrainte d'association d'un

nom à une valeur unique peut être tempérée par l'usage de la notion de bloc dans le langage et par la possibilité de masquer des noms hors de blocs, comme on le verra plus loin.

De ces caractéristiques découlent les deux propriétés essentielles suivantes, très intéressantes dans notre cas :

— *il est possible de déduire aisément la dépendance entre données de l'écriture des opérations dans le programme;*

— *le séquencement dans le programme ne découle que de l'existence de dépendances entre les données correspondantes.*

Un programme flot de données permettra donc aisément de décrire la structure d'un algorithme : possibilités de parallélisme et de pipe-line. En raison de l'absence d'effets de bord, conséquence habituelle de l'utilisation de la notion de variable, il est aisé d'effectuer des transformations sur un programme flot de données tout en préservant le caractère *fonctionnel* de ce programme (à un ensemble de données correspond un ensemble unique de sorties).

Il est clair que la caractéristique (C 1) des algorithmes qui nous intéressent suggère une orientation flot de données pour le langage SIGNAL; dans ce langage, les fonctions opérant aux nœuds du réseau statique seront appelées des *filtres*. La description et la construction des réseaux de filtres sera détaillée plus loin.

2.1.2. *« SIGNAL » est adapté à la description de communications de type synchrone entre filtres*

Nous considérons que les données circulent sur le réseau statique de filtres sous la forme de « signaux ». Un *signal* est, par définition, la donnée d'un couple (flot, horloge), où le mot *flot* désigne une file, éventuellement infinie, de valeurs d'un certain type. Une différence avec le point de vue flot de données réside donc ici dans le caractère synchrone des communications qui sont décrites, par l'introduction de la notion d'*horloge* associée à un flot.

En conclusion, un programme SIGNAL consiste en :

la description d'un réseau statique orienté, sur les arcs duquel circulent des signaux, et dont les nœuds sont des filtres élémentaires (ou générateurs);

la description des générateurs.

Le langage SIGNAL fournit donc :

des moyens de construire ces réseaux;

un ensemble de générateurs, parmi lesquels figurent les générateurs nécessaires à la synchronisation de signaux rythmés par des horloges distinctes (les opérations visées sont le sur- et le sous-échantillonnage dépendant des valeurs observées, les tests, interruptions, décisions sur données...).

L'objet de cet article est de décrire la première partie de SIGNAL, c'est-à-dire la construction des réseaux statiques. Tous les aspects temporels (qui constituent en fait la vraie difficulté du problème) sont volontairement laissés de côté; en particulier, on n'insistera pas ici sur l'opérateur de retard, qui sera seulement introduit furtivement au cours d'un exemple.

2.2. Description des réseaux de filtres

Un filtre est représenté par un réseau dont les nœuds sont des « générateurs » associés aux opérations élémentaires du langage (+, ×, ...), et dont les arcs orientés représentent le flot des données entre ces opérations élémentaires; chaque opération élémentaire possède un certain nombre d'opérandes et de résultats représentés par des ports qui sont distingués par des noms [19]; on peut également voir les ports comme les nœuds de données dans les graphes flots de données. Il n'est pas dans notre propos de décrire l'ensemble des générateurs du langage ni de détailler les types des données circulant entre les opérations élémentaires; ces deux sujets ne seront abordés que lorsque ce sera nécessaire à la compréhension de l'exposé.

Représentation graphique : Sous certaines conditions précisées ultérieurement, certains ports de générateurs constituant un réseau sont visibles (et peuvent être l'objet de connexions) à l'extérieur d'un filtre; celui-ci est représenté par un rectangle auquel aboutissent (respectivement, dont sont issues) des flèches représentant les ports d'entrée (respectivement, de sortie) visibles à l'extérieur du filtre; un générateur est représenté par un demi-cercle muni également de flèches; aux flèches sont associées les noms de ports (*fig.* 2.2).

Fig. 2.2.

Lorsque l'illustration des opérations effectuées sur le filtre le rend utile, on s'autorise à décrire partiellement la structure interne d'un filtre en prolongeant intérieurement les arcs visibles.

A l'inverse des diagrammes de fluence, cette représentation n'est qu'un support graphique pour le langage. Le diagramme de fluence est en effet un outil bien approprié pour la description d'algorithmes simples (cellules du second ordre par exemple), mais il souffre de deux handicaps qui le rendent impropre à la description d'algorithmes complexes :

(1) Il s'agit d'une méthode de description graphique; or chacun sait que la tendance actuelle (en particulier pour les descriptions de masque de VLSI) est de remplacer les moyens de description graphique par des langages structurés au-delà d'un certain degré de complexité.

(2) Cette méthode n'est pas adaptée à la description d'algorithmes faisant intervenir des horloges différentes à synchroniser (c'est le cas quand des tests sont effectués sur les données).

2.3. Caractéristiques des réseaux

Un réseau décrit la circulation des données entre les opérations (arithmétiques ou temporelles) sur signaux. Cette circulation doit respecter les règles qui sont présentées en 2.3.1 et pour cela, les ports d'un filtre doivent vérifier les propriétés énoncées en 2.3.2. Nous terminons ce paragraphe par la description de la structure d'un programme.

2.3.1. *Interconnexions de filtres*

Un arc entre deux opérations élémentaires représente un *flot* formé d'une file ordonnée finie ou infinie de valeurs typées; un *signal* est la donnée d'un flot, d'une horloge indiquant la liste des instants (repérés par rapport à une horloge fondamentale) où les valeurs du flot sont définies, et de conditions initiales. La construction des arcs représentant les flots est obtenue à l'aide des opérateurs définis en 3 par interconnexion de ports de filtre selon des règles portant sur les noms de ces ports :

– un lien entre un port d'entrée et un port de sortie ne peut être établi que si ces deux ports ont un même nom;

– une fois établi, un lien entre ports ne peut être rompu;

– dans un filtre, deux ports de sortie distincts ne peuvent posséder le même nom.

Cette dernière règle interdit d'alimenter un port d'entrée par des signaux issus de deux ports distincts; si, pour la conception de systèmes, on peut donner une interprétation implicite raisonnable de cette opération (par exemple entrelacement non déterministe [13, 20]), il n'en va pas de même en traitement du signal où toute opération concernant des signaux distincts doit être explicite.

2.3.2. *Interface d'un filtre*

Afin de rendre possible la construction des filtres dont la structure interne respecte les règles ci-dessus, leurs ports d'entrée et de sortie doivent vérifier les propriétés suivantes :

– *les ports d'entrée non connectés sont obligatoirement visibles à l'extérieur;*

– *les ports d'entrée connectés sont obligatoirement invisibles de l'extérieur* (afin d'empêcher une seconde connexion);

– *les ports de sortie (connectés ou non connectés) peuvent rester visibles à l'extérieur;* il est en effet admis que le résultat d'un calcul n'est pas nécessairement utilisé (port non connecté devenu invisible) et, inversement le résultat d'un calcul peut être utilisé en diffusion (port connecté visible); l'utilisateur dispose en conséquence de moyens lui permettant de spécifier, parmi les ports de sortie d'un filtre, quels sont ceux qui restent utilisables pour des connexions externes. Cette possibilité est offerte au travers d'opérations de masquage de noms de ports, permettant de considérer un filtre comme une « boîte noire » où seules sont spécifiées les potentialités de communication avec l'extérieur.

2.3.3. *Structuration d'un programme*

Un filtre pourra donc être obtenu à partir des opérations élémentaires du langage par deux classes d'opérateurs :

– des opérateurs permettant de constituer un réseau comme assemblage de sous-réseaux;

– des opérateurs permettant de modifier ou de masquer des noms dans un réseau.

Un programme SIGNAL est une déclaration de filtre constituée d'un identificateur, de paramètres éventuels suivis de la description de ce filtre en terme de ses composants et d'une partie déclarative fournissant la description de ces composants.

3. Construction des réseaux

Les réseaux sont construits à l'aide d'opérateurs sur filtres et de structures de construction de motifs répétitifs.

Il existe deux classes d'opérateurs sur filtres :

– le masquage, le rebouclage et la mutation sont des opérateurs unaires postfixés agissant sur les noms des ports du filtre opérande;

– la composition, la composition collatérale et la séquence sont des opérateurs binaires qui permettent de définir un filtre comme assemblage de deux filtres opérandes; cet assemblage s'appuie sur l'identité des noms de leurs ports respectifs.

3.1. Opérateurs sur les noms de ports

3.1.1. *Masquage*

Le masquage permet d'inhiber des connexions potentielles en rendant invisibles de l'extérieur certains noms de *ports de sortie* d'un filtre. Cet opérateur permet de faire d'un filtre une « boîte noire », les noms masqués pouvant être réutilisés indépendamment.

Rappelons que les ports d'entrée connectés dans un filtre F sont considérés comme invisibles hors de F.

Si F est un filtre et $a_1, \ldots, a_n$ une liste de noms de ports de ce filtre, alors $F\langle !a_1 :, \ldots, a_n :\rangle$ représente le filtre F dans lequel tous les ports de sortie dont les noms apparaissent dans la liste $a_1, \ldots, a_n$ sont masqués.

Inversement, $F\{a_1, \ldots, a_n\}$ représente le filtre F dans lequel tous les ports de sortie n'apparaissant pas dans la liste sont masqués.

Exemple : Soit F le filtre suivant :

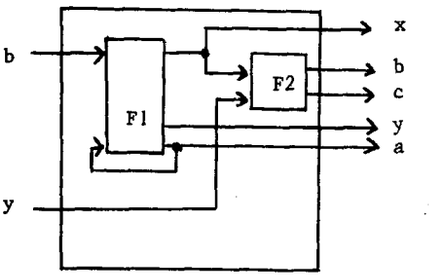


Fig. 3.1.

Remarque : L'arc rebouclant sur F1 n'est en principe pas visible hors de F1.

Alors :

$F\langle !a :, c :, y :\rangle = F\{b, x\} =$

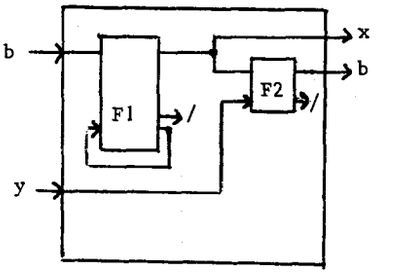


Fig. 3.2.

3.1.2. *Rebouclage*

Le rebouclage permet d'effectuer des connexions entre des entrées et une sortie de même nom d'un filtre. Il est représenté par un nom de port précédé d'une occurrence du caractère ∂.

Exemple : Avec le filtre F dont le schéma est donné sur la figure 3.3.

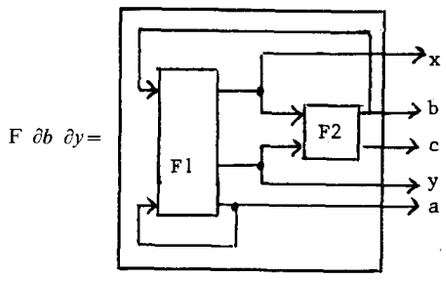


Fig. 3.3.

3.1.3. *Mutation*

La mutation permet de préparer des connexions futures en modifiant dans un filtre les noms de ports (d'entrée ou de sortie) qui ne sont pas masqués. Elle n'a de sens que si elle ne conduit pas à donner un même nom à des ports de sortie distincts.

La mutation est constituée d'un ensemble de redéfinitions de noms de ports entouré des caractères $\langle$ et $\rangle$. Ces redéfinitions peuvent avoir la forme suivante :

(1) $\quad ?x_1 : y_1, \ldots, x_n : y_n,$

(2) $\quad !x_1 : y_1, \ldots, x_n : y_n,$

(3) $\quad x_1 : y_1, \ldots, x_n : y_n.$

La règle (1) [respectivement, (2)] permet d'associer à tout port d'entrée (respectivement, de sortie) de nom x_i, un nouveau nom y_i.

La règle (3) permet d'associer à tout port d'entrée ou de sortie de nom x_i, un nouveau nom y_i.

Exemple : Considérons le filtre F défini ci-dessus (*fig.* 3.1).

$F\langle ?y : c!y : b, b : y\rangle =$

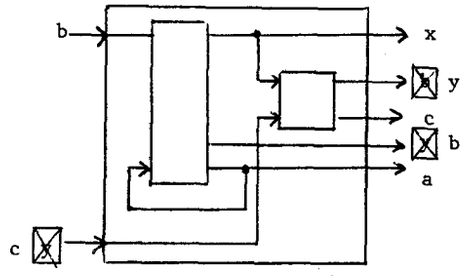


Fig. 3.4.

3.2. EXPRESSIONS BINAIRES DE FILTRES

Une expression binaire est une suite d'au moins deux occurrences de filtres (par exemple des noms de filtre) séparés par un mot-clé dénotant l'opérateur considéré (respectivement « , », « | » et « ; » pour la composition collatérale, la composition et la séquence). Cette définition interdit de mélanger, dans une expression de filtres non parenthésée, les différentes expressions binaires; ce choix évite d'avoir à poser arbitrairement une priorité entre opérateurs et assure une meilleure lisibilité des programmes.

Plutôt que F; G | H on devra écrire :

(F; G) | H s'il s'agit d'une composition d'une séquence et d'un filtre, ou

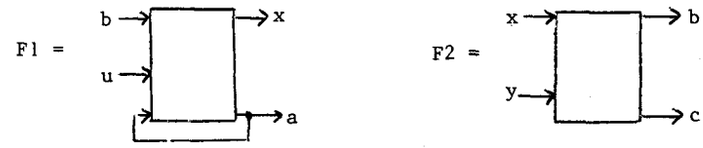
F; (G | H) s'il s'agit d'une séquence formée d'un filtre et d'une composition.

Chacun des opérateurs étant associatif, l'ordre d'évaluation d'une expression formée d'une suite d'opérations de même opérateur est indifférent. Le parenthésage est donc inutile dans une telle expression.

3.2.1. *Composition*

La composition est un opérateur binaire permettant de construire un filtre F par assemblage non ordonné de deux filtres opérandes F1 et F2 (elle est commutative); les sorties de F1 sont reliées aux entrées de F2 possédant le même nom et inversement. Les deux filtres ne doivent pas posséder une sortie de nom identique (diagnostic « erreur » dans le cas contraire).

Exemple :



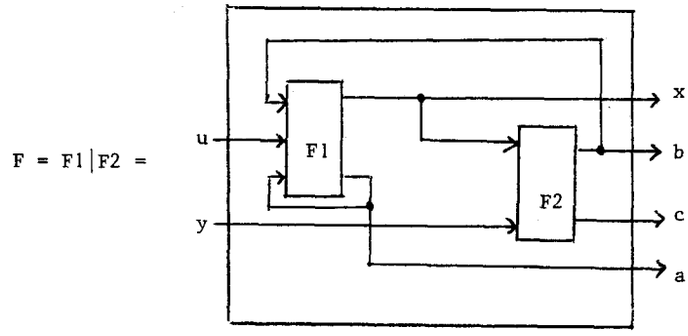


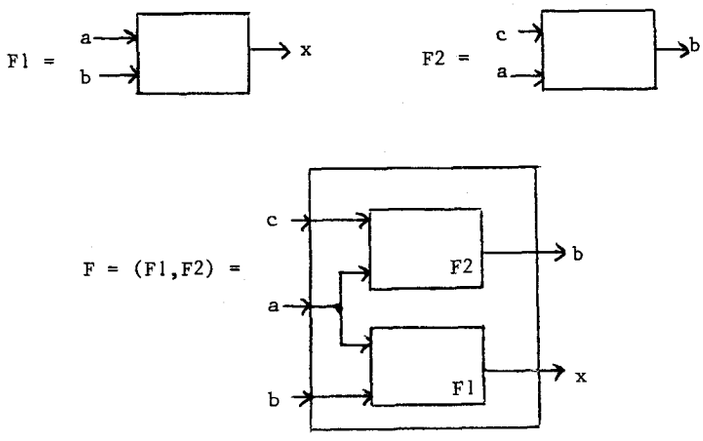
Fig. 3.5.

3.2.2. *Composition collatérale*

La composition collatérale est un opérateur qui, étant donnés deux filtres F1 et F2, confond leurs ports d'entrée respectifs de même nom, en vue d'une diffusion du même signal sur ces ports; le filtre F résultant ne doit pas posséder de ports de sortie distincts ayant un nom identique (diagnostic « erreur » dans le cas contraire).

Propriété : La composition collatérale est commutative.

Exemple :



3.2.3. *Séquence*

La séquence (composition séquentielle) est un opérateur binaire qui, étant donnés deux filtres F1 et F2, connecte les ports de sortie de F1 aux ports d'entrée de F2 de même nom et qui masque les sorties de F1 ayant un nom identique à une sortie de F2; cet opérateur n'est donc pas commutatif. Cet opérateur est utile à la fois pour la construction de motifs répétitifs et pour décrire des calculs intermédiaires par la mise en séquence d'expressions « d'affectation »; *à l'intérieur d'une séquence composée de filtres* F1, F2, ... *dans cet ordre un nom de ports peut être utilisé comme une variable.* Il est à noter cependant que cette assimilation ne peut être qu'approximative dans la mesure où, à la différence des variables :

— une valeur d'un signal sur un port de nom x ne peut être écrasée par la valeur suivante que si elle a été lue;

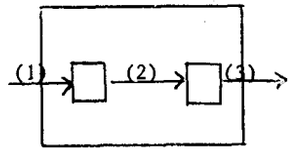
— un port d'entrée non connecté ne peut être masqué (pas de défaut d'initialisation).

Le tableau de la figure 3.9 précise l'effet de cet opérateur. Nous nous intéressons pour cela à un port de nom a pouvant apparaître dans chacun des opérandes et nous considérons alors les trois filtres prototypes de la figure 3.7 :

Fig. 3.7.

Par commodité, le nom a est omis dans le tableau et un port porte ce nom si et seulement si la flèche qui le représente traverse le cadre le plus externe du filtre.

Exemple :



Les ports connectés par (2) ne possèdent pas de nom.
Les ports connectés par (1) et (3) possèdent le nom a.

Fig. 3.8.

TABLEAU F = F1; F2

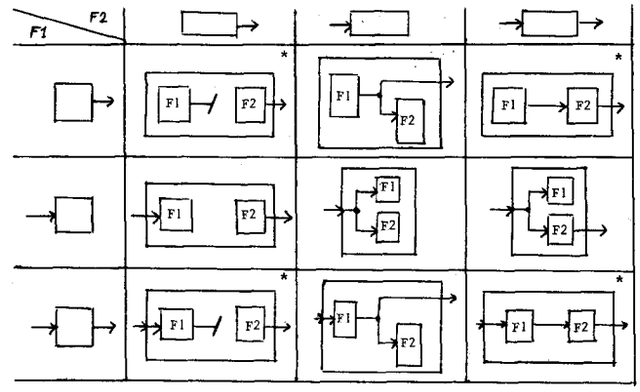


Fig. 3.9.

Exemples :

(1) La séquence ($a := x+y$; $y := x+a$; $a := a+y$) peut être interprétée comme une séquence d'affectations PASCAL où les noms sont regardés comme des variables; le filtre résultant représenté sur la figure 3.10 a pour entrées x et y pour leurs premières occurrences, et pour sorties a et y pour leurs dernières occurrences à gauche du signe d'affectation.

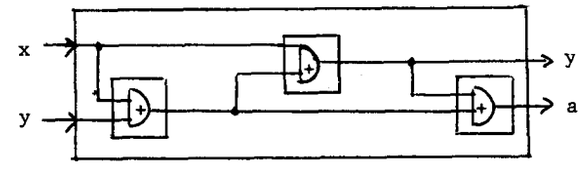


Fig. 3.10.

(2) Pour $i = 1, 2, 3, 4$ soit :

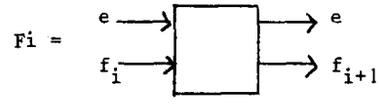


Fig. 3.11.

alors :

F = F1; F2; F3; F4 =

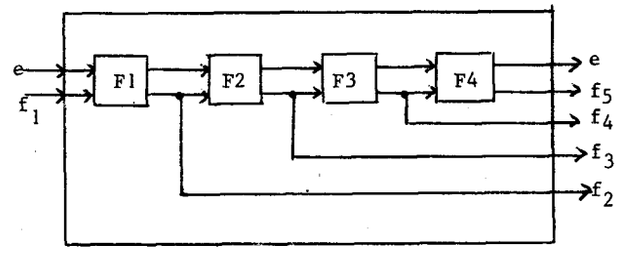


Fig. 3.12.

3.3. MOTIFS À CARACTÈRE RÉPÉTITIF

A chacun des opérateurs binaires définis ci-dessus correspond une structure de définition de motifs à caractère répétitif selon la forme :

mot-clé i in m . . n of F,

où *mot-clé* peut être respectivement :

doseq pour la séquence (;),
dopar pour la composition collatérale (,),
docom pour la composition (|).

Une telle structure définit un motif construit par la répétition ($n-m+1$ fois) de l'expression de filtre F au sein de laquelle l'identificateur i est utilisé dans des opérateurs de mutation. Cette construction est rendue possible par l'existence de noms indicés permettant de donner des noms calculés aux ports.

Sémantique

Nous présentons uniquement la structure séquentielle; les différences respectives avec la structure composée et la structure collatérale étant uniquement liées à l'opérateur binaire associé, sont immédiatement déductibles.

(*a*) L'expression *doseq i in m . . n of* F, où F est une expression de filtre et m, n deux valeurs entières vérifiant $0 \leqq m \leqq n$ est équivalente à la séquence composée de $n-m+1$ occurrences de F dans lesquelles i prend successivement ses valeurs dans l'intervalle $[m, n]$.

Exemple : Nous reprenons l'exemple présenté en 3.2.3 (*fig.* 3.11).

Soit F =

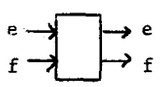


Fig. 3.13.

Soit :

$$G = \textit{doseq } i \textit{ in } 1 \,..\, 4 \textit{ of } F\langle ?\, f : f[i],\ !\, f : f[i+1]\rangle .$$

Avec la définition ci-dessus G est donc le filtre :

$$\begin{aligned} G = F\langle ?\, f : f[1],\ !\, f : f[2]\rangle ; \quad & i=1, \\ F\langle ?\, f : f[2],\ !\, f : f[3]\rangle ; \quad & i=2, \\ F\langle ?\, f : f[3],\ !\, f : f[4]\rangle ; \quad & i=3, \\ F\langle ?\, f : f[4],\ !\, f : f[5]\rangle \quad & i=4, \end{aligned}$$

représenté sur la figure 3.14 :

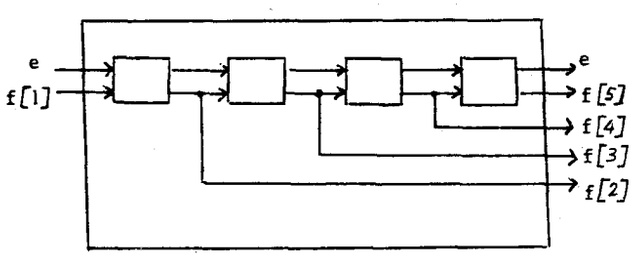


Fig. 3.14.

(*b*) Lorsque les résultats intermédiaires d'une telle séquence sont inutiles, on peut omettre la définition de l'identificateur i. On a alors dans l'exemple ci-dessus :

$$G' = \textit{doseq } 4 \textit{ of } F(=_\Delta G' = F;\ F;\ F;\ F) \quad (^1),$$

soit G' =

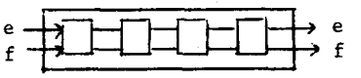


Fig. 3.15.

4. Un exemple simple

L'algorithme de prédiction linéaire en treillis (« adaptive gradient lattice algorithm » de J. Makhoul [16, 4]).

Il est donné (par exemple) par les formules :

$$(4.1) \quad \begin{cases} e_t(n+1) = e_t(n) - k_t(n+1)\, f_{t-1}(n), \\ f_t(n+1) = f_{t-1}(n) - k_t(n+1)\, e_t(n), \end{cases}$$

$$(4.2) \quad e_t(0) = f_t(0) =_\Delta x_t,$$

$$(4.3) \quad k_t(n+1) = \mathrm{Cor}(k_{t-1}(n+1),\ e_t(n),\ f_{t-1}(n)),$$

où la fonction Cor, non spécifiée ici est un estimateur séquentiel de la corrélation entre les signaux $e_t(n)$ et $f_{t-1}(n)$.

Désignons par $zf_t(n+1)$ et $zk_t(n)$ les signaux retardés $f_{t-1}(n)$ et respectivement k_{t-1} (le coefficient étant considéré comme un signal interne); on obtient alors les équations :

$$(4.1) \quad \begin{cases} (1) & e_t(n+1) = e_t(n) - k_t(n+1)\, zf_t(n+1), \\ (2) & f_t(n+1) = zf_t(n+1) - k_t(n+1)\, e_t(n), \\ (3) & zf_t(n+1) = f_{t-1}(n), \end{cases}$$

$$(4.2) \quad (4) \qquad e_t(0) = f_t(0) = x_t,$$

$$(4.3) \quad \begin{cases} (5) & k_t(n+1) = \mathrm{cor}(zk_t(n+1),\ e_t(n),\ zf_t(n+1)), \\ (6) & zk_t(n+1) = k_{t-1}(n+1). \end{cases}$$

Les équations (4.1) et (4.3) peuvent être directement exprimées en SIGNAL, en utilisant, pour chaque valeur de n, les expressions arithmétiques et temporelles suivantes :

(1) → (E1) : $\boxed{e := e - k * zf}$ est une expression arithmétique qui décrit un filtre possédant les entrées e, k, zf et la sortie e (distincte de l'entrée de même nom), que l'on peut représenter par le réseau (R1) de la figure 4.1 :

(R1)

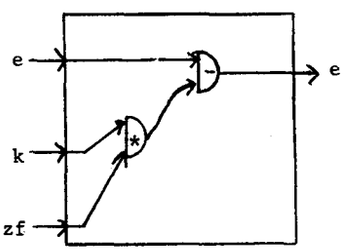


Fig. 4.1.

(1) $=_\Delta$: est égal par définition (n. d. l'auteur).

On a de même pour l'équation (2) l'expression (E2) en SIGNAL et le réseau (R2) de la figure 4.2 :

– (E2) $f := zf - k * e$:

(R2)

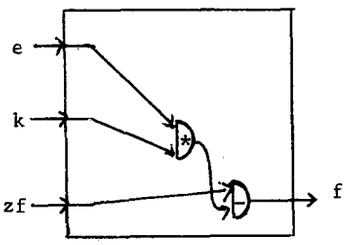


Fig. 4.2.

Les réseaux (R1) et (R2) peuvent alors être superposés (entrées e, k, zf diffusées) en utilisant l'opérateur de composition collatérale SIGNAL :

– (E12) $(e := e - k * zf,\ f := zf - k * e)$:

(R12)

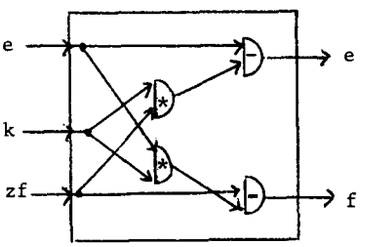


Fig. 4.3.

Le calcul du coefficient k (4.3) est exprimé par la composition de deux filtres :

– (E5) COR occurrence d'un modèle de filtre « cor » possédant les entrées zf, zk, e et la sortie k dont on ne décrit pas ici la structure.

(R5)

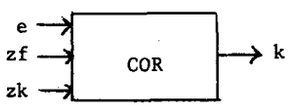


Fig. 4.4.

– (E6) zk *is* k *delay* 1 *init* 0 est une expression temporelle en SIGNAL permettant de définir le signal zk comme étant le signal k retardé de 1 et initialisé à 0. On représente le réseau associé sur la figure 4.5 :

(R6)

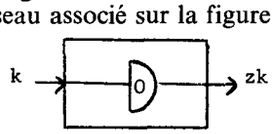


Fig. 4.5.

Le calcul de k est alors écrit en SIGNAL sous la forme suivante (dans laquelle on masque le port zk) :

– (E56) $(zk$ *is* k *delay* 1 *init* 0 | COR) $\langle ! zk : \rangle$

(R56)

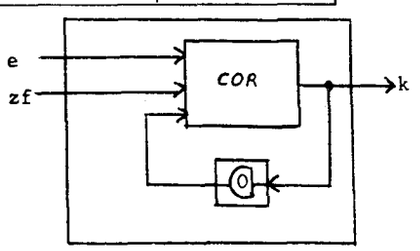


Fig. 4.6.

Finalement, le système d'équations (4.1, 4.3) est décrit par la mise en séquence en SIGNAL de l'expression temporelle définissant zf et des expressions (E56) et (E12) donnant le réseau associé suivant :

– (E12356) $(zf$ *is* f *delay* 1 *init* 0;
$(zk$ *is* k *delay* 1 *init* 0 | COR) $\langle ! zk : \rangle$;
$(e := e - k * zf,\ f := zf - k * e))\{e, f\}$

l'opération $\{e, f\}$ ne laissant visibles que les ports de nom e et f :

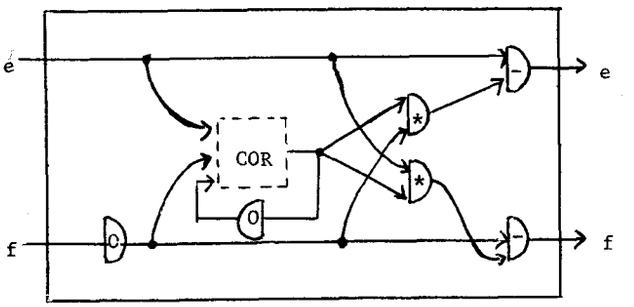


Fig. 4.7.

On décrit l'algorithme en SIGNAL par une déclaration de filtre (ORTHOG) utilisant la structure de construction répétitive séquentielle et comportant une partie déclarative décrivant l'élément construit ci-dessus :

ORTHOG(n) : $\{$*input* e *output* $f\}$ =
(*doseq* n *of* LATTICE) $\langle ? f : e ! e : \rangle$
where
filter LATTICE : $\{$*input* e, f *output* $e, f\}$ =
(zf *is* f *delay* 1 *init* 0;
(zk *is* k *delay* 1 *init* 0 | COR) $\langle ! zk : \rangle$;
$(e := e - k * zf,\ f := zf - k * e))\{e, f\}$
where filter COR : $\{$*input* e, zf, zk *output* $k\}$ = *signa*
end LATTICE.
end ORTHOG.

A cette déclaration correspond le réseau de la figure 4.8 :

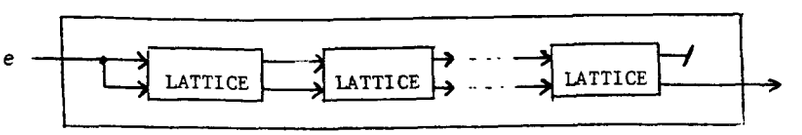


Fig. 4.8.

5. Conclusion

Nous avons présenté une partie du langage SIGNAL : la construction des réseaux statiques orientés (correspondant exactement à la notion de bloc-diagramme bien connue des automaticiens).

L'état actuel (juin 1984) du langage est le suivant :

– la version « temps-réel » de SIGNAL est complètement définie, y compris les aspects temporels (opérateurs permettant de synchroniser des horloges différentes); le

langage est développé sous l'environnement de programmation MENTOR [10] conçu à l'INRIA, ce qui permettra de le faire aisément évoluer;

– un premier compilateur de SIGNAL sera disponible en octobre 1984; il fournira du code FORTRAN; cette étape a pour but de rendre rapidement accessible le langage pour son expérimentation par des usagers. Pour permettre l'étude des effets d'arrondis, des moyens de description et de simulation d'arithmétiques seront fournis en 1985. Les études en vue des implantations sont poursuivies (passer formellement de la description de l'algorithme à la description de son implantation);

– parallélement, une nouvelle version du langage (version « non temps-réel », incluant de la récursivité au sens des informaticiens) sera définie pour inclure la description d'algorithmes de reconnaissance des formes (parole) qui ne sont pas temps-réel *stricto sensu.*

Ce travail a été réalisé dans le cadre de la convention DAII n° 82.35.119 00 790 75 00, passée conjointement avec l'INRIA-Rocquencourt (M. Sorine et al.).

BIBLIOGRAPHIE

[1] W. B. Ackerman, Data Flow Languages, *Proc. AFIPS Conference*, E. Merwin, éd., New York, 1979.

[2] E. A. Ashcroft et W. W. Wadge, Lucid, a Nonprocedural Language with iteration, *CACM*, vol. 20, n° 7, 1977.

[3] J. Backus, Can Programming Be Liberated from the von Neuman Style? A Functional Style and its Algebra of Programs, *CACM*, vol. 2, n° 8, August 1978.

[4] A. Benveniste, exposés 1, 2, 3 dans *Algorithmes rapides pour les systèmes dynamiques linéaires*, vol. 2 de *Outils et Modèles Mathématiques pour l'Automatique et le Traitement du Signal*, Éditions du CNRS, Paris, 1982.

[5] D. D. Chamberlin, The single-assignment approach to parallel processing, *Procs. AFIPS Fall Joint Computer Conference*, 1971.

[6] D. Comte, G. Durrieu, O. Gelly, A. Plas et J. S. Syre, Parallelism Control and Synchronization Expressions in a Single Assignment Language, *SIGPLAN Notices*, vol. 13, n° 1, January 1978.

[7] R. E. Crochiere, R. V. Cox et J. D. Johnston, Real Time Speech Coding, *IEEE-COM*-30, n° 4, 1982.

[8] A. L. Davis et R. M. Keller, Data Flow Program Graphs, *IEEE Computer*, vol. 15, n° 2, February 1982.

[9] J. B. Dennis, First Version of a Data Flow Procedure Language, *Proc. Colloque sur la Programmation*, B. Robinet, éd., Paris, avril 1974, *LNCS*, vol. 19, Springer-Verlag.

[10] V. Donzeau-Gouge, G. Kahn, B. Lang, B. Mélèse et E. Morcos, Outline of a tool for document manipulation, *Information Processing* 83, R. E. A. Mason, éd., North-Holland, 1983.

[11] D. D. Falconer, V. B. Lawrence et S. K. Tewksbury, Processor Hardware Considerations for Adaptive Digital Filter Algorithms, *Proc. ICC*-1980, Seattle.

[12] C. L. Hankin et H. W. Glaser, The Data Flow Programming Language CAJOLE. An Informal Introduction, *SIGPLAN Notices*, vol. 16, n° 16, July 1981.

[13] C. A. R. Hoare, Communicating Sequential Processes, *Comm. ACM*, vol. 21, n° 8, August 1978.

[14] G. Kahn, The Semantics of a Simple Language for Parallel Programming, *Proc. IFIP Congress* 74, J. L. Rosenfeld, éd., 1974.

[15] H. T. Kung, Why Systolic Architectures?, *Computer*, vol. 15, n° 1, January 1982.

[16] J. Makhoul, A class of all-zero lattice digital filters: properties and applications, *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-26, August 1978.

[17] J. McCarthy *et al.*, *LISP* 1.5 *Programmer's Manual*, MIT Press, 1966.

[18] J. R. McGraw, The VAL Language: Description and Analysis, *ACM Trans. on Programming Languages and Systems*, vol. 4, n° 1, January 1982.

[19] R. Milner, Synthesis of Communicating Behaviour, *Proc. Mathematical Foundations of Computer Science*, J. Winkowski, éd., Zakopane, 1978, *LNCS*, vol. 64, Springer-Verlag.

[20] R. Milner, A Calculus of Communicating Systems, *LNCS*, vol. 92, Springer-Verlag, 1980.

[21] J. L. Peterson, Petri nets, *ACM Computing Surveys*, vol. 9, n° 3, September 1977.

[22] P. C. Treleaven, D. R. Brownbridge et R. P. Hopkins, Data-Driven and Demand-Driven Computer Architecture, *Computing Surveys*, vol. 14, n° 1, March 1982.

[23] J. M. Tribolet et R. E. Crochiere, Frequency Domain Coding of Speech, *IEEE-ASSP*-27, n° 5, 1979.