Systèmes réactifs et programmation synchrone

Gérard Berry, Philippe Couronne, Georges Gonthier

INRIA

CENTRE
SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP105
78153 Le Chesnay Cedex
France
Tel. (1) 39 63 55 11

Rapports de Recherche

N° 524

SYSTÈMES RÉACTIFS ET PROGRAMMATION SYNCHRONE

Gérard BERRY
Philippe COURONNÉ
Georges GONTHIER

Mai 1986

SYSTEMES REACTIFS ET PROGRAMMATION SYNCHRONE

Gérard Berry, Philippe Couronné, Georges Gonthier

Ecole Nationale Supérieure des Mines de Paris
(ENSMP)
Centre de Mathématiques Appliquées

Place Sophie Lafitte
Sophia-Antipolis
06565 Valbonne – France

Institut National de Recherche
en Informatique et Automatique
(INRIA)

Route des Lucioles
Sophia-Antipolis
06565 Valbonne – France

Recherche soutenue par le GRECO C3, Pôle Langages

SYSTEMES REACTIFS ET PROGRAMMATION SYNCHRONE

Gérard Berry, Philippe Couronné, Georges Gonthier

Ecole Nationale Supérieure des Mines de Paris (ENSMP)
Centre de Mathématiques Appliquées
Place Sophie Lafitte
Sophia-Antipolis
06565 Valbonne – France

Institut National de Recherche en Informatique et Automatique (INRIA)
Route des Lucioles
Sophia-Antipolis
06565 Valbonne – France

Résumé

On appelle système réactif tout système qui interagit avec son environnement par réception et émission de signaux: système de contrôle en temps réel, automate de commande. Nous présentons les concepts fondamentaux d'un nouveau style de programmation des systèmes réactifs, la programmation synchrone: synchronisme des sorties sur les entrées, exécution sur une machine conceptuellement infiniment rapide, diffusion des signaux, déterminisme. Nous présentons le langage parallèle ESTEREL qui intègre ces concepts de synchronisme, et nous étudions le style de programmation induit. Nous montrons comment on peut compiler un programme ESTEREL en un automate fini déterministe équivalent et très efficace, et comment on peut mesurer la validité de l'hypothèse de synchronisme. Nous comparons sommairement ESTEREL à d'autres langages synchrones comme LUSTRE ou SIGNAL.

Abstract

A reactive system is a system that interacts with its environment by receiving and emitting signals (typically a real time control system, or any kind of control automaton). We present the basic concepts of a new way of programming reactive systems that we call synchronous programming: synchrony of inputs and outputs, execution of a program on a conceptually infinitely fast machine, signal broadcasting, determinism. We discuss the validity of the synchrony hypothesis. We present the ESTEREL synchronous parallel programming language that embodies these features. We study the programming style induced by ESTEREL. We show how to transform any ESTEREL parallel program into an equivalent and efficient deterministic finite automaton. We compare briefly ESTEREL to other synchronous languages such as LUSTRE and SIGNAL.

Recherche soutenue par le GRECO C3, Pôle Langages

SYSTEMES REACTIFS ET PROGRAMMATION SYNCHRONE

G.Berry, P. Couronné, G. Gonthier

Ecole des Mines / INRIA
Sophia-Antipolis
06565 VALBONNE - FRANCE

1. Introduction (*)

Selon D. Harel [14], Le terme *systèmes réactifs* a été introduit par A. Pnueli pour désigner des systèmes pilotés par des signaux provenant de leur environnement et ayant pour charge d'émettre eux-mêmes des signaux vers cet environnement: processus de contrôle en temps-réel, automatismes de contrôles divers (distributeurs de billets de banque, systèmes de contrôle d'interfaces clavier-souris [10]), jeux vidéo etc. La programmation des sytèmes réactifs est du domaine du parallèlisme, et peut certainement être réalisée avec des langages parallèles classiques comme CSP, ADA ou OCCAM. On assiste cependant à plusieurs tentatives de développement de langages ou de formalismes spécifiquement adaptés aux systèmes réactifs, comme ESTEREL [5], LUSTRE [2], SIGNAL [13], les Statecharts de D. Harel [14], SQUEAK [10], ou dans une moindre mesure SML [8]. Ces langages abandonnent tous l'hypothèse d'asynchronisme qui est à la base des langages classiques, pour la remplacer par une hypothèse de *synchronisme* plus ou moins forte. Nous considérerons seulement les formalismes purement synchrones comme ESTEREL, LUSTRE, SIGNAL ou les Statecharts, dans laquelle l'hypothèse est poussée à son maximum: on suppose que les sorties sont fournies de manière absolument synchrone aux entrées, donc que leur calcul "ne prend pas de temps". Cette hypothèse est présentée de façon relativement différente suivant les formalismes, et nous étudierons principalement son traitement dans le langage ESTEREL que nous avons développé, et qui possède une sémantique mathématique rigoureuse et une implémentation complète [5, 12, 6] (ESTEREL est certainement le projet le plus ancien dans le domaine).

L'hypothèse de synchronisme a trois vertus principales:

(i) Elle permet d'éviter des écueils tout à fait sérieux rencontrés dans les langages classiques: par exemple une phrase comme "délai 3 SECONDE; délai 2 SECONDE" ne peut être considérée comme équivalente à "délai 5 SECONDE" dans un univers asynchrone. De même lorsqu'on exécute un programme comme (**)

```
toutes les 1000 MILLISECONDE faire
    emettre SECONDE
fin
```

on est certain que la seconde émise ne sera synchrone avec *aucune* milliseconde, ce qui est pour le moins choquant. Les langages synchrones rendent ces propriétés évidentes, et permettent donc une manipulation beaucoup plus naturelle du temps (ou des temps, comme nous le verrons plus loin).

(ii) Elle permet de rendre les langages *déterministes*. Le non-déterminisme est inscrit par essence dans les formalismes asynchrones, car lui seul peut garantir leur fonctionnement. Or bien des systèmes réactifs sont purement déterministes; les langages synchrones vont permettre de les programmer réellement ainsi, et donc de simplifier leur réalisation et

(*) Cet article reprend et développe l'introduction de [5] sans ajouter d'élément technique véritablement nouveau.

(**) Ceci est le dernier programme en français. Les autres emploient la syntaxe ESTEREL v2, avec des mots-clefs en anglais (pour des raisons de diffusion internationale)

l'étude de leur comportement. (Par ailleurs le synchronisme n'exclut pas le non-déterminisme, mais nous n'en parlerons pas ici).

(iii) Elle fournit un nouveau *style de programmation* que nous considérons comme particulièrement élégant. Nous essaierons de justifier cette affirmation par la suite.

L'hypothèse de synchronisme se heurte pourtant à une objection sérieuse: elle n'est pas directement implémentable. Pour la valider, il faudrait en effet que le programme soit exécuté sur une machine infiniment rapide! Au contraire, l'hypothèse d'asynchronisme est conçue dès le départ en terme d'implémentation et ne pose pas de problème de ce point de vue. Cette objection ne nous paraît cependant pas majeure: le tout est de rapporter la notion de synchronisme à l'utilisateur final du système, ou, dans le formalisme de MEIJE [7], à son *observateur*: si sa perception est que le système fonctionne "comme si" les sorties étaient synchrones aux entrées, alors l'hypothèse est justifiée. Ainsi un informaticien ne se satisfait pas d'un terminal qui écrit l'écho du caractère longtemps après la frappe de la touche: il souhaite réellement voir un système réactif synchrone et oublier autant que faire se peut la mécanique interne de son terminal. Un utilisateur du téléphone n'a aucunement conscience du fait que sa voix a été découpée en paquets et transmise à travers des dizaines de protocoles subtils: il écoute son correspondant comme s'il était à côté de lui, et pour lui le meilleur téléphone est le plus synchrone. Un jeu vidéo qui ne se comporte pas de façon parfaitement synchrone ne trouvera aucun client.

L'hypothèse de synchronisme n'est par ailleurs pas neuve: les physiciens savent bien qu'à niveau assez fin les causes et les effets ne sont pas synchrones; cela ne les empêche pas d'appliquer les équations "instantanées" de Newton ou de Maxwell aux phénomènes macroscopiques. De même les chimistes utilisent constamment la notion de pH qui n'a strictement aucun sens au niveau moléculaire. La seule précaution qu'ils prennent est de vérifier qu'ils sont bien dans le domaine de validité de leurs hypothèses macroscopiques.

Du point de vue des systèmes réactifs, notre hypothèse peut être considérée comme valide dès que la vitesse des calculs à effectuer est raisonnablement plus grande que celle des réactions désirées, et dès que l'on peut éviter les phénomènes d'interférence entre les entrées, les calculs et les sorties. Citons trois cas typiques:

(i) Les systèmes non temps-réel, pour lesquels le problème ne se pose pas du tout. Les langages synchrones peuvent cependant s'y révéler très commodes de par le style de programmation auxquels ils conduisent.

(ii) Les systèmes temps réels lents, qui sont bien plus nombreux qu'il n'y paraît. Il n'y a pas de véritable problème de vitesse de calcul pour assurer la commande d'un train, d'un laminoir ou d'une interface clavier-souris.

(iii) Les circuits intégrés. Le niveau microscopique de la circulation des électrons est extrêmement complexe. Mais l'horloge deux phases est un moyen élégant de nier cette complexité et de se ramener à un cas quasi-synchrone.

L'implémentation d'ESTEREL a montré que l'on pouvait même aller beaucoup plus loin, et traiter des systèmes temps réel rapides: En utilisant la sémantique mathématique du langage [5, 12], On peut effectuer des calculs très profonds sur les programmes, permettant de transformer un programme ESTEREL - donc parallèle - en un automate fini purement séquentiel équivalent. Si l'on veut, tout le parallélisme et la communication inter-tâches sont résolus à la compilation. Il n'y a donc plus besoin de gérer de processus à l'exécution: il suffit de réaliser des transitions d'automates, ce qui est particulièrement simple et s'effectue en temps constant quelle que soit la taille du programme (si l'on ne compte pas le temps des calculs que le programmeur a explicitement écrits dans son programme, et qu'il faudra effectuer de toutes façons quel que soit le formalisme). Pour un exemple complet, voir [3] où nous décrivons la programmation d'une montre digitale avec réveil, chronomètre et affichage sophistiqué. L'automate engendré est petit et tient sans problème le millième de seconde sur n'importe quel processeur. Remarquons de plus que la vitesse de transition maximale est *complètement mesurable* sur un processeur donné puisque le code produit est strictement séquentiel. On peut donc vérifier la validité de l'hypothèse

de synchronisme après compilation d'un programme, en fonction de son contexte d'exécution.

Utiliser des automates pour programmer des systèmes réactifs n'est bien sûr pas nouveau. Mais quiconque a *vraiment* essayé d'en programmer sait combien il est difficile de les écrire, de les mettre au point et surtout de les modifier. Un langage synchrone comme ESTEREL permet de combiner la souplesse d'écriture d'un langage parallèle de haut niveau avec l'efficacité quasi-optimale des automates à l'exécution. Pour tempérer cette affirmation, disons cependant que le style de programmation synchrone n'est pas encore très bien compris, et que nous ne savons pas jusqu'où ces techniques peuvent aller. Mais on peut déjà considérer ESTEREL comme bien adapté à l'écriture de noyaux réactifs de systèmes plus complexes programmés en langages classiques, de même que YACC et LEX sont bien adaptés à l'écriture de noyaux d'analyse syntaxique dans les compilateurs.

Dans la suite nous étudierons comment l'hypothèse de synchronisme est traités dans ESTEREL et à quel style de programmation elle conduit, sans donner de détail sur le langage lui-même (voir [5, 6]). Puis nous indiquerons brièvement comment la même hypothèse est vue dans d'autres langages comme LUSTRE, SIGNAL ou les Statecharts.

2. Le temps multiforme discret

2.1. Systèmes réactifs et notion de temps.

Un système réactif reçoit plusieurs types de signaux à priori non synchronisés entre eux. Par exemple un train peut recevoir un signal toutes les millisecondes, un signal tous les tours de roue, et un signal à chaque crocodile rencontré sur la voie. Un système de contrôle de processus reçoit des mesures de différents capteurs et des commandes du pupitre de contrôle. La plupart des langages parallèles ou temps-réel permettent de traiter ces signaux comme des messages, en n'établissant pas de distinction entre les signaux matériels externes et les signaux logiciels internes; les systèmes d'exploitation temps-réel ont précisément pour rôle de transformer les signaux matériels en signaux logiciels. Nous garderons le même point de vue dans les langages synchrones.

En plus des signaux d'entrée, les langages asynchrones donnent généralement accès à une unité temporelle particulière: l'horloge universelle qui mesure le "temps absolu" lors de l'exécution du programme. Elle est accessible via un symbole spécial (disons la "seconde"), et se manipule par des instructions spécifiques comme les délais ou les chiens de garde. Le rôle particulier du temps universel se justifie dans le cas asynchrone par la nécessité de mettre en relation le temps de calcul du programme et l'arrivée des signaux externes, pour savoir par exemple si ces signaux sont bien pris en compte.

Dans un formalisme synchrone, la situation est toute différente puisque la notion de "temps de calcul" n'existe plus pour une machine infiniment rapide. La seconde n'a plus de raison de jouer un rôle particulier, et peut être traitée comme n'importe quel signal externe. La réciproque est vraie aussi, et on pourra désormais écrire des délais ou des chiens de garde sur n'importe quel signal; nous en ferons un des points essentiels de notre style de programmation.

Puisque la machine calcule infiniment vite, son seul comportement est d'émettre éventuellement certains signaux de sortie sur réception de signaux d'entrées. Les sorties étant strictement synchrones aux entrées, la machine ne fait rien entre les instants où elle reçoit des signaux d'entrée. Une donnée du système est donc simplement une suite discrète d'événements d'entrée, et le système produit en sortie une suite d'événements (éventuellement vides) synchrones aux entrées.

Il nous faut être plus précis sur la notion d'événement. Nous considérerons qu'un événement est formé par l'occurrence d'un signal ou de plusieurs signaux simultanés, comme dans les modèles SCCS [15] ou MEIJE [7]. Il n'y a en effet aucune raison de supposer que les événements d'entrée sont sérialisés comme dans les langages asynchrone, et nous pouvons parfaitement admettre qu'un tour de roue et une seconde se produisent en même temps (en laissant à l'utilisateur le soin de donner un sens à cette phrase dans son application). Par ailleurs

les sorties étant synchrones aux entrées, un observateur extérieur voit bien les signaux d'entrée et de sortie comme simultanés; nous devons prendre en compte cette notion de simultanéité pour pouvoir composer les programmes. Le traitement explicite de la simultanéité sera une autre composante essentielle de notre style de programmation.

2.2. Un formalisme simple

Nous dénoterons les signaux par des identificateurs, utilisant S, S1 etc.. pour des signaux non spécifiés. Un signal pourra être pur, c'est à dire sans valeur associée, ou pourra convoyer une valeur prise dans un type adéquat (défini à la déclaration du signal). Un signal pourra être émis simultanément par plusieurs émetteurs. Ceci ne pose pas de problème pour un signal pur. Pour un signal avec valeur, nous empruntons une idée de Milner [15]. A chaque signal S nous associons une opération associative commutative $*$, et si les émetteurs émettent les valeurs $v_1, v_2, \ldots, v_n$ alors la valeur transportée par le signal est $v_1 * v_2 * \cdots * v_n$. Donnons trois exemples:

(i) Ethernet: tous les organes connectés peuvent effectivement émettre simultanément dans le câble. La combinaison de deux émissions distinctes provoque une erreur. Il suffit de poser $v_1 * v_2 = erreur$ pour tous v_1, v_2.

(ii) Un applaudimètre: chaque spectateur applaudit avec une certaine intensité. L'applaudimètre mesure la somme des intensités (bien que continu, cet exemple est de même nature).

(iii) Un signal pur: il suffit de considérer que le signal transmet une valeur v avec $v * v = v$. Ceci permet de se ramener au cas général des signaux avec valeurs.

D'autres exemples plus pratiques sont donnés en [3], où l'on traite la sonnerie d'une montre à l'aide d'un signal multiple.

L'occurrence d'un signal S portant la valeur v est notée S^v. Un *événement* est un mot de la forme

$$S1^{v_1}\, S2^{v_2} \cdots Sn^{v_n}$$

Il traduit l'arrivée simultanée des signaux S1, S2,..., Sn avec les valeurs correspondantes. Le mot peut aussi être vide, et on considère alors un *événement vide* où aucun signal n'est arrivé.

Une *histoire* du système est une suite d'événements; un programme agit comme un transformateur d'histoires: il transforme son histoire d'entrée en histoire de sortie.

En ESTEREL, il n'y a pas de "temps universel": le comportement d'un programme est insensible à l'insertion d'événements vides dans une histoire d'entrée. L'occurrence d'un événement vide ne provoque d'action du programme que si elle apparaît comme première occurrence de l'histoire d'entrée (elle permet alors d'effectuer les initialisations, et constitue plutôt un événement fictif "lancement du programme"). Dans tout les autres cas une occurrence vide n'a aucun effet. Ceci est une différence importante avec la version actuelle de LUSTRE [2].

3. Les primitives d'ESTEREL

ESTEREL comporte des instructions impératives classiques, un opérateur de parallélisme, un opérateur d'échappement (dont le mariage avec le parallélisme demande un peu de soin), et deux instructions de manipulation d'événements, que nous appellerons *instructions temporelles*.

3.1. Les primitives impératives

Elles comportent l'instruction vide "nothing", l'affectation ":=", l'appel de procédure externe "call", la séquence ";", la conditionnelle "if-then-else-fi", la boucle infinie "loop-end", le parallèle "||", un mécanisme "tag-exit" de sortie de bloc analogue au tag-exit de LELISP (ou au catch-throw de maclisp, ou encore au mécanisme de failure de ML), et enfin la déclaration de variables locales et de signaux locaux. La syntaxe de ces primitives est classique et sera utilisée ici sans autre définition. Le langage donne aussi une (faible) structure modulaire non détaillée ici.

Comme dans tout langage impératif, il y a une notion explicite de *contrôle* et de passage de ce contrôle entre les instructions. Notre machine support étant infiniment rapide, toutes les instructions gèrent le contrôle de façon instantanée. Ainsi l'instruction "nothing" ne fait rien *et ne prend aucun temps*. Une affectation et un appel de procédure sont également instantanés (il faut bien sûr que la procédure soit relativement rapide, rappelons que la validité de l'hypothèse de synchronisme est à vérifier dans chaque cas particulier). Bien que simultanées, les instructions sont effectuées *dans le bon ordre*, et la mémoire est traitée séquentiellement. Donc une séquence

```
X:=0;
X:=X+1
```

rend bien toujours X=1. Dans un parallèle, les branches sont démarrées en même temps, et le parallèle se termine lorsque les deux branches sont terminées.

Il faut imposer une contrainte de finitude, vérifiée à la compilation, pour interdire des instructions comme

```
X:=0;
loop
     X:=X+1
end
```

qui n'ont évidemment aucun sens en univers instantané.

3.2. Les primitives temporelles et la gestion des signaux

Rappelons qu'un programme ESTEREL ne s'exécute que sur réception d'un événement d'entrée. Les signaux d'entrée et les signaux internes sont *diffusés instantanément et fugitivement* à toutes les composantes du programme. Pour émettre un signal S avec comme valeur celle d'une expression <exp>, on écrit simplement

```
emit  S(<exp>)
```

Les valeurs reçues de l'extérieur et les valeurs émises par toutes les instructions "emit" activées sont combinées par l'opération $*$ associée au signal, le résultat étant la valeur convoyée par le signal et reçue par les récepteurs. On peut voir la communication ESTEREL comme une communication par radio: chaque signal détermine une bande de fréquence, les différentes émissions sur une même fréquence étant combinées par les opérations $*$.

Il existe une seule instruction de réception de signal, qui est plus riche que l'attente habituelle. Elle a une des formes

```
do <instruction> upto <occurrence>
do <instruction> upto <occurrence> (<variable>)
```

où le corps <instruction> est une instruction quelconque, où la variable optionnelle <variable> sert à recevoir le valeur convoyée par S, et où <occurrence> a la forme suivante, les crochets indiquant les éléments optionnels

```
[next] [<expression>] <signal>
```

par exemple

```
SECONDE
3 SECONDE
next 3 SECONDE
next MESURE
```

L'occurrence sert à dénoter un événement futur (au sens large), en comptant à partir de l'événement qui a permis au "upto" de prendre le contrôle, que nous appellerons l'événement présent. Ainsi "SECONDE" dénote le premier événement contenant une seconde, en incluant l'événement présent, "next SECONDE" dénote le premier événement contenant une seconde en excluant l'événement présent, et "3 SECONDE" (resp "next 3 SECONDE") dénote le troisième événement contenant une seconde, événement présent inclus (resp. exclu).

La sémantique est la suivante: le corps <instruction> est exécuté jusqu'à l'événement déterminé par <occurrence>, *non inclus*. Sur cet événement le corps est instantanément tué. Si elle est présente, la variable <variable> reçoit la valeur convoyée par le signal S. L'instruction "upto" définit donc la portée temporelle de son corps, l'occurrence représentant pour ce corps l'événement "fin du monde". La terminaison du corps du upto ne provoque pas la terminaison de cet upto, qui doit attendre l'occurrence mentionnée pour se terminer.

Remarquons que dans un cas comme

```
do <instruction> upto SECONDE
```

le corps <instruction> n'est pas exécuté si le signal seconde est présent dans l'événement courant. Dans un cas comme

```
do
    do
        <instruction>
    upto 2 S1
upto next S2
```

le corps <instruction> est tué à la première des occurrences 2 S1 ou next S2, mais l'instruction ne se termine globalement que sur next S2.

3.3. Les instructions temporelles dérivées

Combinée avec les instructions classiques, l'instruction upto permet de définir beaucoup de manipulations temporelles courantes. On définit donc des instructions dérivées, qui se traduisent en instruction de base par macro-génération classique. Ainsi on écrit

```
await MESURE(X)
```

au lieu de

```
do nothing upto MESURE(X)
```

on écrit

```
every next 3 SECOND do
    <instruction>
end
```

pour abréger

```
await next 3 SECOND;
loop
    do
        <instruction>
    upto next 3 SECOND
end
```

Trois instructions dérivées sont particulièrement utiles. La première est le test de présence d'un signal, qui s'écrit

```
present S(X) then
    <inst1>
else
    <inst2>
end
```

Le lecteur pourra vérifier que l'expansion suivante est correcte, en utilisant de façon cruciale le fait que le corps d'un upto n'est pas exécuté si l'occurrence définit l'instant présent

```
tag PRESENT in
    tag ABSENT in
        do exit ABSENT upto S(X);
        <inst1>;
        exit PRESENT
    end;
    <inst2>
end
```

La seconde est le chien de garde classique. Il s'écrit

```
do
    <inst1>
watching <occurrence>
timeout <inst2> end
```

La sémantique est bien classique. Si <inst1> se termine avant l'occurrence mentionnée, le chien de garde se termine avec <inst1>. Sinon dès l'arrivée de l'occurrence attendue, l'instruction <inst1> est tuée et l'instruction <inst2> est lancée. L'expansion est simplement

```
tag T in
    do
        <inst1>;
        exit T
    upto <occurrence>;
    <inst2>
end
```

La troisième est la sélection d'événements, ou attente multiple. Elle s'écrit

```
await
    case <occ1> do <inst1>
    case <occ2> do <inst2>
    ...
    case <occn> do <instn>
end
```

La première des occurrences <occ1>, <occ2>,..., <occn> satisfaite détermine l'action à effectuer, qui est lancée instantanément. Si plusieurs occurrences sont satisfaites simultanément, seule la première instruction dans l'ordre de la liste est lancée. Nous ne donnerons pas l'expansion ici. Nous utiliserons d'autres instructions dérivées évidentes sans les définir.

4. Le style de programmation induit

Nous mettrons en valeur brièvement trois points importants: la manipulation du temps multiforme, l'utilisation de la diffusion, l'utilisation de la simultanéité. Pour plus de détail, le lecteur est invité à consulter les exemples présentés en [4].

4.1. Le temps multiforme

Pour illuster l'importance du fait que les constructions temporelles portent indifféremment sur tous les signaux, donnons d'abord un exemple simple. Une phrase comme "le bouton A doit être appuyé dans un temps limite de 10 secondes, sinon une alarme doit sonner", se traduit en

```
do
    await next A
watching next 10 SECOND
timeout emit ALARM end
```

Une phrase comme "on doit attendre 10 secondes; si pendant ce temps on appuie sur le bouton A, l'alarme doit sonner" se réécrit dualement "10 secondes doit être atteint dans un temps limite de A, sinon une alarme doit sonner", donc en

```
do
    await next 10 SECOND
watching A
timeout emit ALARM end
```

Le chien de garde est donc naturel et utile pour *tous* les signaux. Cet exemple illustrait aussi l'imbrication des unités temporelles. Allons plus loin dans ce sens avec un coureur à pied, qui connaît naturellement quatre unités de temps: la seconde, la foulée, le mètre parcouru, et le tour de piste. Voici un programme d'entraînement: "faire deux tours de la façon suivante: courir doucement 100 mètres, puis, pendant 20 secondes, sauter haut en expirant sur chaque foulée; terminer le tour en courant le plus vite possible"

```
do % pendant 2 tours
    loop
        do % pendant 1 tour
            do
                COURIR_DOUCEMENT
            upto next 100 METRE;
            do
                every FOULEE do
                        SAUTER_HAUT
                    ||
                        EXPIRER
                end
            upto next 20 SECONDE;
            COURIR_LE_PLUS_VITE_POSSIBLE
        upto next TOUR
    end
upto next 2 TOUR
```

Ici les identificateurs comme SAUTER_HAUT et SOUFFLER_FORT font références à d'autres modules, qui peuvent par exemple prendre en compte les battements de coeur.

Ce programme appelle les remarques suivantes, toutes liées à la sémantique du upto:

(i) Les sauts n'ont lieu que si le tour fait plus de 100 mètres. Sinon la partie correspondante du programme n'est jamais activée, étant tuée par le "upto next TOUR".

(ii) De même on ne court vite que si le tour n'est pas terminé après les 100 mètres et les 20 secondes de saut.

(iii) La boucle interne est infinie, mais son exécution est limitée par le "upto next 2 TOUR" externe. On aurait aussi bien pu la limiter par "upto next 1000 SECONDE" ou encore "upto next 1000 METRE".

Nous laissons au lecteur le soin de vérifier qu'un programme aussi simple n'est pas facile à écrire dans les langages classiques.

4.2. La diffusion

Elle a deux avantages: on n'a pas besoin de connaître le ou les destinataire d'un message que l'on émet, on n'a pas besoin de connaître le ou les auteurs d'un message que l'on reçoit. Dans la montre décrite en [3], le module "heure" émet l'heure à chaque modification, sans se soucier de qui l'écoute. Cette heure est attendue d'une part par l'afficheur, d'autre part par le réveil, qui vérifie s'il doit sonner. Remarquons qu'enlever le réveil ou au contraire en ajouter un autre se fait sans aucune modification du module "heure", et que le réveil n'a pas à connaître le module qui lui envoie l'heure.

4.3. L'utilisation de la simultanéité.

Un premier exemple illustre le moyen de gérer une variable consultable à tout instant. Le module de gestion peut recevoir soit une demande d'incrémentation, soit une demande de valeur. Ces demandes correspondent à deux signaux INCREMENTER et DONNER_VALEUR. La valeur est rendue à l'aide d'un signal VALEUR. Supposons d'abord que les deux signaux

INCREMENTER et DONNER_VALEUR ne sont jamais simultanés. Le corps du module de gestion de la variable s'écrit

```
var X:=0 : integer in
    loop
        await
            case next INCREMENTER do X:=X+1
            case next DONNER_VALEUR do emit VALEUR(X)
        end
    end
end
```

Pour mettre la valeur de la variable dans une autre variable Y, un autre module exécutera la séquence

```
emit DONNER_VALEUR;
await VALEUR(Y)
```

Les deux signaux DONNER_VALEUR et VALEUR seront simultanés, et on a donc réalisé un "dialogue infiniment rapide".

Dans le gestionnaire de la variable, il n'est pas nécessaire de supposer que les signaux INCREMENTER et DONNER_VALEUR sont incompatibles. Mais il faut alors choisir d'émettre la valeur avant incrémentation ou la valeur après incrémentation. Dans le premier cas il faut écrire

```
var X:=0 : integer in
    loop
        await
            case next INCREMENTER do
                present DONNER_VALEUR then emit VALEUR(X) end;
                X:=X+1
            case next DONNER_VALEUR do emit VALEUR(X)
        end
    end
end
```

et dans le second cas il faut écrire

```
var X:=0 : integer in
    loop
        await
            case next INCREMENTER do
                X:=X+1;
                present DONNER_VALEUR then emit VALEUR(X) end;
            case next DONNER_VALEUR do emit VALEUR(X)
        end
    end
end
```

Dans le second cas, on peut aussi programmer de la façon suivante, qui est plus élégante car elle met en évidence la priorité de INCREMENTER sur DONNER_VALEUR:

```
var X:=0 : integer in
    loop
        do
            every DONNER_VALEUR do
                emit VALEUR(X)
            end
        upto next INCREMENTER;
        X:=X+1
    end
end
```

Un autre exemple caractéristique est présenté dans le traitement du chronomètre en [3]. Un dialogue infiniment rapide entre deux processus permet à l'un de savoir dans quel état est

l'autre, et d'orienter son comportement en fonction de cette information.

5. Les paradoxes temporels

L'hypothèse de synchronisme peut engendrer certains paradoxes, plus ou moins analogues aux problèmes de court-circuits et d'oscillations rencontrés en électronique. Voici un premier type de paradoxe

```
do emit S upto S
```

Le signal S doit être émis s'il n'est pas présent, et ne doit pas être émis s'il est présent (puisque le corps du upto n'est alors pas exécuté). Ce programme est incohérent, Il se comporte un peu comme une porte "non" dont on branche la sortie sur l'entrée (oscillation). Pour un exemple de court-circuit, considérons un signal S à valeur entière, admettant plusieurs émetteurs avec comme fonction de combinaison des valeurs l'addition usuelle, et écrivons le programme

```
emit S(0);
await S(X);
emit S(X+1)
```

La réception et l'émission étant simultanées, toute réception de n provoque l'émission immédiate de $n+1$. Le court-circuit étant amorcé par l'émission de 0, le signal S ne peut avoir une valeur cohérente.

Ces problèmes apparaissent dès que les signaux d'entrées d'une instruction dépendent des signaux qu'elle peut émettre dans le même instant, et sont donc des problèmes de causalité. Ils sont actuellement détectés par la sémantique statique du langage présentée en [5] et implantée sous une forme améliorée dans compilateur ESTEREL v2. Malheureusement l'interprétation des diagnostics fournis n'est pas aussi simple que nous le souhaiterions. Par ailleurs, à cause de la division sémantique statique - sémantique dynamique, certains programmes parfaitement corrects sont rejetés comme présentant des problèmes de causalité. Nous espérons remédier à cette situation en considérant des sémantiques plus fines (une de ces sémantiques est actuellement étudiée et implantée par F. Boussinot et A. Ressouche).

6. La compilation d'un programme ESTEREL en un automate fini

La sémantique mathématique du langage est présentée en [5, 12]. Elle est donnée par un jeu de règles de réécriture conditionnelle dans le style de Plotkin [16]. Considérons le cas d'un programme P ne comportant que des signaux purs et ne contenant pas de variable (P est alors dit programme en signal pur). Les règles sémantiques permettent de déterminer pour tout événement d'entrée e l'événement de sortie f (formé des signaux émis par P sur réception de e) et un nouveau programme $P\prime$ qui représente la continuation de P après réception de e. Nous employons la notation

$$P \xrightarrow[e]{f} P\prime$$

Les signaux d'entrée de P étant connus et en nombre fini, le nombre d'événements d'entrée de P est également fini, et tout successeur $P\prime$ possède les mêmes événements d'entrée. Appelons ces événements $e_1, e_2, \cdots, e_n$. Posons pour tout i

$$P \xrightarrow[e_i]{f_i} P_i$$

Les transitions correspondantes déterminent tous les comportements de P au "premier instant". Itérons le processus, posant de manière générale pour tout mot w sur l'alphabet $\{1,2,...,n\}$

$$P_w \xrightarrow[e_i]{f_{wi}} P_{wi}$$

Si $w = i_1 i_2 \cdots i_n$, alors P_w représente l'état du programme P après réception de la séquence d'événements $s = e_{i_1} e_{i_2} \cdots e_{i_n}$, et f_{wi} représente l'événement émis par P lorsqu'il reçoit l'entrée

e_i après la séquence s. Nous avons simplement développé le comportement de P en un arbre infini, de façon bien classique.

Théorème: pour tout P, le nombre des P_w syntaxiquement distincts est fini.

Ce résultat (facile) est à rapprocher du théorème de Brzozowski [9] exprimant la terminaison de l'algorithme de la résiduelle sur les expressions rationnelles. Il exprime le fait que l'on peut replier l'arbre de comportement de P en un graphe fini, dont les noeuds sont tous les programmes syntaxiquement distincts engendrés par P. Ce graphe décrit un automate fini *équivalent* à P du point de vue de ses entrées-sorties. Une fois que cet automate est construit, on peut bien entendu jeter les programmes P_w devenus inutiles et les remplacer par de simples numéros d'états.

Pour des programmes généraux avec variables, on peut appliquer la même technique, mais en considérant les opérations sur les variables à un niveau purement symbolique. Les transitions seront alors étiquetées par les opérations à effectuer sur les variables et sur les valeurs des signaux (elles deviennent en fait des arbres, à causes des instructions conditionnelles, mais ceci ne pose pas de difficulté particulière). On obtient alors un automate à contrôle fini couplé à une mémoire contenant les valeurs des variables (qui peuvent appartenir à n'importe quel type).

Cette technique n'est qu'une exploration exhaustive de l'espace d'états du programme. Elle est applicable à tous les langages où l'on ne peut pas créer dynamiquement des processus (ce qui est crucial pour assurer la terminaison), donc aussi à beaucoup de langages asynchrones. Cependant il est bien connu dans le cas asynchrone qu'on assiste immédiatement à une explosion du nombre d'états qui rend la technique inapplicable hors des programmes jouets. Le synchronisme permet la plupart du temps *d'éviter cette explosion*, pour une raison qu'il est essentiel de comprendre: dans un langage asynchrone, *chaque opération interne provoque la création d'une transition et éventuellement d'un nouvel état.* Ceci se voit clairement sur les équations sémantiques des langages, qui sont toujours fondées sur des entrelacements d'actions. Au contraire dans le cas synchrone, *les transitions ne sont engendrées que par l'arrivée d'événements externes, et factorisent toutes les opérations internes exécutées simultanément.* Tous les états sont de vrais états au sens comportement d'entrée-sortie, et il n'exise pas d'état intermédiaire dû au fonctionnement interne du programme.

Le compilateur ESTEREL v2 implémente exactement cet algorithme, avec une efficacité raisonnable. Donnons un exemple: la montre décrite en [3], qui comporte un chronomètre, un réveil et un affichage sophistiqué, produit un automate à 41 états et 273 transitions (en 120 s sur un sps9). Cet automate est produit par le compilateur sous la forme d'un code C, qui occupe après compilation moins de 5 K octets sur un VAX.

Notre technique de compilation appelle un certain nombre de remarques.

(i) Il n'y a pas besoin de minimiser l'automate obtenu, qui est très généralement *déjà minimal* (il est facile de construire des contre-exemples, faisant appel à des situations qui ont peu de chances de se produire dans des programmes normalement écrits). Ce phénomène est évidemment très important. Il est lié à des propriétés encore mal comprises de l'algorithme employé (il semble se produire aussi dans l'algorithme original de la résiduelle [9]).

(ii) L'algorithme transforme le programme parallèle initial en un programme strictement séquentiel équivalent. Il n'y a donc plus de processus à gérer à l'exécution, d'où simplicité et gain de vitesse. Toute la communication inter-processus est résolue à la compilation et codée dans l'automate.

(iii) Produire un automate est idéal pour la plupart des programmes temps-réel. Le temps de transition est extrêmement faible (quelques instructions machine suffisent), et indépendant de la taille de l'automate. Si l'on connaît les temps des processus de calculs introduits explicitement par l'utilisateur (appels de fonctions et procédures externes), on peut mesurer *exactement* le temps de transition maximal et donc connaître exactement la validité de l'hypothèse de synchronisme. De plus le code généré est tellement simple qu'on ne voit pas bien quelle technique pourrait fonctionner lorsque celle-ci ne suffit pas.

(v) Beaucoup d'instructions, et en particulier les dialogues infiniment rapides, n'engendrent aucun code à exécuter lors des transitions. C'est un excellent moyen d'être infiniment rapide.

(vi) Une fois l'automate produit, on peut utiliser des outils de vérification de propriétés sur les automates. Nous avons interfacé le système ESTEREL v2 avec le système EMC [11], et nous comptons l'interfacer aussi avec les systèmes CESAR [17] et MEC [1].

(vii) La technique peut certainement s'appliquer à tous les langages synchrones; elle n'est pas particulière à ESTEREL.

Voici pour les avantages. Il reste quelques inconvénients:

(i) Le style de programmation synchrone reste à mettre au point. Nos expériences actuelles sont probantes mais limitées. Les problèmes de causalité sont plus fréquents et plus gênants que nous ne l'aurions souhaité.

(ii) Le problème de la compilation séparée est ouvert. Nous ne savons pas comment réutiliser de façon efficace le code compilé de sous-modules, à cause de difficultés théoriques encore mal comprises (voir cependant [18]).

(vi) La distribution du code sur plusieurs processeurs n'est pour l'instant pas possible. Nous cherchons actuellement des moyens de faire cohabiter synchronisme et asynchronisme. Un moyen pragmatique fort utile est de compiler séparément les programmes physiquements séparés, et de les intégrer à des programmes asynchrones classiques.

7. Comparaison avec d'autres formalismes

Le langage LUSTRE [2] est un langage fonctionnel synchrone, où les événements sont représentés par des suites de valeurs, les suites étant synchronisées par une "horloge universelle". On peut cependant définir d'autres notions d'horloges et des suites liées à ces horloges, reconstruisant ainsi nos temps multiples. Un programme est un système d'équations aux suites, écrit avec un petit nombre de primitives. La sémantique mathématique est particulièrement simple. L'aspect synchrone apparaît dans la modélisation même de la notion d'événement, et dans le fait que les calculs ne provoquent pas de décalage sur l'horloge universelle. Ainsi une expression comme

$$X = Y + Z$$

signifie qu'à tout instant n on a l'égalité

$$X_n = Y_n + Z_n$$

C'est une moyen élégant de dire que l'opération "+" ne prend pas de temps. La diffusion instantanée des signaux est exprimée par le fait que toute équation peut faire apparaître toute variable de suite en partie droite. Les problèmes de causalité sont très semblables en ESTEREL et LUSTRE. Contrairement à ESTEREL, LUSTRE ne comporte pas de mémoire permanente explicite, mais utilise un mécanisme de référence aux valeurs passées d'une variable (par les opérateurs *pre* et *current*). Les deux langages ont des puissance analogues et sont certainement inter-traduisibles. Mais le style de programmation auxquels ils conduisent est assez différent.

Le langage SIGNAL [13] est un aussi un langage fonctionnel fondé sur une approche flot de données, mais avec une approche plus géométrique pour la construction des programmes (opérateurs de construction de réseaux). Comme en LUSTRE, il n'y a pas de notion de mémoire; mais au lieu de considérer un temps universel pour l'indexation des suites, SIGNAL considère les événements externes comme définissant des horloges, et un calcul d'horloge permet de savoir sur quelle horloge est cadencée une expression. Encore une fois le synchronisme pur est introduit au niveau des opérateurs. La diffusion est possible, et réalisée par exemple dans l'opérateur de parallélisme.

Les Statecharts [14] présentent un moyen hiérarchique de décrire des automates finis, fondé sur une représentation graphique. Ce moyen conduit naturellement à décrire le comportement visible d'un processus, non sa réalisation. Il s'agit donc plus d'un mécanisme de spécification,

utilisable de façon orthogonale à un langage comme ESTEREL, car produisant encore un autre style de spécification. Le synchronisme et la diffusion apparaissent à tous les niveaux du système. La sémantique n'est pas toujours très claire ni très utilisable pour l'instant à notre avis. Il n'y a pas de production de code.

8. Conclusion

Nous avons montré comment l'hypothèse de synchronisme permet de simplifier la programmation des systèmes réactifs déterministes, comment elle s'exprime en ESTEREL, et comment elle peut être "réalisée" par une transformation des programmes en automates à contrôle fini. Nous avons mentionné d'autres formalismes aussi adaptés au même genre d'applications. Deux travaux importants sont en cours:

- L'étude pratique d'applications, qui est fondamentale pour la mise au point du style de programmation synchrone et pour comprendre les limites de la technique; de ce point de vue nous ne considérons pas ESTEREL comme adapté à l'écriture directe de très gros programmes, mais plutôt à l'écriture de noyaux de traitement d'événements, fournissant un code interfaçable avec n'importe quel langage classique. Le compilateur ESTEREL v2 est un support essentiel de ces expérimentations.
- Une étude sémantique plus complète du phénomène de synchronisme, nécessaire en particulier pour résoudre les problèmes de compilation séparée.

Remerciements

Nous tenons à remercier F. Boussinot, V. Lecompte, J-P. Marmorat, S. Moisan, A. Ressouche, J-P. Rigault et J-M. Tanzi qui participent au développement d'ESTEREL.

References

1. A. Arnold, "Construction et analyse des systèmes de transitions : le système MEC," Actes du colloque C3 d'Angoulème ().
2. J-L. Bergerand, P. Caspi, N. Halbwachs, D. Pilaud, and E. Pilaud, "Outline of A Real Time Data Flow Language," Rapport IMAG (1985).
3. G. Berry, "Programming a Digital Watch in ESTEREL v2.1," Collection "ESTEREL v2.1 Programming Examples", rapport technique INRIA/ENSMP (1986).
4. G. Berry, F. Boussinot, P. Couronné, G. Gonthier, and J-P. Marmorat, "ESTEREL v2.1 Programming Examples," Collection de Rapports techniques ENSMP/INRIA (1986).
5. G. Berry and L. Cosserat, "The Synchronous Programming Language ESTEREL and its Mathematical Semantics," Rapport INRIA 327, paru dans "Seminar on Concurrency", Springer-Verlag LNCS 197, à paraître dans Science of Computer Programming (1985).
6. G. Berry, P. Couronné, and G. Gonthier, "ESTEREL v2.1 System Manuals," Collection de rapports techniques ENSMP/INRIA (1986).
7. G. Boudol, "Notes on Algebraic Calculi of Processes," Rapport INRIA 395 (1985).
8. M.C. Browne, E.M. Clarke, and D. L. Dill, "Automatic Circuit Verification Using Temporal Logic : Two New Examples," Carnegie-Mellon University Report (1985).
9. J. A. Brzozowski, "Derivatives of Regular Expressions," *JACM, vol 11, no 4*, (1964).
10. L. Cardelli, "SQUEAK, A Language for Communicating with Mice," AT&T Bell Laboratories Report, Bell Laboratories, Murray Hill, New Jersey 07974 (1985).
11. E.M. Clarke, E.A. Emerson, and A.P. Sistla, "Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach," Department of Computer Science Report, Carnegie-Mellon University (September 1983).
12. L. Cosserat, "Sémantique Opérationnelle du Langage Synchrone ESTEREL," Thèse de Docteur Ingénieur, Université de Nice (1985).
13. P. Le Guernic, A. Benveniste, P. Bournal, and T. Gauthier, "SIGNAL : A Data Flow Oriented Language For Signal Processing," Publication IRISA 246 (1985).
14. D. Harel, "Statecharts : A visual Approach to Complex Systems," Weizmann Institute of Science, Rehovot, Israel (1984).
15. R. Milner, "Calculi for Synchrony and Asynchrony," *Theoretical Computer Science* **25**(3) pp. 267-310 (1983).
16. G.D. Plotkin, "A Structural Approach to Operational Semantics," Lectures Notes, Aarhus University (1981).
17. J-P. Queille and J. Sifakis, "Specification and Verification of Concurrent Systems in CESAR," Proc. International Symposium on Programming, Springer-Verlag LNCS 137 (1982).
18. J-M. Tanzi, "Traduction Structurelle des Programmes ESTEREL en Automates," Thèse de Troisième Cycle, Université de Nice (1985).

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique