

M. HUOU

J.-P. ELLOY

## **Sémantique du parallélisme et du choix du langage Electre**

*Informatique théorique et applications*, tome 29, n° 4 (1995), p. 315-338.

[http://www.numdam.org/item?id=ITA\\_1995\\_\\_29\\_4\\_315\\_0](http://www.numdam.org/item?id=ITA_1995__29_4_315_0)

© AFCET, 1995, tous droits réservés.

L'accès aux archives de la revue « Informatique théorique et applications » implique l'accord avec les conditions générales d'utilisation (<http://www.numdam.org/legal.php>). Toute utilisation commerciale ou impression systématique est constitutive d'une infraction pénale. Toute copie ou impression de ce fichier doit contenir la présente mention de copyright.

NUMDAM

Article numérisé dans le cadre du programme  
Numérisation de documents anciens mathématiques  
<http://www.numdam.org/>

## SÉMANTIQUE DU PARALLÉLISME ET DU CHOIX DU LANGAGE ELECTRE (\*)

par M. HUOU <sup>(1)</sup> et J.-P. ELLOY <sup>(1)</sup>

Communiqué par A. ARNOLD

---

**Résumé.** – *Le langage Electre a été développé à la suite de l'expérience du laboratoire dans la mise en œuvre de lois de commande automatisées en vue du contrôle des processus temps réel. L'application est structurée en tâches dont l'activation et l'interruption sont conditionnées par des occurrences d'événements.*

*On propose ici une sémantique de ce langage, en particulier de ses opérateurs fondamentaux que sont le **parallélisme** des tâches, le **parallélisme** et le **choix** des interruptions de tâches.*

*Il est établi ici le caractère déterministe et complet de cette sémantique, montrant ainsi que pour toute application temps-réel décrite par un programme du langage Electre et dans tout état du système, toute occurrence d'événement provoque le passage déterministe à un nouvel état.*

*Pour obtenir ce résultat, un système de transitions est construit à partir de la syntaxe du langage, dont les états traduisent un état courant du processus contrôlé sous la forme du programme Electre décrivant la suite du contrôle, et dont les transitions sont provoquées par les occurrences d'événements ou les terminaisons de tâches. On établit pour cela un algorithme (à l'aide d'un système d'attributs) du calcul effectif de toute transition.*

**Mots clés :** *Langage réactif, sémantique opérationnelle, système de transitions.*

**Abstract.** – *The Electre language has been developed as a result of the experience of the automatic control department in the implementation of the real time control applications. These applications consist of many tasks that are conditionally executed and interrupted according to some events'occurrences.*

*We propose a language semantic and in particular that of its fundamental operators: parallelism of tasks, parallelism of interruptions, and choice of tasks up on interruption.*

*The deterministic characteristics of this semantic is presented to show that any real time application described by the Electre language can have a deterministic state transition from any state to another one according to an event occurrence.*

*A transition system, based on the language syntax, is constructed to obtain the previous results.*

### 1. INTRODUCTION

Cette étude s'inscrit dans le cadre des travaux sur les langages temps réel destinés à la commande en ligne de procédés industriels complexes, c'est-

---

(\*) Reçu en octobre 1994; révisé en février 1995.

(<sup>1</sup>) LAN École Centrale/université de Nantes (URA, CNRS n° 823), 1, rue de la Noë, 44072 Nantes Cedex 03, France, e-mail: elloy@lan.ec-nantes.fr.

à-dire dont l'action est conditionnée par un grand nombre d'événements engendrés par le procédé. Plus particulièrement sont ciblées ici les applications dites « embarquées », c'est-à-dire dont on connaît *a priori* l'ensemble des événements susceptibles d'en affecter le comportement ainsi que l'ensemble des actions que leur occurrence doit engendrer. Sont ainsi exclues les applications impliquant la création dynamique de tâches et/ou d'événements. Dans ce contexte applicatif, un souci industriel très actuel est de pouvoir modéliser hors-ligne, à des fins de preuve, le comportement de ces applications, en raison du coût prohibitif qu'engendre toute erreur de conception du logiciel qui peut affecter l'intégrité du procédé et de son environnement technique et humain. C'est pourquoi, le langage Electre a été défini à la suite de l'expérience acquise au Laboratoire d'Automatique de Nantes (France) dans la conception et la mise en œuvre de lois de commande de procédés industriels continus et manufacturiers : le langage Electre cible d'ailleurs plus volontiers cette dernière catégorie de procédés, dont l'évolution est en grande partie scandée par des occurrences de signaux émis par les nombreux équipements constitutifs de ces installations.

Le langage Electre ne comporte pas, dans sa version actuelle, de spécification du temps : il en constitue la première phase et se « cantonne » à la conception du comportement des activités du système de commande en fonction des signaux : c'est pourquoi ce langage est qualifié de réactif. A la différence des autres propositions actuelles de langages réactifs, Electre est un langage « asynchrone » de conception des applications temps réel, en ce sens qu'il autorise la spécification de la préemption d'une action en cours par un nouvel événement. Ceci souligne que dans les hypothèses implicites d'utilisation de ce langage, les opérations (tâches) du système de commande ne sont pas perçues comme atomiques du point de vue temporel, mais sont, au contraire, susceptibles de durer « longtemps » par rapport à la fréquence présumée des événements : cette vision est propre à la conception des applications temps réel au niveau macroscopique : c'est le cadre privilégié d'utilisation du langage Electre ; la programmation des tâches (ou actions) temps réel dans les langages réactifs asynchrones incite ainsi le concepteur à introduire dans le corps même des tâches l'expression de leurs conditions événementielles d'exécution au cours de leur évolution ; avec le langage Electre le programmeur doit extraire ces conditions de l'expression des tâches pour les incorporer dans le langage lui-même, et cela en descendant jusqu'à un niveau de spécification choisi en fonction de l'application temps-réel contrôlée.

Electre est donc un langage réactif asynchrone ; mais sa particularité ne s'arrête pas là. Il s'agit en outre d'un langage « squelette », c'est-à-dire non dépendant du langage de programmation employé pour l'implémentation effective des lois de commande. On a constaté expérimentalement que, dans cette optique, on peut réduire les objets fondamentaux manipulés par un tel langage à deux objets essentiels : les tâches (elles ne détaillent pas leur contenu procédural) et les événements. Non seulement cette démarche lève toute dépendance du langage Electre (langage de conception) par rapport à l'implémentation, mais on peut ainsi réaliser (travaux en cours) des passerelles entre langages de spécification classiques (comme SA-RT, Hood) et Electre, ainsi que entre Electre et un langage d'implémentation impératif classique, comme Ada ou C. Prouver un programme conçu en Electre devient possible grâce aux travaux présentés dans cet article. Complété de la preuve (indépendante) des algorithmes des tâches, on pourra ainsi, à terme, aboutir à la preuve complète (comportementale) des applications temps réel.

Cet article propose la sémantique opérationnelle du langage Electre, et en particulier il traite de ses opérateurs fondamentaux que sont le parallélisme des tâches, le parallélisme et le choix des interruptions de tâches, ce qui n'avait pas été fait dans les travaux précédents sur le langage Electre ([PR], [Hu]).

La méthode choisie pour définir cette sémantique s'inspire de l'approche structurelle de Plotkin [PI] et du calcul de réécritures introduit par Knuth et Bendix ([Kn], [KB]). Notre langage est suffisamment simple pour que le calcul de réécritures se fasse à l'aide d'un système d'attributs définis sur la grammaire.

A la différence des langages synchrones ESTEREL ([BC], [BCo]), LUSTRE [CO] et SIGNAL [GB], basés sur des modélisations discrètes du temps, le langage ELECTRE ([ER], [RC], [Ro]) est né de l'idée d'utiliser un formalisme inspiré de celui des expressions de chemin [CH]. Une application temps-réel spécifiée en Electre est découpée en tâches appelées modules de telle façon qu'il ne figure plus dans celles-ci de points de synchronisation blocants. Le langage spécifie l'activation et la préemption de tâches par des événements indépendamment de leur origine, qu'elle soit logicielle (occurrences émises par les tâches) ou matérielle (occurrences émises par l'application contrôlée) ; il décrit la prise en compte des occurrences des événements mais ne décrit par leur genèse : il ne précise pas leurs sources et il ne traite pas de leur production. En outre tout événement peut engendrer la préemption de certains modules au bénéfice d'autres modules

dont l'exécution est alors requise ; les conditions de reprise d'exécution de toute action ainsi suspendue doivent alors être définies : un identificateur de module peut être répété (ou figurer dans une structure répétitive), et dans ce cas l'exécution d'un module sera reprise au début si elle est terminée et sinon au point où elle a été interrompue. Mais comme les modules sont des atomes du langage Electre le point de reprise est de toute façon hors de portée d'une sémantique opérationnelle qui ne peut qu'identifier le module à activer.

L'activation d'un module peut être liée à la fin naturelle d'un autre module, c'est pourquoi le langage prendra en compte les événements « fin de module ».

Des occurrences d'événements peuvent être mémorisées en attendant que leur prise en compte devienne possible, et si un choix doit être fait lorsqu'elle le devient, priorité sera toujours donnée au traitement commandé par l'événement le plus ancien. Nous ferons donc l'hypothèse d'asynchronisme suivante :

*Deux occurrences d'événements ne sont jamais simultanées.*

Cette hypothèse paraît évidemment contradictoire avec la possibilité que plusieurs événements partagent la même source : l'hypothèse est faite au niveau des sources et donc il faudra, au niveau du générateur, dans tous les cas de ce genre remplacer l'ordre temporel par une précédente logique : il est prévu la possible interposition, entre les sources d'occurrences et les événements traités par le langage, d'un « générateur d'événements » permettant de faire partager à plusieurs événements la même source et de définir de nouveaux événements par des opérations logiques ou temporelles sur les événements déjà connus. Un tel générateur d'événements est en dehors du langage et n'apparaît dans la sémantique développée ici qu'à travers les hypothèses faites sur les conditions d'occurrences des événements.

L'évolution de l'application contrôlée est alors rythmée par les occurrences d'événements et les fins naturelles de modules : un état courant de l'application est spécifié en Electre par le programme « de ce qui reste à faire » et la liste des événements mémorisés, un changement d'état est provoqué par une occurrence d'événement ou une fin de module.

La sémantique du langage prend la forme d'un *système de transitions*. Il a été démontré ([PR], [Hu]) que pour un noyau du langage le système de transitions est déterministe et complet ce qui signifie que pour tout état courant de l'application contrôlée et pour toute occurrence d'événement (ou une fin de module) il existe une et une seule transition possible représentant l'évolution de l'application contrôlée. Nous montrerons que

ces propriétés du système de transitions sont conservées pour le langage étendu aux opérateurs que sont le parallélisme des tâches, le parallélisme et le choix des interruptions de tâches. L'étude de la sémantique est donc essentiellement restreinte ici à ces opérateurs.

Le comportement de l'application temps réel spécifié par un programme Electre est alors entièrement décrit par la partie de ce système de transitions accessible à partir de l'état initial associé à ce programme, et ce comportement sera entièrement déterministe même dans le cas parallèle.

Ce travail sur la sémantique formelle s'inscrit dans un ensemble d'activités sur le projet Electre. Nous citerons seulement quelques unes d'entre elles :

Un compilateur [CR] fondé sur le système de règles de cette sémantique opérationnelle a été réalisé : il transforme tout programme Electre en un système de transitions dont il a été démontré qu'il était fini. L'étude de certaines des propriétés de ce système se fait à l'aide de l'outil MEC [Ar]. Un environnement d'exécution pour Electre a été réalisé suite à un appel d'offres STRIN émis par le Ministère de l'Industrie [CL].

Le projet Electre a été soutenu par le C.N.R.S. (PRC C3) et le Groupement de Recherche Automatique (pôle C2A). Des contrats de recherche ont été établis dans ce cadre avec le Ministère de la Recherche et de la Technologie et plusieurs compagnies industrielles françaises telles que Renault-automobiles, Trialog...

## 2. PRÉSENTATION DU LANGAGE

### 2.1. Présentation informelle

Nous présentons ici les principaux aspects du langage, la totalité de celui-ci étant présenté dans [Hu2]. Les entités de base du langage sont les modules et les événements. Les modules sont représentés par des lettres minuscules : **a**, **b**, **c...z**. Les événements sont représentés par la lettre **e** indicée :  $e_1, e_2 \dots e_n \dots$

Nous parlerons de la **fin naturelle** d'un module pour la distinguer des arrêts d'exécution dus aux préemptions. La fin naturelle du module **a** sera notée  $f_a$ .

L'évolution du processus contrôlé est rythmée par les *occurrences* d'événements et les *fins naturelles* de modules.

#### 2.1.1. Séquentialité et répétition indéfinie

L'exécution de modules en séquence est exprimée par la concaténation de leurs identificateurs. Si l'on convient provisoirement (la représentation

définitive étant faite en 2.1.4) de représenter un état courant du processus contrôlé par le programme Electre décrivant la suite du contrôle, alors dans l'état représenté par le programme **abc**, le module **a** est en cours d'exécution et à la fin de **a**, le nouvel état sera **bc**, puis, à la fin de **b**, le nouvel état sera **c**. (On rappelle que le langage ne détaille pas le contenu procédural des modules mais il contrôle leur exécution). L'évolution du processus contrôlé peut donc être représentée par des transitions de la forme :

$$abc - f_a \rightarrow bc \rightarrow f_b \rightarrow c - f_c \rightarrow \text{NIL}$$

où nous introduisons le vocable NIL pour représenter la terminaison du processus contrôlé.

La répétition infinie d'une séquence est exprimée par une \*, par exemple :

$$[abc]^* - f_a \rightarrow [bc][abc]^* - f_b \rightarrow [c][abc]^* - f_c \rightarrow [abc]^* \dots$$

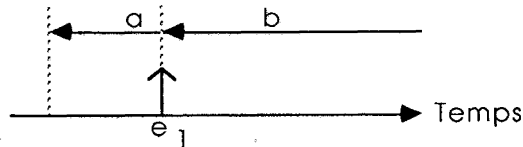
2.1.2. *Préemptions*

Une occurrence d'un événement peut agir sur un module de deux façons : par *préemption* et par *activation*.

La préemption d'un module par un événement est représentée par le symbole « / ». Par exemple  $a/e_1 - e_1 \rightarrow \text{NIL}$  signifie qu'une occurrence de  $e_1$  survenant durant l'exécution de **a** préempte **a**.

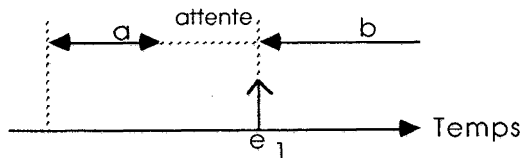
L'activation est représentée par le symbole « : ». Par exemple

$a/e_1 : b - e_1 \rightarrow b$  signifie qu'une occurrence de  $e_1$  survenant durant l'exécution de **a** préempte **a** et active **b**.



Dans le programme  $a/e_1 : b$ , si la fin naturelle de **a** survient avant la première occurrence de  $e_1$ , cette occurrence est attendue pour lancer l'exécution de **b**, et cette attente est exprimée en Electre par un module spécial, noté **1** qui ne fait rien et qui n'a pas de fin naturelle :

$$a/e_1 : b - f_a \rightarrow 1/e_1 : b - e_1 \rightarrow b.$$



Les préemptions ne portent pas seulement sur les modules, mais aussi sur des parties plus complexes de l'application contrôlée; pour ce faire cette partie sera entourée de *crochets* :  $[ab]/e_1 : c - e_1 \rightarrow c$  tandis que  $ab/e_1 : c - e_1 \rightarrow ab/e_1 : c$  et que  $a[b/e_1 : c] - e_1 \rightarrow a[b/e_1 : c]$  puisque dans ces deux derniers exemples seule une occurrence de  $e_1$  peut interrompre **b**.

Remarquons dans ce dernier exemple qu'un changement d'état provoqué par une occurrence de  $e_1$  peut être réduit à un bouclage sur l'état de départ.

Dans l'exemple  $a[b/e_1 : c] - e_1 \rightarrow a[b/e_1 : c] - f_a \rightarrow c$ , l'occurrence de  $e_1$  ne peut préempter que le module **b** et n'aura donc d'effet qu'à la fin de **a**. On constate donc que si on définit l'instant de prise en compte d'un événement comme étant l'instant où la préemption a lieu alors l'instant de prise en compte n'est donc pas nécessairement le même que l'instant d'occurrence.

### 2.1.3. *Parallélisme et choix*

– Plusieurs modules peuvent s'exécuter simultanément, ce qui sera exprimé en Electre par le symbole de parallélisme  $\parallel$ . Ainsi le programme  $a/e_1 : [b\parallel c]$  signifie qu'une occurrence de  $e_1$  interrompt **a** pour activer simultanément **b** et **c**. Comme pour les préemptions, le parallélisme n'opère pas seulement sur les modules, mais peut porter sur des parties plus complexes de l'application contrôlée.

– Par ailleurs plusieurs événements peuvent avoir la faculté d'interrompre le même module, ceci sera fait en parenthésant la structure d'interruption :

a) Dans un tel cas si un *choix* (représenté par le symbole de choix  $\{ \}$ ) doit être fait lorsque leur prise en compte devient possible alors la chronologie des occurrences des événements joue un rôle essentiel dans l'évolution de l'application contrôlée : priorité sera toujours donnée au traitement commandé par l'événement dont l'occurrence arrive la première : ainsi le programme  $a/\{e_1 : b|e_2 : c\}$  signifie qu'une occurrence de  $e_1$  interrompt **a** pour activer **b** ou qu'une occurrence de  $e_2$  interrompt **a** pour activer **c**. Si l'occurrence de  $e_2$  est postérieure à celle de  $e_1$ , elle sera sans effet :

$$a/\{e_1 : b|e_2 : c\} - e_1 \rightarrow b - e_2 \rightarrow b \quad \text{et} \quad a/\{e_1 : b|e_2 : c\} - e_2 \rightarrow c.$$

b) La prise en compte d'une structure *parallèle* d'interruption se traduit par une structure parallèle de contrôle : sur occurrence de  $e_2$  pendant l'exécution de **a**, le programme  $a/\{e_1 : b|e_2 : c\}$  sera remplacé par  $[1/e_1 : b]\parallel c$ , programme qui sur occurrence de  $e_1$  est remplacé par  $b\parallel c$ ; les traitements parallèles sont lancés au fur et à mesure des occurrences d'événements.

#### 2.1.4. Typage des événements

Les événements peuvent être qualifiés de façon différente en fonction du mode de mémorisation souhaité.

a) Événement standard. Nous avons vu que dans l'expression  $a[b/e_1]$  une occurrence de  $e_1$  ne peut interrompre que  $b$ ; si elle survient pendant l'exécution de  $a$  elle doit être *mémorisée* afin de pouvoir être prise en compte à la fin de  $a$ , et  $b$  ne sera pas exécuté. Ainsi la description d'un état courant du processus contrôlé devra comporter, en plus du programme Electre, une mémoire où sont enregistrées les occurrences d'événements survenues. Cette représentation se fera sous la forme d'une suite finie d'identificateurs d'événements, suite finie que nous désignerons par *narration*. Nous représentons désormais un état courant par un couple de la forme

(programme Electre, Narration).

Nous aurons par exemple les transitions suivantes :

$$(a[b/e_1 : c], \text{NIL}) - e_1 \rightarrow (a[b/e_1 : c], e_1) - f_a \rightarrow (c, e_1).$$

La première transition est provoquée par l'occurrence de  $e_1$  qui est mémorisée; la deuxième transition est provoquée par la fin naturelle du module  $a$ ; l'occurrence de  $e_1$  est alors prise en compte par le fait qu'il y a préemption de  $b$  par  $e_1$ . Remarquons que le changement d'état provoqué par l'occurrence de  $e_1$  est réduit à une simple mémorisation de  $e_1$ .

Nous donnons un exemple illustrant, dans la structure de choix le traitement commandé par l'occurrence la plus ancienne :

$$\begin{aligned} (ab/\{e_1 : c|e_2 : d\}, \text{NIL}) - e_2 &\rightarrow (ab/\{e_1 : c|e_2 : d\}, e_2) - e_1 \\ &\rightarrow (ab/\{e_1 : c|e_2 : d\}, e_2 e_1) - f_a \rightarrow (d, e_2 e_1). \end{aligned}$$

b) Événement fugace. On peut souhaiter imposer qu'un événement ne soit jamais mémorisé, la prise en compte ne peut donc avoir lieu qu'à l'instant d'occurrence. Ainsi si  $e_1$  est *fugace* :

$$\begin{aligned} (ab/e_1 : c, \text{NIL}) - e_1 &\rightarrow (ab/e_1 : c, \text{NIL}) - f_a \\ &\rightarrow (b/e_1 : c, \text{NIL}) - e_1 \rightarrow (c, \text{NIL}). \end{aligned}$$

c) Nous voulons que le type d'un événement soit intrinsèque, c'est-à-dire une propriété de l'événement lui-même, indépendante des instances de

l'identificateur dans le programme. Pour chaque programme  $\sim s$  du langage, nous supposons donc donné une partition de l'ensemble  $E = \{e_1, e_2, \dots, e_n\}$  des identificateurs d'événements en deux sous-ensembles  $E_{stan}, E_{@}$  où  $E_{stan}$  désigne l'ensemble des événements du programme  $\sim s$  ayant la propriété standard, les autres événements étant considérés fugaces.

La construction que nous proposons associera un système de transitions à chaque partition de  $E$ .

### 2.1.5. Consommations des événements

Les effacements d'occurrences d'événements dans la mémoire seront gérés sous le nom de consommations.

a) Si lors de la prise en compte il n'y a pas d'activation de module, c'est-à-dire s'il s'agit d'une simple préemption par un événement de type standard alors la consommation est effectuée à la prise en compte :  $(ab/e_1, e_1) - f_a \rightarrow (NIL, NIL)$  et  $(a/e_1, NIL) - e_1 \rightarrow (NIL, NIL)$ .

b) Nous avons vu que les préemptions (dans le cas d'une activation) ont pour effet de substituer un traitement à un autre, et ce traitement de substitution est dit prioritaire. Et nous voulons qu'il le reste jusqu'à sa fin naturelle. Pour ce faire, l'occurrence de  $e_1$  est gardée en mémoire jusqu'à cette fin naturelle, la consommation est effectuée à la fin naturelle du traitement activé. Ainsi, dans l'exemple suivant où  $e_1$  est standard et où  $e_2$  est fugace :

$$(a/e_1 : b/e_2 : ca/e_1 : b, NIL) - e_1 \rightarrow (b/e_2 : ca/e_1 : b, e_1) - e_2 \\ \rightarrow (ca/e_1 : b, e_1) - f_c \rightarrow (b, e_1).$$

nous voyons que  $e_1$  active  $b$  (première transition) qui est interrompu par  $e_2$  qui active  $c$  (deuxième transition) et (puisque  $b$  a été interrompu par  $e_2$ ), la mémorisation de  $e_1$  permet d'assurer, à la fin de  $c$ , la reprise du traitement  $b$  prioritaire sur  $a$  (troisième transition).

Le « traitement de substitution » activé par un événement peut ne pas être un simple module mais un traitement plus complexe : pour le spécifier ce traitement prioritaire sera entouré de **crochets** et la consommation sera effectuée à la fin de celui-ci; exemple (où  $e_1$  est standard) :

$$a/e_1 : [bcd]d - e_1 \rightarrow (bcd, e_1) - f_b \rightarrow (cd, e_1) - f_c \rightarrow (d, NIL).$$

Une transition telle que  $(a/e_1 : bc, e_2) - e_1 \rightarrow (bc, e_2 e_1)$  perdrait l'information que la fin naturelle de  $b$  doit consommer  $e_1$  :  $(bc, e_2 e_1) - f_b \rightarrow$

$(c, ?)$ , information qui est implicite dans le programme  $a/e_1 : bc$  mais qui n'apparaît plus dans le programme  $bc$ . C'est pourquoi nous voulons que cette information apparaisse explicitement dans le programme  $a/e_1 : bc$  à l'aide du symbole  $\odot$ . On écrira le programme  $a/e_1 : bc$  sous la forme  $a/e_1 : b\odot e_1 c$  (et le programme  $a/e_1 : [bc]$  sous la forme  $a/e_1 : [bc]\odot e_1$ ). Ainsi :

$$(a/e_1 : b\odot e_1 c, e_2) - e_1 \rightarrow (b\odot e_1 c, e_2 e_1) - f_b \rightarrow (c, e_2).$$

Comme nous l'avons déjà vu, plusieurs occurrences d'un même événement peuvent s'accumuler en mémoire avant que la prise en compte de la première occurrence ait lieu :

$$(ab/e_1, \text{NIL}) - e_1 \rightarrow (ab/e_1, e_1) - e_1 \rightarrow (ab/e_1, e_1 e_1).$$

Si  $e_1$  a la quantité standard, toutes les occurrences sont effacées. Ainsi :

$$\begin{aligned} (ab/e_1 : c\odot e_1, e_1) - e_1 &\rightarrow (ab/e_1 : c\odot e_1, e_1 e_1) - f_a \\ &\rightarrow (c\odot e_1, e_1 e_1) - f_c \rightarrow (\text{NIL}, \text{NIL}). \end{aligned}$$

*Remarque* : dans le cas standard le comportement est équivalent à celui d'une mémorisation unique de l'événement. On a toutefois souhaité maintenir une mémorisation multiple pour des extensions futures du langage.

Il n'y a pour les événements fugaces ni prise en compte différée ni reprise prioritaire des modules qu'ils déclenchent : ils ne sont pas mémorisables, ni donc consommables.

### 2.1.6. Une grammaire engendrant Electre

Dans ce paragraphe on présente une grammaire hors-contexte engendrant le langage Electre considéré dans la suite. Toutes les constructions visant à formaliser la sémantique du langage seront entièrement établies sous forme de calculs d'attributs sur la grammaire.

Le type des événements est donnée par une partition de  $E$ .

Dans la suite une lettre minuscule précédée d'un *tilde* représentera toujours une chaîne terminale, c'est-à-dire une suite finie de symboles terminaux, c'est-à-dire encore une suite de symboles du langage. Une lettre majuscule précédée d'un tilde représentera une chaîne non-terminale, c'est-à-dire une suite finie de symboles, terminaux ou non.

**Axiome : S Règles de production:**

$$E ::= e_1 \mid e_2 \mid \dots \mid e_n$$

$K ::= E \mid \{J\}$  Un événement composé est un événement ou une structure d'interruption  $\{J\}$ . Plusieurs événements peuvent avoir la faculté

d'interrompre le même module, ceci sera fait en parenthésant la structure d'interruption.

$$\mathbf{J} ::= \mathbf{I} \parallel \mathbf{I} \parallel \mathbf{U} \parallel \mathbf{I} \parallel \mathbf{V}$$

$$\mathbf{U} ::= \mathbf{I} \parallel \mathbf{I} \parallel \mathbf{U}$$

$$\mathbf{V} ::= \mathbf{I} \parallel \mathbf{I} \parallel \mathbf{V}$$

$\mathbf{I} ::= \mathbf{K} \parallel \mathbf{K} : \mathbf{C}$ . Une structure simple d'interruption est constituée d'un événement composé déclenchant éventuellement une structure de contrôle.

$$\mathbf{M} ::= \mathbf{a} \parallel \mathbf{b} \parallel \dots \parallel \mathbf{z} \parallel \mathbf{1}$$

$\mathbf{G} ::= \mathbf{M} \parallel [\mathbf{P}] \parallel \mathbf{M} \odot \mathbf{E} \parallel [\mathbf{P}] \odot \mathbf{E}$ . Un module généralisé est un module ou un module composé entre crochets. Les consommations (symbole  $\odot$ ) sont incorporées à la fin d'un module ou d'un module composé.

$$\mathbf{P} ::= \mathbf{C} \parallel \mathbf{C} \parallel \mathbf{P}$$

$$\mathbf{R} ::= \mathbf{G} \parallel \mathbf{G}^*$$

$$\mathbf{C} ::= \mathbf{R} \parallel \mathbf{R} \mathbf{C} \parallel \mathbf{R} / \mathbf{I}$$

$\mathbf{S} ::= \mathbf{C}$ . Un programme Electre est une structure de contrôle  $\mathbf{C}$  suivie d'un point.

*Exemple* : Dérivation à gauche (à partir de l'axiome) du programme  $\mathbf{ab}/e_1 : \mathbf{c} \odot e_1$ . (avec  $e_1$  appartenant à  $E_{\text{stan}}$ )

$$\begin{aligned} S &\rightarrow C. \rightarrow RC. \rightarrow GC. \rightarrow MC. \rightarrow aC. \rightarrow aR/I. \rightarrow aG/I. \rightarrow aM/I. \\ &\rightarrow ab/I. \rightarrow ab/K : C. \rightarrow ab/E : C. \rightarrow ab/e_1 : C. \\ &\rightarrow ab/e_1 : R. \rightarrow ab/e_1 : G. \rightarrow ab/e_1 : M \odot E. \\ &\rightarrow ab/e_1 : c \odot E. \rightarrow ab/e_1 : c \odot e_1. \end{aligned}$$

**DÉFINITION** : Dans l'exemple ci-dessus  $\mathbf{a}$  est appelé module de tête du programme; si le programme commence par une structure parallèle, comme  $[\mathbf{a} \parallel \mathbf{b}] / e_1 : \mathbf{c} \odot e_1$ , alors  $\mathbf{a}$  et  $\mathbf{b}$  sont appelés modules de tête du programme.

## 2.2. Correction syntaxique

Si l'une des trois hypothèses suivantes n'est pas satisfaite par le programme, alors celui-ci devra être refusé. En fait, il s'agit de corrections syntaxiques, qui ne sont pas exprimables dans la grammaire choisie. La détection des programmes interdits (et leur rejet) n'est pas du ressort du travail effectué ici; ces corrections sont assez simples et naturelles pour ne pas poser de problème à l'écriture d'un programme.

### 2.2.1. Première hypothèse de correction syntaxique

« Tout identificateur d'événement apparaissant dans plusieurs branches parallèles doit être un identificateur d'événement fugace ».

Si un même identificateur d'événement de type standard apparaît dans plusieurs branches parallèles, et si parmi les modules déclenchés par l'événement, la consommation se fait à la fin du premier à se terminer, alors la reprise n'est pas assurée dans les autres branches.

*Exemple* : (afin de ne pas alourdir la notation, nous avons volontairement écrit  $a/e_1 : b$  (où  $e_1$  est standard) à la place de  $a/e_1 : b \odot e_1$ )

$$\begin{aligned}
 & ([a/e_1 : b]^* \parallel [c/e_1 : d/e_2 : g]^*, \text{NIL}) \\
 - e_1 & \rightarrow ([b [a/e_1 : b]^* \parallel [d/e_2 : g] [c/e_1 : d/e_2 : g]^*, e_1) \\
 - f_b & \rightarrow ([a/e_1 : b]^* \parallel [d/e_2 : g] [c/e_1 : d/e_2 : g]^*, \text{NIL}) \\
 - e_2 & \rightarrow ([a/e_1 : b]^* \parallel g [c/e_1 : d/e_2 : g]^*, e_2) \\
 - f_g & \rightarrow ([a/e_1 : b]^* \parallel [c/e_1 : d/e_2 : g]^*, \text{NIL}).
 \end{aligned}$$

Comme  $e_1$  a été consommé par la fin de  $b$ , il n'y a pas de reprise de  $d$  prioritaire sur celle de  $c$ , ce qu'on attendait normalement de la structure de contrôle  $[c/e_1 : d/e_2 : g]^*$  (puisque  $d$  a été interrompu par  $e_2$ ).

De même si parmi les modules déclenchés par l'événement, la consommation se fait à la fin du dernier à se terminer, alors il y aura dans les autres branches des reprises injustifiées :

$$\begin{aligned}
 & ([a/e_1 : b]^* \parallel [c/e_1 : d/e_2 : g]^*, \text{NIL}) \\
 - e_1 & \rightarrow ([b [a/e_1 : b]^* \parallel [d/e_2 : g] [c/e_1 : d/e_2 : g]^*, e_1) \\
 - f_b & \rightarrow ([b [a/e_1 : b]^* \parallel [d/e_2 : g] [c/e_1 : d/e_2 : g]^*, e_1).
 \end{aligned}$$

Comme  $d$  ne s'est pas terminé,  $e_1$  n'a pas été consommé et, dans la branche gauche, à la fin de  $b$  il y a reprise prioritaire de  $b$  sur  $a$  ce qu'on n'attendait pas de la structure  $[a/e_1 : b]^*$ .

On ne peut admettre (pour un événement standard) que la fin d'un module dans une branche influence (par l'intermédiaire des consommations) la possibilité de prise en compte dans une autre. La façon la plus simple de gérer branche par branche mémorisations, changements d'état et consommations, nous paraît être de traiter l'événement sous des noms différents selon les branches, puisque rien n'empêche plusieurs événements (différenciés par leurs identificateurs) de partager la même source.

Exemple de programme rejeté comme syntaxiquement incorrect (avec  $e_1$  appartenant à  $E_{\text{stan}}$ ) :  $[(a/e_1 : b \odot e_1) \parallel (c/e_1 : d \odot e_1)]$ , puisque l'événement standard  $e_1$  figure dans les deux branches parallèles.

On observera que chaque occurrence d'un événement fugace induira en fait une synchronisation entre les branches où elle peut être prise en compte : dans l'exemple suivant, sur occurrence de  $e_2$  fugace, les modules  $b$  et  $d$  se déclenchent simultanément dans les deux branches :

$$\begin{aligned} & (([a/e_2 : b]^* || [c/e_2 : d]^*), NIL) - e_2 \\ & \rightarrow ([b(a/e_2 : b)^* || d(c/e_2 : d)^*], NIL). \end{aligned}$$

### 2.2.2. Deuxième hypothèse de correction syntaxique

« Pour qu'un programme du langage soit syntaxiquement correct il faudra interdire qu'un même identificateur de module (excepté  $1$ ) n'apparaisse dans plusieurs branches parallèles ».

Les exécutions simultanées d'un même module dans des branches parallèles n'ont pas de sens ; ainsi le programme suivant  $[a/e_2 : b||a]$  (où  $e_2$  est fugace) n'a pas de sens car sur occurrence de  $e_2$  le module  $a$  se terminerait dans la branche de gauche tandis que l'exécution de  $a$  se poursuivrait dans celle de droite : et le langage ne pourrait faire la différence entre la fin de  $a$  dans une branche et son exécution dans l'autre.

### 2.2.3. Troisième hypothèse de correction syntaxique

« Pour qu'un programme du langage soit syntaxiquement correct il faudra interdire que la fin naturelle d'un traitement activé consomme un événement qui ne l'a pas activé. La grammaire n'interdit pas un programme tel que  $a/e_1 : [b||c]@e_2$  mais la fin naturelle du traitement  $[b||c]$  activé par  $e_1$  ne peut consommer que l'événement susceptible de l'activer, c'est-à-dire  $e_1$ . »

*Désormais on ne considère que des programmes corrects.*

## 3. CONSTRUCTION DE LA SÉMANTIQUE DU LANGAGE ELECTRE

INTRODUCTION : Nous avons vu (§ 2.1.4) qu'un état courant de l'application temps-réel est un couple (**programme** « de ce qui reste à faire », Narration), la narration étant une suite finie d'identificateurs d'événements qui représentent la chronologie des occurrences d'événements survenus et encore mémorisés et qui sera notée  $N$ . L'évolution de l'application contrôlée est rythmée par les occurrences d'événements et les fins naturelles de modules : un changement d'état est donc provoqué par une occurrence d'événement  $e_i$  ou une fin de module  $f_a$ . Différents cas peuvent se produire :

1. Le changement d'état est provoqué par l'occurrence d'un événement, et l'instant d'occurrence n'est pas le même que l'instant de prise en compte ;

par exemple, considérons l'état  $(\mathbf{ab}/e_1 : \mathbf{c} \odot e_1, \text{NIL})$  une occurrence de  $e_1$  (où  $e_1$  est standard) ne sera prise en compte qu'à la fin de  $\mathbf{a}$ . Dans ce cas, puisqu'en présence de  $e_1$  le programme de ce qui reste à faire est inchangé, le nouvel état est  $(\mathbf{ab}/e_1 : \mathbf{c} \odot e_1, e_1)$ , la seule modification étant la concaténation de  $e_1$  à  $N$ , ceci afin de mettre à jour la narration; la transition est donc :

$$(\mathbf{ab}/e_1 : \mathbf{c} \odot e_1, \text{NIL}) - e_1 \rightarrow (\mathbf{ab}/e_1 : \mathbf{c} \odot e_1, e_1).$$

2. Le changement d'état est provoqué par l'occurrence d'un événement et l'instant d'occurrence est le même que l'instant de prise en compte : dans ce cas il y a préemption, et le programme de ce qui reste à faire est modifié; par exemple, considérons l'état  $(\mathbf{a}/e_1 : \mathbf{b} \odot e_1, \text{NIL})$ . Une occurrence de  $e_1$  est immédiatement prise en compte, le programme de ce qui reste à faire est  $\mathbf{b} \odot e_1$  et le couple  $(\mathbf{a}/e_1 : \mathbf{b} \odot e_1, e_1)$  ne représente pas un état de l'application. Dans ce cas le nouvel état est  $(\mathbf{b} \odot e_1, e_1)$  et la transition est :  $(\mathbf{a}/e_1 : \mathbf{b} \odot e_1, \text{NIL}) - e_1 \rightarrow (\mathbf{b} \odot e_1, e_1)$ .

Et c'est le calcul permettant d'obtenir le couple  $(\mathbf{b} \odot e_1, e_1)$  à partir du couple  $(\mathbf{a}/e_1 : \mathbf{b} \odot e_1, e_1)$  que nous appelons *calcul de réévaluation*.

3. Parfois dans les hypothèses de 2, la prise en compte d'un événement entraîne la prise en compte d'autres événements : par exemple considérons l'état  $(\mathbf{a}/e_1 : \mathbf{b} \odot e_1/e_2 : \mathbf{c} \odot e_2, e_2)$ . Une occurrence de  $e_1$  est immédiatement prise en compte et, en l'absence de  $e_2$ , le programme de ce qui reste à faire serait  $\mathbf{b} \odot e_1/e_2 : \mathbf{c} \odot e_2$ ; mais en présence de  $e_2$  le programme de ce qui reste à faire est  $\mathbf{c} \odot e_2$  : nous verrons que dans ce cas plusieurs réévaluations sont nécessaires; et la transition est :  $(\mathbf{a}/e_1 : \mathbf{b} \odot e_1/e_2 : \mathbf{c} \odot e_2, e_2) - e_1 \rightarrow (\mathbf{c} \odot e_2, e_2 e_1)$ .

4. Reprenons maintenant l'exemple de la transition  $(\mathbf{a}/e_1 : \mathbf{b} \odot e_1, \text{NIL}) - e_1 \rightarrow (\mathbf{b} \odot e_1, e_1)$  dans laquelle l'occurrence et la prise en compte de  $e_1$  sont simultanées. Le calcul distingue cependant l'occurrence, exprimée par la concaténation de  $e_1$  à  $N$ , de la prise en compte, exprimée par la réévaluation de  $(\mathbf{a}/e_1 : \mathbf{b} \odot e_1, e_1)$  en  $(\mathbf{b} \odot e_1, e_1)$ .

On y voit que la réévaluation effectue la mise à jour de la narration conformément au type de l'événement. Pour  $e_1$  fugace, le couple  $(\mathbf{a}/e_1 : \mathbf{b} \odot e_1, e_1)$  se réévalue en  $(\mathbf{b} \odot e_1, \text{NIL})$  et la transition est  $(\mathbf{a}/e_1 : \mathbf{b} \odot e_1, \text{NIL}) - e_1 \rightarrow (\mathbf{b} \odot e_1, \text{NIL})$  où n'apparaît pas de mémorisation de  $e_1$ .

De fait, en l'absence de prise en compte, la réévaluation consistera simplement en la mise à jour de  $N$ . La transition  $(\mathbf{ab}/\mathbf{e}_1 : \mathbf{c} \odot \mathbf{e}_1, \text{NIL}) - \mathbf{e}_1 \rightarrow (\mathbf{ab}/\mathbf{e}_1 : \mathbf{c} \odot \mathbf{e}_1, \mathbf{e}_1)$  a été écrite pour un événement de type standard. Pour  $\mathbf{e}_1$  de type fugace, la mise à jour de  $N$  est effectuée par la réévaluation du couple  $(\mathbf{ab}/\mathbf{e}_1 : \mathbf{c} \odot \mathbf{e}_1, \mathbf{e}_1)$  en  $(\mathbf{ab}/\mathbf{e}_1 : \mathbf{c} \odot \mathbf{e}_1, \text{NIL})$  et la transition est :

$$(\mathbf{ab}/\mathbf{e}_1 : \mathbf{c} \odot \mathbf{e}_1, \text{NIL}) - \mathbf{e}_1 \rightarrow (\mathbf{ab}/\mathbf{e}_1 : \mathbf{c} \odot \mathbf{e}_1, \text{NIL})$$

où l'on voit que l'événement fugace  $\mathbf{e}_1$  n'est pas mémorisé.

*Ainsi, on ne confondra pas l'expression d'une occurrence, constatée par sa concaténation à la narration d'un couple, avec sa mémorisation qui ne peut être constatée que dans le nouvel état résultant de la transition.*

On voit que les réévaluations doivent effectuer la mise à jour de la narration conformément à la partition de  $E$ .

5. Les fins de module sont traitées comme des événements fugaces :

- elles n'apparaissent jamais dans le deuxième terme d'un état,
- mais elles sont concaténées à l'environnement pour permettre le calcul de réévaluation qui détermine le programme de ce qui reste à faire tout en assurant leur effacement.

Par exemple, dans l'état  $(\mathbf{ab}, \text{NIL})$ , l'exécution du module  $\mathbf{a}$  est en cours, et à la fin de  $\mathbf{a}$ , le couple  $(\mathbf{ab}, f_a)$  est réévalué en  $(\mathbf{b}, \text{NIL})$ . La transition est  $(\mathbf{ab}, \text{NIL}) - f_a \rightarrow (\mathbf{b}, \text{NIL})$ .

6. Nous avons vu en 3 que la prise en compte d'un événement peut entraîner la prise en compte d'autres événements et dans ce cas plusieurs réévaluations sont nécessaires ; et ces réévaluations cessent lorsqu'un couple se réévalue en lui-même (est *stable*) : nous obtenons alors le nouvel état.

En résumé, la marche à suivre pour le calcul d'une transition sera la suivante.

Pour  $\mathbf{e}'$  de la forme  $\mathbf{e}_1$  ou  $\mathbf{f}_a$ , les transitions (s'il en existe) d'étiquette  $\mathbf{e}'$  à partir d'un état  $(\mathbf{S} \rightarrow \sim s, \mathbf{N})$  sont déterminées comme suit :

1° On forme le couple  $(\mathbf{S} \rightarrow \sim s, \mathbf{N}.\mathbf{e}')$  où  $\mathbf{N}.\mathbf{e}'$  est obtenu par concaténation de l'étiquette  $\mathbf{e}'$  à la narration  $\mathbf{N}$ .

2° A partir du couple ainsi obtenu, le calcul de réévaluation en fournit un autre et l'on répète ce calcul jusqu'à l'obtention d'un couple stable  $(\mathbf{S} \rightarrow \sim s', \mathbf{N}')$  (ou  $(\text{NIL}, \mathbf{N}')$ ).

On a alors obtenu la transition :

$$(\mathbf{S} \rightarrow \sim s, N) - e' \rightarrow (\mathbf{S} \rightarrow \sim s', N') \text{ (ou } (NIL, N'))$$

Trois questions fondamentales se posent alors :

1° Étant donné un état  $(\mathbf{S} \rightarrow \sim s, N)$ , et si une occurrence d'événement (ou une fin de module) survient, engendre-t-elle toujours un nouvel état ?

2° Ce nouvel état est-il unique ?

3° Existe-t-il un algorithme permettant de calculer ce nouvel état ?

La réponse à ces trois questions est positive, ce qui sera justifié au § 3.3.

La méthode pour trouver ce nouvel état consiste à utiliser les systèmes de transition (ce système étant présenté formellement en 3.2), les outils pour le calcul du nouvel état sont donnés en 3.1 et les propriétés du système de transitions (existence et unicité du nouvel état) le sont en 3.3 (le calcul effectif du nouvel état étant présenté en 3.3.3).

### 3.1. Calcul d'une transition

#### 3.1.1. Présentation des attributs

Le calcul du nouveau couple stable nécessite le calcul de deux attributs : l'un, noté DCL (DéCLenchement d'une préemption) qui donne pour les structures d'interruption la liste des événements interrupteurs qui sera défini en 3.1.2 et l'autre noté ETR (Évaluation d'une TRansition) qui fournira le couple stable et qui sera défini en 3.1.3.

Chaque programme a une unique dérivation à gauche : afin de faciliter l'expression des calculs d'attributs nous ferons précéder le programme de sa dérivation à gauche à partir de l'axiome S.

#### 3.1.2. Calcul du déclenchement d'une préemption : l'attribut DCL

Dans le programme  $\mathbf{a}/\{\mathbf{e}_1 : \mathbf{b} \odot \mathbf{e}_1 | \mathbf{e}_2 : \mathbf{c} \odot \mathbf{e}_2\}$  une occurrence de  $\mathbf{e}_1$  interrompt  $\mathbf{a}$  et le programme de ce qui reste à faire est  $\mathbf{b}$ , tandis qu'une occurrence de  $\mathbf{e}_2$  interrompt  $\mathbf{a}$  et le programme de ce qui reste à faire est  $\mathbf{c}$ . Dans une structure d'interruption il importe donc de savoir l'information

suivante : quel est l'événement interrupteur (appelé aussi déclencheur car il déclenche en général une structure de contrôle).

Une structure de contrôle sous la menace d'une structure d'interruption ne peut évoluer qu'en l'absence de préemption. Il importe donc aussi de définir formellement l'absence de préemption.

Ces deux informations seront fournies à l'aide d'un attribut synthétisé défini sur les non terminaux d'interruption E, K, J, U, V, I, cet attribut est à valeurs dans l'ensemble des suites extraites de N (y compris NIL), et comme il dépend de la narration N, nous le notons  $DCL_N$  et ses règles de calcul sont les suivantes :

$$\begin{aligned} E &::= e_i & DCL_N(E) &::= e_i \text{ si } e_i \text{ figure dans } N, \text{ NIL sinon} \\ K &::= E & DCL_N(K) &::= DCL_N(E) \\ K &::= \{J\} & DCL_N(K) &::= DCL_N(J) \\ X &::= I \\ (\text{avec } X = J, U \text{ ou } V) & DCL_N(X) &::= DCL_N(I) \end{aligned}$$

La règle suivante exprime le fait que dans une structure d'interruption parallèle les traitements parallèles sont lancés au fur et à mesure des occurrences d'événements.

$$\begin{aligned} X &::= I \parallel U \\ (\text{avec } X = J, U \text{ ou } V) & \text{ si } DCL_N(I) = DCL_N(U) = \text{NIL} \\ & \text{ alors } DCL_N(X) := \text{NIL} \\ & \text{ si } DCL_N(I) \neq \text{NIL} \text{ et } DCL_N(U) = \text{NIL} \text{ alors} \\ & DCL_N(X) := DCL_N(I) \\ & \text{ si } DCL_N(U) \neq \text{NIL} \text{ et } DCL_N(I) = \text{NIL} \text{ alors} \\ & DCL_N(X) := DCL_N(U) \text{ sinon } DCL_N(X) \text{ est} \\ & \text{ égale à la suite extraite de } N \text{ obtenue en conservant} \\ & \text{ tous les termes figurant dans l'une au moins des} \\ & \text{ deux suites } DCL_N(I), DCL_N(U). \end{aligned}$$

(ce qui signifie par exemple que si  $N = e_1 e_1 e_4 e_2 e_4 = t_1 t_2 t_3 t_4 t_5$ , si  $DCL_N(I) = t_2 t_4 t_5$  et si  $DCL_N(U) = t_1 t_2 t_4 t_5$ , alors  $DCL_N(X) = t_1 t_2 t_4 t_5$ , où l'on voit que le troisième terme de N ( $t_3 = e_4$ ) ne figure pas dans  $DCL_N(X)$ ).

La règle suivante exprime le fait que dans une structure de choix le traitement est commandé par l'occurrence la plus ancienne.

$X ::= I \parallel V$   
(avec  $X = J$  ou  $V$ )

si  $DCL_N(I) = DCL_N(V) := NIL$   
alors  $DCL_N(X) := NIL$   
si  $DCL_N(I) \neq NIL$  et  $DCL_N(V) = NIL$  alors  
 $DCL_N(X) = DCL_N(I)$  si  $DCL_N(V) \neq NIL$   
et  $DCL_N(I) = NIL$  alors  $DCL_N(X) = DCL_N(V)$   
sinon :

a) dans le cas où dans la suite  $N$  le premier terme de  $DCL_N(I)$  précède celui de  $DCL_N(V)$  alors  
 $DCL_N(X) = DCL_N(I)$

b) dans le cas où dans la suite  $N$  le premier terme de  $DCL_N(V)$  précède celui de  $DCL_N(I)$  alors  
 $DCL_N(X) = DCL_N(V)$

c) Sinon alors on décide arbitrairement  
 $DCL_N(X) = DCL_N(I)$

$I ::= K$   
 $I ::= K : C$

$DCL_N(I) = DCL_N(K)$   
 $DCL_N(I) = DCL_N(K)$

### 3.1.3. Le calcul des réévaluations : l'attribut $ETR_N$

Pour réévaluer un couple  $(S \rightarrow \sim s, N)$  nous calculerons sur la dérivation  $S \rightarrow \sim s$  un attribut dépendant non seulement de la partition de  $E$  mais aussi du terme  $N$  du couple considéré, attribut dont la valeur en  $S$  sera un nouveau couple  $(S \rightarrow \sim s', N')$ . Cet attribut dont la répétition sert à calculer les transitions sera appelé **E**valuation des **T**ransitions et comme il dépend de la narration  $N$ , nous le notons  $ETR_N$ .

L'attribut  $ETR_N$  est synthétisé, il dépend de l'attribut  $DCL_N$  (mais  $DCL_N$  ne dépend pas de  $ETR_N$ ) et il est défini sur toutes les instances de non-terminaux.

Les valeurs de l'attribut  $ETR_N$  sur les dérivations à gauche  $X \rightarrow \sim x$  sont des couples  $(Y \rightarrow \sim y, N')$  ou  $(NIL, N')$ .

Dans la suite on écrira indifféremment  $(X \rightarrow \sim x, N) - ETR \rightarrow (X' \rightarrow \sim x', N')$  pour  $ETR_N(X \rightarrow \sim x) = (X' \rightarrow \sim x', N')$ , et l'on abrègera par exemple la règle :

“Si  $ETR_N(N_i \rightarrow \sim n_i) = (N'_i \rightarrow \sim n'_i, N')$  et  $ETR_N(N_k \rightarrow \sim n_k) = (N'_k \rightarrow \sim n'_k, N'')$  alors  $ETR_N(N_0 \rightarrow \sim n) = (N'_0 \rightarrow \sim n', N''')$ ” sous la forme

$$\frac{\left\{ \begin{array}{l} (N_i \rightarrow \sim n_i, N) - ETR \rightarrow (N'_i \rightarrow \sim n'_i, N'), \\ (N_k \rightarrow \sim n_k, N) - ETR \rightarrow (N'_k \rightarrow \sim n'_k, N'') \end{array} \right\}}{(N_0 \rightarrow \sim n, N) - ETR \rightarrow (N'_0 \rightarrow \sim n', N''')}$$

Toutes les règles de calcul ont été définies [(Hu 2)] et [PR].

Nous énonçons les règles de calcul de l'attribut synthétisé  $\mathbf{ETR}_N$  pour les seules règles de grammaire  $\mathbf{X} ::= \mathbf{I} \parallel \mathbf{U}$  (avec  $X = J$  ou  $U$ ) et  $\mathbf{X} ::= \mathbf{I} \mid \mathbf{V}$  (avec  $X = J$  ou  $V$ ).

*Parallélisme d'une structure d'interruption*

$\mathbf{X} ::= \mathbf{I} \parallel \mathbf{U}$  avec  $\mathbf{X} = \mathbf{J}$  ou  $\mathbf{U}$

Nous examinons le cas où il y a préemption, c'est-à-dire  $\text{DCL}_N(X) \neq \text{NIL}$ . Alors  $\text{DCL}_N(I) \neq \text{NIL}$  ou  $\text{DCL}_N(U) \neq \text{NIL}$  et on discute selon que  $\text{DCL}_N(I)$  et  $\text{DCL}_N(U)$  sont NIL ou non et dans le cas où le(s)  $\text{DCL}_N$  est (sont) différent de NIL on discute selon les valeurs de  $\mathbf{ETR}_N$ . On obtient par exemple :

a)  $\text{DCL}_N(I) = \text{NIL}$  et  $\text{DCL}_N(U) \neq \text{NIL}$ .

S'il y a préemption par la seule branche U qui se termine, l'évolution de X exprime l'attente de l'évolution de I dans la branche gauche et donc on a la règle suivante :

$$\frac{(U \rightarrow \sim u, N) - \text{ETR} \rightarrow (\text{NIL}, N')}{\left\{ \begin{array}{l} (X \rightarrow I \parallel U \rightarrow \sim i \parallel \sim u, N) - \text{ETR} \\ \rightarrow (P \rightarrow C \rightarrow R/I \rightarrow 1/I \rightarrow 1/ \sim i, N') \end{array} \right\}}$$

S'il y a préemption par la seule branche U qui évolue, l'évolution de X exprime l'évolution de U dans la branche droite et l'attente de l'évolution de I dans la branche gauche d'où la règle suivante :

$$\frac{(U \rightarrow \sim u, N) - \text{ETR} \rightarrow (P \rightarrow \sim p, N')}{\left\{ \begin{array}{l} (X \rightarrow I \parallel U \rightarrow \sim i \parallel \sim u, N) - \text{ETR} \\ \rightarrow (P \rightarrow C \parallel P \rightarrow R/I \parallel P \rightarrow 1/I \parallel P \rightarrow 1/ \sim i \parallel P \rightarrow 1/ \sim i \parallel \sim p, N') \end{array} \right\}}$$

b)  $\text{DCL}_N(I) \neq \text{NIL}$  et  $\text{DCL}_N(U) \neq \text{NIL}$ .

$$\frac{(I \rightarrow \sim i, N) - \text{ETR} \rightarrow (\text{NIL}, N') \text{ et } (U \rightarrow \sim u, N) - \text{ETR} \rightarrow (\text{NIL}, N'')}{(X \rightarrow I \parallel U \rightarrow \sim i \parallel \sim u, N) - \text{ETR} \rightarrow (\text{NIL}, N' \Delta N'')}$$

où  $N' \Delta N''$  est la suite extraite de N obtenue en conservant les termes figurant à la fois dans  $N'$  et dans  $N''$  : le nouvel environnement tient compte des suppressions effectuées dans chaque branche. (Ce qui signifie par exemple que si  $N = e_1 e_2 e_3 e_4 e_1 = t_1 t_2 t_3 t_4 t_5$ , si  $N' = t_1 t_2 t_3 t_5$  et si  $N'' = t_1 t_2 t_4 t_5$ , alors  $N' \Delta N'' = t_1 t_2 t_5 = e_1 e_2 e_1$ , où l'on voit que le nouvel environnement tient compte des suppressions de  $e_4$  dans la branche gauche et de  $e_3$  dans la branche droite.)

$$\frac{\left\{ \begin{array}{l} (I \rightarrow \sim i, N) - \text{ETR} \rightarrow (\text{NIL}, N') \\ \text{et } (U \rightarrow \sim u, N) - \text{ETR} \rightarrow (P \rightarrow \sim p, N'') \end{array} \right\}}{(X \rightarrow I || U \rightarrow \sim i || \sim u, N) - \text{ETR} \rightarrow (P \rightarrow \sim p, N' \Delta N'')}$$

$$\frac{\left\{ \begin{array}{l} (I \rightarrow \sim i, N) - \text{ETR} \rightarrow (C \rightarrow \sim c, N') \\ \text{et } (U \rightarrow \sim u, N) - \text{ETR} \rightarrow (\text{NIL}, N'') \end{array} \right\}}{(X \rightarrow I || U \rightarrow \sim i || \sim u, N) - \text{ETR} \rightarrow (P \rightarrow C \rightarrow \sim c, N' \Delta N'')}$$

$$\frac{\left\{ \begin{array}{l} (I \rightarrow \sim i, N) - \text{ETR} \rightarrow (C \rightarrow \sim c, N') \\ \text{et } (U \rightarrow \sim u, N) - \text{ETR} \rightarrow (P \rightarrow \sim p, N'') \end{array} \right\}}{\left\{ \begin{array}{l} (X \rightarrow I || U \rightarrow \sim i || \sim u, N) - \text{ETR} \\ \rightarrow (P \rightarrow C || P \rightarrow \sim c || P \rightarrow \sim c || C \rightarrow \sim c || \sim p, N' \Delta N'') \end{array} \right\}}$$

*Choix d'une structure d'interruption*

**X ::= I|V** avec **X = J** ou **V**

Si  $\text{DCL}_N(X) \neq \text{NIL}$  alors  $\text{DCL}_N(I) \neq \text{NIL}$  ou  $\text{DCL}_N(V) \neq \text{NIL}$

Par exemple si  $\text{DCL}_N(I) = \text{NIL}$  et si  $\text{DCL}_N(V) \neq \text{NIL}$ , l'évolution de X est celle de V. Ainsi

$$\frac{(V \rightarrow \sim v, N) - \text{ETR} \rightarrow (C \rightarrow \sim c, N')}{(X \rightarrow I | V \rightarrow \sim i | \sim v, N) - \text{ETR} \rightarrow (C \rightarrow \sim c, N')}$$

### 3.1.4. Couples stables

Pour introduire le calcul du nouvel état (§ 3) nous avons évoqué la notion de couple stable; nous pouvons maintenant en donner une définition rigoureuse : les couples stables sont, pour une partition donnée de E définissant les propriétés des événements,

- d'une part les couples  $(\text{NIL}, N)$ , où N une suite finie d'éléments de E,
- d'autre part les couples  $(S \rightarrow \sim s, N)$  pour lesquels les deux affirmations suivantes sont vraies :

1.  $S \rightarrow \sim s$  est une dérivation selon la grammaire et N est une suite finie d'éléments de E,

2.  $\text{ETR}_N(S \rightarrow \sim s) = (S \rightarrow \sim s, N)$ .

*Remarque:* Les règles de calcul de [(Hu 2)] et [PR] ont été définies de façon à correspondre à la sémantique intuitive donnée par les créateurs du langage; elles sont telles que, lorsque N ne contient pas de fin de module

ou d'événement fugace, ni un identificateur d'événement susceptible de préempter un module de tête de  $\sim s$ , alors  $(S \rightarrow \sim s, N)$  est un couple stable.

Nous pouvons maintenant donner une présentation rigoureuse du système de transitions.

### 3.2. Le système de transitions décrivant la sémantique

Il dépend du typage des événements (c'est-à-dire d'une partition de l'ensemble  $E$  des identificateurs d'événements, définissant les propriétés des événements). Les transitions de ce système représentent les changements d'état provoqués par des événements ou des fins de module.

Il est étiqueté par les identificateurs d'événements ou de fins de modules : son alphabet est l'ensemble  $\bar{E} = E \cup \dot{E}$  avec  $\dot{E} = \{f_a, f_b, f_c, \dots, f_z\}$  et  $E = \{e_1, e_2, \dots, e_n\}$ .

Ses états seront les couples  $(\mathbf{NIL}, N)$  ou les couples stables  $(S \rightarrow \sim s, N)$ .

Ses transitions sont définies à partir des attributs  $\text{ETR}_N$  de la façon suivante :

Étant donné un état  $(S \rightarrow \sim s, N)$ , une étiquette  $e'$  (pour  $e'$  de la forme  $e_1$  ou  $f_a$ ) et  $N.e'$  la chaîne obtenue par concaténation de  $e'$  au bout de la chaîne  $N$ , il existe une transition d'étiquette  $e'$  de l'état  $(S \rightarrow \sim s, N)$  vers un état  $(S \rightarrow \sim s', N')$  (respectivement  $(\mathbf{NIL}, N')$ ) si, et seulement si,

– ou bien  $(S \rightarrow \sim s', N') = (S \rightarrow \sim s, N.e')$ ,

– ou bien il existe une suite finie des réévaluations pour des valeurs successives du paramètre  $N$  :

$$\begin{aligned} (S \rightarrow \sim s, N.e') - \text{ETR} &\rightarrow (S \rightarrow \sim s_1, N_1) - \text{ETR} \rightarrow \dots - \text{ETR} \\ &\rightarrow (S \rightarrow \sim s_n, N_n) - \text{ETR} \rightarrow (S \rightarrow \sim s', N') \\ &(\text{respectivement } (S \rightarrow \sim s_n, N_n) - \text{ETR} \rightarrow (\mathbf{NIL}, N')). \end{aligned}$$

On a alors obtenu la transition,

$$(S \rightarrow \sim s, N) - e' \rightarrow (S \rightarrow \sim s', N') \text{ (ou } (\mathbf{NIL}, N')).$$

### 3.3. Propriétés du système de transition

Nous pouvons maintenant justifier les réponses données au début du § 3 : étant donné un état  $(S \rightarrow \sim s, N)$ , si une occurrence d'événement  $e'$  survient (pour  $e'$  de la forme  $e_1$  ou  $f_a$ ), le nouvel état est l'état  $(S \rightarrow \sim s', N')$

(respectivement  $(NIL, N')$ ) évoqué ci-dessus et nous allons voir que cet état est *unique* (3.3.1), qu'il existe (3.3.2) et comment le *calculer* (3.3.3).

### 3.3.1. Système déterministe

Comment l'attribut  $ETR_N$  a été défini de façon à donner un résultat unique sur toute dérivation  $S \rightarrow \sim s$ , et comme tout programme du langage a une unique dérivation à gauche à partir de l'axiome  $S$  [CPE] il existe au plus une transition  $e'$  à partir d'un état  $(S \rightarrow \sim s, N)$  : en ce sens, pour le langage étendu au parallélisme et au choix, le système de transitions est déterministe.

### 3.3.2. Système complet

Pour le langage étendu au parallélisme et au choix on a démontré [Hu 2] le résultat suivant : Pour toute étiquette  $e'$  et tout état  $(S \rightarrow \sim s, N)$ , il existe une transition partant de  $(S \rightarrow \sim s, N)$ .

Par définition un état mort est un état d'où ne part aucune transition : les seuls états morts sont donc les états de la forme  $(NIL, N)$  et le système de transitions est complet aux états morts près. (Ces états représentent la terminaison du processus contrôlé).

D'après les § 3.1 et 3.2 on en déduit l'algorithme ci-dessous de recherche du nouvel état évoqué au début du § 3 :

### 3.3.3. Algorithme de calcul des transitions

Étant donné l'état  $(S \rightarrow \sim s, N)$ , et l'étiquette  $e'$ ,

1° poser  $(S \rightarrow \sim s_0, N_0) = (S \rightarrow \sim s, N.e')$ ,

2° tant que  $ETR(S \rightarrow \sim s_i, N_i)$  est différent de  $(S \rightarrow \sim s_i, N_i)$  et de  $(NIL, N')$ , poser  $(S \rightarrow \sim s_{i+1}, N_{i+1}) = ETR(S \rightarrow \sim s_i, N_i)$ .

3° il existe  $k$  tel que  $ETR(S \rightarrow \sim s_k, N_k)$  est égal à  $(S \rightarrow \sim s_k, N_k)$  ou à  $(NIL, N')$ , et donc on a trouvé la transition  $(S \rightarrow \sim s, N) - e' \rightarrow ETR(S \rightarrow \sim s_k, N_k)$ .

### 3.3.4. Remarque concernant la sémantique d'un programme Electre

Est à construire seulement la partie accessible à partir de l'état initial  $(S \rightarrow \sim s, NIL)$ . Afin que le système de transitions ne soit pas inutilement grand, dans les exemples on pourra restreindre les transitions issues de l'état  $(S \rightarrow \sim s, N)$  d'une part aux seuls événements qui figurent dans  $\sim s$  et d'autre part à la fin d'un module figurant en tête de  $\sim s$ .

Par exemple, pour l'état

$$(S \rightarrow \sim s, NIL) = (S \rightarrow [a||b]/\{e_1 : c \odot e_1 | e_2 : d \odot e_2\}, NIL),$$

on ne regardera que les quatre transitions d'étiquettes  $e_1$ ,  $e_2$ ,  $f_a$ ,  $f_b$ . Sur occurrence de  $e_1$  on a la transition :

$$(S \rightarrow [a||b]/\{e_1 : c\odot e_1 | e_2 : d\odot e_2\}, \text{NIL}) - e_1 \rightarrow (S \rightarrow c\odot e_1, e_1),$$

et, pour l'état  $(S \rightarrow c\odot e_1, e_1)$ , on ne regardera que les transitions d'étiquettes  $e_1$ ,  $f_c$ .

#### 4. CONCLUSION ET PERSPECTIVES

Cette sémantique peut s'étendre sans difficultés majeures au langage incluant la préemption non nécessaire et des propriétés supplémentaires concernant les événements (on peut considérer six types selon la mémorisation avant ou (et) après la prise en compte et selon que l'on consomme la première occurrence ou toutes les occurrences) et les tâches-modules (pour un module en cours d'exécution on peut en interdire son interruption par toute occurrence d'une partie donnée de  $E$ ) [Hu 2].

Ainsi on dispose maintenant d'une base formelle sur laquelle on va désormais s'appuyer vers les extensions prévues et en cours suivantes : prouver des propriétés de tout programme Electre via un outil comme MEC [Ar], introduire le paramétrage temporel des tâches dans Electre et dériver du système de transitions les échéances calculées de l'application en fonction des instants d'occurrence des événements et ainsi prouver, à terme, le respect des contraintes temporelles des applications embarquées. Ces évolutions futures ne sont pas déconnectées des travaux effectués par les tenants des langages synchrones : une passerelle vers le code OC (commun à tous ces langages) est déjà en cours de réalisation, et par ailleurs une introduction à l'ambisynchronisme a également été effectuée qui intègre les approches synchrone et asynchrone dans une approche globale [Ri]. Une expérimentation en vraie grandeur est certainement la prochaine étape majeure de ce projet pour en valider la pertinence industrielle.

#### RÉFÉRENCES

- [Ar] A. ARNOLD, *MEC: A system for constructing and analysing transition systems*, LABRI, Université de Bordeaux-1, 1988.
- [BC] G. BERRY et L. COSSERAT, *The ESTEREL Synchronous Programming Language and its Mathematical Semantics*, Seminar on Concurrency, S. BROOKES and G. WINSKEL eds., Springer-Verlag, *Lecture Notes in Computer Science*, 1985, n° 197, p. 389-448.

- [BCo] G. BERRY, P. COURONNE et G. GONTHIER, *Synchronous Programming of reactive systems: an introduction to ESTEREL*, Rapport INRIA n° 646, 1986.
- [CH] R. H. CAMPBELL et N. HABERMAN, The specification of process synchronization by path expressions, *Lecture Notes in Computer Science*, 1973, n° 16, p. 89-102.
- [CL] D. CREUSOT, LEMOINE, ROUX et TRINQUET, *Un environnement d'exécution pour Electre*, contrat n° STR ELE1, janvier 1991.
- [CP] P. CASPI, D. PILAUD, N. HALBWACHS et J.A. PLAICE, *LUSTRE: a declarative language for programming synchronous systems*, 14th ACM Symposium on principles of programming languages, Munich 1987.
- [CPe] D. CREUSOT et J. PERRAUD, *Grammaire et analyse syntaxique du langage Electre*, Rapport de contrat VEH-ELE-D2, Convention Renault, 1988.
- [CR] F. CASSEZ et O. ROUX, *Compilation du langage Electre*, Rapport interne n° 91-11, LAN, École Centrale de Nantes.
- [ER] J.-P. ELLOY et O. ROUX, Electre; a Language for Control Structuring in Real Time, *The Computer Journal*, 1985, 28, n° 5, p. 229-234.
- [GB] P. LE GUERNIC, A. BENVENISTE, P. BOURNAI et T. GAUTIER, SIGNAL: a data-flow oriented language for signal processing, *IEEE Trans. on ASSP*, ASSP-34, 1986, 2, p. 362-374.
- [Hu] M. HUOU, Contribution à la sémantique du langage Electre, Thèse de Doctorat de l'Université de Nantes et de l'École Centrale de Nantes, 1991.
- [Hu2] M. HUOU, *Une sémantique étendue du langage Electre*, Rapport interne n° 92-17, LAN, École Centrale de Nantes, 1992.
- [Kn] D. E. KNUTH, Semantics of Context-Free Languages, *Mathematical Systems Theory*, 1988, 2, n° 2, p. 127-145; *Mathematical Systems Theory*, 1971, 5, n° 1, p. 95-96, Correction.
- [KB] D. E. KNUTH et P. BENDIX, Simple word problems in universal algebras in J. LEECH, ed., *Computational problems in abstract algebra*.
- [PR] J. PERRAUD, O. ROUX et M. HUOU, Opérationnel semantics of a kernel of the langage Electre, *Theoretical Computer Science*, in volume 97, 1992.
- [Pl] G. D. PLOTKIN, *A Structural Approach to Operational Semantics*, Lecture Notes, Computer Science Department, Aarhus University, 1981.
- [Ri] M. RICHARD, Étude de la conjonction des approches synchrone et asynchrone dans les langages réactifs : Application à Electre, Thèse de Doctorat de l'Université de Nantes et de l'École Centrale de Nantes, 1992.
- [RC] O. ROUX, F. CASSEZ, D. CREUSOT et J.-P. ELLOY, Le langage réactif asynchrone Electre, *Technique et Science Informatiques*, 1992, vol. 11, n° 5, p. 35-66.