Introduction au langage LOTOS

Hubert GARAVEL

VERILOG Rhône-Alpes
VERILOG Centre d'Etudes Rhône-Alpes
Forum du Pré Milliet
MONTBONNOT
38330 SAINT-ISMIER
tel: +(33) 76 52 18 36 fax: +(33) 76 52 12 39

1 Introduction

Un consensus s'est établi sur la manière d'utiliser l'informatique pour résoudre un problème : "*spécifier, puis valider, enfin implémenter*". Bien souvent, cependant, la mise en œ uvre de ces excellents principes se heurte à des difficultés pratiques. S'il est possible, à la rigueur, de se dispenser de cette démarche pour de petits programmes, elle s'avère indispensable dans le domaine des protocoles, des automatismes industriels et des systèmes temps-réel, à cause de la complexité souvent combinatoire de ces applications et des exigences de fiabilité auxquelles elles sont soumises. En faisant l'impasse sur les phases de spécification et de vérification, on court le risque de se retrouver devant une implémentation qui ne fonctionne pas correctement — ou pire, seulement de temps en temps —, que ses développeurs ne parviennent plus à maîtriser, qui est difficile à maintenir et à faire évoluer car mal documentée, et qui en définitive aura coûté fort cher.

On peut être convaincu des vertus de la spécification et hésiter devant le choix du formalisme à adopter. La question n'est pas simple : un bon langage de spécification doit fournir à la pensée des représentations mentales simples, intuitives, précises et cohérentes pour des problèmes difficiles qui font appel aux notions de temps (concurrence, simultanéité, asynchronisme, synchronisme, ...) et d'espace (répartition, communication, ...). En outre il faut que ce langage puisse être supporté par des outils de génie logiciel : la démarche de spécification doit être assistée par l'emploi d'interpréteurs, de compilateurs, de débogueurs et d'outils de vérification.

2 Le langage LOTOS

LOTOS[1] est un langage permettant de spécifier l'architecture et le fonctionnement de systèmes distribués. La définition formelle de LOTOS fait l'objet de la norme ISO 8807 [ISO88]. Il existe plusieurs présentations "pédagogiques" de LOTOS, par exemple [BB88] et [Gar89a, annexes A et B].

Historiquement, LOTOS a été défini pour permettre la description des services et des protocoles de télécommunications, en particulier pour les systèmes OSI[2]. En effet les organismes de normalisation, confrontés aux problèmes posés par l'emploi du langage naturel (même complété par des tables d'états) dans les normes [LS88] , ont recherché des moyens d'expression mieux adapté à leurs besoins. C'est ainsi

[1] Langage Of Temporal Ordering Specification
[2] Open Systems Interconnection

que l'ISO[3] et le CCITT[4] ont normalisé trois langages de description formelle : LOTOS, ESTELLE[5] et SDL[6]. En fait leur domaine d'application s'étend bien au-delà du cadre des protocoles OSI. Plusieurs raisons poussent un nombre toujours croissant d'utilisateurs à adopter ces langages :

- LOTOS, ESTELLE et SDL possèdent un pouvoir d'expression suffisant pour les applications réelles, car ils ont été conçus pour décrire des systèmes complexes, par des experts qui sont quotidiennement confrontés à ces questions. Le risque est donc faible de se trouver bloqué par des problèmes que l'on ne peut pas exprimer

- leur définition (syntaxe, sémantique statique et sémantique dynamique) est suffisamment rigoureuse pour avoir franchi avec succès les étapes difficiles imposées par les comités internationaux de normalisation. En particulier leur sémantique dynamique est décrite formellement, contrairement à beaucoup de "méthodologies" pour la spécification de systèmes, dont la sémantique n'est pas toujours très claire. Ceci supprime les risques d'incompréhension ou d'ambiguïté, en particulier entre ceux qui spécifient et ceux qui implémentent

- ces langages sont acceptés et utilisés par une communauté particulièrement dynamique. Des conférences internationales leur sont consacrées chaque année (*FORTE Formal Description Techniques*, *IFIP WG 6.1 Protocol Specification, Testing, and Verification*). La CEE apporte son soutien actif par le biais des programmes communautaires ESPRIT (projets SEDOS, SEDOS-Demo, PANGLOSS) et RACE (projets BEST et SPECS)

- on assiste actuellement à un regroupement autour des standards qui évite la dispersion des efforts. Les utilisateurs peuvent donc profiter des progrès techniques, tout en étant assurés de la pérennité de leurs investissements, puisque l'évolution des normes se fait de manière très contrôlée

- il existe des outils de génie logiciel capables de seconder les utilisateurs dans leur travail. En choisissant un langage normalisé, ceux-ci bénéficient d'une offre multi-vendeurs[7] qui leur évite de se lier trop étroitement à un formalisme "propriétaire"

Contrairement à ESTELLE et SDL dont les concepts traditionnels semblent familiers, LOTOS est souvent perçu — en France surtout — comme un langage "futuriste", ce qui suffit au moins à lui assurer une place dans cette revue. Toutefois cette réputation est largement injustifiée, comme cette présentation entend le montrer.

3 Les spécifications LOTOS

De prime abord, une spécification LOTOS est un texte ASCII qui regroupe un ensemble de définitions de processus (encadrées par les mots-clés "**process**" et "**endproc**") et de définitions de types (délimitées par les mots-clés "**type**" et "**endtype**").

LOTOS possède une structure de blocs imbriqués : chaque définition de processus peut contenir des définitions de processus ou de types qui lui sont locales ; en revanche une définition de type ne peut pas englober d'autres définitions de types ni de processus. Au plus haut niveau, une spécification LOTOS se comporte comme une définition de processus, bien que la syntaxe soit légèrement différente de celle des processus ordinaires (les mots-clés "**specification**" et "**endspec**" sont utilisés).

Les définitions de processus décrivent le contrôle et les définitions de types décrivent les données. Cette dichotomie qui apparaît au niveau syntaxique rend bien compte du fait que le langage LOTOS se com-

[3] Organisation Internationale de Normalisation
[4] Comité Consultatif International pour la Téléphonie et la Télégraphie
[5] Extended Finite State Machine Language
[6] Specification and Description Language
[7] il serait regrettable de ne pas mentionner ici les outils GEODE et VEDA pour SDL et ESTELLE, développés et diffusés par la société VERILOG

pose de deux parties tout à fait orthogonales qui seront, pour cette raison, présentées séparément : les structures de contrôle et les structures de données.

Les identificateurs utilisés dans une spécification LOTOS se répartissent en six classes (processus, portes, variables, types, sortes, opérations) qui seront définies tour à tour. Dans tous les exemples, les mots-clés seront écrits en minuscules et les identificateurs en majuscules.

Pour rendre cette introduction à LOTOS plus accessible, les exemples ont été délibérement choisis en dehors du domaine des protocoles ; on trouvera dans la littérature de nombreuses applications de LOTOS aux problèmes des télécommunications.

4 La partie contrôle de LOTOS

Les structures de contrôle dont dispose LOTOS sont issues des recherches sur les *algèbres de processus*, inspirées notamment par les travaux de Hoare sur CSP [Hoa78] et Milner sur CCS [Mil80].

Syntaxiquement parlant, le contrôle est décrit en LOTOS par les termes d'une algèbre : ce sont des expressions mathématiques, appelées *comportements*, construites à partir des opérateurs de contrôle qui sont fournis par LOTOS.

Sémantiquement parlant, chaque comportement représente un automate d'états finis ou infinis, les règles de traduction des comportements en automates constituant la sémantique dynamique de LOTOS.

4.1 Portes et signaux

Plus concrètement, LOTOS permet de décrire des comportements qui s'exécutent en parallèle et qui correspondent par rendez-vous. En LOTOS, le rendez-vous est le moyen unique pour exprimer la synchronisation et la communication.

On appelle *signal* ou *action* la proposition effectuée par un comportement qui désire participer à un rendez-vous. Un signal se compose d'une *porte* et une liste d'*offres* pour l'émission ou la réception de valeurs typées (dans la terminologie ISO pour LOTOS, le type d'une valeur ou d'une variable porte le nom de *sorte*). Par exemple, le signal :

```
OUTPUT !2 !X1 !X2
```

signifie que l'on cherche à émettre simultanément, sur la porte `OUTPUT`, les trois valeurs `2`, `X1` et `X2`. De même, le signal :

```
INPUT ?A:REAL ?B:REAL ?C:REAL
```

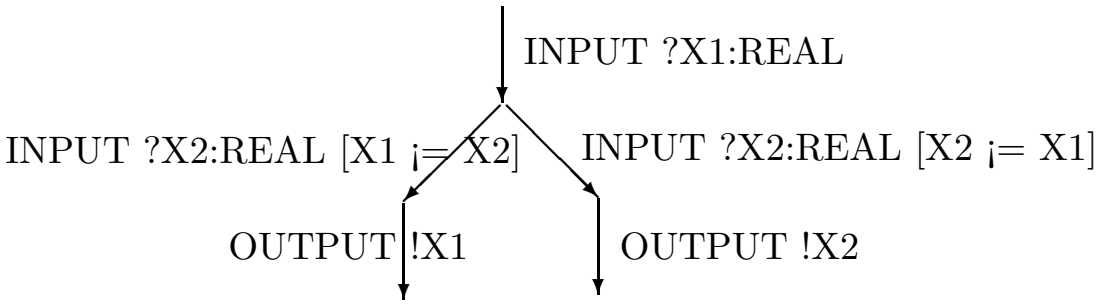
indique que l'on s'attend à recevoir simultanément, sur la porte `INPUT`, trois valeurs réelles qui seront rangées dans les variables `A`, `B` et `C`. Enfin on peut combiner émissions et réceptions au cours d'un même rendez-vous et spécifier des conditions sur les valeurs émises ou reçues. C'est ainsi que le signal :

```
EXCHANGE !2 ?X1:REAL ?X2:REAL [X1 < X2]
```

exprime que, sur la porte `EXCHANGE`, on désire émettre la valeur `2` tout en recevant deux valeurs réelles `X1` et `X2`. En outre le rendez-vous est conditionné par une garde booléenne : il ne sera accepté que si la valeur `X1` est plus petite que la valeur `X2`.

4.2 Opérateurs séquentiels

Tout programme LOTOS décrit un automate dont chaque transition correspond à un signal ; cet automate peut aussi être considéré comme un graphe orienté dont chaque arc est étiqueté par un signal. Ce modèle est bien adapté aux besoins de spécification pour les systèmes répartis asynchrones. Voici un exemple assez simple d'automate qui, après avoir reçu deux valeurs réelles `X1` et `X2` sur la porte `INPUT`, émet le plus petit de ces deux nombres sur la porte `OUTPUT` :



Si les valeurs `X1` et `X2` s'avèrent être égales, on est en présence d'un choix non-déterministe : l'environnement laisse le système décider librement entre les deux évolutions possibles (dans cet exemple, le fait de choisir une évolution plutôt que l'autre ne change pas le résultat).

Pour spécifier un tel automate dans un langage de programmation, il existe classiquement deux approches : soit l'on énumère la liste des états et des transitions, soit l'on décrit l'automate par une expression régulière, procédé bien connu des utilisateurs du système UNIX dont les langages de commandes et les éditeurs de texte font un emploi intensif.

LOTOS s'inspire de la seconde solution puisqu'il possède deux opérateurs binaires "`;`" et "`[]`"[8] qui expriment respectivement la composition séquentielle et le choix non-déterministe et qui sont analogues aux opérateurs "`.`" et "`|`" des expressions régulières, à ceci près qu'en LOTOS tout opérande figurant en partie gauche de "`;`" doit obligatoirement être un signal et que l'arrêt doit être indiqué explicitement par l'opérateur nullaire "**stop**". C'est ainsi que le comportement de l'automate ci-dessus peut être décrit en LOTOS, de manière simple et concise :

```
INPUT ?X1:REAL;
   (
   INPUT ?X2:REAL [X1 <= X2];
      OUTPUT !X1;
         stop
   []
   INPUT ?X2:REAL [X2 <= X1];
      OUTPUT !X2;
          stop
   )
```

Pour exprimer les comportements cycliques, LOTOS ne dispose pas de l'opérateur d'itération "`*`" des expressions régulières, mais il possède un mécanisme plus général d'appel de processus récursif qui sera détaillé ci-après.

4.3 Opérateurs sur les valeurs

Il existe plusieurs opérateurs LOTOS permettant de manipuler les valeurs et de créer des *variables*.

L'opérateur "`[...] ->` ..." permet de conditionner l'exécution d'un comportement par une garde booléenne ; il s'apparente à une instruction "**if** ... **then** ...".

L'opérateur "**let** ... **in** ..." permet de donner un nom à une expression en définissant une variable. Toutefois, les règles de syntaxe et de sémantique statique font qu'il est impossible de modifier la valeur d'une variable : LOTOS est un langage fonctionnel, sans affectation, ce qui élimine du même coup toutes les difficultés posées par l'existence de variables partagées.

L'opérateur "**choice** ... `[]` ..." permet de choisir, de manière non-déterministe, une valeur dans un domaine et de la nommer en définissant une variable. On peut l'utiliser, entre autres, pour spécifier que la valeur exacte d'un résultat renvoyé n'est pas significative.

[8] "`[]`" se veut la représentation ASCII de la fameuse "barre carrée" proposée par Dijkstra

L'exemple suivant, consacré au calcul des racines d'un polynôme du 2nd degré, illustre l'emploi de ces opérateurs :

```
INPUT ?A:REAL ?B:REAL ?C:REAL;
   (
   let DELTA:REAL = (B ^ 2) - (4 * A * C) in
      (
      [DELTA > 0] ->
         (
         let X1:REAL = (-B - sqrt (DELTA)) / (2 * A) in
            let X2:REAL = (-B + sqrt (DELTA)) / (2 * A) in
              OUTPUT !2 !X1 !X2;
                 stop
         )
      []
      [DELTA = 0] ->
         (
         let X0:REAL = -B / (2 * A) in
            OUTPUT !1 !X0 !X0;
               stop
         )
      []
      [DELTA < 0] ->
         (
         choice X1, X2:REAL []
           OUTPUT !0 !X1 !X2;
              stop
         )
      )
   )
```

4.4 Processus

Il est possible de donner un nom à un fragment de programme LOTOS en définissant un *processus*, de la même manière que les langages algorithmiques permettent de définir des procédures pour regrouper un ensemble d'instructions. La définition d'un processus est paramétrée par la liste (entre crochets) des portes sur lesquelles le processus peut effectuer des rendez-vous. On peut ainsi définir un processus `ROOT` pour encapsuler le fragment de programme défini ci-dessus (la signification du mot-clé "**noexit**" sera expliquée plus loin) :

```
process ROOT [INPUT, OUTPUT] : noexit :=
   INPUT ?A:REAL ?B:REAL ?C:REAL;
      ...
endproc
```

L'appel d'un processus se fait tout naturellement en donnant le nom du processus et la liste de portes passées en paramètres effectifs. Les appels récursifs sont permis et, grâce à eux, on peut exprimer les comportements cycliques. Par exemple le processus ci-dessous acquiert deux valeurs réelles sur la porte `INPUT` et renvoie leur minimum sur la porte `OUTPUT`, après quoi il recommence indéfiniment ; il suffit de reprendre le comportement défini plus haut en remplaçant "**stop**" par un appel récursif pour indiquer qu'après l'émission du résultat sur la porte "`OUTPUT`" le comportement ne s'arrête pas, mais repart dans l'état où il était initialement :

```
process MIN [INPUT, OUTPUT] : noexit :=
   INPUT ?X1, X2:REAL;
      (
      [X1 <= X2] ->
         OUTPUT !X1;
            MIN [INPUT, OUTPUT]
      []
      [X2 <= X1] ->
         OUTPUT !X2;
            MIN [INPUT, OUTPUT]
      )
endproc
```

Une définition de processus LOTOS peut également être paramétrée par une liste (entre parenthèses) de variables formelles. En pratique ces paramètres ont deux utilisations : il peut s'agir soit d'informations constantes propres au processus considéré (par exemple, un numéro unique attribué à chaque tâche parallèle du système), soit de variables d'état, locales au processus, dont la valeur peut être mise à jour au moyen d'appels récursifs.

L'exemple suivant illustre ce second point. Il décrit le fonctionnement d'un chronomètre simplifié[9]. Il s'agit d'un processus `CHRONO` possédant deux variables paramètres formels et réagissant à quatre signaux : la variable booléenne `EN_MARCHE` mémorise si l'on est en train de chronométrer ou non ; la variable entière `TEMPS` comptabilise le nombre de centièmes de secondes qui se écoulés depuis le début du chronométrage ; le signal `AFF` permet d'afficher le temps chronométré ; le signal `M_A` met en marche ou arrête le chronométrage ; le signal `RAZ` remet à zéro le temps chronométré ; le signal `C_SEC` indique que le temps vient de progresser d'un centième de seconde. L'appel initial de ce processus est :

```
CHRONO [AFF, M_A, RAZ, C_SEC] (false, 0)
```

et sa définition est la suivante :

```
process CHRONO [AFF, M_A, RAZ, C_SEC] (EN_MARCHE:BOOL, TEMPS:NAT) : noexit :=
   AFF !TEMPS;
      CHRONO [AFF, M_A, RAZ, C_SEC] (EN_MARCHE, TEMPS)
   []
   M_A;
      CHRONO [AFF, M_A, RAZ, C_SEC] (not (EN_MARCHE), TEMPS)
   []
   RAZ;
      CHRONO [AFF, M_A, RAZ, C_SEC] (EN_MARCHE, 0)
   []
   C_SEC;
      (
      [EN_MARCHE = true] ->
         CHRONO [AFF, M_A, RAZ, C_SEC] (EN_MARCHE, TEMPS + 1)
      []
      [EN_MARCHE = false] ->
         CHRONO [AFF, M_A, RAZ, C_SEC] (EN_MARCHE, TEMPS)
      )
endproc
```

LOTOS possède un opérateur "**exit**" qui permet à un comportement de se terminer en renvoyer comme résultat une liste de valeurs. Les sortes de ces valeurs doivent être déclarés dans l'en-tête du processus après le mot-clé "**exit**" (le mot-clé "**noexit**" signifiant que le processus ne retourne aucun résultat). Dans certains cas, l'utilisation conjointe des paramètres variables et des résultats permet d'avoir un style de programmation applicatif, qui ne fait pas intervenir explicitement les notions de portes, de signaux et de rendez-vous. Par exemple le processus LOTOS ci-dessous calcule le minimum et le maximum des deux nombres qui lui sont passés en paramètres :

[9] on trouvera également une description en LUSTRE de ce même exemple dans l'article de N. Halbwachs et D. Pilaud

```
process MIN_MAX (X:REAL, Y:REAL) : exit (REAL, REAL) :=
   [X <= Y] ->
      exit (X, Y)
   []
   [Y <= X] ->
      exit (Y, X)
endproc
```

Il est possible de récupérer les résultats renvoyés par un processus grâce à l'opérateur de composition séquentielle "... `>>` **accept** ... **in** ...". Cet opérateur réalise ce qui correspondrait, dans un langage algorithmique, à un appel de sous-programme avec passage de paramètres par valeur et par résultat (la transmission des résultats se faisant au moyen d'un rendez-vous implicite). C'est ainsi que l'on pourrait redéfinir le processus `MIN` en fonction du processus `MIN_MAX` :

```
process MIN [INPUT, OUTPUT] : noexit :=
   INPUT ?X1, X2:REAL;
      MIN_MAX (X1, X2) >> accept Y1:REAL, Y2:REAL in
         OUTPUT !Y1;
            MIN [INPUT, OUTPUT]
endproc
```

Il existe d'autres opérateurs séquentiels qui ne seront pas présentés ici, notamment un opérateur d'interruption, qui se note "`[>`".

4.5 Opérateurs parallèles

Ce qu'on a vu jusqu'à présent de LOTOS permet de décrire des programmes séquentiels, dans un style fonctionnel certes, mais relativement proche de la programmation algorithmique classique. Mais cela seul ne suffirait pas à faire de LOTOS un langage adapté aux systèmes répartis.

Dans beaucoup de langages couramment utilisés, on constate que les aspects parallèles ne sont pas appréhendés dans leur généralité, mais plutôt considérés comme des "extensions" apportées au cadre séquentiel, ce qui conduit à des demi-solutions dont la sémantique, camouflée sous une syntaxe plus ou moins baroque, n'est pas toujours très claire.

La solution adoptée par LOTOS — et plus généralement dans les algèbres de processus depuis CCS — séduit au contraire par sa rigueur et son élégance. Le principe est simple : comme les aspects séquentiels, les aspects parallèles s'expriment par des opérateurs. Cette idée, qui peut sembler novatrice au regard des langages de programmation existants, est pourtant familière aux utilisateurs d'UNIX qui ont recours quotidiennement à ces notions, puisque le *shell* leur offre, outre la composition séquentielle "`;`", deux opérateurs de composition parallèle "`&`" et "`|`".

En LOTOS l'opérateur équivalent à l'opérateur "`&`" du shell se note "`|||`" : il exprime l'exécution simultanée de deux comportements sans aucune synchronisation entre eux (sauf en ce qui concerne la terminaison par "**exit**" : le processus qui se termine le premier attend l'autre). Comme il n'existe aucune variable partagée entre les comportements qui s'exécutent en parallèle, on évite les problèmes liés à l'emploi de "`&`" (par exemple lorsque deux processus UNIX concurrents écrivent dans le même fichier).

Pour permettre aux comportements concurrents de se synchroniser et de communiquer, LOTOS possède un opérateur noté "`|[`$G_1, \ldots\ G_n$`]|`", qui généralise l'opérateur "`|`" du shell puisqu'il autorise n canaux de communication au lieu d'un seul. L'opérateur "`|[`$G_1, \ldots\ G_n$`]|`" indique que les deux comportements auxquels il s'applique doivent fonctionner en parallèle, tout en se synchronisant par rendez-vous sur les portes $G_1, \ldots\ G_n$ et sur ces portes-ci exclusivement (en outre la terminaison par "**exit**" est toujours synchrone). Par exemple la commande shell suivante :

```
(P1 & P2) ; (P3 | P4)
```

qui lance l'exécution simultanée des processus `P1` et `P2` et, lorsqu'ils ont terminé, exécute les processus `P3` et `P4` en parallèle, la sortie de `P3` servant d'entrée à `P4`, a comme équivalent en LOTOS :

```
(P1 ||| P2) >> (P3 [G] |[G]| P4 [G])
```

où l'opérateur "`>>`" exprime la composition séquentielle et où l'opérateur "`|[G]|`" explicite la communication entre `P3` et `P4` par l'emploi d'une porte `G`.

L'opérateur "`|[`$G_1, \ldots\ G_m$`]|`" est très général puisqu'il combine le synchronisme (sur les portes $G_1, \ldots\ G_n$) et l'asynchronisme (sur toutes les autres portes). En fait l'opérateur complètement asynchrone "`|||`" (ainsi que l'opérateur complètement synchrone "`||`" qui existe aussi en LOTOS) sont définis comme des cas particuliers de cet opérateur.

Les opérateurs parallèles de LOTOS offrent un moyen naturel pour décrire l'architecture d'un ensemble de processus communicants. On peut facilement exprimer les topologies classiques des réseaux, par exemple, le schéma *producteur-consommateur* dans lequel un processus producteur envoie des informations à un processus consommateur via un processus tampon :

```
PRODUCER [G1]
|[G1]|
BUFFER [G1, G2]
|[G2]|
CONSUMER [G2]
```

de même que le schéma *client-serveur*, où plusieurs processus clients sont en concurrence pour l'accès à une ressource fournie par un processus serveur :

```
(
CLIENT [G]
|||
CLIENT [G]
||| ... |||
CLIENT [G]
)
|[G]|
SERVER [G]
```

de même que le *réseau en étoile*, où plusieurs processus sites communiquent par l'intermédiaire d'un processus central :

```
(
SITE [G1]
|||
SITE [G2]
||| ... |||
SITE [Gn]
)
|[G1, G2, ..., Gn]|
NODE [G1, G2, ..., Gn]
```

de même que le *réseau en anneau*, où chaque processus site ne peut communiquer qu'avec ses voisins immédiats :

```
(
SITE [G0, G1]
|[G1]|
SITE [G1, G2]
|[G2]| ... |[Gn-1]|
SITE [Gn-1, Gn]
)
|[Gn, G0]|
SITE [Gn, G0]
```

La sémantique du parallélisme en LOTOS peut être résumée comme suit. Si, à un instant donné, deux comportements qui s'exécutent en parallèle proposent chacun un signal, deux cas sont à considérer, selon qu'un rendez-vous est possible ou non entre ces deux signaux :

- si oui, les deux signaux peuvent être exécutés en même temps, ce qui permet aux deux comportements de s'attendre et de s'échanger des valeurs
- si non, les deux signaux ne peuvent avoir lieu simultanément. Ils doivent être ordonnés dans le temps : l'un d'eux est choisi, de manière non-déterministe, pour être exécuté avant l'autre. Il n'y a alors ni synchronisation ni communication entre les deux comportements. On retrouve ici la notion d'*entrelacement* dont l'origine remonte aux systèmes d'exploitation en temps partagé

Un rendez-vous entre deux signaux n'est possible que si les conditions suivantes sont réunies :

- les deux signaux ont la même porte
- l'opérateur parallèle qui compose les deux comportements autorise le rendez-vous sur cette porte
- les deux signaux ont le même nombre d'offres
- les offres des deux signaux sont deux à deux compatibles, étant donné que :
  - une offre "`!V`" est compatible avec une offre "`?X:S`" si la valeur `V` a pour sorte `S`. Si le rendez-vous a lieu, alors la variable `X` prendra la valeur `V`
  - une offre "`?X:S`" est compatible avec une offre "`!V`" si la valeur `V` a pour sorte `S`. Si le rendez-vous a lieu, alors la variable `X` prendra la valeur `V`
  - une offre "`!V1`" est compatible avec une offre "`!V2`" si les valeurs `V1` et `V2` ont la même sorte et sont égales
  - une offre "`?X1:S1`" est compatible avec une offre "`?X2:S2`" si les sortes `S1` et `S2` sont identiques. Si le rendez-vous a lieu, les variables `X1` et `X2` prendront une même valeur `V`, de sorte `S`, choisie de manière non-déterministe
- les gardes booléennes associées aux signaux sont satisfaites (ces gardes peuvent porter sur les valeurs échangées au moment du rendez-vous)

LOTOS permet aussi le rendez-vous n-aire, c'est-à-dire la possibilité d'avoir plus de deux sites qui participent à un même rendez-vous. La sémantique du rendez-vous n-aire se déduit de celle du rendez-vous binaire par associativité et commutativité des opérateurs parallèles. On peut alors avoir des échanges multi-directionnels de valeurs très sophistiqués, la diffusion d'une valeur par un site vers tous les autres sites n'étant qu'un cas particulier du mécanisme général.

Enfin il existe un opérateur "**hide** ... **in** ..." qui permet de rendre "invisibles" certains signaux et que l'on utilise comme moyen d'abstraction, afin de cacher les rendez-vous effectués dans certaines parties du système.

5 La partie données de LOTOS

Les structures de données de LOTOS sont issues des travaux sur les *types abstraits algébriques* et, en particulier, sur le langage ACT-ONE [EM85].

Elles s'articulent autour du concept de *type*, ce qui correspond à la notion usuelle de module, classe ou objet dans les autres langages. Un type LOTOS regroupe des *sortes* qui sont des noms donnés à des domaines de valeurs (les sortes sont analogues aux types dans les langages algorithmiques classiques), des *opérations* qui sont des fonctions définies sur ces sortes et des *équations* qui caractérisent la sémantique des opérations au moyen d'égalités mathématiques.

L'exemple ci-dessous, tiré de la bibliothèque des types LOTOS prédéfinis, concerne le type `BOOLEAN`. La syntaxe LOTOS pour la déclaration des types fait apparaître trois sections, introduites respectivement par les mots-clés "**sorts**", "**opns**" et "**eqns**". Ceci reflète bien le fait qu'ici le type `BOOLEAN` encapsule une sorte `BOOL`, un ensemble d'opérations (`FALSE`, `TRUE`, `NOT`, ...) dont on précise les sortes des opérandes

et du résultat, et un ensemble d'équations (le mot-clé "**forall**" introduit des variables quantifiées universellement et le mot-clé "**ofsort**" indique de quelle sorte sont les membres gauche et droit des équations qu'il précède) :

```
type BOOLEAN is
   sorts
      BOOL
   opns
      FALSE, TRUE : -> BOOL
      NOT : BOOL -> BOOL
      _AND_, _OR_, _XOR_, _IMPLIES_, _IFF_ : BOOL, BOOL -> BOOL
   eqns
      forall X, Y : BOOL
      ofsort BOOL
         NOT (TRUE)   =  FALSE;
         NOT (FALSE)  =  TRUE;
         X AND TRUE   =  X;
         X AND FALSE  =  FALSE;
         X OR TRUE    =  TRUE;
         X OR FALSE   =  X;
         X XOR Y      =  (X AND NOT (Y)) OR (Y AND NOT (X));
         X IMPLIES Y  =  Y OR NOT (X);
         X IFF Y      =  (X IMPLIES Y) AND (Y IMPLIES X);
endtype
```

Les possibilités offertes par LOTOS sont riches :

- il existe une bibliothèque normalisée de types de base que l'on peut importer
- l'utilisateur peut définir ses propres notations d'opérations (par exemple "`%`", "`**`", "`//`", "`<=>`", ...) et spécifier qu'une opération binaire s'utilise de manière infixée et non pas préfixée (ainsi les opérations `AND`, `OR`, ... de l'exemple précédent)
- la surcharge d'opérateurs est permise : on la possibilité de définir des opérateurs ayant même nom et des profils différents. La résolution des surcharges se fait de manière contextuelle ; en cas d'ambiguïté, on a recours aux informations fournies par la structure de blocs
- les équations conditionnelles sont autorisées : toute équation peut être gardée par une liste de prémisses
- LOTOS possède la notion d'héritage multiple : un type peut importer les sortes, les opérations et les équations définies dans un ou plusieurs autres types. Il n'y a pas de conflit lorsqu'une même sorte ou une même opération est importée par deux chemins différents : les règles de sémantique statique de LOTOS permettent de reconnaître et de traiter ces situations. De même la possibilité d'avoir des surcharges fait qu'il est possible d'importer sans conflit deux opérations de même nom définies dans des types différents
- l'enrichissement des types est possible : il suffit de définir un nouveau type qui en importe d'autres et qui les complète en leur ajoutant des sortes, des opérations et des équations
- LOTOS autorise le renommage des types, grâce auquel on peut créer un nouveau type en fonction d'un type existant, simplement en renommant les sortes et les opérations de ce dernier. Ceci permet, d'une part, de réutiliser des types déjà construits en changeant leurs notations et, d'autre part, de pouvoir définir des sortes dérivées[10] à partir de sortes existantes. Par exemple on peut définir le type `BINARY` des entiers modulo 2 en fonction du type `BOOLEAN` : les valeurs de sorte `BIN` sont alors isomorphes — moyennant le renommage des opérateurs — aux valeurs de sorte `BOOL`, mais elles ne sont pas compatibles avec elles :

[10] au sens des types dérivés du langage ADA

```
type BINARY is BOOLEAN renamedby
   sortnames
      BIN for BOOL
   opnnames
      0 for FALSE
      1 for TRUE
      . for AND
      <+> for XOR
endtype
```

- LOTOS possède un concept de généricité très puissant, puisqu'il permet de définir des types paramétrés par des sortes formelles, des opérations formelles. On peut même spécifier, grâce à des équations formelles, des contraintes sur les sortes et les opérations formelles. L'exemple suivant décrit une file d'attente à priorités, dont les éléments sont d'une sorte formelle `ITEM`, pour laquelle il existe une opération formelle "`<<`" dont le profil est celui d'une relation binaire et qui satisfait les équations formelles caractérisant une relation d'ordre :

```
type ITEM_QUEUE is BOOLEAN
   formalsorts
      ITEM
   formalopns
      _<<_ : ITEM, ITEM -> BOOL
   formaleqns
      forall X, Y, Z:ITEM
         ofsort BOOL
            X << X = TRUE;
         ofsort ITEM
            (X << Y) AND (Y << X) => X = Y;
         ofsort BOOL
            (X << Y) AND (Y << Z) => X << Z = TRUE;
   sorts
      QUEUE
   opns
      {} : -> QUEUE
      _._ : ITEM, QUEUE -> QUEUE
      EMPTY : QUEUE -> BOOL
      TOP : QUEUE -> ITEM
      GET : QUEUE -> QUEUE
      PUT : ITEM, QUEUE -> QUEUE
   eqns
      forall X, Y:ITEM, Q:QUEUE
         ofsort BOOL
            EMPTY ({}) = TRUE;
            EMPTY (X . Q) = FALSE;
         ofsort ITEM
            TOP (X . Q) = X;
         ofsort QUEUE
            GET (X . Q) = Q;
         ofsort QUEUE
            PUT (X, {}) = X . {};
            X << Y => PUT (X, Y . Q) = X . (Y . Q);
            not (X << Y) => PUT (X, Y . Q) = Y . PUT (X, Q);
endtype
```

Il est ensuite possible de créer des exemplaires instanciés de ce type générique en substituant aux sortes et aux opérations formelles des sortes et des opérations effectives. LOTOS autorise l'instanciation partielle : on peut avoir des types semi-instanciés, dans lesquels certaines sortes et certaines opérations demeurent formelles. Pour obtenir, à partir du type générique `ITEM_QUEUE`

défini précédemment, une file d'attente dont les éléments sont des nombres réels, il suffit d'instancier respectivement la sorte formelle `ITEM` et l'opération formelle "`<<`" par la sorte effective `REAL` et l'opération effective "`<=`" :

```
type REAL_QUEUE is ITEM_QUEUE actualizedby REAL_NUMBER using
   sortnames
      REAL for ITEM
   opnnames
      <= for <<
endtype
```

6 Conclusion

Cette présentation s'achève par une rapide évaluation de LOTOS et de son adéquation comme langage de spécification ; elle se replace ensuite dans le cadre plus vaste des méthodes de génie logiciel en donnant quelques indications sur la manière d'interpréter, de compiler et de vérifier formellement les spécifications LOTOS.

6.1 Spécification

Il semble que la syntaxe ésotérique, bien que concise, de LOTOS constitue un écueil pour certains utilisateurs qui souhaiteraient plutôt s'exprimer dans une syntaxe graphique comparable à celle de SDL. Plusieurs prototypes ont été développés et les comités de normalisation se penchent actuellement sur la question.

Pour exprimer le contrôle, LOTOS offre des constructions de haut niveau possédant une sémantique formelle particulièrement simple et élégante (la sémantique dynamique de LOTOS se résume en une quinzaine de règles). Le mécanisme du rendez-vous en LOTOS est suffisamment général pour permettre de retrouver les autres modes de synchronisation et de communication (sémaphores, moniteurs, files d'attente, ...)

En dépit d'une syntaxe cryptique, les concepts de LOTOS ont un caractère très intuitif que cette présentation a tenté de mettre en lumière ; il serait toutefois excessif de prétendre que l'on peut maîtriser LOTOS en faisant l'économie d'une réflexion approfondie sur le parallélisme. Il s'agit là d'un exercice salutaire : LOTOS apparaît à beaucoup comme un véritable outil de pensée qui permet de se concentrer sur le problème à résoudre, sans se perdre dans les détails d'implémentation.

Il faut cependant être conscient que LOTOS n'est pas "le" langage universel. Comme SDL et ESTELLE, c'est essentiellement un langage asynchrone[11], bien qu'il possède certaines caractéristiques typiquement synchrones, par exemple la diffusion. Il ne permet pas de décrire les aspects quantitatifs (performances, temps de réponse) d'un système, ni les problèmes de temporisation : on peut simuler l'expiration d'un délai par une transition interne spontanée, mais cela implique que le fonctionnement du système doit rester correct quelle que soit les valeurs choisies pour les délais ; plusieurs propositions d'extension ont été faites pour introduire la notion de délai dans LOTOS.

En ce qui concerne la partie données, le choix des types abstraits, de préférence à des solutions plus classiques, ne va pas sans réticences. Les membres de l'ISO ont sans doute estimé que l'approche types abstraits convenait mieux à un langage de spécification[12] parce qu'elle n'impose pas de contraintes indues aux implémenteurs et parce qu'en obligeant celui qui spécifie à définir complètement les propriétés des données, on évite les ambiguïtés classiques du genre : sur combien d'octets sont codés les entiers ? que se passe-t-il si une addition provoque un débordement arithmétique ?

[11] par opposition aux langages synchrones tels que LUSTRE, ESTEREL et SIGNAL

[12] un choix identique a été fait en 1988 par le CCITT au moment de la dernière révision de SDL

En fait, pour décrire les données de manière portable, on n'a guère d'autre choix actuellement que les types abstraits. L'ISO a bien normalisé un autre langage, ANS.1[13], qui est très utilisé en tant que format d'échange de données dans les systèmes OSI. Mais ASN.1 convient mal pour la spécification, car il exprime uniquement la syntaxe des données et non les opérations sémantiques pour manipuler ces données.

Enfin, il faut souligner la richesse de LOTOS comme langage de spécification. Entre les deux attitudes extrêmes qui consistent à utiliser seulement la partie contrôle ou seulement la partie données, il existe une grande variété de styles de spécification (impératif, applicatif, orienté-objet, orienté-contraintes, ...) dont beaucoup restent à étudier.

6.2 Interprétation et compilation

La syntaxe et la sémantique statique de LOTOS peuvent être analysées grâce à des techniques de compilation classiques [BH88]. La sémantique dynamique de LOTOS en fait un langage "exécutable", contrairement à d'autres formalismes de spécification, VDM[14] par exemple.

Côté interprétation, plusieurs simulateurs interactifs ont été développés : ils facilitent la mise au point en permettant de faire évoluer une spécification LOTOS pas à pas.

Côté compilation, il existe des générateurs de code séquentiel capables de produire automatiquement un prototype en langage C à partir d'une spécification LOTOS. En ce qui concerne la génération de code parallèle, on sait traduire un sous-ensemble important de LOTOS vers un modèle de type réseau de Petri [Gar89a] [GS90] et l'on voit apparaître des algorithmes pour implémenter le rendez-vous LOTOS de manière distribuée. Enfin, il est possible d'engendrer un code C d'excellente qualité, souvent optimal, à partir des types abstraits algébriques de LOTOS, moyennant certaines restrictions sur la manière d'écrire les équations [Bar88] [Gar89b].

Ces résultats laissent espérer que l'on puisse un jour dériver automatiquement une implémentation efficace à partir d'une spécification LOTOS, ce qui supprimerait ou réduirait la part de codage manuel et le risque d'erreurs qu'elle comporte.

6.3 Vérification

Les systèmes répartis sont généralement soumis à de sévères contraintes quant à leur sûreté de fonctionnement. On pense immédiatement aux logiciels embarqués pour l'avionique ou aux dispositifs de surveillance des centrales nucléaires, mais une défaillance dans un transfert de fichier bancaire peut également avoir des conséquences désastreuses.

Pour de tels systèmes, on ne saurait se contenter de l'intime conviction des implémenteurs relativement à la correction de leur système. L'analyse *de visu* du code ne suffit pas ; seule une vérification rigoureuse, fondée sur une approche théorique solide et assistée d'outils de génie logiciel, peut apporter quelques certitudes dans ce domaine.

Malheureusement les langages d'implémentation actuels sont trop complexes pour permettre une vérification suffisamment précise. Le fait que les utilisateurs du langage C ne disposent pas d'outils pour l'analyse statique du flux de données — même sans prendre en compte les problèmes posés par le parallélisme — est significatif à cet égard. Pour de tels langages, seules les approches de qualimétrie, de test et de mesure de performances sont réalistes.

Or les propriétés de bon fonctionnement auxquelles on s'intéresse sont beaucoup plus ambitieuses. On cherche généralement à établir que le système ne peut pas se bloquer, qu'il reste toujours dans une situation globale cohérente, qu'aucun message n'est perdu, qu'aucun événement non prévu ne peut se produire, qu'une séquence d'événements attendue doit obligatoirement avoir lieu, ... La vérification de

[13] Abstract Syntax Notation One
[14] Vienna Development Method

telle propriétés nécessite une analyse globale — statique et dynamique — du système. Autant dire qu'elle est quasiment impossible à mettre en œ uvre pour des langages qui n'ont pas de sémantique formelle et qui reposent sur des mécanismes compliqués.

LOTOS réalise un heureux compromis entre la richesse d'expression voulue par les utilisateurs et la simplicité indispensable aux outils de vérification. C'est un langage d'inspiration mathématique faisant appel à des concepts "propres" (algèbres de processus et types abstraits), s'appuyant sur des modèles éprouvés (automates, graphes) et possédant des caractéristiques qui simplifient l'analyse statique des programmes (langage fonctionnel, propriétés algébriques des opérateurs).

Ceci suffit à expliquer le nombre considérable de travaux de recherche consacrés à la vérification pour LOTOS. On peut distinguer plusieurs approches : la transformation de programmes LOTOS par application de lois algébriques, la preuve de propriétés à l'aide de démonstrateurs de théorèmes, la traduction de LOTOS vers des réseaux de Petri et/ou des automates d'états finis [Gar89a] [GS90] sur lesquels on peut ensuite évaluer efficacement les propriétés désirées [Fer88].

On constate actuellement un fort besoin d'outils de vérification. Mais les utilisateurs ne sont pas toujours conscients que leur demande a un prix : il faut consentir à l'effort d'apprendre les méthodes formelles. En retour chaque étape de leur travail pourra être supportée par des outils et l'on peut envisager avec optimisme l'avenir de la conception de programmes corrects assistée par ordinateur.

Références

[Bar88] Christian Bard. *CÆSAR.ADT Reference Manual*. Laboratoire de Génie Informatique — Institut IMAG, Grenoble, août 1988.

[BB88] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–29, janvier 1988.

[BH88] Pascal Bouchon and Jean-Michel Houdouin. Analyseur LOTOS pour CÆSAR. Rapport de fin d'études ensimag, Laboratoire de Génie Informatique — Institut IMAG, Grenoble, juin 1988.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1 — Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, Berlin, 1985.

[Fer88] Jean-Claude Fernandez. *ALDEBARAN : un système de vérification par réduction de processus communicants*. Thèse de Doctorat, Université Joseph Fourier (Grenoble), mai 1988.

[Gar89a] Hubert Garavel. *Compilation et vérification de programmes LOTOS*. Thèse de Doctorat, Université Joseph Fourier (Grenoble), novembre 1989.

[Gar89b] Hubert Garavel. Compilation of LOTOS Abstract Data Types. In Son T. Vuong, editor, *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*, pages 147–162, Amsterdam, décembre 1989. North-Holland.

[GS90] Hubert Garavel and Joseph Sifakis. Compilation and Verification of LOTOS Specifications. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification (Ottawa, Canada)*, pages 379–394, Amsterdam, juin 1990. IFIP, North-Holland.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, août 1978.

[ISO88] ISO. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, septembre 1988.

[LS88] Jeroen van de Lagemaat and Giuseppe Scollo. On the Use of LOTOS for the Formal Description of a Transport Protocol. In Kenneth J. Turner, editor, *Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88 (Stirling, Scotland)*, pages 247–261, Amsterdam, septembre 1988. North-Holland.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1980.