CFIP'93 Ingénierie des Protocoles
R. Dssouli et G. v. Bochmann, eds.
Hermès, Paris, 1993.

Comment rendre LOTOS apte à spécifier des systèmes temps réel ?

Guy Leduc, Luc Léonard

Chercheur qualifié et Aspirant du F.N.R.S.,
Université de Liège,
Institut d'Electricité Montefiore, B 28,
B-4000 Liège 1, Belgique

Résumé. *Nous expliquons et justifions par des exemples notre proposition d'extension temporelle de LOTOS. Les exemples couvrent des fonctions de protocole et des facilités de service que l'on rencontre fréquemment en pratique ou qui en sont inspirés, tels que les temporisations, les chiens de garde, les délais, l'isochronisme, le multimédia, le contrôle de débit, … La sémantique du langage est présentée ensuite et ses propriétés discutées de manière plus formelle. Nous pensons que l'application systématique des techniques formelles temporelles à ce type d'exemples est le passage obligé pour convaincre du bien-fondé des choix de conception. L'efficacité pratique d'un formalisme est tout aussi importante que ses propriétés mathématiques associées.*
Mots-clés: FDT, langage de spécification, LOTOS, algèbre de processus, temps réel.
Abstract. *We explain and justify by examples our proposed timed extension of LOTOS. The examples cover protocol functions and service facilities that are frequently encountered in practice or are inspired by existing ones, such as timers, watchdogs, delays, isochronism, multimedia, rate control, … The semantics of the language is presented after, and its properties discussed more formally. We think that the systematic application of formal temporal techniques to this kind of examples is unavoidable to convince of the pertinence of the design choices. The practical effectiveness of a formalism is as important as the associated mathematical properties.*
Key words: FDT, specification language, LOTOS, timed process algebra, real time.

1. Introduction

Le besoin de spécifier formellement des systèmes dont le comportement dépend étroitement du temps n'est pas nouveau. La plupart des protocoles contiennent des mécanismes de temporisation essentiels pour la sûreté de fonctionnement. La retransmission de messages dans un protocole de liaison de données ou de transport en est un exemple évident.

Dès 1976, Merlin et Farber [Merlin 76] proposent une extension temporelle des réseaux de Petri visant précisément à spécifier ces mécanismes de récupération d'erreur basés sur des temporisations.

Depuis lors, plusieurs nouveaux mécanismes de protocoles, ainsi que les facilités de service correspondantes, ont fait leur apparition et rendent ce besoin encore plus fondamental. Citons par exemple les transferts de données isochrones, le contrôle de débit d'émission, la resynchronisation de flux multimédia, …

Vu l'importance du problème, la recherche d'une technique de description formelle apte à spécifier ces systèmes s'est intensifiée durant les cinq dernières années. La plupart des travaux portent sur l'extension de modèles existants et en particulier sur les algèbres de processus [Nicollin 91a]. Parmi ces travaux, certains, dont les nôtres, portent spécifiquement sur l'extension de LOTOS [Bolognesi 92, Leduc 93, Léonard 93b, Miguel 93a, Quemada 92].

Malgré le fait que, du point de vue fondamental, les articles publiés à ce jour aient quasiment fait le tour de la question, plusieurs écoles s'affrontent toujours sur quelques concepts de base, sans vraiment comparer leurs propositions sur des exemples types. Il nous apparaît pourtant que le principal critère pour départager ces différentes options devrait être celui de leur efficacité pratique, vérifiée par une application à des cas concrets. Les enseignements tirés de cet exercice aideraient alors à converger vers une proposition standard d'extension de LOTOS.

Cet article apporte sa contribution à cette tâche. Nous tentons d'y justifier de façon très pragmatique (mais parfois aussi théorique) notre langage ET-LOTOS, et ses choix ou rejets, sur base d'une collection d'exemples retenus pour leur réalisme mais aussi pour la difficulté inhérente à leur description. Notons que c'est par le biais d'exemples de ce genre que nous avons convergé vers la proposition actuelle. Ainsi, en même temps que l'extension de LOTOS, nous avons conçu le protocole TIC-TAC [Léonard 93a], qui est une agrégation de plusieurs mécanismes temporels réels et complexes. Dans cet article, nous ne reprenons pas l'entièreté de cet exemple, mais plutôt quelques morceaux choisis, complétés par d'autres exemples simples.

Nous nous intéresserons tout d'abord à l'extension du LOTOS séquentiel, puis aux processus parallèles qui nécessitent une attention particulière. Après cette présentation largement informelle, nous donnerons la sémantique formelle de ET-LOTOS et nous discuterons quelques propriétés importantes du langage. Enfin, par une comparaison avec des travaux analogues, nous tenterons de faire apparaître les avantages de notre proposition.

2. Description de processus temporisés séquentiels

Qu'y a-t-il de plus banal qu'un mécanisme de temporisation dans un protocole de communication ? Et pourtant, la plupart des formalismes existants sont bien souvent incapables de le spécifier correctement. Ainsi, en LOTOS, il est habituel de trouver des expressions du type: `send; (ack; send_next [] i; re-send)` pour évoquer ou approcher ce mécanisme élémentaire. Dans cette expression, l'événement interne `i` est censé se produire "au bon moment", ni avant, ni après. Cette lacune n'est pas problématique si la logique du protocole ne dépend pas de la valeur de la temporisation. Toutefois, il est clair qu'une telle approximation peut souvent être la source de difficultés. Par exemple, s'il existe deux valeurs de temporisation dans un même système (l'une côté émetteur, l'autre côté récepteur) comment exprimer que l'une doive être supérieure à l'autre ou égale à 3 fois l'autre ?

Avant de présenter les principes et les constructeurs temporels que nous proposons d'adopter, nous devons introduire la façon de modéliser le temps, c'est-à-dire le

domaine temporel. Celui-ci définit un ensemble de valeurs (temporelles) et doit, pour être temporel, respecter certaines propriétés.

Certains auteurs ont par exemple choisi un domaine temporel *discret*, isomorphe à l'ensemble des nombres naturels $N \cup \{\infty\}$ et muni de l'opération + et de la relation d'ordre $\leq$. Nous estimons qu'un tel domaine n'est pas simple à utiliser en pratique car il nécessite de définir un grain de temps de référence a priori. Cela n'est pas toujours possible. Par exemple la spécification d'un système qui gère un débit de transmission qui peut croître sans borne prévisible et donc requérir des intervalles de temps entre émissions arbitrairement petits. De plus, lorsque l'on désire raffiner la description d'un système, changer de niveau d'abstraction, il n'est pas rare de devoir changer le grain de temps de référence, et cela nécessite une réécriture complète de la spécification. Pour ces raisons, nous avons conçu ET-LOTOS de façon à ce qu'il supporte aussi un domaine de temps dense : ni la syntaxe, ni la sémantique ne dépendent du modèle temporel. Un modèle temporel est *dense* s'il est toujours possible de trouver une valeur comprise entre deux valeurs arbitraires. Deux exemples de domaines denses sont des domaines isomorphes aux nombres réels et aux nombres rationnels. Pour des raisons techniques de définition de la sémantique d'ET-LOTOS par des systèmes de transitions étiquetées, nous nous limiterons cependant à des domaines dénombrables; ce qui est le cas des rationnels mais pas des réels.

En fait, avec ET-LOTOS, il est possible de définir à sa guise le domaine temporel pour autant qu'il respecte certaines contraintes. On pourrait par exemple étendre la librairie standard avec un type de donnée comportant la sorte `time` et quelques opérations usuelles. Si l'on désigne par D le domaine temporel, par + l'addition sur D, par 0 l'élément neutre de D, (D,+,0) sera un monoïde commutatif, (D,$\leq$) sera un ordre total et ∞ sera l'élément absorbant pour l'addition et tel que $\forall\, d \in D.\ d \leq \infty$.

2.1. Deux constructions de base : le limiteur d'offre et l'opérateur de délai

2.1.1. La temporisation

Partant de cet exemple anodin, nous pouvons déjà faire apparaître très précisément deux besoins d'extension de LOTOS: *la durée d'une offre d'interaction* et *le préfixage par un délai*. En effet, si nous analysons le comportement d'un émetteur, nous constatons qu'après l'émission d'un message, il est disposé à attendre un acquit *pendant un certain temps* (notion de durée d'une offre) ou de réémettre son message *après ce même temps*. Dans notre formalisme, cette description se traduit littéralement et donne le processus de la figure 1. Remarquons que nous ne respectons pas scrupuleusement la syntaxe de LOTOS [ISO 8807], mais que nous adoptons une notation plus dépouillée dans laquelle nous omettrons souvent les mots clés "process", "endproc", ainsi que les arguments des processus, lorsque cela ne nuit pas à la compréhension.

```
SAFE:= Req?s:SDU; SEND(s)
SEND(s):= Send!s; (Ack{waiting-time}; SAFE
                  []
                  Δ^waiting-time SEND(s))
```

Figure 1. *Temporisation*

Le préfixage d'action est donc étendu: l'expression `g{time}`, où `time` ∈ D, exprime que l'offre `g` cessera après une durée `time`. Cet attribut temporel est appelé le *limiteur d'offre*. Le préfixage d'une action munie d'un limiteur d'offre, soit `g{time}`, doit s'interpréter comme suit. Si après le délai `time`, `g` ne s'est pas produite, le processus se comporte comme `stop`; ce qui signifie que l'offre est retirée mais que le comportement ultérieur n'est pas exécuté. Toutefois, ceci n'empêche pas que les alternatives de comportement (s'il y en a), spécifiées par un choix `[]`, soient exécutables. Le limiteur d'offre n'exprime donc pas une obligation (ou une nécessité) d'exécution de `g` dans un délai imparti, mais plutôt une interdiction d'exécution *après* ce délai imparti. Lorsque l'action n'est pas complétée d'un limiteur d'offre, nous retrouvons la sémantique de LOTOS où l'action peut se produire n'importe quand. Le limiteur d'offre est donc par défaut égal à ∞ pour les actions observables. Pour les actions internes, nous verrons plus loin que cette valeur par défaut est 0.

L'opérateur $\Delta^{\texttt{time}}$ exprime le préfixage par un délai de valeur déterminée `time`. On notera en outre qu'aucune action interne n'intervient dans la description de la figure 1.

Notre approche pour décrire des délais et des durées d'offre n'est pas la seule possible. On aurait pu aussi tout simplement généraliser le préfixage d'action en y incluant *deux* attributs ou un *intervalle* temporel du type `g{min,max}` : `g` n'est offerte qu'après un temps `min` et jusqu'à un temps `max`. C'est d'ailleurs fait de cette manière dans [Bolognesi 92]. Nous n'avons pas retenu cette option pour une raison très simple. Dissocier l'introduction d'un délai de la limitation de l'offre par l'opérateur Δ donne une combinaison plus puissante. Δ n'est pas attaché simplement à une action, comme la valeur `min` dans `g{min,max}`, mais peut aussi bien s'appliquer à un processus. Il nous apparaît aussi que la notation $\Delta^{\texttt{delay}}$ `g{lifetime}`, où le délai précède l'action, est plus claire que l'association d'un intervalle écrit après l'action. Nous admettrons toutefois la notation compacte: `g{min,max}` pour $\Delta^{\texttt{min}}$ `g{max-min}`.

Si nous reprenons l'exemple de la figure 1, on voit que Δ préfixe le processus `send` qui peut être considéré dans son entièreté, c'est-à-dire sans que l'on doive le développer pour faire apparaître ses actions et ensuite les contraindre par une borne inférieure.

La figure 2 illustre un autre cas où les processus `accept_data` et `accept_exp_data` sont préfixés par des délais différents. Il s'agit d'un système qui met plus de temps à accepter une nouvelle donnée ordinaire qu'à accepter une nouvelle donnée urgente.

Cet exemple nous permet aussi d'insister sur le fait que l'expiration d'un délai ne doit pas résoudre un choix. En effet, si c'était le cas, le processus `accept_data` n'aurait jamais l'occasion de s'exécuter à cause de "l'effet de bord" de l'expiration du délai `small_delay`. Il faut donc que l'expiration d'un délai soit sans conséquence vis-à-vis d'autres actions offertes. Il est dès lors exclu d'interpréter cette expiration comme un événement interne.

```
SP:= Δ^high-delay ACCEPT-DATA >> SP
     []
     Δ^small-delay ACCEPT-EXP-DATA >> SP
```

Figure 2. *Δ ne résout pas les choix*

Dans LOTOS-T [Miguel 93a], un mécanisme plus simple que ceux que nous venons d'évoquer est proposé. Le préfixage peut être complété aussi par un label temporel, comme notre limiteur d'offre, mais avec une signification très différente. Il s'agit d'une estampille temporelle qui signifie que l'action n'est offerte qu'à un instant précis et non pas pendant une durée précise. Cette estampille est en fait un intervalle de durée nulle. Dans LOTOS-T, les mêmes critiques s'appliquent donc avec le désavantage supplémentaire qu'il n'y a plus de notion de durée de l'offre. Exprimer avec LOTOS-T l'effet de notre limiteur d'offre s'avère plus lourd : soit on recourt à un opérateur `choice` généralisé sur des offres ponctuelles (p.ex. `choice t:time [] [t ≤ max] -> g[t]; …`), soit on utilise un mécanisme similaire à une temporisation où l'offre est arrêtée par une action alternative après un délai donné.

2.1.2. L'isochronisme

La paire (limiteur d'offre, opérateur Δ) s'avère être également très puissante pour spécifier ce type de comportement. La figure 3 illustre un processus qui n'accepte les primitives `DataReq` qu'à des instants précis régulièrement espacés dans le temps. D'où le terme isochronisme.

```
ISOCHRONOUS:= Δ^period DataReq{0}; ISOCHRONOUS
```

Figure 3. *L'isochronisme*

Notons que l'offre est ici de durée nulle. L'utilisateur doit être prêt à interagir à cet instant pour que l'action ait lieu. Dans le cas contraire, le processus se comporte comme `stop` et n'accepte plus aucune action ultérieure. En d'autres termes, ce processus ne prévoit pas le cas où l'action `DataReq` ne se produirait pas. On peut bien sûr prévoir explicitement ce cas comme le montre la figure 4. Le processus dispose maintenant d'un mécanisme de récupération en cas d'offre insatisfaite.

```
ISOCHRONOUS:= Δ^period  (DataReq{0}; ISOCHRONOUS
                         []
                        ISOCHRONOUS)
```

Figure 4. *L'isochronisme avec tolérance à la faute*

Notons à nouveau qu'aucun de ces processus n'impose l'occurrence de la primitive. Toutefois, la figure 3 nous montre un processus confiant, qui n'imagine pas que celle-ci ne puisse pas se produire. Par contre, la figure 4 nous montre un processus prudent, plus robuste en cas de faute. Si notre processus modélise un service de communication qui n'accepte les primitives qu'à des instants précis, la deuxième approche semble mieux convenir car l'utilisateur peut ne pas avoir de donnée à transmettre à chaque fois. Quand peut-on alors vouloir spécifier des processus "confiants" ? Si le concepteur du fournisseur de service est également le concepteur de l'utilisateur de service, il se peut qu'il puisse être sûr que l'utilisateur sera toujours prêt. Ce cas serait plus plausible avec une primitive d'indication plutôt qu'une primitive de requête. Un autre exemple est toutefois plus parlant. Supposons que l'interaction consiste à émettre un rayon de lumière dans l'air ou verser une certaine quantité de café (normalement dans une tasse), alors le concepteur peut être confiant, l'air ne stoppera pas la lumière et l'absence de tasse n'empêchera pas le café de se déverser.

A partir de ces exemples, on voit très bien que le concepteur peut introduire une tolérance à la faute dans son système lorsqu'il juge que l'environnement est susceptible de ne pas être coopérant. Dans le cas contraire, aucune précaution n'est prise; ce qui indique implicitement que l'environnement est toujours disposé à se comporter comme le système le souhaite. L'un des principaux points de discorde entre les différents concepteurs d'extensions temporelles est sans doute celui de savoir comment, dans la sémantique, exprimer cette notion de confiance dans l'environnement. Certains soutiennent qu'il faut imposer à une action dont l'offre est réduite, de se produire durant sa période de validité. Cela se traduit par un artifice mathématique curieux qui, le cas échéant, consiste à bloquer le passage du temps pour éviter de dépasser l'instant ultime de l'offre d'une action. C'est ce qui se passe dans [Bolognesi 92] avec le mécanisme d'intervalle évoqué plus haut. Nous ne sommes pas en faveur de cette option gênante en pratique et contre-intuitive. On peut vouloir exprimer la confiance en son environnement, mais cela ne signifie pas que l'on puisse effectivement contraindre celui-ci à mériter cette confiance s'il ne le désire pas. Bloquer le temps en cas de non participation de l'environnement n'apporte rien et s'avère une contrainte d'usage très peu pratique. Pour s'en persuader, il suffit de se rendre compte que si les mécanismes des figures 1 et 4 sont simples, c'est justement grâce au fait que le temps ne se trouve pas stoppé à l'échéance des délais exprimés par les limiteurs d'offre.

En réalité, la seule chose à prendre en compte est beaucoup moins contraignante: il faut que le système donne la *préférence à l'interaction* plutôt qu'à une progression dans le temps sans interaction. Cet élément indispensable sera toutefois discuté plus loin. Sur base des exemples des figures 3 et 4, notons déjà que cela se traduira par le fait que, si l'environnement et le système sont prêts à exécuter l'action `DataReq`, alors celle-ci aura lieu. En effet, on ne pourrait pas admettre que le temps passe sans qu'une telle interaction (devenue autonome - interne - puisque l'environnement y est associé) ne s'exécute.

2.2. Délai non déterministe

L'opérateur Δ que nous avons introduit permet de préfixer un processus par un délai déterminé. Si nous voulons modéliser un milieu de transmission pouvant introduire un délai compris dans un intervalle donné [min, max], Δ n'est pas suffisant, car il ne peut imposer que la contrainte min par le biais de Δ^{min}. Toute offre apparaissant après cet intervalle est susceptible de se produire dès que l'environnement est disposé à le faire. Ce que nous voulons exprimer, c'est que le milieu de transmission puisse choisir de manière interne le délai dans l'intervalle [min, max] sans que l'environnement ne puisse intervenir. Il s'agit d'une forme temporelle de non déterminisme interne.

```
TRANSM := DataReq?s:SDU; Δ^min i{max-min}; DataInd!s; stop
```

Figure 5. *Délai non déterministe ou aléatoire*

Considérons la figure 5. Le processus introduit un délai minimum `min` par l'opérateur Δ^{min} entre une requête et l'indication correspondante. De plus, il peut introduite un délai supplémentaire aléatoire dans l'intervalle [0, max-min] qui est matérialisé par l'événement interne `i` muni d'un limiteur d'offre `{max-min}`. Comme

pour les actions observables, ce limiteur d'offre rend l'occurrence de `i` impossible passé ce délai. De plus, il laisse cette occurrence aléatoire dans l'intervalle [0, max-min] puisque l'action interne est autonome. Ce qui est fondamental dans cette construction, c'est que l'environnement n'a aucune prise sur ce processus en ce qui concerne ce délai aléatoire. Le processus décide seul. Dans `i{delay};a;stop`, à tout moment dans l'intervalle [0, delay] le processus peut de manière autonome exécuter l'action interne `i` et rendre `a` possible. Notons la différence avec $\Delta^{\texttt{delay}}$ `a;stop` où l'environnement ne peut interagir avant le délai `delay`.

Le fait que nous nous trouvions en présence d'une forme nouvelle de non déterminisme justifie l'emploi de l'action interne `i`, contrairement au cas du délai fixé. Rappelons que l'opérateur Δ ne fait pas intervenir `i`, même de manière cachée.

Il nous reste une propriété à évoquer concernant cet événement interne `i`, sans quoi la spécification de la figure 5 n'a pas l'effet recherché. Il s'agit de l'*obligation* pour `i` de s'être produit *au plus tard* à la fin de l'intervalle [0, max-min]. Rappelons que pour les actions observables, nous n'avons pas imposé cette contrainte car nous la jugeons inutile en pratique et contre-intuitive. Pour l'action interne, il n'en va pas de même. Cette action étant interne, elle est autonome et il n'est donc pas contre nature de l'obliger à s'exécuter à un instant précis. De plus, cette obligation (ou nécessité) s'avère très pratique. Le cas que nous venons de présenter en est un exemple: il nous permet d'imposer une fin à un délai aléatoire. Dans la section suivante nous illustrons trois autres usages de cette caractéristique de `i`.

Lorsque l'action interne n'est pas complétée d'un limiteur d'offre, nous considérons par défaut qu'il vaut 0. On dira qu'un tel `i` est *urgent*. Par opposition, dans `i{∞}`, `i` est dit *non contraint*. Nous aurions pu adopter pour `i` la même convention que pour les actions observables. Deux raisons nous ont conduits à changer. La raison principale sera donnée dans la section 3.4. La deuxième raison est que les spécifications comportent beaucoup plus de i{0} que de i{∞} en pratique.

Notons que nous aurions pu ne pas introduire une action interne pour modéliser la fin d'un délai aléatoire, mais cela nous aurait conduit soit à un modèle de temps non déterministe, soit à l'introduction d'une autre action interne spéciale [Leduc 93]. Nous avons voulu éviter ces deux alternatives qui s'avèrent moins adéquates.

Remarquons que l'on retrouve la faculté d'imposer l'occurrence de i tant dans [Miguel 93a] que dans [Bolognesi 92]. Dans [Miguel 93a] cependant, l'absence d'intervalle complique la spécification d'un délai non déterministe. Il faut à nouveau recourir à l'opérateur `choice` généralisé. Par exemple, `choice t:time [t isin interval] -> i; i[t]; P`. Le premier `i` étant urgent par défaut, le choix du `t` est résolu au temps 0.

2.3. Sur la nécessité d'exécuter l'action interne

Dans cette section, nous reprenons trois exemples qui illustrent l'intérêt de disposer de la possibilité de forcer l'exécution de l'action interne à un moment précis.

2.3.1. Le double chien de garde

Le premier de ces mécanismes temporels a été modélisé dans [Nicollin 91b]. Il s'agit d'un double chien de garde. Pour modéliser un chien de garde, X. Nicollin et J. Sifakis introduisent un nouvel opérateur dans ATP. Nous serions donc en droit de

nous interroger sur la nécessité d'introduire un tel opérateur dans ET-LOTOS. En fait, cet opérateur n'est pas nécessaire ici. Considérons la figure 6. Il s'agit d'une procédure d'entrée en session. Par `prompt` le système invite l'utilisateur à introduire son nom et son mot de passe en moins de `logtime` unités de temps. En cas de dépassement de temps ou de mot de passe incorrect, le système reprend au début, sinon le système accepte la requête. L'utilisateur peut ainsi avoir plusieurs possibilités d'entrer en session, mais le temps total imparti pour l'ensemble des tentatives ne peut dépasser `maxtime`. Il y a donc bien deux chiens de garde imbriqués paramétrés respectivement par `logtime` et `maxtime`.

```
SESSION :=
(LOGIN-PROCEDURE [> Δ^maxtime i; stop) >> SESSION-PHASE
where
LOGIN-PROCEDURE :=
PHASE1 >> accept b:bool in
          ([b]-> exit) [] ([not(b)]-> LOGIN-PROCEDURE)
where
PHASE1 :=
prompt?log:login; prompt?pwd:password; exit(correct(log,pwd))
[> Δ^logtime i; PHASE1
```

Figure 6. *Procédure d'entrée en session*

Les raisons pour lesquelles un opérateur spécifique n'est pas nécessaire en ET-LOTOS sont les suivantes: LOTOS possède déjà l'opérateur `[>` d'interruption qui, couplé avec un délai et la possibilité d'imposer l'exécution de `i` à un moment donné donne la même expressivité.

Notons qu'en ET-LOTOS le processus `exit` peut avoir un limiteur d'offre qui est par défaut égal à ∞ (cas de la figure 6). Ce limiteur signifie simplement que la terminaison ne pourra avoir lieu passé ce délai. Nous traitons en fait l'action de terminaison réussie δ comme une action observable ordinaire.

2.3.2. Temporisation

Comme deuxième exemple, nous reprenons la temporisation de la section 2.1 mais dans un cas plus général. Considérons la figure 7 qui illustre la même temporisation que celle de la figure 1 mais où l'on suppose l'existence d'un processus `receive_ack` que l'on s'interdit de modifier. Ce processus `receive_ack` représente le comportement normal, tandis que le processus `send` représente la récupération lors de l'expiration du délai d'attente.

```
SAFE:= Req?s:SDU; SEND(s)
SEND(s):= Send!s; (RECEIVE_ACK  []  Δ^waiting-time i; SEND(s))
```

Figure 7. *Temporisation - Variante*

Ici, la présence de `i` implique que le système va nécessairement basculer (c.-à-d. résoudre le choix) directement après l'expiration du délai $\Delta^{\texttt{waiting\_time}}$. Comme nous l'avons expliqué, nous préférons que l'opérateur Δ n'induise pas l'occurrence de l'action `i` lui-même, vu la résolution des choix que cela impliquerait. Toutefois,

dans certains cas, comme celui-ci, on apprécie de disposer d'un moyen de résoudre les choix.

2.3.3. Contrôle de débit

Comme troisième exemple, nous avons extrait du service de transport OSI95 [Baguette 92] une facilité de service assez subtile. Il s'agit d'un fournisseur de service qui est tenu de contrôler le débit d'entrée des données conformément à des critères de qualité de service qui ont été négociés. Dans un premier temps, nous considérons trois critères à respecter:

- Le fournisseur empêche l'utilisateur d'émettre à une cadence trop élevée en espaçant les offres d'interaction `DataReq` d'au moins `min` unités de temps.
- Le fournisseur peut ne pas être prêt tout juste après ce délai `min`.
- Le fournisseur doit offrir une cadence minimale à l'utilisateur ou, dès qu'il n'en est plus capable, initier la déconnexion. Cela se traduit par le délai maximal `max` entre deux offres de `DataReq`. Remarquons que si l'interaction `DataReq` n'a pas lieu parce que l'utilisateur n'a pas de données à émettre, le fournisseur ne doit pas initier la déconnexion. Seule la prise de conscience par le fournisseur de ne pas pouvoir tenir son engagement le force à déconnecter, même s'il s'avère que l'utilisateur n'avait rien à émettre à ce moment là.

```
THROUGHPUT-CONTROL1:=   Δ^min i{∞}; DataReq; THROUGHPUT-CONTROL1
                        []
                        Δ^max i; DisconnectInd; stop
```

Figure 8. *Contrôle de débit avec déconnexion*

La figure 8 en montre la description en ET-LOTOS. On notera la présence des deux délais Δ^{min} et Δ^{max}, qui ne résolvent pas les choix quand ils expirent, et la présence de deux actions internes : la première étant non contrainte et la deuxième urgente. Elles sont nécessaires pour modéliser respectivement la deuxième et la troisième propriétés. Notons la nécessité de résoudre les choix dans ce cas.

L'exemple suivant est une variante du précédent. On y a ajouté le critère suivant: le fournisseur doit indiquer à l'utilisateur qu'il ne peut (temporairement) tenir un débit de référence (compris entre le minimum et le maximum) qui a été négocié. La figure 9 montre la solution. `Thres` est le seuil d'indication (threshold).

```
THROUGHPUT-CONTROL2:=
Δ^thres ReportInd; stop
[> (Δ^min i{∞}; DataReq; THROUGHPUT-CONTROL2
    []
    Δ^max i; DisconnectInd; stop)
```

Figure 9. *Contrôle de débit avec indication et déconnexion*

2.4. Mesure du temps d'attente

Il s'avère parfois nécessaire en pratique de mesurer le temps pendant lequel un processus propose une interaction avant que celle-ci ne se produise. Pour réaliser

cela, nous proposons d'enrichir le préfixage d'action par un attribut spécial `@t,` "at t", qui sauve dans la variable `t` la valeur du temps d'attente. Nous l'appelerons "la mesure du temps d'attente". Cette solution a été proposée par Wang Yi [Wang 91] pour permettre à son extension temporelle de CCS de disposer d'un théorème d'expansion en présence d'un temps dense. On pourrait motiver son introduction dans ET-LOTOS de la même manière, mais en fait, cette construction s'avère aussi très utile en pratique comme vont l'illustrer quelques exemples.

2.4.1. Isochronisme

Nous reprendrons tout d'abord une variante de l'exemple sur l'isochronisme présenté dans la section 2.1.2. Ici, nous remplaçons l'offre ponctuelle par une offre de durée `d`, inférieure à `period`, et nous imposons de comptabiliser les offres réussies et les non réussies séparément afin de pouvoir adapter la période. La figure 10 montre l'utilisation de l'attribut `@t`. La nouvelle période est calculée par la fonction `new` non détaillée.

```
behaviour ISOCHRONOUS(0,0,period,0) where
ISOCHRONOUS(successful,unsuccessful,period,t1):=
Δ^period-t1 (a@t{d}; ISOCHRONOUS (successful+1,unsuccessful,
                          new(period,successful,unsuccessful),t)
            []
            ISOCHRONOUS (successful,unsuccessful+1,
                         new(period,successful,unsuccessful),0))
```

Figure 10. *Isochronisme - Variante*

L'attribut `@t` est une forme particulière de déclaration de variable `t` dont la portée se définit comme pour une déclaration ordinaire de variable en LOTOS. Cette construction `@t` nous permet d'éviter de devoir recourir à l'opérateur `choice` généralisé comme c'est le cas dans [Miguel 93a]. Enfin, nous voyons plus difficilement comment le temps pourrait être mesuré dans [Bolognesi 92] à cause de problèmes liés au blocage du temps.

2.4.2. Contrôle de gigue

Un autre exemple très typique de service à fournir aux applications multimédia est le contrôle de la gigue sur le délai de transmission. Ces applications ne peuvent tolérer de trop grandes variations sur ces délais.

```
JITTER_CONTROL :=
Control?t:time; JITTER_CONTROL2 (t,t)
where
JITTER_CONTROL2 (tmin,tmax:time) :=
  Control?t:time;
       ([max(t,tmax) - min(t,tmin) ≤ maxjit]->
              JITTER_CONTROL2 (min(t,tmin),max(t,tmax))
        []
        [max(t,tmax) - min(t,tmin) > maxjit]-> DiscInd; stop)
```

Figure 11. *Contrôle de gigue*

La figure 11 illustre un processus chargé de contrôler la gigue et d'initier la déconnexion en cas d'écart trop important. On suppose que ce processus reçoit les informations sur chaque délai de transit (mesuré par un autre processus communicant) par le point d'interaction `control` et qu'il initie la déconnexion par `DiscInd`. Ce processus ne contient en fait aucune construction temporelle mais illustre bien le type de traitement que l'on désire effectuer sur des temps mesurés.

2.4.3. Mesure du temps entre pannes

Nous modélisons un système dans lequel une panne peut survenir n'importe quand, mais le système ne redémarre que si cette panne n'est pas trop proche de la panne précédente (écart minimal `min`). La figure 12 illustre cet exemple.

```
P:= Normal_behaviour [> (i@t; [t > min] -> P)
```

Figure 12. *Mesure du temps entre pannes*

2.5. Conclusion sur les processus séquentiels

Jusqu'ici tous nos exemples ne faisaient pas intervenir de synchronisation entre processus. Le cas de la composition parallèle de processus sera discuté dans la section 3. Nous pouvons remarquer que les extensions de LOTOS que nous proposons sont très simples à mettre en œuvre et très flexibles. Nous les rappelons ci-dessous.

- Un opérateur Δ^{t} permettant de préfixer un processus par un délai `t`. Sa sémantique est telle qu'aucune action interne n'intervient en fin de délai.
- Un limiteur d'offre dans le temps à ajouter au préfixage par une action, dont la valeur par défaut est ∞ (ou 0 pour i). Aucune action interne n'intervient en fin d'offre et il n'y a pas d'obligation d'effectuer l'action en fin d'offre sauf pour l'action `i`.
- L'ajout d'un attribut optionnel `@t` aux offres d'action permettant de mesurer le temps pendant lequel l'offre a été proposée avant d'être exécutée.

Nous avons montré pourquoi la dissociation entre l'opérateur de délai et le limiteur d'offre a notre faveur, et combien l'obligation imposée à des actions observables est non seulement inutile, mais aussi encombrante.

3. Extension aux processus parallèles

Dans cette section, nous abordons la description de systèmes plus complexes composés de plusieurs processus communicants. Nous verrons sur base de plusieurs exemples qu'en pratique, il nous paraît tout à fait suffisant d'imposer que les *synchronisations internes* entre processus soient urgentes pour spécifier des systèmes distribués temps réel.

3.1. Temporisation symétrique

Notre premier exemple est la temporisation symétrique proposée par T. Bolognesi dans [Bolognesi 92]. Cet exemple met en présence deux processus dont chacun:

- exécute des tâches propres de durée inconnue (représentées respectivement par les actions `d1` et `d2`);

- puis propose à l'autre de se synchroniser (via la porte `sync`) dès que possible (c.-à-d. dès que *l'autre* est prêt);

- mais n'est pas disposé à attendre l'autre plus d'un certain temps (respectivement `t01` et `t02`).

Cet exemple est présenté comme étant difficile à résoudre car aucun des processus en présence ne peut déterminer quand l'autre devient prêt à interagir. En particulier, l'autre processus peut être prêt avant. Il devient donc impossible pour un processus d'imposer le moment où l'interaction doit avoir lieu. L'exigence d'urgence de l'interaction ne peut donc se matérialiser qu'à un niveau plus global où les deux processus sont vus comme une paire de processus composés.

Pour réaliser cela, nous avons adopté l'hypothèse de *progression maximale*, mais celle-ci est limitée aux *interactions internes*. Cela signifie que la sémantique de l'opérateur `hide` est telle que toute action cachée devient urgente. En d'autre termes, si une synchronisation interne peut se produire, elle se produira immédiatement. En fait, il faut être un peu plus nuancé, car cette interaction interne peut être en compétition avec une autre action également exécutable et, dans ce cas, ne pas avoir lieu. Ce qu'il faut retenir de l'hypothèse de progression maximale des interactions internes, c'est que l'interaction interne est prioritaire vis-à-vis du passage du temps (sans interaction), mais pas vis-à-vis d'autres actions.

Cette progression maximale sur l'interaction cachée `sync` nous permet d'écrire une spécification en ET-LOTOS (figure 13) d'une extraordinaire simplicité. Notons également l'intérêt du limiteur d'offre dans ce cas.

```
SYMMETRICAL_TIMEOUT [d1,d2] (to1,to2:time) :=
hide sync in (PROCESS[d1,sync](to1) |[sync]| PROCESS[d2,sync](to2))
where
PROCESS [d,sync] (t:time) := d; sync{t}; PROCESS [d,sync](t)
```

Figure 13. *La temporisation symétrique*

Il existe une seule alternative à la progression maximale des interactions internes pour pouvoir exprimer cette temporisation symétrique. Il s'agit de la proposition de [Bolognesi 92] qui consiste à introduire un opérateur spécifique qui a pour but de rendre certaines interactions urgentes.

3.2. Synchronisation entre un émetteur et un temporisateur

La figure 14 illustre une autre façon moins abstraite de spécifier la temporisation de la figure 1. Ici, nous avons isolé un processus `Timer` qui réalise la fonction de temporisation et interagit avec le processus émetteur `Sender` par trois points d'interaction: `set` (pour armer la temporisation), `reset` (pour stopper la temporisation) et `timeout` (pour indiquer l'expiration du timer).

On voit que les trois interactions `set`, `reset` et `timeout` sont internes, non accessibles à l'environnement. De plus, pour que la temporisation fonctionne comme on le souhaite, il est nécessaire que les interactions entre `Sender` et `Timer` se produisent dès qu'elles sont rendues possibles par les deux processus. Autrement dit, il convient que les interactions internes soient urgentes.

```
SYSTEM :=
hide set,reset,timeout in (SENDER
                           |[set,reset,timeout]|
                           TIMER)
where
SENDER := req?s:sdu; SENDER2 (s)
SENDER2 (s:sdu) :=
send!s; set!waiting-time; (ack; reset; SENDER
                          []
                          timeout; SENDER2 (s))
TIMER := set?t:time; (reset; TIMER
                     []
                     Δ^t timeout; TIMER)
```

Figure 14. *Une temporisation moins abstraite*

Notons que dans un cas comme celui-ci, d'autres formalismes, qui ne disposent ni de la progression maximale, ni d'un opérateur d'urgence (p.ex. TIC [Quemada 92], ATP [Nicollin 90]) peuvent quand même spécifier correctement cet exemple. Ceci est dû au fait que, pour chaque interaction, on connaît d'avance le processus qui se fera attendre: le `sender` pour un `set` ou un `reset`, le `timer` pour un `timeout`. On peut alors imposer l'urgence de manière locale dans un des processus uniquement pour chaque interaction.

3.3. Terminaison urgente

La progression maximale sur interaction interne s'étend naturellement aux synchronisations sur l'action de terminaison réussie δ. Pour illustrer les bienfaits induits par cette caractéristique, nous nous sommes inspirés de l'exemple de resynchronisation des composantes d'un flux multimédia (voix et image) proposé dans [Regan 93]. La spécification est donnée à la figure 15. Elle est composée de deux processus parallèles dont chacun gère une composante du flux (la voix ou l'image). Chaque processus est censé recevoir un flux de trames quasi-isochrone (c.-à-d. isochrone à une gigue près) et réémettre ce flux de façon totalement isochrone. Il introduit pour ce faire un délai égal à la gigue maximale. Toutefois, si la gigue du flux d'entrée est telle qu'une trame se fait attendre trop longtemps, alors le processus ne pourra pas la resynchroniser et signale une erreur. Le fait de disposer d'une synchronisation urgente sur δ, et donc d'une terminaison urgente de processus permet une modélisation très élégante de ce système. La figure 15 nous montre la solution. Chaque processus resynchronise son flux entrant jusqu'au moment où il est dans l'impossibilité de le faire à cause d'un retard trop important d'une trame. Dans ce cas, il propose la terminaison (exit) avec un code d'erreur. Cette terminaison va interrompre également le processus qui s'occupe de l'autre flux grâce à la construction `[> exit`. On voit que, par sa symétrie, cette structure permet un arrêt simultané et immédiat à l'initiative de l'un ou l'autre. Notons encore que les deux processus `exit (video)` et `exit (sound)` ne peuvent pas se synchroniser car `video` ≠ `sound`.

Cet exemple est à rapprocher de la temporisation symétrique de la section 3.1, car on ne peut connaître d'avance le processus qui initiera la déconnexion. Les mêmes remarques sont dès lors d'application.

```
SYNC-CONTROL [soundrec,videorec,out1,out2,errorsignal]:=
  ((CONTROL [soundrec,out1] (sound,speriod,sjitter) [> exit(video))
   |||
   (CONTROL [videorec,out2] (video,vperiod,vjitter) [> exit(sound))
  ) >> accept err:ersource in errorsignal!err; stop
where
  CONTROL [r,s] (ers:ersource, per,jit:time):=
        r@t{jit}; Δ^jit-t s{0}; Δ^per-jit CONTROL [r,s] (ers,per,jit)
        []
        Δ^jit exit(ers)
```

Figure 15. *Resynchronisation des composantes d'un flux multimédia*

3.4. Conclusions sur le parallélisme

Un seul concept à été introduit dans cette section : l'urgence sur les synchronisations internes. Remarquons que plusieurs autres formalismes adoptent également ce concept [Hansson 90, Hennessy 90, Miguel 93a, Reed 88, Regan 93, Wang 91].

Nous pouvons également mieux justifier la décision de prendre 0 comme valeur par défaut du limiteur d'offre associé aux actions internes `i`, par opposition à la valeur par défaut ∞ pour les limiteurs d'offre des actions observables. Nous avons voulu être consistants avec le fait que la progression maximale induit nécessairement l'urgence sur les `i` générés par abstraction. Plus précisément, si nous voulons que deux processus qui étaient fortement bissimilaires en LOTOS le restent en ET-LOTOS (par exemple `i;exit` et `hide a in (a;exit |[a]| a;exit)`), il importe de considérer que les `i` non munis de limiteur d'offre ont en fait un limiteur d'offre nul.

Comme nous l'avons déjà mentionné, certains formalismes sont moins expressifs que le nôtre parce qu'ils ne peuvent imposer l'urgence que de manière locale (c.-à-d. un processus décide du moment de l'occurrence pour les autres) [Nicollin 90, Quemada 92].

D'autres formalismes ont adopté une autre façon de spécifier l'urgence de manière globale : en ajoutant un nouvel opérateur qui peut rendre urgentes des actions (y compris observables), et donc en particulier des interactions [Bolognesi 92]. Si ces interactions urgentes sont ensuite rendues internes par un `hide`, alors on dispose d'un moyen similaire à la progression maximale. Cette approche a l'avantage de découpler l'urgence de l'abstraction (`hide`) et est plus générale car elle permet de spécifier un intervalle pour l'occurrence des interactions internes plutôt que la simple urgence. Toutefois, nous ne retenons pas cette formule car elle conduit à obliger l'occurrence d'actions observables (Cf. section 2). De plus, il n'est pas évident que la généralité de cette approche soit utile en pratique. Si c'était le cas, nous pourrions comme T. Bolognesi ajouter un nouvel opérateur pour l'exprimer. Nous avons en fait conçu la sémantique pour permettre la définition de ce genre d'opérateurs.

4. Sémantique et propriétés de ET-LOTOS

Dans cette section, nous présentons la sémantique opérationnelle de ET-LOTOS sous la forme habituelle d'axiomes et de règles d'inférence pour chaque opérateur.

Nous adoptons la présentation en deux colonnes de Moller et Tofts [Moller 90] afin de séparer clairement les transitions décrivant le passage du temps des actions standard.

4.1. Sémantique

Notations

L est l'alphabet des actions observables; i et δ représentent respectivement l'action interne et la terminaison réussie. $L^i = L \cup \{i\}$ et $L^{i,\delta} = L \cup \{i,\delta\}$. D est le domaine temporel ou encore l'alphabet des actions temporelles. $D_0 = D - \{0\}$.

$P \xrightarrow{a} P'$, où $a \in L^{i,\delta}$, signifie que P peut exécuter a et ensuite se comporter comme P'. $P \xrightarrow{a}$ signifie $\exists\, P'.\ P \xrightarrow{a} P'$ et $P \not\xrightarrow{a}$, où $a \in L^{i,\delta}$, signifie $\nexists\, P'.\ P \xrightarrow{a} P'$, c.-à-d. P ne peut pas exécuter l'action a. $P \xrightarrow{d} P'$, où $d \in D$, signifie que P peut rester inactif (c.-à-d. ne pas exécuter d'actions de $L^{i,\delta}$) pendant une période de d unités de temps et ensuite se comporter comme P'. Enfin, dans les règles d'inférence ci-dessous, d et $d_1 \in D$ et $a \in L^{i,\delta}$ par défaut d'autre mention.

	(S) $\mathrm{stop} \xrightarrow{d} \mathrm{stop}$
(AP1) $a\{d\}; P \xrightarrow{a} P$ $(a \in L^i)$	(AP2) $a\{d_1+d\}; P \xrightarrow{d} a\{d_1\}; P$ $(d_1+d \in D_0)$ (AP3) $a\{0\}; P \xrightarrow{0} \mathrm{stop}$ $(a \in L)$
(TM1) $a@t\{d\}; P \xrightarrow{a} P\,[0/t]$ $(a \in L^i)$	(TM2) $a@t\{d_1+d\}; P \xrightarrow{d} a@t\{d_1\}; P[t+d/t]$ $(d_1+d \in D_0)$ (TM3) $a@t\{0\}; P \xrightarrow{0} \mathrm{stop}$ $(a \in L)$
(D1) $\dfrac{P \xrightarrow{a} P'}{\Delta^0 P \xrightarrow{a} P'}$	(D2) $\Delta^{d_1+d} P \xrightarrow{d} \Delta^{d_1} P$ $(d_1+d \in D_0)$ (D3) $\dfrac{P \xrightarrow{d} P'}{\Delta^0 P \xrightarrow{d} P'}$
(Ex1) $\mathrm{exit}\{d\} \xrightarrow{\delta} \mathrm{stop}$	(Ex2) $\mathrm{exit}\{d_1+d\} \xrightarrow{d} \mathrm{exit}\{d_1\}$ (Ex3) $\mathrm{exit}\{0\} \xrightarrow{0} \mathrm{stop}$
(Ch1) $\dfrac{P \xrightarrow{a} P'}{P \,[]\, Q \xrightarrow{a} P'}$ (+ Ch1')	(Ch2) $\dfrac{P \xrightarrow{d} P',\ Q \xrightarrow{d} Q'}{P \,[]\, Q \xrightarrow{d} P' \,[]\, Q'}$
(PC1) $\dfrac{P \xrightarrow{a} P'}{P \,\|[\Gamma]\|\, Q \xrightarrow{a} P' \,\|[\Gamma]\|\, Q}$ $(a \in L^{i,\delta} - \Gamma)$ (+ PC1')	(PC3) $\dfrac{P \xrightarrow{d} P',\ Q \xrightarrow{d} Q'}{P \,\|[\Gamma]\|\, Q \xrightarrow{d} P' \,\|[\Gamma]\|\, Q'}$

(PC2) $\dfrac{P \xrightarrow{a} P',\ Q \xrightarrow{a} Q'}{P \,|[\Gamma]|\, Q \xrightarrow{a} P' \,|[\Gamma]|\, Q'}$ $(a \in \Gamma \cup \{\delta\})$	
(H1) $\dfrac{P \xrightarrow{a} P'}{\text{hide } \Gamma \text{ in } P \xrightarrow{a} \text{hide } \Gamma \text{ in } P'}$ $(a \in L^{i,\delta} - \Gamma)$ (H2) $\dfrac{P \xrightarrow{a} P'}{\text{hide } \Gamma \text{ in } P \xrightarrow{i} \text{hide } \Gamma \text{ in } P'}$ $(a \in \Gamma)$	(H3) $\dfrac{P \xrightarrow{d} P',\ P \not\xrightarrow{a}\ \forall\ a \in \Gamma}{\text{hide } \Gamma \text{ in } P \xrightarrow{d} \text{hide } \Gamma \text{ in } P'}$
(En1) $\dfrac{P \xrightarrow{a} P'}{P >> Q \xrightarrow{a} P' >> Q}$ $(a \in L^i)$ (En2) $\dfrac{P \xrightarrow{\delta} P'}{P >> Q \xrightarrow{i} Q}$	(En3) $\dfrac{P \xrightarrow{d} P',\ P \not\xrightarrow{\delta}}{P >> Q \xrightarrow{d} P' >> Q}$
(Di1) $\dfrac{P \xrightarrow{a} P'}{P \ [> Q \xrightarrow{a} P' \ [> Q}$ $(a \in L^i)$ (Di2) $\dfrac{Q \xrightarrow{a} Q'}{P \ [> Q \xrightarrow{a} Q'}$ (Di3) $\dfrac{P \xrightarrow{\delta} P'}{P \ [> Q \xrightarrow{\delta} P'}$	(Di4) $\dfrac{P \xrightarrow{d} P',\ Q \xrightarrow{d} Q'}{P \ [> Q \xrightarrow{d} P' \ [> Q'}$
(In1) $\dfrac{P[g_1/h_1,\ldots g_n/h_n] \xrightarrow{a} P',\ Q[h_1,\ldots h_n]:=P}{Q[g_1,\ldots g_n] \xrightarrow{a} P'}$	(In2) $\dfrac{P[g_1/h_1,\ldots g_n/h_n] \xrightarrow{d} P',\ Q[h_1,\ldots h_n]:=P}{Q[g_1,\ldots g_n] \xrightarrow{d} P'}$

Nous n'expliquerons pas en détail toutes ces règles. Notons que celles de la colonne de gauche sont en fait celles de LOTOS sauf bien sûr TM1 et D1 relatives à la mesure du temps d'attente et au délai. TM1 exprime que la variable t prend la valeur 0 lorsque l'action s'exécute. D1 indique qu'un délai de valeur nulle est sans effet sur les actions exécutables. En ce qui concerne la deuxième colonne, S signifie que `stop` est un processus qui laisse le temps s'écouler. Ch2, PC3 et Di4 imposent que le temps s'écoule de manière identique pour tout processus. AP2 indique qu'un limiteur d'offre permet au temps de s'écouler jusqu'au moment où le limiteur atteint la valeur 0. AP3 indique que si le limiteur est à 0, l'offre peut disparaître en un temps nul. TM2 et TM3 sont similaires à AP2 et AP3. On y explique comment la variable `t` de l'attribut `@t` est établie. D2 indique qu'un délai permet au temps de passer jusqu'au moment où le délai atteint 0. D3 complète D1 pour indiquer qu'un délai de valeur 0 est sans effet sur un processus. Ex2 et Ex3 sont les transpositions de AP2 et AP3 au cas de l'action δ. H3 exprime la progression maximale sur interaction cachée par le fait qu'il donne la priorité à l'exécution d'une interaction interne sur le

passage du temps. En3 est la transposition de H3 au cas de l'abstraction de δ, et rend les terminaisons réussies urgentes.

Notons que nos règles de sémantique ne nécessitent pas, comme dans [Bolognesi 92], l'ajout d'une estampille temporelle aux actions, ni de fonction auxiliaire. Le prix à payer semble être la présence de deux prémisses négatives (dans H3 et En3) mais uniquement sur des actions observables et non temporelles. Nous estimons que c'est un bon compromis car ces prémisses négatives n'induisent pas d'inconsistances (Cf. section 4.2).

4.2. Propriétés

Nous nous contentons ici de signaler les propriétés principales.

1. La sémantique de ET-LOTOS est consistante.

2. La bissimulation forte est une congruence. Notons que la bissimulation forte est définie de manière standard en considérant toutes les transitions y compris les transitions temporelles.

3. Les transitions temporelles sont déterministes. Cela signifie qu'après une transition temporelle $\xrightarrow{d}$, un processus ne peut se trouver que dans un seul état.

4. Les transitions temporelles sont fermées selon la relation ≤. De manière formelle: si $P \xrightarrow{d} P'$ alors $\forall\, d' \leq d.\ \exists\, P''.\ P \xrightarrow{d'} P''$.

5. Les transitions temporelles *ne sont pas* additives. Cela veut dire qu'en général, $P \xrightarrow{d1} P'$ et $P' \xrightarrow{d2} P''$ n'implique pas $\exists\, P''.\ P \xrightarrow{d1+d2} P''$.

L'absence d'additivité est d'ailleurs un choix délibéré qui simplifie la sémantique et induit les propriétés de persistance suivantes.

6. Lorsqu'un processus P exécute une transition temporelle $P \xrightarrow{d} P'$, alors nous sommes certains que :

a) si les actions possibles de P sont identiques aux actions possibles de P', alors aucun changement n'a eu lieu durant la transition $\xrightarrow{d}$;

b) si les actions possibles de P sont différentes des actions possibles de P', alors les changements n'ont pu se produire qu'à la fin de la transition $\xrightarrow{d}$.

Sans entrer dans les détails, ces propriétés de persistance permettent, en cas de besoin, de définir aisément des opérateurs temporels plus élaborés. Elles assurent en fait une forme d'atomicité des transitions temporelles.

5. Comparaison avec d'autres propositions

Nous nous limiterons à une synthèse des quelques comparaisons avec [Miguel 93a] et [Bolognesi 92] qui ont été présentées dans les sections 2 et 3.

Par rapport à [Miguel 93a], nous estimons que le limiteur d'offre et l'opérateur de délai de ET-LOTOS sont plus puissants que les estampilles temporelles pour exprimer des intervalles et des délais. Nous pouvons aussi plus naturellement exprimer des délais non déterministes sans avoir recours à l'opérateur `choice` généralisé. La même remarque s'applique pour la mesure du temps.

Par rapport à [Bolognesi 92], nous ne permettons pas de forcer l'occurrence d'une action observable avant ou à un moment donné (en bloquant le temps le cas

échéant). Nous estimons que c'est inutile et peu commode en pratique (Cf. l'exemple sur l'isochronisme). Ici aussi notre combinaison (limiteur d'offre, opérateur Δ) est plus souple que l'ajout d'un intervalle de temps au préfixage d'action. En ce qui concerne la mesure du temps, nous n'avons pas trouvé comment ces auteurs peuvent l'exprimer, même avec l'aide de l'opérateur `choice` généralisé. Enfin, nous avons vu que notre hypothèse de progression maximale est en théorie moins générale que la solution proposée par [Bolognesi 92], mais que cela ne nous préoccupe pas pour deux raisons. La première est qu'en pratique cette expressivité additionnelle ne semble pas nécessaire. La deuxième est que, si d'aventure elle s'avérait utile, nous avons conçu la sémantique de ET-LOTOS pour pouvoir facilement ajouter l'opérateur manquant (Cf. propriété 6 de la section 4.2).

6. Conclusion

Le langage ET-LOTOS que nous avons présenté nous semble prêt à affronter la réalité des systèmes temps réels. En plus des nombreux exemples d'applications que nous avons proposés dans cet article, nous avons spécifié le protocole TIC-TAC [Léonard 93a] qui s'inspire de l'ATM et des applications multimédia. Le protocole présenté dans [Regan 93] ne semble pas poser de problème à ce stade de notre étude. Enfin, nous avons montré les points forts de notre proposition par rapport à d'autres extensions de LOTOS.

Remerciements

Nous remercions T. Bolognesi, C. Pecheur, J. Quemada et les chercheurs du LAAS-CNRS (Toulouse) pour les discussions très intéressantes que nous avons eues au sujet de ET-LOTOS et de ses prédécesseurs.

Bibliographie

[Baguette 92] Y. Baguette, L. Léonard, G. Leduc, A. Danthine, O. Bonaventure, "OSI95 Enhanced Transport Service", Rept. No. OSI95/Deliverable ULg-A/ P/V1, Université de Liège, B28, B-4000 Liège, Belgium, Dec. 1992.

[Bolognesi 92] T. Bolognesi, F. Lucidi, "LOTOS-like process algebras with urgent or timed interactions", K. Parker, G. Rose, eds., Formal Description Techniques, IV (North-Holland, Amsterdam, 1992) 249-264.

[Hansson 90] H. Hansson, B. Jonsson, "A calculus for communicating systems with time and probabilities", 11th IEEE Real-Time Systems Symposium, Orlando, Florida, 1990, IEEE Computer Society Press

[Hennessy 90] M. Hennessy, T. Regan, "A temporal process algebra", J. Quemada, J. Mañas, E. Vazquez, eds., Formal Description Techniques, III (North-Holland, Amsterdam, 1991) 33-48.

[ISO 8807] ISO/IEC-JTC1/SC21/WG1/FDT/C, "IPS - OSI - LOTOS, a Formal Description Technique Based on the Temporal Ordering of Observational Behaviour", IS 8807, February 1989.

[Leduc 93] G. Leduc, L. Léonard, "A timed LOTOS supporting a dense time domain and including new timed operators", M. Diaz, R. Groz, eds., FORTE '92, Perros-Guirec, France, Oct. 92 (North-Holland, Amsterdam, 1993) 87-102.

[Léonard 93a] L. Léonard, G. Leduc, A. Danthine, "The Tick-Tock case study for the assessment of Timed FDTs", The OSI95 project (Springer-Verlag, 1993), à paraître.

[Léonard 93b] L. Léonard, G. Leduc, "An Enhanced Version of Timed LOTOS and its Application to a Case Study", Rept. No. SART 93/09/13, Université de Liège, B28, B-4000 Liège, Belgium, May 1993.

[Merlin 76] P.M. Merlin, D.J. Farber, "Recoverability of Communication Protocols - Implications of a theoretical Study", IEEE Transaction on Communication, 24, Sept. 76, 1036-1043.

[Miguel 93a] C. Miguel, A. Fernandez, L. Vidaller, "Extending LOTOS towards performance evaluation", M. Diaz, R. Groz, eds., Formal Description Techniques, V (North-Holland, Amsterdam, 1993) 103-118.

[Miguel 93b] C. Miguel, A. Fernandez, L. Vidaller, "Assessment of Extended LOTOS", The OSI95 project (Springer-Verlag, 1993), à paraître.

[Moller 90] F.Moller, C. Tofts, "A temporal calculus of communicating systems", J.C.M. Baeten, J.W. Klop, eds., CONCUR '90, Theories of Concurrency: Unification and Extension, LNCS 458 (Springer - Verlag, Berlin, 1990) 401-415.

[Nicollin 90] X. Nicollin, J.-L. Richier, J. Sifakis, J. Voiron, "ATP : An algebra for timed processes", M. Broy, C.B. Jones, eds., IFIP Working Conference on Programming Concepts and Methods, Sea of Gallilee, Israel (North-Holland, Amsterdam, 1990).

[Nicollin 91a] X. Nicollin, J. Sifakis, "An Overview and Synthesis on Timed Process Algebras", K.G. Larsen, A. Skou, eds., Computer-Aided Verification, III (LNCS 575, Springer-Verlag, Berlin, 1992) 376-398. Aussi dans: LNCS 600.

[Quemada 92] J. Quemada, D. de Frutos, A. Azcorra, "TIC: A TImed Calculus", Formal Aspects of Computing (1992) 3.

[Reed 88] G.M.Reed, A.W. Roscoe, "A Timed Model for Communicating Sequential Processes", Theoretical Computer Science 58 (1988) 249 - 261 (North-Holland, Amsterdam).

[Regan 93] T. Regan, "Multimedia in Temporal LOTOS: a Lip-Synchronization Algorithm", à paraître dans A. Danthine, G. Leduc, P. Wolper, eds., Protocol Specification, Testing and Verification, XIII (North-Holland, Amsterdam, 1993).

[Wang 91] Y. Wang, "CCS + Time = an Interleaving Model for Real Time Systems", J. Leach Albert, B. Mounier, M. Rodríguez Artalego, eds., Automata, Languages and Programming, 18 (LNCS 510, Springer-Verlag, Berlin, 1991) 217-228.

Guy Leduc est ingénieur civil électricien (électronique, 1983), et agrégé de l'enseignement supérieur en Sciences Appliquées (1991) de l'Université de Liège. Depuis 1983, il travaille pour le Fonds National belge de la Recherche Scientifique (F.N.R.S.) à l'Université de Liège, dans le service du Professeur A. Danthine, où il occupe depuis 1990 un poste de chercheur qualifié. Ses activités et intérêts de recherche incluent les techniques formelles de description (FDT) et les réseaux de communication. Depuis 1985, ses recherches se sont focalisées sur LOTOS, et plus particulièrement sur les relations d'implémentation, la théorie du test, la recherche d'une sémantique dénotationelle et d'une extension temporelle du langage. En plus de ces travaux théoriques, il a appliqué LOTOS dans divers projets européens, tels que OSI95 sur la conception de protocoles à hautes performances.

Luc Léonard est ingénieur civil électricien (informatique), sorti de l'Université de Liège en 1991. Depuis cette date, il travaille comme Aspirant du F.N.R.S dans le service de Systèmes et Automatique du professeur A. Danthine. Son intérêt se porte sur la Technique de Description Formelle LOTOS, et plus particulièrement sur les possibilités d'extension temporelle. Il a aussi participé au projet OSI95, au cours duquel il a eu l'occasion d'appliquer LOTOS a la spécification du service fourni par le protocole de transport TPX.