Une nouvelle méthode pour la spécification en LOTOS

Pascal Poizat, Christine Choppy & Jean-Claude Royer
IRIN
Université de Nantes & Ecole Centrale
2 rue de la Houssinière
B.P. 92208
44322 Nantes cedex 3

7 février 1998
révisé le 9 avril 1998
révisé le 23 juillet 1998

***Résumé**

Nous présentons une nouvelle méthode pour la spécification en LOTOS. Notre méthode présente deux aspects : elle est orientée contraintes pour ce qui concerne la décomposition d'un processus en sous-processus séquentiels, et elle est orientée états abstraits pour la spécification des composants séquentiels. Ce second aspect est basé sur la construction d'un automate à partir de la description du comportement externe du processus, puis sur l'extraction d'une spécification LOTOS correspondant à cet automate. Nous illustrons notre méthode à travers un exemple simple, celui d'un hôpital.

Mots-clés : LOTOS, méthode, spécification, orienté contraintes, orienté états, automate

***Abstract**

We present a new method for LOTOS specification. Our method is both constraint oriented (for the processes decomposition into sequential sub-processes) and state oriented (for the design of the sequential components). This latter aspect is based on (i) the design of an automaton from the external behavior description, (ii) the generation of a LOTOS specification associated with this automaton. We illustrate our method through a simple example, a hospital.

Key-words: LOTOS, method, specification, constraint oriented, state oriented, automaton

Table des matières

1 Introduction

L'emploi de méthodes formelles de spécification est d'une grande importance pour le développement de systèmes logiciels et entre autres lors de la conception de systèmes distribués. En effet, elles permettent une description non ambiguë, précise, complète et indépendante de toute implémentation. La possibilité de pratiquer des tests, vérifications et validations sur ces spécifications est aussi un des points qui concourent à leur utilisation de plus en plus fréquente.

Si l'importance de l'utilisation de spécifications formelles lors du développement de systèmes logiciels est largement accepté, il existe encore un besoin certain de méthodes permettant de les appliquer. En cela, nous aimerions citer [AR97] : "a formalism does not provide a method by fiat". Il semble encore plus nécessaire de fournir une méthode pour les spécifications qui prennent en compte à la fois la spécification de l'activité concurrente et celle des types de donnée impliqués dans cette activité.

LOTOS est l'un des langages de spécification de systèmes distribués normalisés par l'ISO. Nous présentons ici une nouvelle méthode de conception des spécifications LOTOS. Cette méthode se base à la fois sur une approche orientée contraintes pour la décomposition des processus en sous-processus communiquants et une approche orientée états pour la description du comportement des processus séquentiels.

Dans la première partie nous présentons LOTOS qui nous sert de langage cible de spécification. La partie statique et la partie dynamique sont introduites puis un panorama des méthodes de conception LOTOS existantes est fait.

Dans la seconde partie, notre méthode est présentée sur un exemple simple, celui d'un hôpital. Le point principal de cette méthode est la construction semi-automatique de l'automate du comportement interne des processus séquentiels. Deux possibilités de création de spécifications LOTOS à partir de cet automate sont aussi expliquées. Enfin, le problème de la parallélisation de certains comportements purement séquentiels est abordé.

2 Présentation de LOTOS

Dans cette section nous faisons une brève présentation de LOTOS.

2.1 Introduction à LOTOS

LOTOS [ISO89b](Language Of Temporal Ordering Specification) est l'une des trois techniques de description formelles normalisées par l'ISO pour la spécification formelle des systèmes ouverts distribués et en particulier les protocoles de communication. Les autres techniques sont Estelle (Extended Finite State Machine Language) [ISO89a] et SDL (Specification and Description Language) [Tur93].

L'idée de départ de LOTOS est que les systèmes peuvent être spécifiés en définissant les relations temporelles entre les interactions qui constituent le comportement externe observable d'un système [BB88].

La partie de LOTOS qui traite des comportements des processus et de leurs interactions est basée sur les algèbres de processus et s'inspire en cela fortement de CCS [Mil80] et de CSP [Hoa78]. La seconde partie de LOTOS traite des données internes aux processus et des traitements qui s'y rattachent et s'appuie sur ACT ONE [EM85].

Les objectifs de LOTOS sont ceux des spécifications formelles en général : définition de descriptions non ambiguës, précises, complètes, indépendantes de toute implémentation, lisibles et bien définies formellement (dans l'idée de les vérifier, les valider et de tester le bien fondé d'implémentations).

2.2 Partie statique

La partie statique concerne la définition et la manipulation des données. [Poi91] précise qu'en LOTOS les données sont attachées à la notion de portes de communication (voir plus bas). Le formalisme est celui des types abstraits algébriques, inspiré d'ACT ONE [EM85]. Pour [Gar89], LOTOS évite ainsi les problèmes rencontrés en Estelle [ISO89a] où les données sont spécifiées au moyen des types Pascal. La sémantique est donnée par réécriture de termes [Led87].

LOTOS permet de définir des types directement mais aussi en en important d'autres (utilisation de la bibliothèque LOTOS) ou en en combinant d'autres (notons que [Gar89] appelle cette combinaison de l'héritage ; cela est différent de la notion d'héritage en objet[1]). La généricité est permise par l'emploi de types paramétrés par des sortes, des opérateurs et des équations formels. Ces types formels peuvent alors être instanciés (actualisés). L'instanciation peut avoir lieu lors de l'instanciation des processus (voir partie dynamique). Cela est d'ailleurs préalable à toute utilisation dans les expressions comportementales. Notons enfin qu'il est possible de renommer les types (sortes et opérations du type). Cela permet une forme de réutilisation.

LOTOS est fortement typé [Gar89].

Dans le cas de conflits de noms liés à l'importation, l'"héritage" ou le renommage, la résolution se fait en fonction des sortes des arguments et du résultat. En cas d'ambiguïté, on a recours aux règles de sémantique statique de LOTOS.

1. Il n'y a pas par exemple de notion de sous typage entre les sortes ou de polymorphisme. Il est aussi possible de définir des sortes dérivées mais elles ne sont pas compatibles entre elles.

2.3 Partie dynamique

La partie dynamique concerne les structures de contrôle. Comme en CCS [Mil80] et CSP [Hoa78], le contrôle en LOTOS est décrit syntaxiquement par les termes d'une algèbre ([Gar90]) appelés expressions comportementales (on dit aussi comportement). La synchronisation et la communication s'effectuent exclusivement par rendez-vous, sans partage de mémoire et via des portes. Sémantiquement parlant tout comportement représente un automate d'états finis ou infinis [Gar90].

[LFHH92] précise que la synchronisation en LOTOS est :

1. *multi directionnelle* : n processus peuvent participer au rendez-vous. Or, dans les articles sur LOTOS, tous les opérateurs sont toujours décrits par une sémantique binaire. Les opérateurs sont souvent associatifs (en particulier sur une même porte).

2. *symétrique* : tous les processus ont la même importance lors d'un rendez-vous. Il n'y a pas un initiateur et des répondeurs. C'est différent de la sémantique de l'envoi de message en objet. Il semble toutefois possible de faire une analogie entre l'envoi de message et la réception de valeur (via une porte de communication au niveau de l'objet client) d'une part et entre la réception de message et l'émission de valeur (via une porte de communication au niveau du receveur) d'autre part. Dans l'exemple de la figure 1, l'objet A envoie un message m à l'objet B (figure 1a). L'analogie consiste à considérer qu'il y a communication par une porte m entre un processus PA représentant A et un processus PB représentant B : PA recevant une valeur via m et PB envoyant la valeur correspondante (figure 1b).

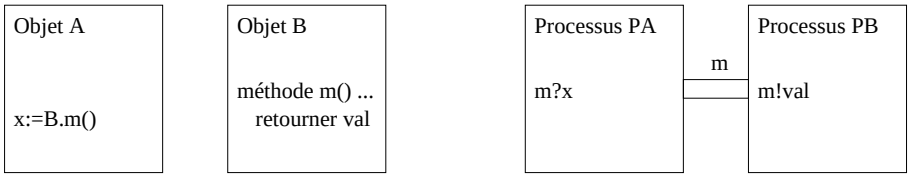


Figure 1 : Analogie entre sémantique objet et processus LOTOS

3. *anonyme* : la synchronisation est proposée à l'environnement. Il est impossible de s'adresser à un processus particulier. Remarque : il est possible d'y remédier en donnant des identifiants aux processus et en communiquant ces valeurs lors du rendez-vous. L'utilisation de l'opérateur de masquage permet aussi d'y remédier.

4. *non déterministe* : quand plus d'une synchronisation est possible, l'une d'entre elles est choisie de façon non déterministe.

Les portes sont des canaux de communication permettant le rendez-vous entre n processus (ou taches) se déroulant en parallèle. Il existe deux portes spéciales : `i` (opération interne au processus) et δ (sert à la définition sémantique de LOTOS dans le cas de la terminaison des processus ; elle ne doit pas être utilisée dans une spécification).

Les événements sur ces canaux sont atomiques et constituent les éléments de base de la synchronisation. On parle d'actions[2] observables. L'ensemble des processus avec lequel un processus communique constitue son environnement. Un processus particulier, l'observateur humain, peut éventuellement faire partie de cet environnement [BB88].

Les portes peuvent être accompagnées d'offres (émission (`!Valeur`) ou réception (`?Variable:Sorte`) de valeur, choix aléatoire d'une valeur, correspondance de valeurs). Des gardes permettent de restreindre les valeurs possibles dans le cas du choix aléatoire, ou de contrôler la valeur reçue dans le cas d'une réception. Par exemple, dans l'expression suivante : `échange!Put?Get:NAT [Get>0]`,

2. Quand cette action concerne n processus, on parle plutôt d'interaction [BB88].

la valeur `Put` est émise lors du rendez-vous pendant qu'une valeur de type NAT (qui doit être supérieure à zéro) est reçue dans la variable `Get`.

Le tableau 1 est extrait de [Gar89].

offre 1	offre 2	condition	action
$!V_1$	$!V_2$	$V_1 = V_2$	
$!V_1$	$?X_2 : S_2$	$V_1 \in domain(S_2)$	$X_2 := V_1$
$?X_1 : S_1$	$!V_2$	$V_2 \in domain(S_1)$	$X_1 := V_2$
$?X_1 : S_1$	$?X_2 : S_2$	$S_1 = S_2$	$X_1, X_2 := VtqV \in domain(S_1)$

TAB. 1 – *Correspondance entre offres*

Dans le cas de rendez-vous n-aires, les offres ne peuvent s'unifier que si l'intersection des ensembles de valeurs permis par les offres est non vide.

Dans le cas d'offres n-aires (ex: `p !`e_1 `?x:S !`e_2), il faut en plus que les attributs soient en nombre égal et compatibles. Une offre est toujours atomique, même quand elle comporte plusieurs attributs [Led87].

Des opérateurs permettent de composer les expressions comportementales pour en obtenir de nouvelles (les expressions ont une définition récursive). [BB88, Poi91] utilisent les transitions étiquetées pour donner la sémantique (dénotationnelle) de LOTOS.

Cette sémantique donne un moyen de dériver automatiquement les expressions comportementales. Il peut y avoir non déterminisme dans la réduction des transitions étiquetées, cependant dans de nombreux cas cet indéterminisme est réduit par l'interaction du processus avec son environnement.

Remarque : dans ce qui suit, nous faisons parfois l'analogie entre un processus et son comportement.

Les opérateurs de composition de LOTOS sont :

- `stop` : comportement inactif (blocage)
- `exit` : dénote un processus qui se termine normalement (contrairement à `stop`). `exit` a comme sémantique le franchissement d'une porte δ :

  `exit` $\equiv \delta$`;stop`

- `;` : `p;E` rendez-vous sur la porte p en préalable à l'expression comportementale E. On dit aussi qu'on a une transition étiquetée par la porte p. Si p est `i`, l'action n'est pas observable ; pour toute autre porte elle l'est.
- `[]` : choix non déterministe. L'un des comportements se déclenchera en fonction de l'état des communications sur leurs portes associées.

  Remarque : noter la différence entre `a;b;stop [] a;c;stop` et `a;(b;stop [] c;stop)`. Dans les deux cas il y a indéterminisme, mais dans le premier cas, il est indépendant de l'environnement : une fois entré dans une branche, il n'y a plus de choix d'interaction sur b ou c selon la branche choisie.

- `|[liste de portes]|` : composition parallèle. Les processus se synchronisent sur certaines portes. Les transitions étiquetées par des portes n'appartenant pas à la liste se font de manière asynchrone. Deux opérateurs complémentaires existent : `||` (la liste des portes contient toutes les portes ; les processus sont entièrement synchronisés) et `|||` (la liste des portes ne contient que δ ; il y a entrelacement (interleaving) ; ils ne se synchronisent que sur leur terminaison).

  Remarque : l'opérateur `||` peut provoquer un blocage dans certains cas, par exemple :

  `a;c;exit || a;d;exit`

puisque soit c soit d n'est pas dans une des branches du `||`. Donc le comportement est équivalent à `a;stop`.

- [> : permet de spécifier l'interruption d'un comportement par un autre. Lorsque le second débute (passe sa première porte), le comportement de la composition des deux processus est équivalent à celui du second processus. Ceci est vrai tant que le premier comportement n'a pas atteint une porte δ. S'il se bloque avant, le second comportement est inévitablement exécuté.

- `hide` : cet opérateur permet de cacher certaines portes d'un comportement et ainsi de gagner en abstraction. En général, cela sert quand on a un processus "boite noire" composé d'un certain nombre de sous-processus communiquant par des portes qui ne doivent pas être connues/vues de l'extérieur.

- $\rightarrow$: cet opérateur permet de conditionner un comportement par une garde.

  Remarque : dans le cas de gardes non exclusives, il peut y avoir non déterminisme. Quand aucune garde n'est évaluée à vrai le processus est équivalent à `stop`.

  Ainsi : `P` $\equiv$ `p?x:T;([x>0]` $\rightarrow$ `p!f(x) [] [x<0]` $\rightarrow$ `p!g(x))` $\equiv$ `stop` dans le cas où $x = 0$.

- *les prédicats de sélection* permettent d'imposer des restrictions sur les valeurs échangées lors d'une communication.

  Exemple : `porte1?x:NAT[x<3];porte2!x`

- `let` : bloc lexical habituel pour assigner des valeurs aux variables (locales).

Au niveau de la nécessité ou non, pour deux processus, de se synchroniser sur une porte, voir le tableau 2, extrait de [Gar89].

portes	$\|\|\|$	$\|[liste]\|$	$\|\|$
δ	oui	oui	oui
$p \in liste$	non	oui	oui
$p \notin liste$	non	non	oui
i	non	non	non

TAB. 2 – *Composition parallèle et synchronisation*

Il faut aussi, pour être exhaustif, noter l'existence de deux opérateurs permettant d'écrire l'un le choix non déterministe sous forme concise (`choice`), et l'autre adapté à la composition parallèle (`par`). On peut se reporter à [Gar89] par exemple pour la syntaxe et la sémantique exacte.

Les processus, lorsqu'ils se terminent ont la possibilité de transmettre des résultats (fonctionnalité). Cela correspond à l'ajout d'offres sur la porte δ. Il doit donc y avoir correspondance entre les types, voire les valeurs de deux processus composés en parallèle (voir [Poi91]).

La fonctionnalité `noexit` dénote la fonctionnalité d'un processus ne terminant pas (`stop`). Remarque : le fait qu'un processus ait la fonctionnalité `exit` n'assure pas sa terminaison. C'est une condition nécessaire mais non suffisante. Le comportement résultant de la composition parallèle de deux sous-comportements ne se termine donc correctement que si l'un des deux a la fonctionnalité `noexit` (ne termine pas), ou si tous les deux terminent par `exit` en offrant sur δ des offres compatibles. On dit que les fonctionnalités des comportements composés en parallèle doivent être compatibles.

Il est possible de terminer avec une offre qualifiée "`any`". Le `any` au niveau de la fonctionnalité d'un processus (porte δ) a le rôle du `?` (dans `p?x:T` par exemple) au niveau des offres sur les portes (autres que δ). `any S` permet d'indiquer qu'un processus peut retourner n'importe quelle valeur du domaine de S. La valeur effective provient donc soit de l'indéterminisme (valeur prise au hasard

dans le domaine de S), soit d'un processus composé en parallèle qui retourne une valeur du type correspondant (voir le tableau 1).

L'opérateur » permet à un processus de se déclencher après la terminaison réussie d'un autre. Il est possible de passer des valeurs en résultat de l'un à l'autre. La fonctionnalité du premier doit être compatible avec l'attente déclarée du second. Cette communication se fait via la porte δ et est cachée aux autres processus.

En LOTOS, les comportements peuvent être paramétrés par des portes et/ou des variables formelles (limitation au premier ordre). Des processus peuvent ensuite en appeler d'autres en instanciant les paramètres formels. La récursion est autorisée. Notons que les sortes des paramètres doivent correspondre.

Remarque : il n'y a ni héritage, ni sous-typage autorisé entre les sortes. Ni au niveau de l'opérateur » ni à celui de l'instanciation.

2.4 Propriétés

[BB88] introduit l'intérêt de pouvoir comparer deux spécifications LOTOS. Il s'agit de comparer deux spécifications, éventuellement à deux niveaux d'abstraction (respectivement d'implémentation) d'un problème donné. L'approche consistant à comparer les spécifications de deux comportements via une relation d'équivalence est bien adaptée à LOTOS et à son but initial (modélisation et spécification des comportements des systèmes ouverts distribués). Notons que les problèmes de construction, raffinage ou comparaison de spécifications LOTOS interviennent aussi au niveau de la méthodologie de conception (voir plus bas).

[Gar90] cite plusieurs approches pour vérifier les spécifications LOTOS : transformation de programmes par application de lois algébriques, preuve de propriétés à l'aide de démonstrateurs de théorèmes, traduction de LOTOS vers les réseaux de Pétri et/ou les automates d'états finis sur lesquels on peut ensuite évaluer les propriétés voulues.

Équivalence observationnelle

CCS [Mil80] a introduit l'idée d'*équivalence observationnelle*. Elle est basée sur l'idée que le comportement d'un système est déterminé par la façon dont il interagit avec l'environnement externe.

L'idée, en LOTOS, est que cette interaction est représentée par les actions du processus sur ses canaux de communication (portes). Il s'agit donc de construire un arbre d'actions du processus, avec les noeuds symbolisant des *états* du processus (au sens ensemble de *portes* où le processus est capable de communiquer à un instant t) et les branches étiquetées par les *communications* faites pour passer d'un état à un autre. Il s'agit ensuite de prouver l'équivalence de deux arbres.

Remarque : [BB88, Poi91] utilisent des arbres infinis pour représenter le comportement de processus récursifs. Il semble plus pratique d'utiliser des graphes ou automates d'états finis. Cette approche est d'ailleurs à mettre en relation avec la construction d'un automate dans la démarche de conception de spécifications LOTOS (voir [GR90] ou plus bas).

[BB88, Led87] introduisent d'abord l'idée de *bissimulation*. Il existe une bissimulation entre deux arbres de comportements si et seulement s'ils acceptent les mêmes séquences d'événements (aux `i` près) et restent équivalents après avoir chacun exécuté la même séquence. Il y a équivalence observationnelle entre deux comportements s'il existe une bissimulation entre leurs deux arbres.

La définition d'une *congruence observationnelle* permet d'obtenir des lois utiles. Ces lois rendent possible une forme de réécriture de graphes. Cette réécriture (règles exprimées sous forme graphique) permet aussi de prouver l'équivalence de deux comportements. Selon [Led87] pour obtenir une congruence il est nécessaire de restreindre la bissimulation.

L'équivalence de deux comportements exige la découverte d'une bissimulation, ce qui n'est concevable que pour des comportements simples car il n'existe pas de méthode systématique. Le problème est encore plus complexe pour les comportements récursifs [Led87].

Des outils comme Caesar/Aldebaran [Fer88] permettent (par extraction et animation de graphe) d'étudier l'équivalence observationnelle[3] (voir [Gar89]).

Équivalence de test

Dans le cas où l'on cherche à comparer deux spécifications LOTOS pour montrer que l'une est une implémentation de l'autre il faut disposer d'autres relations que l'équivalence observationnelle. Il faut une relation asymétrique. La *relation de réduction-extension* ([BB88, Led87] l'appellent *relation d'implémentation*) est asymétrique (contrairement à l'équivalence vue plus haut). L'idée en est qu'un comportement B réduit une spécification S ($B\,red\,S$) si B ne peut exécuter que des actions que S exécute et si B ne peut refuser que des actions que S refuse. Il s'agit d'actions observables (le cas de i n'est pas traité).

Il est possible d'obtenir une équivalence entre deux comportements A et B en utilisant cette relation. A et B sont équivalents si $A\,red\,B$ et $B\,red\,A$. Cette équivalence est d'ailleurs plus puissante que l'équivalence observationnelle car elle ne différencie pas deux processus indifférentiables d'un point de vue opérationnel (contrairement à l'équivalence observationnelle). [BB88] prend le cas de `a;(a;a;stop[]a;stop)` et `a;a;a;stop[]a;a;stop`, non équivalents d'un point de vue observationnel mais équivalents d'un point de vue opérationnel. Pour cela, cette équivalence est nommée équivalence de test (ils répondent aux mêmes séquences de test).

Plus de détails sur ces relations asymétriques, l'équivalence de test et les références s'y rapportant peuvent être trouvés dans [BB88].

Remarque : l'étude des relations d'équivalence est en général présentée avant la partie statique dans les articles traitant de LOTOS. Pourtant l'ajout d'offres sur les portes complique la notion d'équivalence de deux actions observables. Deux offres sont équivalentes si elles sont compatibles.

Notons pour terminer que comme le fait remarquer [Led87] ces principes d'équivalence ne sont valables qu'en l'absence de la partie statique (ACT ONE).

Contraintes temporelles en LOTOS

La spécification asynchrone impose des contraintes supplémentaires d'ordonnancement des processus destinées à éviter l'interblocage ([Gar89]).

Les contraintes temporelles ne peuvent pas s'exprimer simplement en LOTOS. LOTOS ne traite que l'ordre relatif des événements. Il n'y a pas de temps universel. Il est toutefois possible de remplacer une contrainte de délai par un événement interne, `i`. [Gar89] note que, dans ce cas, le nouveau comportement obtenu est un sur-ensemble de l'ancien. D'autre part, les bons algorithmes ne dépendent pas des valeurs des délais et conservent donc leurs bonnes propriétés (pas de blocages, pas de famine, ...). La modélisation d'un délai peut cependant modifier le comportement du système (voir [Gar89]).

Remarque : il semble possible d'exprimer les délais par l'interaction avec un processus Horloge (figure 2) en parallèle et l'opérateur [> pour réaliser la temporisation d'un processus. De toutes façons, le fait de modéliser les délais dans l'horloge semble continuer à obliger d'utiliser `i` dans ce processus. L'idée d'un processus Horloge commun est d'ailleurs reprise dans [Cla92].

3. D'autres algorithmes et d'autres moyens pour prouver l'équivalence de deux comportements sont aussi présentés dans [BB88].

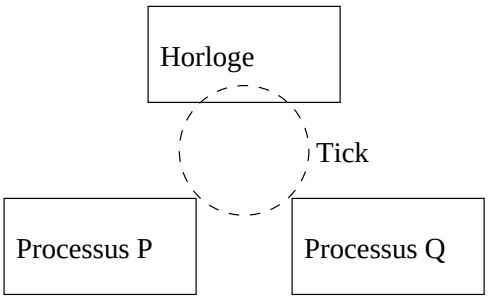


Figure 2 : Proposition de modélisation des délais

2.5 Méthodes de conception des spécifications

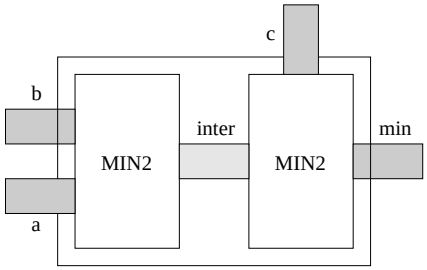
Une spécification LOTOS consiste en des définitions de types, de processus (éventuellement avec paramètres formels) et d'une expression comportementale (éventuellement avec instanciation de paramètres formels).

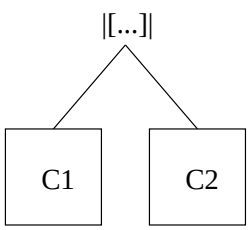
Peu d'articles parlent d'une méthode de conception des spécifications LOTOS. Il existe deux grandes tendances : orienté contraintes et orienté états.

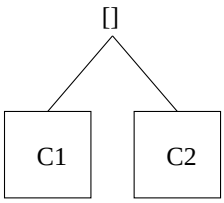
[Poi91] donne des contextes d'utilisation de chaque opérateur de LOTOS. Des liens entre opérateurs et concepts de programmation concurrente sont faits.

[BB88] présente l'approche *orientée contraintes*. Il s'agit d'une méthode de type *diviser pour régner* : les processus sont composés en parallèle, le processus résultant ayant un ensemble de contraintes égal à l'union des contraintes de synchronisation des deux sous-processus, ensemble auquel il faut rajouter les contraintes de synchronisation entre ces deux sous-processus. Ceci permet [LFHH92] de structurer la spécification en termes de décomposition du problème en plusieurs processus et de spécifier séparément les différents sous-processus.

Ces approches orientées contraintes (figure 3), utilisées essentiellement pour la spécification de protocoles de communication, privilégient souvent l'aspect dynamique par rapport à l'aspect statique (souvent les processus n'ont pas de partie donnée correspondante). Cependant, leur point fort est que la décomposition des processus en composants séquentiels est assez aisée.







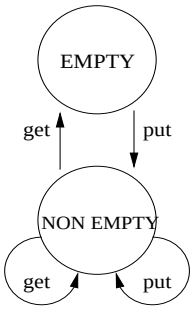

```
process MIN3[a,b,c,min]: noexit:=
   hide inter in
       ( MIN2[a,b,inter]
   |[inter]|
       MIN2[c,inter,min] )
endproc
```

```
process MIN2 [in1,in2,out]: noexit:=
       in1?x1:NAT; in2?x2:NAT;
         out!min(x1,x2); stop
   []
       in2?x2:NAT; in1?x1:NAT;
         out!min(x1,x2); stop
endproc
```

Figure 3 : Approche orientée contraintes

[GR90] tente, sur l'exemple des matrices d'interrupteurs[4], de donner une méthode de spécification. Les auteurs notent cependant que LOTOS est suffisamment expressif pour autoriser un grand éventail de styles de spécification. [BB88] précise que LOTOS permet de plus la spécification à différents niveaux d'abstraction et tend à relativiser les différences entre spécification et implémentation. Ce ne sont pas des spécificités de LOTOS.

[Led87] propose l'approche hybride consistant à définir un objet sous forme d'un processus gérant un type de donnée. Nous reprenons cette idée en section 3. C'est l'approche *orientée états* (figure 4).




```
process BUFFER[put,get](b:Buffer):noexit:=
   EMPTY[put,get](b)
endproc
process EMPTY[put,get](b:Buffer):noexit:=
   put?x:T;NON-EMPTY[put,get](push(b,x))
endproc
process NON-EMPTY[put,get](b:Buffer):noexit:=
   [nb(b)=1]→get!top(b);
      EMPTY[put,get](pop(b))
[] [nb(b)>1]→get!top(b);
      NON-EMPTY[put,get](pop(b))
[] put?x:T;NON-EMPTY[put,get](push(b,x))
endproc
```

Figure 4 : Approche orientée états

[Cla92] donne une méthode mixte de conception de systèmes embarqués. La spécification est basée sur la notion d'objets concurrents communiquants. Tout objet est un processus ayant un paramètre indiquant son état interne (caché). Cet objet communique avec d'autres objets vus comme son environnement. La méthode s'appuie aussi sur la composition parallèle de processus dont le comportement correspond à un ensemble de contraintes et sur l'utilisation de *cycles* pour la définition du comportement des sous-processus les plus simples. Un cycle correspond à un comportement qui peut rendre un service particulier (une ou plusieurs communications séquentielles) puis qui se rappelle récursivement. [Cla92] explique aussi que spécifier l'interaction des sous-processus avec leur environnement permet de tester ces processus en tant que prototypes.

4. Il faut noter ici que de nombreux exemples servant à expliquer LOTOS sont connexes au domaine des télécommunications.

[Tur93] explique que du point de vue conception, les trois styles les plus pertinents sont le style orienté contraintes, le style orienté états (appelé style orienté automate) et le style orienté ressources. Pour les auteurs, les spécifications sont écrites dans la majorité des cas en utilisant un style mixte et le point important consiste à trouver un bon équilibre entre les styles possibles de façon à obtenir une spécification bien structurée.

Plusieurs indications sont données :

- utilisation du style orienté contraintes pour représenter les systèmes comme des *boites noires* sans considérer leur structure interne. Ce style est recommandé pour spécifier à un haut niveau d'abstraction.
- utilisation du style orienté états comme modèle d'implémentation pour dériver des implémentations efficaces (rapides).
- utilisation du style orienté ressources pour spécifier les systèmes où les sous parties sont bien définies et où la communication entre ces sous parties est cachée (style appelé *boite blanche* dans [Tur93]).

En ce qui concerne la partie algébrique, les auteurs proposent une discipline par constructeurs.
Pour le reste, la méthode est articulée autour de trois phases : conception des spécifications, vérification puis implémentation et prototypage.

L'ISO a défini des orientations pour l'utilisation de LOTOS. [VSVSB91] précise tout d'abord que trois principes doivent être respectés lors de la conception de systèmes distribués :

- orthogonalité : les besoins indépendants doivent être spécifiés indépendamment
- généralité : les définitions génériques et paramétriques doivent être préférées aux définitions répondant à un problème particulier
- flexibilité : les conceptions doivent pouvoir être mises à jour, c'est à dire étendues et modifiées.

Les auteurs préconisent de penser d'abord le système de façon abstraite et en fonction des besoins (description *extensionnelle*). Deux styles de spécification correspondent à cette attente :

- monolithique : seules les différentes suites d'actions possibles sont étudiées. Ce style est très indépendant de toute implémentation mais il manque de structure et est peu compréhensible.
- orienté contraintes : les actions observables (et elles seules) sont présentées selon un ordonnancement temporel défini comme la conjonction de contraintes. Ce style répond bien aux trois principes cités plus haut.

Dans un second temps, le spécifieur pourra s'attacher à la façon dont le système répond aux besoins, c'est à dire sa structure (interne) en termes de sous parties qui interagissent ou en termes d'états abstraits. Les deux styles suivants permettent cette description *intentionnelle* :

- orienté états : le système est vu comme une ressource à état interne explicite variant avec les interactions. Ce style est utile en phase finale de conception, quand la ressource peut être vue comme un objet séquentiel.
- orienté ressources : les interactions internes et observables sont présentées. Le comportement est vu comme une communication cachée entre plusieurs sous systèmes composés. Contrairement à l'approche orientée contraintes, la décomposition se fait ici en fonction de la ressource (type) et pas des contraintes entre interactions. Les sous systèmes sont spécifiés dans n'importe lequel des styles existants.

[Gar89, Gar90] citent comme autres styles possibles :

- impérative

– logique et algébrique, en utilisant la partie données

– fonctionnelle, en modélisant les processus comme des fonctions et en n'utilisant ni portes ni rendez-vous (pas d'opérateurs de composition parallèle entre autres)

– programmation parallèle pure

– programmation "objet" (structurée) : intégration des concepts d'objet et de comportement/processus et d'interface avec les portes sur lesquelles s'effectuent les rendez-vous d'un processus.

  Remarque : on n'a pas ici de mécanisme d'héritage proprement dit, uniquement renommage, généricité et importation.

Remarque : [BB88] pose aussi dans sa conclusion le problème de la possibilité de modéliser certains composants de système entièrement d'une façon ou d'une autre et explique que c'est avant tout un choix de style.

Nous avons analysé la méthode[5] de [GR90] et nous l'avons schématisée sur la figure 5.

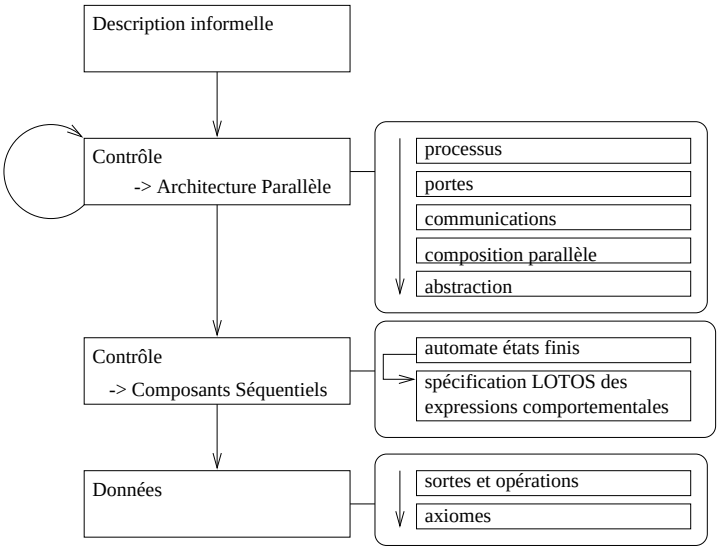


Figure 5 : Méthode de spécification LOTOS [GR90]

Les phases de cette méthode sont décrites dans ce qui suit.

Description informelle

Il s'agit ici de décrire en langage naturel le système à spécifier.

Architecture parallèle

Remarque : le terme activité concurrente semble plus approprié qu'architecture parallèle car les termes architecture et parallèle sont en général réservé à la partie matérielle. Il s'agit de trouver les composants exécutés en parallèle. Chacun est modélisé par un processus LOTOS. Les interactions entre composants sont modélisées par des portes formelles. On détermine ensuite les communications entre ces composants et on les relie entre eux par l'intermédiaire des opérateurs de composition parallèle de LOTOS. Enfin, si nécessaire, on masque certaines communications non pertinentes à un certain niveau d'abstraction grâce à l'opérateur *hide* de LOTOS.

Cette phase peut être appliquée récursivement aux sous-systèmes de façon à obtenir des composants strictement séquentiels.

5. Notons tout de suite que cette démarche permet d'obtenir une spécification, mais pas de la raffiner. Pour cela, il faut en plus disposer d'une relation permettant de savoir si une spécification est une implémentation correcte moins abstraite d'une autre spécification ; par exemple la relation de réduction-extension.

Composants séquentiels

Chaque composant séquentiel peut être décrit par un automate d'états finis. Concrètement, il semble intéressant d'associer les communications sur les portes aux transitions entre états. Un état correspondant pour sa part à un état dans lequel un processus peut recevoir (en fait les communications sont symétriques, donc il faudrait plutôt dire communiquer) certains messages. En fait cela correspond aux portes auxquelles écoute le processus.

Notons ici, que certains aspects des données vont apparaître au niveau des transitions (en général, on se contentera de donner l'interface de ces opérations).

Dans un second temps, il est possible de transformer cet automate en expressions comportementales LOTOS, [GR90] donne les grandes lignes de cette transformation. Les auteurs font remarquer qu'il faut éviter de créér un processus par état puisque cela conduit à des spécifications non structurées.

Données

Les auteurs de [GR90], en se basant sur leur expérience, remarquent qu'il est plus sage de traiter la spécification de la partie données après celle de la partie comportement. Nous remarquons qu'en effet cela permet d'avoir un ensemble minimal et complet (en termes de réponse aux besoins exprimés dans la description informelle) d'opérations dans la partie données (voir section 3.4).

Il s'agit tout d'abord de recenser les sortes et opérations apparues au niveau de l'automate. Ces sortes et opérations sont traduites en LOTOS/ACT ONE.

Pour finir, les équations correspondant aux opérations non constructrices sont créées (les auteurs parlent de "discipline par constructeurs" ou axiomatisation constructive au niveau de la construction des axiomes). Des méthodes existent pour extraire un groupe minimal de constructeurs à partir d'un automate. Il doit être possible de se baser sur les idées de [And95].

Comme le fait remarquer [BB88], il est (pour l'instant ?) impossible de donner une mesure absolue de la qualité (lisibilité, facilité pour faire des preuves, ...) d'une spécification LOTOS[6]. Toutefois, il est possible d'utiliser les propriétés des spécifications LOTOS pour comparer des spécifications (montrer l'équivalence entre deux spécifications ou montrer que l'une est une implémentation moins abstraite correcte de l'autre).

2.6 Conclusion

LOTOS est un langage permettant la spécification formelle de comportements concurrents. Il semble bien correspondre aux besoins ayant donné lieu à sa création, et ce, même si certains comportements parallèles sont difficilement ou pas du tout modélisables en LOTOS (contraintes temps réel, communication asynchrone, synchronisation non atomique ou faible, ...).

De nombreux exemples sont disponibles et montrent l'intérêt que LOTOS suscite. LOTOS est utilisé par un grand nombre d'équipes sur plusieurs continents [GLO91]. Cela met l'accent sur la nécessité de définir un style (voire une méthode) de conception des spécifications LOTOS. De plus, il parait intéressant, sinon nécessaire, d'étudier la réutilisation des composants LOTOS, et ce peut être en relation avec l'intégration des concepts de certains paradigmes existants comme la concurrence et l'orientation objet.

De plus, une notation graphique comme dans le cas de SDL pourrait encore plus répandre l'utilisation de LOTOS. Un travail en ce sens a d'ailleurs commencé en tant que projet commun à l'ISO et au CCITT en 1988 et devrait être presque terminé ([LFHH92]).

3 Une nouvelle méthode pour la spécification en LOTOS

Nous donnons ici notre méthode pour la spécification en LOTOS. Elle est inspirée de [GR90].

6. C'est également le cas dans d'autres formalismes de spécification.

Les éléments apportés par notre méthode sont :

- la description informelle est orientée en fonction des services du processus.
- la description des communications est fortement typée et se fait lors de la description des fonctionnalités externes des processus (et non pas lors du travail sur les composants séquentiels et leur activité interne). Il nous apparaît en effet que le type des données émises et reçues via une porte font partie de la description externe d'un processus et est utile lors de la composition parallèle.
- la composition des processus utilise éventuellement des schémas.
- l'automate du comportement est obtenu de façon semi-automatique. Dans [GR90] rien n'est dit sur la façon d'obtenir cet automate. On ne connaît pas l'état initial.
- la création des spécifications est différente (mais utilise le même sous ensemble d'opérateurs de LOTOS).
- une des possibilités est de créer un sous-processus par état (les auteurs de [GR90] considèrent que cela conduit à créer des spécifications non structurées).
- une automatisation partielle et des guides facilitent la production de la partie statique.

Notre méthode est décrite dans la figure 6.

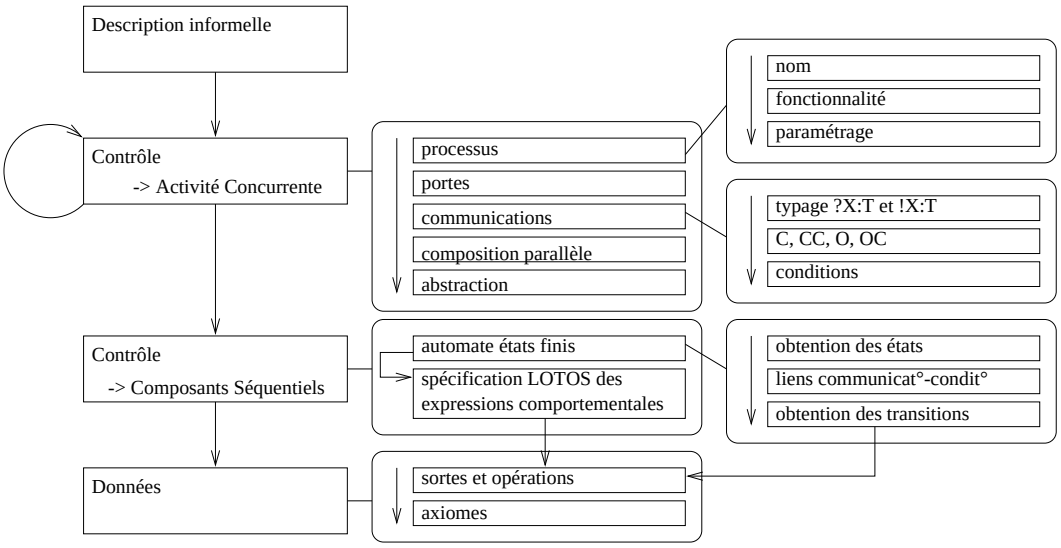


Figure 6 : Notre méthode de spécification LOTOS

L'approche est la suivante :

- description informelle du système à spécifier, fonctionnalités externes : il s'agit de définir les services que le processus peut rendre (à son environnement externe)
- découpage du problème en plusieurs processus gérant chacun un type de données
- définition du comportement externe du chaque processus
  - définition d'un type interne (nom)
  - détermination de l'ensemble des portes de communication du processus

  - typage des communications sur les portes (choix des données échangées : amorphes ou à vie propre (voir plus bas))
  - répartition des portes en 4 ensembles distincts (C, CC, O et OC) et définition d'éventuelles conditions associées
    * préconditions (la communication peut elle ou non avoir lieu)
    * conditions de comportement (effets de conditions sur la communication)

- composition parallèle des processus
- abstraction : masquage à l'environnent externe du processus de communications internes (entre sous-processus).
- définition du comportement interne de chaque processus par un automate
  - obtention des états abstraits du processus en fonction des combinaisons possibles des conditions associées à ses portes de communication
  - obtention des transitions entre états abstraits
  - obtention d'un état initial
  - création de la spécification LOTOS de la partie dynamique
    * obtention des signatures des opérations de la partie statique
    * création par simplification des expressions logiques
    * création par sous-processus
- définition du type interne de chaque processus
  - écriture des axiomes correspondant aux opérations apparues lors de la construction de l'automate du processus
  - utilisation du principe d'Ω-dérivation pour l'obtention des parties gauches des axiomes et ce en liaison avec l'automate du comportement
  - définition éventuelle d'autres types de données internes au processus : si deux sous-processus se communiquent une donnée générique, il y a deux possibilités:
    * cette donnée a une vie propre et doit être modélisée par un nouveau processus et le type de donnée correspondant
    * cette donnée est amorphe et est définie comme un nouveau type (privé) présent dans le processus résultant de la composition des deux sous-processus

Dans les sections qui suivent nous allons détailler toutes ces étapes. Tout au long des explications nous prendrons comme exemple le cas d'un hôpital.

3.1 Description informelle

Il s'agit ici de décrire en langage naturel le système à spécifier. L'idée est d'essayer de dégager les composants du système et leurs interactions.

Il s'agit d'étudier un système simple d'hôpital décrit informellement comme suit. Le système simule un hôpital dans lequel il y a des entrées normales en nombre borné et des entrées en urgence en nombre illimité[7]. Toutes les entrées sont traitées selon le principe de la première arrivée d'abord mais les urgences sont prioritaires sur les entrées normales. Les patients qui entrent dans l'hôpital sont définis par un type Patient disposant de toutes les fonctionnalités nécessaires.

Du point de vue d'un observateur externe, celui-ci peut constater les actions (fonctionnalités) suivantes :

- PRESENT : transmet la valeur booléenne correspondant à la présence ou non d'un patient dans l'hôpital

7. Bien que dans le cas général, le nombre d'admissions en urgence soit aussi limité, les hôpitaux n'étant pas extensibles.

- ADMIT : admission d'un patient normal (non urgent)
- EMERGENCY : admission d'un patient aux urgences
- EXIT : sortie du patient qui vient d'être soigné : s'il y a une urgence il s'agit de la plus ancienne des urgences ; sinon du plus ancien des patients non urgents.

3.2 Activité concurrente

3.2.1 Processus

Il s'agit de trouver les composants exécutés en parallèle. Chacun est modélisé par un processus LOTOS. Nous indiquons aussi la fonctionnalité des processus. Il s'agit ici de processus ne terminant pas; ils ont donc la fonctionnalité *noexit* (voir page 6) et sont modélisés par des processus récursifs. Cette décomposition s'inspire de la méthode de spécification orientée contraintes. Chaque processus dispose de données internes (partie statique) et d'une partie contrôle (partie dynamique). Un type est créé pour chaque type de processus et un objet de ce type constitue le paramètre formel du processus. Ce type devra disposer d'une opération d'initialisation utilisée à l'instanciation du processus. Nous l'appelerons `init`. Un exemple d'utilisation peut être trouvé en 3.2.4 page 19.

De la description ci-dessus, il en résulte un unique processus : l'hôpital
Ce processus est ne termine pas (il ne termine jamais son service, n'est jamais détruit, ...). Il aura donc la fonctionnalité noexit.
L'HOPITAL est paramétré par un objet de type Hôpital.
Il est possible d'obtenir la spécification suivante automatiquement.

```
process HOPITAL [...] (h: Hôpital) : noexit :=
...
endproc
```

3.2.2 Portes

Les interactions entre composants sont modélisées par des portes formelles. On peut voir cela comme les services (interface) du composant. Les portes externes sont données par la description informelle.

Ici nous avons : PRESENT, EMERGENCY, EXIT, ADMIT.
La spécification LOTOS est automatiquement modifiée.

```
process HOPITAL[PRESENT,EMERGENCY,EXIT,ADMIT] (h: Hôpital) : noexit :=
...
endproc
```

3.2.3 Communication

On détermine les communications entre ces composants. Les données émises ou reçues sur les portes sont typées. Le processus ne change pas. L'émission d'une donnée de type T est notée !X:T. La réception d'une donnée de type T est notée ?X:T. Cette syntaxe est celle de LOTOS à ceci près que nous typons aussi les émissions.

Dans le cas qui nous intéresse :

- PRESENT : ?P:Patient !X:Bool
- EMERGENCY : ?X:Patient

– EXIT : !X:Patient

– ADMIT : ?X:Patient

Tous les types de données définis dans cette phase doivent avoir une spécification algébrique associée. Il est possible d'avoir des types génériques mais ils devront alors être instanciés (voir 3.4.4).

Pour chaque processus, nous déterminons les quatre ensembles suivants :

– C, *constructeurs non conditionnés*: la communication sur ces portes modifie la valeur des données du processus et n'est pas soumise à condition ;

– CC, *constructeurs conditionnés*: la communication sur ces portes modifie la valeur des données du processus et est soumise à certaines conditions ;

– O, *observateurs non conditionnés*: la communication sur ces portes ne modifie pas la valeur des données du processus et n'est pas soumise à condition ;

– OC, *observateurs conditionnés*: la communication sur ces portes ne modifie pas la valeur des données du processus et est soumise à certaines conditions.

Ces ensembles sont récapitulés dans le tableau 3. Par condition nous entendons aussi bien ce qui autorise une communication (précondition) que ce qui peut modifier l'effet d'une communication (condition de comportement). Dans le cas de OC et de CC nous donnons et nommons les conditions.

Le but de cette distinction est de faciliter la création de l'automate du processus. De plus, ces ensembles correspondent aussi à des ensembles d'opérations du type sous-jacent au processus.

$\sigma = C + CC + O + OC$ est l'ensemble des portes du processus. Il faut noter qu'une porte reliant un processus P à un processus Q peut appartenir à deux ensembles différents selon qu'on s'occupe de P ou de Q.

	non conditionnés	conditionnés
modification état interne	C	CC
conservation état interne	O	OC

TAB. 3 – *Ensembles C, CC, O et OC*

Dans le cas du processus hôpital :

- O : PRESENT : ?P:Patient !X:Bool
- OC : vide
- C : EMERGENCY : ?X:Patient
- CC :
  - EXIT : !X:Patient, c-nonVide (précondition, hôpital non vide), c-urgence (condition de comportement : s'il existe au moins une urgence alors c'est la plus ancienne qui sort ; sinon c'est le patient non urgent le plus ancien qui sort)
  - ADMIT : ?X:Patient, c-nonPlein (précondition, hôpital non plein)

Ce typage des communications sert non seulement en tant qu'interface du processus mais aussi lorsqu'il s'agit de travailler sur la partie statique de la spécification[8] (voir 3.3.2).

8. C'est pourquoi nous typons explicitement les émissions, ce qui n'est pas le cas en LOTOS.

3.2.4 Composition parallèle

Les composants sont reliés entre eux par les opérateurs de composition parallèle de LOTOS.

Nous créons un processus représentant l'environnement extérieur (cf [Led87]) et qui consiste à faire admettre de façon indéterminée des patients urgents et normaux à l'hôpital et à les en faire sortir (une fois soignés).

```
process ENVIRONNEMENT [σ] : noexit :=
  EXIT?p:Patient; ENVIRONNEMENT [σ] (* sortie d'un patient *)
[]
  let p := creerUrgent in EMERGENCY!p; ENVIRONNEMENT [σ] (* admission urgente *)
[]
  let p := creerNormal in ADMIT!p; ENVIRONNEMENT [σ] (* admission non urgente *)
endproc
```

Nous ne décrivons pas ici les données d'ENVIRONNEMENT ni ses opérations internes. Nous supposons seulement que son type de données associé comporte des opérations permettant de créer un patient urgent ou un patient non urgent. Ses portes correspondent à celles de l'hôpital, comme indiqué figure 7.

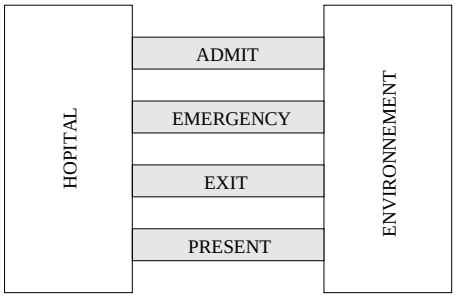


Figure 7 : Composition parallèle de l'hôpital et de son environnement

Nous pouvons maintenant nous occuper de la composition. Les compositions de processus entrent souvent dans des cadres déjà définis. Nous proposons d'utiliser une bibliothèque de "schémas de compositions" (instanciation). Si la composition ne se trouve pas dans cette bibliothèque, il est bien sur possible de l'étendre. Quelques schémas (appelés schémas de synchronisation ou styles d'architectures) peuvent être trouvés dans [Led87, Gar90, LLS97, HL97].

Notons que LOTOS dispose de moyens de paramétrer un type de données mais pas la partie comportement (hormis les noms de portes et les données paramètres) d'un processus. Notre mécanisme d'instanciation doit donc être simulé (voir [Tur97] par exemple) par renommage manuel des noms de processus et des listes de portes dans les schémas.

L'hôpital est un problème de type de client-serveur. Nous utiliserons donc par exemple le schéma suivant (un serveur, deux clients) :

```
specification CLIENT-SERVEUR [σ_CS] : noexit
behaviour
        SERVEUR [σ_CS] (initServeur)
|[σ_CS]|
                (CLIENT [σ_CS] (initClient)
        |||
                CLIENT [σ_CS] (initClient))
endspec
```

L'instanciation donne :

```
specification HOPITAL-CLIENT-SERVEUR [σ] : noexit
behaviour
        HOPITAL [σ] (init(10) [9])
|[σ]|
                (ENVIRONNEMENT [σ]
        |||
                ENVIRONNEMENT [σ])
        where ...
endspec
```

On peut voir qu'à l'instanciation des processus gérant un type de donnée (ici HOPITAL), un appel à l'opération `init` (voir 3.2.1) est substitué au paramètre formel des processus.

3.2.5 Abstraction

Si nécessaire, on masque certaines communications non pertinentes à un certain niveau d'abstraction grâce à l'opérateur *hide* de LOTOS.

Ici il n'y en a pas, mais si nous avions voulu modéliser un hôpital fonctionnant avec une file d'attente tout en masquant les communications entre ces deux composantes, nous aurions eu :

```
process HOPITAL-AVEC-ATTENTE [σ] (h:Hôpital,f:FileAttente) : noexit :=
    hideporteEchanges in
        HOPITAL [σ,porteEchanges] (h)
    |[σ,porteEchanges]|
        FILE-ATTENTE [σ,porteEchanges] (f)
    where ...
endproc
```

3.3 Composants séquentiels

3.3.1 Automate d'états finis

L'utilisation d'un automate se justifie pleinement en ce que, comme le fait remarquer [Tur93], "la représentation sous forme d'arbre du comportement entier d'un système n'est pratique que lorsque le nombre d'états est limité".

Obtention des états

L'idée est qu'un processus peut se trouver dans différents états abstraits dans lesquels il est en mesure ou non de rendre un service (c'est à dire d'autoriser une communication sur une porte). Les états sont donc obtenus par composition des conditions de communication (un recensement des conditions a été fait lors du travail sur les portes, voir 3.2.3) dans une table de vérité.

Nous obtenons le tableau 4.
Des formules logiques expriment des propriétés reliant ces conditions et permettent de supprimer les états incohérents.

Ici :

1. c-urgence$\Longrightarrow$c-nonVide (l'hôpital ne peut pas être vide s'il contient une urgence)

9. L'opération init : Nat $\rightarrow$Hôpital (voir 3.2.1), crée un hôpital vide de n places.
10. L'idée d'état "intermédiaire" correspond à un hôpital ni vide, ni plein.

c-nonVide	c-urgence	c-nonPlein	interprétation	référence
V	V	V	hôpital intermédiaire avec urgence	InterAvecUrgence[10]
V	V	F	hôpital plein avec urgence	PleinAvecUrgence
V	F	V	hôpital intermédiaire sans urgence	InterSansUrgence[10]
V	F	F	hôpital plein sans urgence	PleinSansUrgence
F	V	V	impossible (1)	
F	V	F	impossible (2)	
F	F	V	hôpital vide	Vide
F	F	F	impossible (2)	

TAB. 4 – *Composition des conditions et obtention des états*

2. c-nonVide$\vee$c-nonPlein (l'hôpital ne peut être vide et plein à la fois)

La propriété c-nonVide $\vee$ c-nonPlein permet par exemple de supprimer les états correspondant aux triplets <F,$\forall$,F> (<F,V,F> et <F,F,F>).

Un automate est construit avec un état pour chaque cas différent dans la table de composition des états (table 4). Les n-uplets correspondant à la table de vérité y sont notés (figure 8).

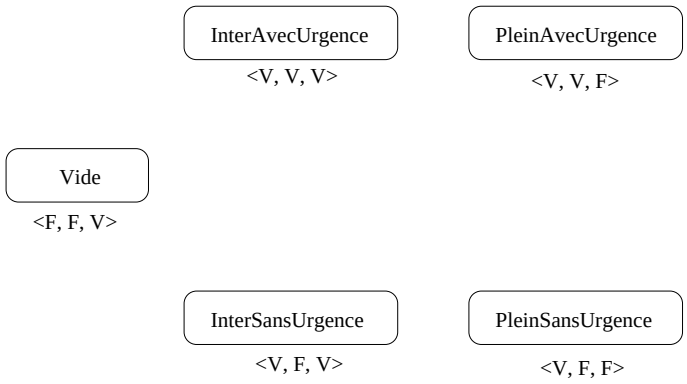


Figure 8 : Etats de l'automate

Liens communications-conditions

Pour chacune des portes de C ou CC, on donne les préconditions et les postconditions en fonction des conditions trouvées auparavant. c désigne la valeur de la condition c avant communication et c' sa valeur après communication.

L'*idée générale* est qu'un état abstrait est identifié par un ensemble de services que peut rendre le processus. Les transitions correspondent à des opérations qui sont possibles sous certaines conditions (et donc à partir de certains états ; nous avons ici les points de départ des transitions) et conduisent ou non à modifier l'état abstrait (c'est le cas des constructeurs qui font changer d'état abstrait ; nous avons ici les points d'arrivée des transitions).

Dans le cas de l'hôpital nous obtenons les tables 5 et 6.

	précondition		
porte	c-nonVide	c-urgence	c-nonPlein
EMERGENCY	$\forall$	$\forall$	$\forall$
ADMIT	$\forall$	$\forall$	vrai
EXIT	vrai	$\forall$	$\forall$

TAB. 5 – *Liens communications-conditions : préconditions*

	postcondition		
porte	c-nonVide'	c-urgence'	c-nonPlein'
EMERGENCY	vrai	vrai	c-nonPlein
ADMIT	vrai	c-urgence	¬ c-presquePlein
EXIT	¬ c-unSeulPatient	¬ c-uneSeuleUrgence	¬ c-urgence ∨ c-nonPlein

TAB. 6 – *Liens communications-conditions : postconditions*

Les préconditions décrivent l'ensemble des états dans lesquels une communication sur une porte est possible ou non. Les éléments de C ou O ne sont pas préconditionnés et ceux de CC ou OC ne sont préconditionnés que par leur condition d'application définie plus tôt par le spécifieur.

Les postconditions décrivent l'effet des opérations sur la valeur des conditions et donc le changement d'état abstrait.

Il existe en général des *cas critiques* et cet effet ne peut donc pas être exprimé uniquement en fonction des conditions d'état. Il est nécessaire d'utiliser de nouvelles conditions pour tester si le type de données correspondant au processus est ou non dans un état critique. Ces nouvelles conditions vont apparaître lorsqu'on cherche à exprimer les postconditions.

Ici c-unSeulPatient signifie qu'il n'y a qu'un seul patient (en prenant en compte les patients urgents et ceux qui ne le sont pas) dans l'hôpital, c-uneSeuleUrgence qu'il n'y a qu'une urgence et c-presquePlein qu'il ne reste plus qu'une place pour un seul patient normal.

Pour déterminer l'*état initial*, on examine les services (qui sont associés à des portes) que le processus doit rendre ou non (contraintes). A partir des préconditions des portes et de la table de création des états on trouve les états susceptibles d'être initiaux. En ce qui concerne la spécification LOTOS, il est nécessaire de n'avoir qu'un seul état initial (état dans lequel "démarre" le processus lors de son instanciation). Dans le cas où il y a plusieurs état initiaux possibles, un choix peut être fait en ajoutant une contrainte sur les services ou en faisant un choix arbitraire.

L'état initial de l'hôpital doit permettre d'admettre un patient normal (ADMIT : c-nonPlein), d'admettre une urgence (EMERGENCY), mais pas de pouvoir faire sortir un patient (EXIT : c-nonVide). Il faut ¬ c-nonVide ∧ c-nonPlein soit le triplet $< F, \forall, V >$. Seul l'état VIDE correspond (voir table d'obtention des états).

Transitions

Nous allons présenter séparément les automates puis les intégrer, en prenant dans l'ordre :

- état initiaux possibles du processus ainsi que l'ensemble des observateurs sans conditions
- chacun des observateurs avec conditions
- chacun des constructeurs sans conditions
- chacun des constructeurs avec conditions.

Les état initiaux indiquent dans quel(s) état(s) sont instanciés les processus (figure 9).

Les observateurs sans conditions (PRESENT dans l'exemple) correspondent à des transitions d'un état vers lui-même (un observateur ne modifiant pas l'état abstrait) et sont possibles dans tous les états. Pour cela, on les groupe en o (figure 9).

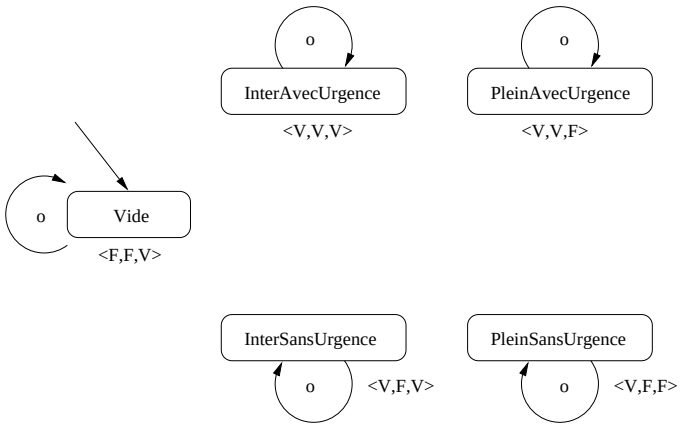


Figure 9 : Automate avec état initial et transitions correspondant à O

Les observateurs conditionnels ne sont autorisés qu'à partir des états ayant la bonne valeur de vérité pour la condition correspondante. Ils correspondent à des transitions de chacun de ces états vers eux mêmes.

Il n'y en a pas dans le cas de l'hôpital.

L'idée générale pour obtenir les transitions correspondant aux constructeurs est la suivante : à l'aide d'un n-uplet représentant les valeurs des conditions avant une opération (répondant donc à sa précondition) et de la table des postconditions, il est possible d'obtenir le n-uplet correspondant à la valeur des conditions après cette opération (postcondition).

Les transitions correspondant aux constructeurs non conditionnés permettent de passer d'un état à un autre mais n'ont pas de précondition associée (EMERGENCY, figure 10).

Pour les trouver on procède comme suit :

- prendre un état e (n-uplet de booléens)
- à l'aide de la table des liens communications-conditions (ligne correspondant au constructeur considéré) et en supposant que ce n-uplet corresponde aux valeurs de précondition, trouver le n-uplet correspondant aux valeurs de postcondition et l'état f correspondant
- on a une transition entre e et f
- recommencer avec un état non encore choisi

Prenons comme exemple l'opération EMERGENCY.

- Nous choisissions un premier état : VIDE. Il correspond au n-uplet <F,F,V>. Ce n-uplet peut s'unifier avec la précondition d'EMERGENCY : $< \forall, \forall, \forall >$ et nous avons donc dans ce cas { c-nonVide=F, c-urgence=F, c-nonPlein=V }.
- En appliquant cette substitution à la postconditon d'EMERGENCY (voir table 6) nous obtenons : <V, V, c-nonPlein> soit <V, V, V> ce qui correspond à l'état InterAvecUrgence.
- nous avons donc une transition étiquetée par EMERGENCY entre Vide et InterAvecUrgence.
- nous continuons avec un autre état ...

Chaque n-uplet correspondant à un et un seul état, on a donc une relation entre états représentant les transitions de l'automate.

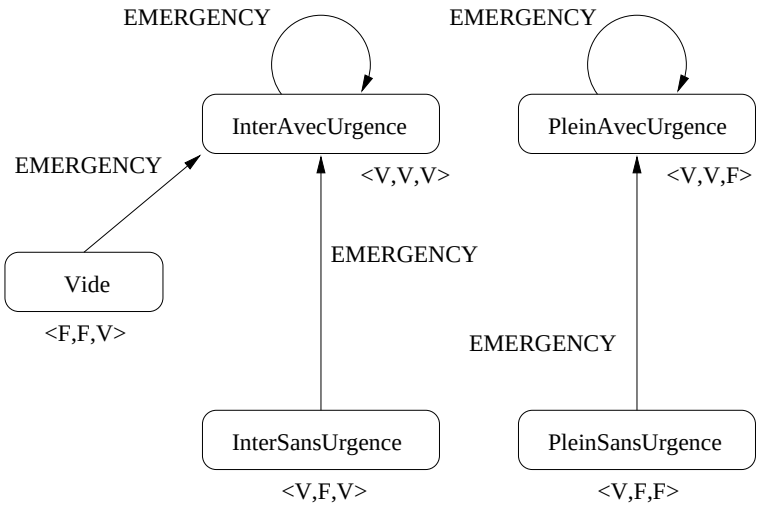


Figure 10 : Automate EMERGENCY

Les constructeurs conditionnés permettent de passer d'un état à un autre lorsque certaines préconditions sont vérifiées (figures 11 et 12). Pour les trouver on procède comme pour les constructeurs non conditionnés mais les conditions apparaissant au niveau des *si* des postconditions sont reportées en tant que préconditions sur les transitions. Pour des raisons de place, les préconditions ont été numérotées (et sont décrites dans le tableau 7).

Dans certains cas, les conditions inhérentes aux états impliquent celles ajoutées dans la table des liens communications-conditions et simplifient ainsi les tests à faire.

Ici, nous avons : c-uneSeuleUrgence $\Longrightarrow$ c-urgence, c-unSeulPatient $\Longrightarrow$ c-nonVide, c-unSeulPatient $\wedge$ c-urgence $\Longrightarrow$ c-uneSeuleUrgence, c-unSeulPatient $\wedge$ c-uneSeuleUrgence $\Longrightarrow$ c-nonPlein, ...

Ainsi dans le cas d'EXIT, pour l'état InterAvecUrgence ({c-nonVide=V, c-urgence=V, c-nonPlein=V}) nous obtenons théoriquement:

1. {c-nonVide=F, c-urgence=F, c-nonPlein=V} (Vide)
   si c-unSeulPatient et c-uneSeuleUrgence
2. {c-nonVide=F, c-urgence=V, c-nonPlein=V} (Impossible)
   si c-unSeulPatient et ¬c-uneSeuleUrgence
3. {c-nonVide=V, c-urgence=F, c-nonPlein=V} (InterSansUrgence)
   si ¬c-unSeulPatient et c-uneSeuleUrgence
4. {c-nonVide=V, c-urgence=V, c-nonPlein=V} (InterAvecUrgence)
   si ¬c-unSeulPatient et ¬c-uneSeuleUrgence

Le cas 2 est éliminé car il est incohérent puisque c-urgence $\Longrightarrow$ c-nonVide (voir relations entre conditions lors de la construction de la table des états). Le cas 1 peut être simplifié en c-unSeulPatient car c-unSeulPatient $\wedge$ c-urgence $\Longrightarrow$ c-uneSeuleUrgence et car dans l'état InterAvecUrgence c-urgence est vrai. De même, le cas 4 peut être simplifié en ¬ c-uneSeuleUrgence.

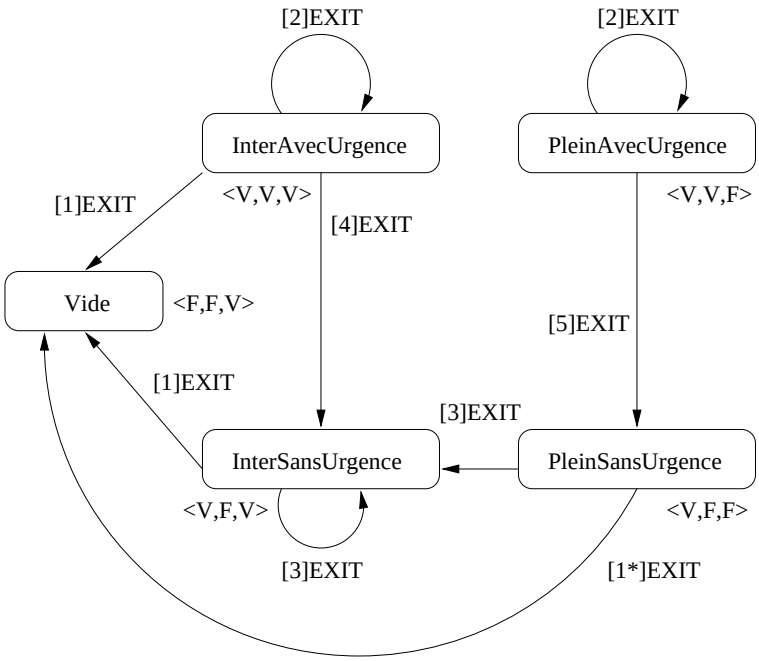


Figure 11 : Automate EXIT

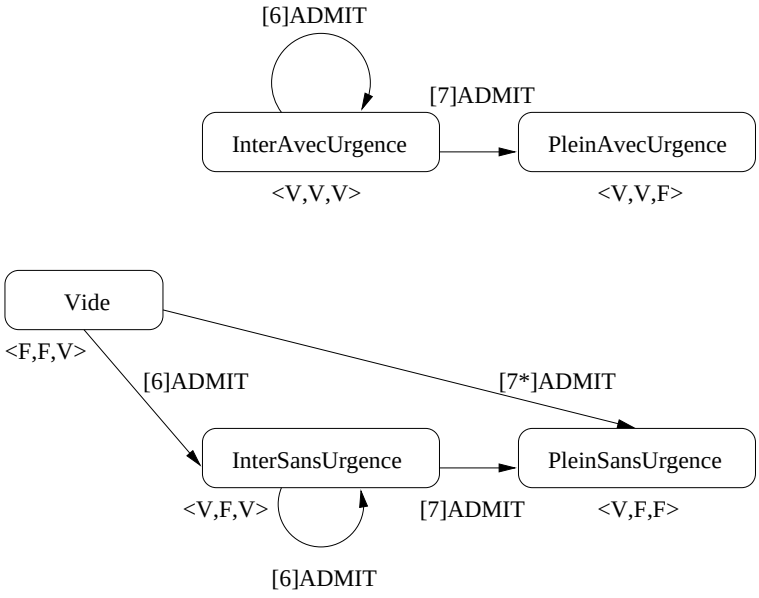


Figure 12 : Automate ADMIT

On obtient l'automate de la figure 13 en intégrant tous les autres.

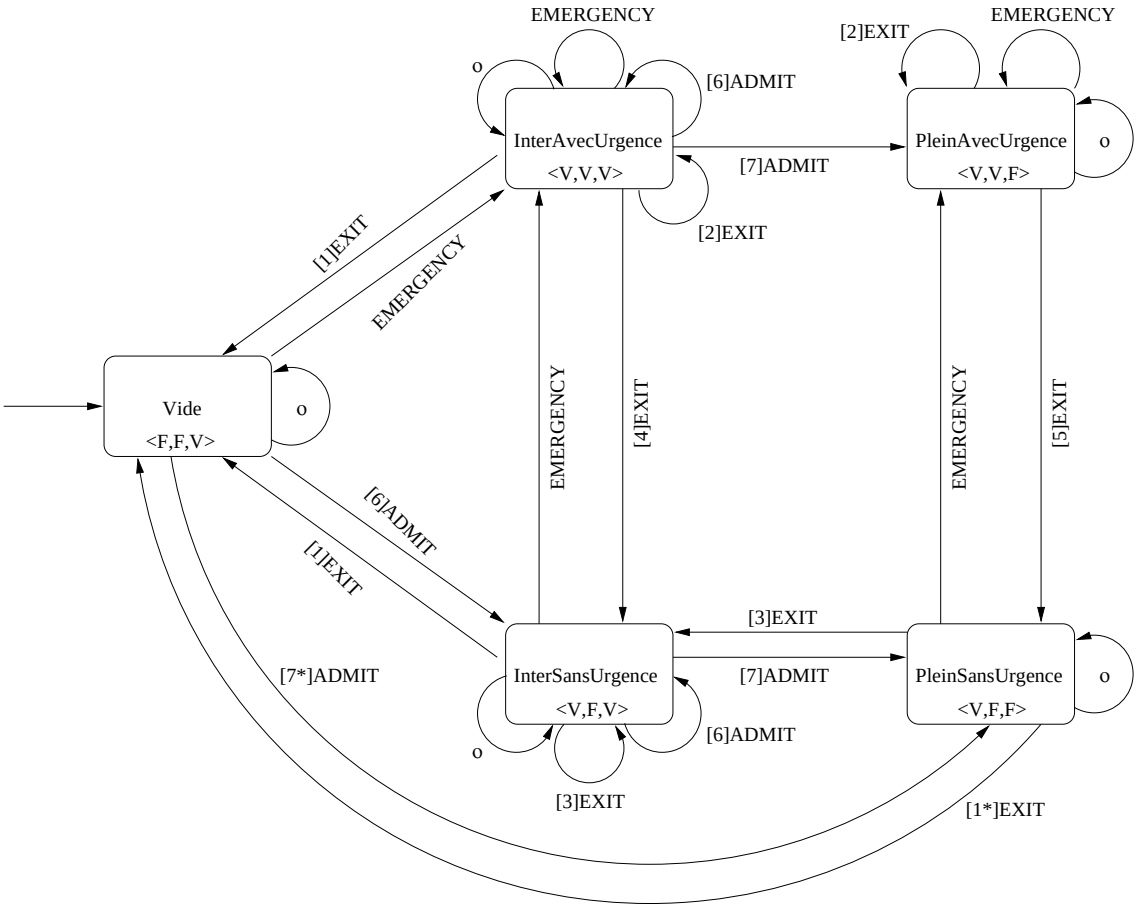


Figure 13 : Automate de l'Hôpital

numéro	condition
1, 1*	c-unSeulPatient
2	¬ c-uneSeuleUrgence
3	¬ c-unSeulPatient
4	¬ c-unSeulPatient ∧ c-uneSeuleUrgence
5	c-uneSeuleUrgence
6	¬ c-presquePlein
7,7*	c-presquePlein

TAB. 7 – *Conditions des transitions*

Remarque : cette démarche automatique nous a permis de repérer les cas [1*] (sortie du seul patient non urgent d'un hôpital plein à une place) et [7*] (entrée d'un patient non urgent dans un hôpital vide à une place) de l'automate que nous n'avions pas détectés sans suivre la méthode. Ces cas correspondent à un hôpital réduit à une seule place. De façon plus générale, la méthode permet de ne pas oublier les "cas critiques".

3.3.2 Création des spécifications LOTOS

Nous donnons ici notre méthode semi-automatique d'obtention des spécifications LOTOS à partir de l'automate. Il est possible de procéder de deux façons (voir les schémas de transformation des figures 14 et 18) : un processus est associé à chaque automate et (i) pour chaque état une

branche conditionnelle est créée puis des simplifications sont effectuées afin d'obtenir une spécification plus lisible, ou (ii) un sous-processus est créé pour chaque état de l'automate (cette dernière approche nécessitant/autorisant beaucoup moins de simplifications). La spécification complète peut être trouvée en annexe A.

Signatures

Il est possible ici d'extraire automatiquement les signatures des opérations correspondant aux communications sur les portes en fonction du typage de ces portes fait plus tôt (voir 3.2.3) :

- pour toute porte de O ou OC,
  `nomPorte` : $!X_{s_1}{:}T_{s_1}$ ... $!X_{s_n}{:}T_{s_n}$ $?X_{r_1}{:}T_{r_1}$ ... $?X_{r_m}{:}T_{r_m}$
  devient une famille d'opérations
  $\texttt{nomOper}_i : T \times T_{r_1} \times ... \times T_{r_m} \rightarrow T_{s_i}$, $i \in \mathbb{N}$, $1 \leq i \leq n$
  où T est le nom du type associé au processus.

- pour toute porte de C ou CC,
  `nomPorte` : $!X_{s_1}{:}T_{s_1}$ ... $!X_{s_n}{:}T_{s_n}$ $?X_{r_1}{:}T_{r_1}$ ... $?X_{r_m}{:}T_{r_m}$
  devient une famille d'opérations
  $\texttt{nomOper}_i : T \times T_{r_1} \times ... \times T_{r_m} \rightarrow T_{s_i}$, $i \in \mathbb{N}$, $1 \leq i \leq n$
  et une opération
  $\texttt{nomOper}_\omega : T \times T_{r_1} \times ... \times T_{r_m} \rightarrow T$

Dans le cas d'opérations ayant plus d'un type résultat, différentes nouvelles opérations sont créées de façon à ce que chacune ne retourne qu'une seule valeur. Cela servira plus tard lors de la génération des parties gauches des axiomes.
Ainsi, dans le cas qui nous intéresse :

- à la porte PRESENT nous associons l'opération `present` : $H\hat{o}pital \times Patient \rightarrow Bool$
- à EMERGENCY, `emergency` : $H\hat{o}pital \times Patient \rightarrow H\hat{o}pital$
- à EXIT nous associons les opérations :
  - `first` : $H\hat{o}pital \rightarrow Patient$ (sortie de l'hôpital d'un patient soigné)
  - `cure` : $H\hat{o}pital \rightarrow H\hat{o}pital$ (hôpital après la sortie du patient soigné)
- à ADMIT,`admit` : $H\hat{o}pital \times Patient \rightarrow H\hat{o}pital$

Nous faisons la distinction entre les *opérations internes* (le type résultat est le type d'intérêt) et les *opérations externes* (le type résultat n'est pas le type d'intérêt). Les opérations internes pour lesquelles le type d'intérêt n'apparaît pas dans les arguments sont qualifiées d'*opérations internes de base*. Les opérations sont *totales* lorsqu'elles s'appliquent dans tous les états et *partielles* dans le cas contraire.

Nous appelerons β l'application faisant correspondre à une porte P la (ou les) opération(s) (signature) associée(s). Ainsi, $\beta(EXIT)$ est égal à l'ensemble {`first` : $H\hat{o}pital \rightarrow Patient$, `cure` : $H\hat{o}pital \rightarrow H\hat{o}pital$}. β_ω désignera l'opération interne $\texttt{nomOper}_\omega$ ($\beta_\omega(EXIT)$ est `cure` : $H\hat{o}pital \rightarrow H\hat{o}pital$) et β_o les autres.

De plus, pour chaque état de l'automate, ainsi que pour chacune des conditions (conditions d'état et conditions de transitions), une opération externe "observateur d'état" est créée avec le profil :
`oper` : $H\hat{o}pital \rightarrow Bool$.

profil		description
observateurs (opérations externes) de conditions		
c-nonVide:	$H\hat{o}pital \rightarrow Bool$	hôpital non vide
c-urgence:	$H\hat{o}pital \rightarrow Bool$	il existe au moins une urgence
c-nonPlein:	$H\hat{o}pital \rightarrow Bool$	hôpital non plein
c-presquePlein:	$H\hat{o}pital \rightarrow Bool$	hôpital presque plein
c-unSeulPatient:	$H\hat{o}pital \rightarrow Bool$	hôpital avec un seul patient
c-uneSeuleUrgence:	$H\hat{o}pital \rightarrow Bool$	hôpital avec une seule urgence
observateurs (opérations externes) d'état		
vide:	$H\hat{o}pital \rightarrow Bool$	état Vide
interAvecUrgence:	$H\hat{o}pital \rightarrow Bool$	état InterAvecUrgence
interSansUrgence:	$H\hat{o}pital \rightarrow Bool$	état InterSansUrgence
pleinAvecUrgence:	$H\hat{o}pital \rightarrow Bool$	état PleinAvecUrgence
pleinSansUrgence:	$H\hat{o}pital \rightarrow Bool$	état PleinSansUrgence
opérations externes		
present:	$H\hat{o}pital \times Patient \rightarrow Bool$	patient présent dans l'hôpital.
first:	$H\hat{o}pital \rightarrow Patient$	patient à soigner
opérations internes initiales		
init:	$Nat \rightarrow H\hat{o}pital$	crée un hôpital vide de n places
opérations internes totales		
emergency:	$H\hat{o}pital \times Patient \rightarrow H\hat{o}pital$	admission d'un patient en urgence
opérations internes partielles		
admit:	$H\hat{o}pital \times Patient \rightarrow H\hat{o}pital$	admission d'un patient normal
cure:	$H\hat{o}pital \rightarrow H\hat{o}pital$	soigne le prochain patient et l'ôte des patients

TAB. 8 – *Signatures des opérations obtenues*

Toutes les opérations[11] sont récapitulées dans le tableau 8.

Création par simplification des expressions logiques

Les éléments de O sont codés selon le schéma suivant :

$$\boxed{\texttt{o?x}_{r_j}\texttt{:T}_{r_j}\ \texttt{!x}_{s_i}\texttt{; PROCESSNAME [portes] (paramètres)}}$$

A priori, la valeur des $\texttt{x}_{s_i}$ est fonction des données du processus et des valeurs reçues : $\beta_{o_i}(o)(h, \{x_{r_j}\})$. Les listes de portes et de paramètres sont celles de l'appel.

Dans l'exemple de PRESENT, nous obtenons :

$$\texttt{PRESENT?p:Patient!(present(h,p)); HOPITAL [}\sigma\texttt{] (h)}$$

Les éléments de OC sont codés selon le schéma suivant :

$$\boxed{\texttt{[c-cond(h)]} \rightarrow \texttt{c?x}_{r_j}\texttt{:T}_{r_j}\ \texttt{!x}_{s_i}\texttt{; PROCESSNAME [portes] (paramètres)}}$$

c-cond correspond à la condition correspondant à l'élément en question. A priori, la valeur des $\texttt{x}_{s_i}$ est fonction des données du processus et des valeurs reçues : $\beta_{o_i}(c)(h, \{x_{r_j}\})$. Les listes de portes et de paramètres sont celles de l'appel.

On crée une branche conditionnelle correspondant à chaque état. La précondition associée est la condition d'état du processus. Ces conditions peuvent être trouvées dans la table des cas ayant

11. Dans ce qui suit, h désigne le paramètre du processus correspondant à son type associé.

servi à trouver les états (table 4 page 20).

En ce qui concerne les éléments de C et CC, des branches conditionnelles sont utilisées pour chacune des transitions partant des états. Le principe est décrit dans la figure 14. g(x) représente une communication sur la porte g avec une éventuelle réception de valeurs (x_{r_j}) et une éventuelle émission de valeurs ($\beta_{o_i}(g)(p, x_{r_j})$). f(p,x) (avec f= $\beta_\omega(g)$) est fonction des données internes du processus et des données éventuellement reçues via la porte g). σ désigne l'ensemble des portes du processus.

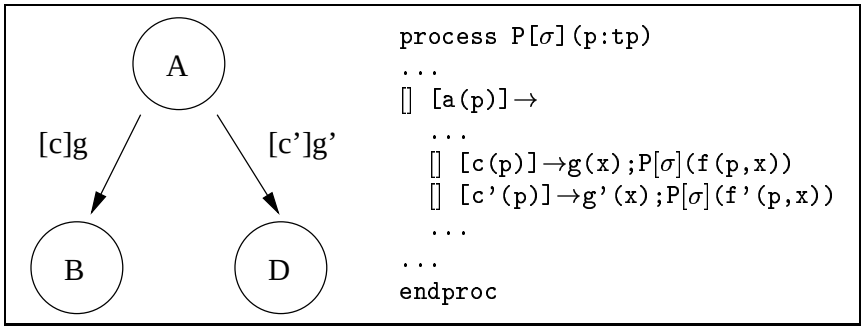


Figure 14 : Spécification par simplification pour les éléments de C et CC

Enfin, il y a appel récursif du processus dans le cas des états qui ne sont pas des puits (un puits correspondant à la terminaison avec blocage – `stop` – du processus : aucune transition ne peut être faite).

Il est possible de regrouper les conditions des transitions étiquetées par une même porte.
Le processus obtenu est représenté dans la figure 15. Les conditions ont été vues plus haut.

```
process HOPITAL
                [PRESENT,EMERGENCY,EXIT,ADMIT]
        (h: Hôpital) : noexit :=
   PRESENT?p:Patient !(present(h,p)); HOPITAL [σ] (h)
[] [vide(h)]→
         (EMERGENCY?p:Patient; HOPITAL [σ] (emergency(h,p))
      [] ([6] ∨ [7*])→ADMIT?p:Patient; HOPITAL [σ] (admit(h,p))
[] [interAvecUrgence(h)]→
         ([1] ∨ [2] ∨ [4])→ EXIT !first(h); HOPITAL [σ] (cure(h))
      [] ([6] ∨ [7])→ ADMIT?p:Patient; HOPITAL [σ] (admit(h,p))
      [] EMERGENCY?p:Patient; HOPITAL [σ] (emergency(h,p))
[] [interSansUrgence(h)]→
         ([1] ∨ [3])→ EXIT !first(h); HOPITAL [σ] (cure(h))
      [] ([6] ∨ [7])→ ADMIT?p:Patient; HOPITAL [σ] (admit(h,p))
      [] EMERGENCY?p:Patient; HOPITAL [σ] (emergency(h,p))
[] [pleinAvecUrgence(h)]→
         ([2] ∨ [5])→ EXIT !first(h); HOPITAL [σ] (cure(h))
      [] EMERGENCY?p:Patient; HOPITAL [σ] (emergency(h,p))
[] [pleinSansUrgence(h)]→
         ([1*] ∨ [3])→ EXIT !first(h); HOPITAL [σ] (cure(h))
      [] EMERGENCY?p:Patient; HOPITAL [σ] (emergency(h,p))
endproc
   où σ dénote l'ensemble des portes PRESENT,EMERGENCY,EXIT,ADMIT
```

Figure 15 : Spécification du processus HOPITAL

La spécification obtenue est "orientée état" : elle est construite autour de tests de l'état abstrait (courant) du processus, la liste des communications autorisées dépendant de cet état.

Ici, un regroupement plus poussé peut être fait sur les portes. On aurait pu obtenir une spécification plus concise en travaillant directement sur les automates séparés (et pas sur l'automate intégré).

Le processus obtenu est représenté dans la figure 16.

```
process HOPITAL
                [PRESENT,EMERGENCY,EXIT,ADMIT]
        (h: Hôpital) : noexit :=
   PRESENT?p:Patient !(present(h,p)); HOPITAL [σ] (h)
[] EMERGENCY?p:Patient; HOPITAL [σ] (emergency(h,p))
[] [(vide(h) ∧ ([6] ∨ [7*]))
   ∨ (interAvecUrgence(h) ∧ ([6] ∨ [7]))
   ∨ (interSansUrgence(h) ∧ ([6] ∨ [7]))]→
      ADMIT?p:Patient; HOPITAL [σ] (admit(h,p))
[] [(interAvecUrgence(h) ∧ ([1] ∨ [2] ∨ [4]))
   ∨ (interSansUrgence(h) ∧ ([1] ∨ [3]))]
   ∨ (pleinAvecUrgence(h) ∧ ([2] ∨ [5]))]
   ∨ (pleinSansUrgence(h) ∧ ([1*] ∨ [3]))]→
      EXIT!first(h); HOPITAL [σ] (cure(h))
endproc
   où σ dénote l'ensemble des portes PRESENT,EMERGENCY,EXIT,ADMIT
```

Figure 16 : Spécification simplifiée du processus HOPITAL

Il est possible de simplifier encore la spécification obtenue en travaillant sur les conditions : l'emploi par exemple d'un outil de simplification (et de preuve) d'expressions logiques permet d'obtenir une expression plus concise des fonctions booléennes de la spécification.

Cette simplification conduit en fait à une spécification équivalente (figure 17) à celle qui aurait été obtenue en travaillant directement sur les conditions d'application des portes et en traitant - au niveau de la création de la spécification - les éléments de CC comme des éléments de OC.

```
process HOPITAL
                [PRESENT,EMERGENCY,EXIT,ADMIT]
        (h: Hôpital) : noexit :=
   PRESENT?p:Patient !(present(h,p)); HOPITAL [σ] (h)
[] EMERGENCY?p:Patient; HOPITAL [σ] (emergency(h,p))
[] [c-nonPlein(h)]→ADMIT?p:Patient; HOPITAL [σ] (admit(h,p))
[] [c-nonVide(h)]→EXIT!first(h); HOPITAL [σ] (cure(h))
endproc
   où σ dénote l'ensemble des portes PRESENT,EMERGENCY,EXIT,ADMIT
```

Figure 17 : Spécification concise du processus HOPITAL

Le fait de passer par un automate permet toutefois de mieux appréhender le comportement interne des processus.

Création par sous-processus

Une autre possibilité pour obtenir les spécifications est d'ajouter un paramètre au processus représentant son état actuel. C'est cet état qui sert de précondition aux branches conditionnelles. Dans le cas d'une transition d'un état e vers f, nous avons une branche testant e et un appel récursif remplaçant e par f.

En allant plus loin, il est possible de créer un sous-processus par état. [Tur93] explique que ce style permet une implémentation rapide, efficace et précise quand la spécification est traduite dans un langage séquentiel et donne des règles de transformation automatique de l'automate en LOTOS. [GR90] fait cependant remarquer que cela conduit à obtenir des spécifications non structurées.

Nous utilisons les règles suivantes :

- le processus principal appelle le processus correspondant à l'état initial
- pour toute transition [c] p allant d'un état A à un état B, nous créons une branche conditionnelle correspondante dans le processus A (voir figure 18). Ceci est à comparer avec l'approche de la figure 14 page 28.

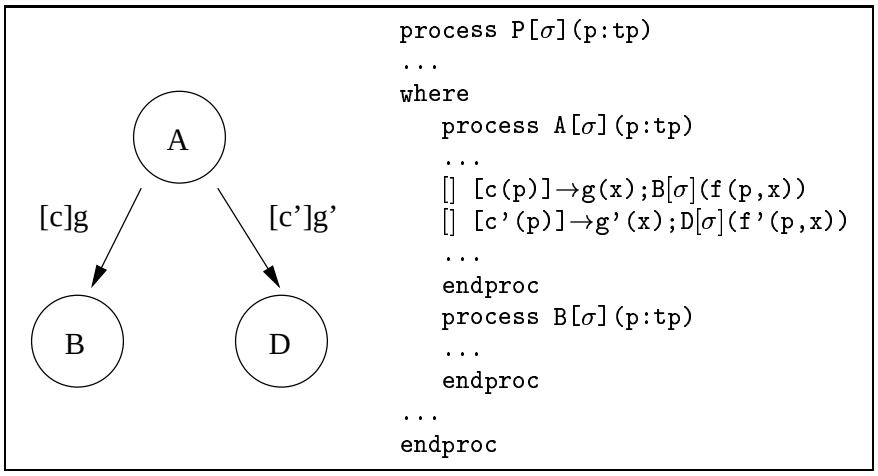


Figure 18 : Spécification en sous-processus

Le résultat est un processus éclaté en autant de sous-processus que l'automate contient d'états. Le processus LOTOS obtenu est donné dans la figure 19.

```
process HOPITAL
                [PRESENT,EMERGENCY,EXIT,ADMIT]
        (h: Hôpital) : noexit :=
   VIDE(h)[σ]
where
process VIDE [σ] (h:Hôpital) : noexit :=
   PRESENT?p:Patient !(present(h,p)); VIDE [σ] (h)
[] EMERGENCY?p:Patient; InterAvecUrgence [σ] (emergency(h,p))
[] [6]→ADMIT?p:Patient; InterSansUrgence [σ] (admit(h,p))
[] [7*]→ADMIT?p:Patient; PleinSansUrgence [σ] (admit(h,p))
endproc
process InterSansUrgence [σ] (h:Hôpital) : noexit :=
   PRESENT?p:Patient !(present(h,p)); InterSansUrgence [σ] (h)
[] EMERGENCY?p:Patient; InterAvecUrgence [σ] (emergency(h,p))
[] [6]→ADMIT?p:Patient; InterSansUrgence [σ] (admit(h,p))
[] [7]→ADMIT?p:Patient; PleinSansUrgence [σ] (admit(h,p))
[] [1]→EXIT !first(h); VIDE [σ] (cure(h))
[] [3]→EXIT !first(h); InterSansUrgence [σ] (cure(h))
endproc
process InterAvecUrgence [σ] (h:Hôpital) : noexit :=
   PRESENT?p:Patient !(present(h,p)); InterAvecUrgence [σ] (h)
```

```
[]  EMERGENCY?p:Patient; InterAvecUrgence [σ] (emergency(h,p))
[]  [6]→ADMIT?p:Patient; InterAvecUrgence [σ] (admit(h,p))
[]  [7]→ADMIT?p:Patient; PleinAvecUrgence [σ] (admit(h,p))
[]  [1]→EXIT!first(h); VIDE [σ] (cure(h))
[]  [2]→EXIT!first(h); InterAvecUrgence [σ] (cure(h))
[]  [4]→EXIT!first(h); InterSansUrgence [σ] (cure(h))
endproc
process PleinSansUrgence [σ] (h:Hôpital): noexit:=
   PRESENT?p:Patient!(present(h,p)); PleinSansUrgence [σ] (h)
[]  EMERGENCY?p:Patient; PleinAvecUrgence [σ] (emergency(h,p))
[]  [3]→EXIT!first(h); InterSansUrgence [σ] (cure(h))
[]  [1*]→EXIT!first(h); VIDE [σ] (cure(h))
endproc
process PleinAvecUrgence [σ] (h:Hôpital): noexit:=
   PRESENT?p:Patient!(present(h,p)); PleinAvecUrgence [σ] (h)
[]  EMERGENCY?p:Patient; PleinAvecUrgence [σ] (emergency(h,p))
[]  [2]→EXIT!first(h); PleinAvecUrgence [σ] (cure(h))
[]  [5]→EXIT!first(h); PleinSansUrgence [σ] (cure(h))
endproc
endproc
   où σ dénote l'ensemble des portes PRESENT,EMERGENCY,EXIT,ADMIT
```

Figure 19 : Spécification du processus HOPITAL avec sous-processus des états abstraits

Problèmes de parallélisation des composants séquentiels

La méthode conduit à l'obtention de composants séquentiels. Cependant, dans de certains cas, plusieurs méthodes peuvent être servies en parallèle. C'est le cas par exemple pour l'hôpital de deux urgences, ou (sous certaines conditions) d'une admission et d'une sortie,... C'est le cas des éléments de C et O en général.

L'approche naïve consisterait à remplacer les opérateurs conditionnels ([]) par des opérateurs parallèles (|||) dans le processus serveur (figure 20). Toutefois, les branches correspondant aux éléments de CC et OC doivent être évaluées en section critique (ce qu'assurait l'emploi de l'opérateur [] mais pas |||). De plus, les expressions comportementales composées en parallèle ne peuvent partager des variables. Il ne pourrait y avoir de propagation des modifications des données internes. [GR90] fait remarquer que la récursivité dans les compositions parallèles devrait être évitée car elle est souvent difficile à comprendre, à implémenter efficacement et à vérifier automatiquement.

```
process HOPITAL
                [PRESENT,EMERGENCY,EXIT,ADMIT]
         (h: Hôpital): noexit:=
   PRESENT?p:Patient!(present(h,p)); HOPITAL [σ] (h)
|||  EMERGENCY?p:Patient; HOPITAL [σ] (emergency(h,p))
|||  [c-nonPlein(h)]→ADMIT?p:Patient; HOPITAL [σ] (admit(h,p))
|||  [c-nonVide(h)]→EXIT!first(h); HOPITAL [σ] (cure(h))
endproc
   où σ dénote l'ensemble des portes PRESENT,EMERGENCY,EXIT,ADMIT
```

Figure 20 : Spécification parallèle du processus HOPITAL

La solution que nous adoptons est donc la suivante :

– Dans les cas où un processus gère un (ou plusieurs) type de données, il ne contient que des opérateurs séquentiels. Dans la réalité, on se rend compte que ce n'est pas gênant car

pour avoir une cohérence dans l'accès aux données il faudrait de toutes façons recourir à un mécanisme de type moniteur/sections critiques qui rendrait alors le processus séquentiel.

– Dans les cas où un processus résulte de la composition de plusieurs autres processus composés ou non en parallèle, il ne peut gérer de types de données. Les compositions de processus peuvent être instanciées à partir d'une bibliothèque. Il serait aussi intéressant d'étudier les compositions de processus LOTOS comme résultant de la traduction d'une composition d'automate. Ces deux possibilités sont représentées dans la figure 21.

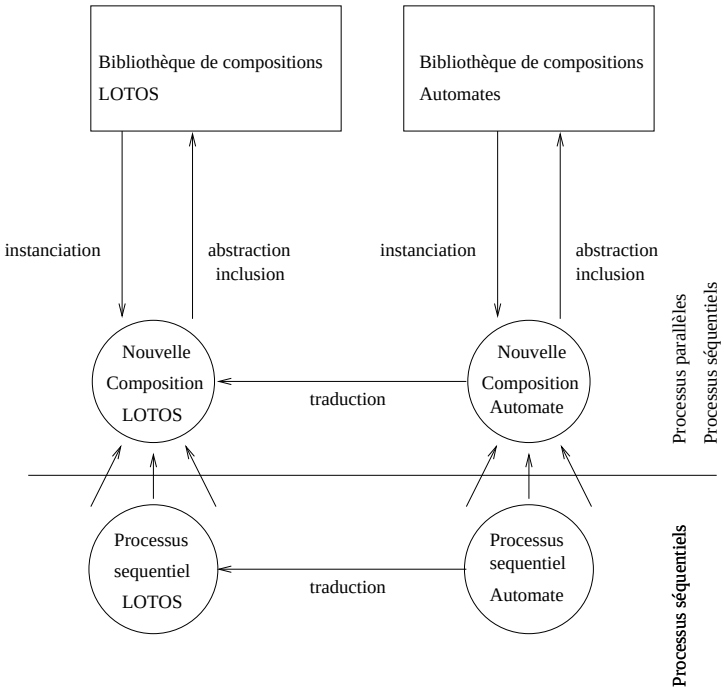


Figure 21 : Composition des processus séquentiels

3.4 Données, sortes et opérations

Cette partie traite de la création des types algébriques associés aux processus. L'idée est d'utiliser une discipline par constructeurs. Cela permet entre autres, comme le fait remarquer [GR90], de dériver des implémentations concrètes de la spécification grâce à des outils comme Caesar.

Les opérations ont été trouvées dans la phase précédente (tableau 8). Il s'agit de créer les axiomes leur correspondant. Des opérations auxiliaires sont nécessaires pour exprimer les axiomes relatifs aux observateurs de conditions (voir annexe A pages 39 et 40). L'idée est d'obtenir un type abstrait gracieusement présenté, au sens de [Bid82]. En effet, sous certaines conditions [Der95], cette forme assure de bonnes propriétés de consistance et de suffisante complétude.

Dans un premier temps, nous caractérisons les opérations de façon à obtenir un ensemble Ω d'opérateurs appelés *générateurs*. Cet ensemble va nous permettre dans un second temps de définir une présentation gracieuse du type d'intérêt (ici l'hôpital).

3.4.1 Caractérisation des opérations

Le modèle des TAG (*Types Abstraits Graphiques*) [And95] donne les principes permettant de caractériser les opérations et d'extraire d'un automate un ensemble minimal d'opérations permettant de définir toutes les valeurs d'un type d'intérêt (ici l'hôpital).

L'idée des TAG a été de remarquer que les différentes structures possibles d'un terme du type d'intérêt sont obtenues en parcourant les transitions qui arrivent dans l'état correspondant.

Nous considérons, comme dans ce modèle, que notre automate est la représentation dynamique d'un type de donnée appelé *type d'intérêt*. Les états représentent les états abstraits du type d'intérêt et les transitions les opérations sur ce type d'intérêt.

Les transitions sont représentées par un état origine, un état destination, une garde (qui doit être évaluée à vrai) et une opération.

Il y a correspondance (via β) entre les opérations du type d'intérêt et les transitions (gardées) de l'automate :

- les transitions d'un état sur lui même (`PRESENT`) correspondent aux opérations externes
- les transitions sans état d'origine (`INIT`) correspondent aux opérations internes de base
- les autres transitions (`EXIT`, `EMERGENCY`, `ADMIT`) correspondent aux opérations internes qui ne sont pas de base.

Les opérations sont séparées en opérations *secondaires* (elles s'expriment à l'aide d'autres opérations) et *primitives*. Un algorithme permet d'extraire du modèle dynamique un ensemble d'opérations internes primitives appelées *générateurs*. Cet ensemble est nécessaire pour construire toutes les valeurs (termes) du type d'intérêt. Il faut noter que cet ensemble peut ne pas suffire dans certains cas. Il faut donc le valider.

Cet algorithme consiste à trouver un ensemble minimal de transitions permettant de couvrir l'automate (c'est à dire de pouvoir atteindre tous les états à partir de l'état initial). Les générateurs sont ensuite obtenus en appliquant β_ω (cf 3.3.2) aux portes correspondant aux transitions.

Dans notre cas, l'ensemble des générateurs est constitué d'`init`, `emergency` et `admit`. Notons qu'`init` (voir 3.2.1), opération interne primitive de base, fait toujours partie des générateurs.

La construction des axiomes se fait ensuite en deux phases :

- extraction des préconditions à partir du modèle dynamique
- écriture des axiomes sous forme d'équations conditionnelles positives à l'aide du principe d'Ω-*dérivation* [Bid82].

3.4.2 Type abstrait gracieusement présenté et application de l'Ω-dérivation

Nous donnons ici les grands principes et définitions des types abstraits gracieusement présentés [Bid82].

$< T_i, \Sigma_i, E_i >$ est la présentation du type (abstrait) d'intérêt T_i dans le contexte spécifié par $< S, \Sigma, E >$. Ω est une famille d'opérations du type d'intérêt.

Un ensemble Ω-*complet* de termes est :

- soit réduit à un singleton $\{x\}$, avec x variable de type T_i;
- soit obtenu à partir d'un ensemble Ω-complet en dérivant au moins un terme par rapport à Ω.

Le principe de la *dérivation* en x d'un terme t par rapport à un ensemble G d'opérateurs (ayant un même codomaine) consiste à substituer à toutes les occurrences de la variable x dans t le terme $G(y_1, ..., y_n)$ avec y_i des nouvelles variables de type conforme au profil de G.

Prenons un exemple. Soit G_{nat} l'ensemble constitué des opérateurs $\{zero, succ\}$. $\{x\}$ (x de type nat) est un ensemble G_{nat}-complet de termes. En dérivant x en x par rapport à G_{nat} nous

obtenons l'ensemble $\{zero, succ(x')\}$, autre ensemble G_{nat}-complet de termes.

Un ensemble Ω-*complets de n-uplets* est constitué des n-uplets obtenus par combinaison et permutation d'un ensemble Ω-complet de termes avec renommage éventuel pour qu'aucune variable n'ait plus d'une occurrence dans le n-uplet.
Un ensemble Ω-*complet de F-termes* est l'ensemble des termes obtenus en substituant successivement dans le terme $F(x_1, ..., x_n)$ tous les n-uplets d'un ensemble Ω-complet de n-uplets au n-uplet $(x_1, ..., x_n)$.

On appelle *terme prédéfini* un terme composé uniquement d'opérateurs de Σ et de variables qui ne sont pas du type d'intérêt.

L'ensemble des Ω-*termes* est constitué :

- des variables de type T_i;
- des termes $F(t_1, ..., t_n)$ avec F appartenant à Ω, et t_i étant soit un Ω-terme soit un terme prédéfini.

Une Ω-*équation* est une équation de la forme $M = N$ avec M et N étant des Ω-termes.

Une *présentation Ω-complète d'un opérateur F* est un ensemble $\{E_1...E_p\}$ d'équations orientables de gauche à droite en $G_i = D_i$ se sorte que :

- l'ensemble des G_i constitue un ensemble Ω-complet de termes;
- les D_i sont des termes uniquement constitués d'opérateurs de Σ, d'Ω et de F-termes.
- le système de réécriture de termes $G_i \rightarrow D_i$ est à terminaison finie

Les ensembles Ω-complets étant minimaux, le système de réécriture correspondant est sans paires critiques.

L'idée de *présentation gracieuse* est basée sur l'idée que les équations d'un type abstrait d'intérêt T_i peuvent être partagées en :

- un ensemble éventuellement vide d'Ω-équations;
- pour chaque opération n'appartenant pas à Ω un ensemble d'équations constituant une définition Ω-complète de F.

Les présentations gracieuses sont donc, à la fois, une méthode d'obtention de spécification et un cadre qui assure que sous certaines conditions [Der95] les spécifications écrites ont de bonnes propriétés (consistance et suffisante complétude).

TAG et Ω-dérivation

Le spécifieur doit donner les parties droites des règles. Si cela est impossible, un pas d' Ω-dérivation est effectué. Les TAG s'appliquent dans un contexte objet et la dérivation se fait sur une variable particulière, celle correspondant à l'objet défini par le TAG. Il faut noter que l'Ω-dérivation ne permet pas, à la base, d'obtenir les prémisses des axiomes. Celles-ci sont obtenues grâce aux conditions des transitions de l'automate.
Une Ω-dérivation dans un état donné correspond à remplacer la variable h (le type d'intérêt) d'un terme par l'application d'un générateur qui arrive dans cet état. Pour cela, il suffit d'appliquer β_ω à la porte correspondant à la transition de l'automate.

Prenons le terme correspondant à `cure` (l'opération interne correspondant aux transitions `EXIT`) dans l'état PleinSansUrgence. Nous obtenons :

PleinSansUrgence(h) == vrai ==> cure(h) ==?

Trois transitions arrivent dans l'état PleinSansUrgence (figure 13, page 25). Nous ne tenons pas compte de la transition étiquetée par [5]EXIT car $\beta_\omega(EXIT)$: exit n'est pas un générateur. Les axiomes induits par la dérivation sur des opérations secondaires ne présentent pas d'intérêt car les opérations secondaires ont elles-mêmes des axiomes donnant, directement ou pas, leur définition en fonction des opérations primaires. Pour l'Ω-dérivation nous ne prenons donc en compte que la transition [7]ADMIT provenant de l'état InterSansUrgence et la transition [7*]ADMIT provenant de l'état Vide.

En Ω-dérivant ce terme sur PleinSansUrgence et en tenant compte du fait que $\beta_\omega(ADMIT) = admit : Hôpital \times Patient \rightarrow Hôpital$, nous obtenons donc :

InterSansUrgence(h) $\wedge$ [7] == vrai ==> cure(β_ω(ADMIT)(h,p)) == β_ω(ADMIT)(cure(h),p)

Vide(h) $\wedge$ [7*] == vrai ==> cure(β_ω(ADMIT)(h,p)) == h

soit

InterSansUrgence(h) $\wedge$ [7] == vrai ==> cure(admit(h,p)) == admit(cure(h),p)

Vide(h) $\wedge$ [7*] == vrai ==> cure(admit(h,p)) == h

avec [7]=[7*]=c-presquePlein(h)

Il faut remarquer que les axiomes des observateurs d'état peuvent être extraits de la table des conditions d'état (table 4, page 20), en ce qui les concerne il n'est pas nécessaire d'avoir recours à l'Ω-dérivation.

3.4.3 Commentaires sur les TAG

Le modèle des TAG (*Types Abstraits Graphiques*) est une description abstraite et formelle d'objets. Cette description est composée d'un modèle dynamique et d'un modèle fonctionnel.

Dans les TAG, le modèle fonctionnel est l'interprétation fonctionnelle du modèle dynamique. Les états sont des ensembles distincts de valeurs du type d'intérêt et les opérations des fonctions définies sur ces ensembles. Le modèle est complété par l'écriture des axiomes de la spécification algébrique correspondante.

Il faut noter que par notre méthode nous obtenons un automate du processus (modèle dynamique) équivalent à la partie dynamique du TAG HOPITAL [AR94]. La différence essentielle entre le TAG et notre automate réside dans le fait que les transitions dans l'automate ont une sémantique de communication (synchronisation entre un objet serveur et un ou plusieurs objets clients). Des différences existent aussi au niveau du traitement des observateurs (définis dans nos automates comme une transition d'un état vers ce même état).

3.4.4 Types génériques

Le type Patient de la spécification est un type générique. Il doit être instancié dans la spécification LOTOS. Pour cela il existe deux possibilités:

- Patient a une vie propre et un processus PATIENT paramétré par le type de données Patient est créé. C'est dans ce processus que la spécification du type Patient est faite.
- Patient est un type "amorphe" (par exemple juste un numéro de sécurité sociale) et est défini dans l'hôpital. Le type lui est alors propre.

3.4.5 Remarques

Le fait de traiter la partie données après la partie comportement permet, comme nous l'avons fait remarquer plus haut, d'obtenir un ensemble *minimal et complet* d'opérations dans la partie données. Ceci est entendu par rapport aux besoins préalablement exprimés dans la description informelle du processus.

La minimalité est assurée par le fait que seules des opérations servant dans la spécification de la partie dynamique sont spécifiées dans la partie algébrique.

La "complétude" résulte du couplage qui a été fait entre un processus et le type de données qu'il gère. Un processus peut être vu comme l'interface (au sens des services) d'un type de données. Les opérations constructrices du type de donnée résultent ainsi directement des communications sur les portes (de C ou CC) du processus associé.

On peut remarquer que dans notre approche la partie comportement et la partie spécification algébrique sont fortement liées : les opérations du type influent par exemple sur la possibilité ou non de faire une opération et le comportement dynamique a aussi des influences sur le fait qu'une opération du type de donnée soit possible (admise) ou non.

Nous pensons que les difficultés de réutilisation séparée du comportement ou du type de données et la plus grande difficulté au niveau des preuves et vérifications (il n'est pas suffisant d'effectuer des preuves et vérifications au niveau du comportement ou du type algébrique, mais il faut aussi prendre en compte les relations entre les deux) ne sont pas inhérentes à notre méthode mais à toute spécification intégrant à la fois les aspects contrôle et structures données.

Nous pensons qu'il est plus pertinent d'associer les structures de données à des composants séquentiels mais nous prévoyons toutefois d'étendre notre méthode, prévue pour des données gérées par des composants séquentiels, aux composants parallèles.

4 Conclusion

La spécification formelle présente de nombreux intérêts pour la conception des systèmes distribués. Nous pensons que la définition de méthodes simples et claires de conception de ces spécifications permettra de développer plus encore leur usage.

Nous avons ici défini une méthode pour la spécification en LOTOS tirant parti à la fois de l'approche orientée contraintes et de l'approche orientée état, les deux méthodes les plus couramment utilisées pour la conception en LOTOS.

Dans un premier temps, le spécifieur décrit informellement le problème en s'attachant aux services que rend le processus solution. Ce processus est ensuite décomposé en sous-processus (approche orientée contraintes) disposant d'interfaces externes bien définies (portes et typage des communications). Ces interfaces permettent de composer les sous-processus.

Dans un second temps, le spécifieur décrit le comportement interne des sous-processus les plus simples : les processus séquentiels. Ces derniers gèrent des types de données. L'étude des communications et leur effet sur ce type de donnée permettent de construire de façon semi-automatique un automate représentant le comportement interne du processus. Cet automate est ensuite traduit dans le langage de spécification LOTOS.

Notre méthode permet donc à la fois de facilement décomposer les processus et d'avoir des types de données cohérents avec les processus séquentiels. Comme avec les "agendas" décris par [GHD98], elle établit clairement ce qu'il est nécessaire de faire lors de chaque phase, quelles informations sont obtenues et comment il est possible de les utiliser automatiquement pour simplifier la tache du spécifieur.

Notre méthode est proche en termes de propriétés (abstraction de données, décomposition hiérarchique) de celle de [Cla92] permettant de développer des spécifications pour systèmes embarqués ; mais elle gagne en clarté en n'employant pas de porte unique de communication (avec un argument se référant à la méthode requise par le client).

Notons que notre approche par automate permet d'utiliser les outils disponibles pour vérifier les propriétés ou l'équivalence de systèmes (voir par exemple [Arn92]).

Notre méthode a été utilisée sur des exemples de taille réduite mais elle doit être testée sur des études de cas grandeur nature.

Références

[And95] Pascal André. *Méthodes formelles et à objets pour le développement du logiciel : Etudes et propositions.* PhD Thesis, Université de Rennes I (Institut de Recherche en Informatique de Nantes), Juillet 1995.

[AR94] Pascal André and Jean-Claude Royer. Introduction de concepts formels dans le développement à objets, Septembre 1994. Journées du GDR/PRC "Programmation" à Lille.

[AR97] E. Astesiano and G. Reggio. Formalism and method. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97*, volume 1214 of *Lecture Notes in Computer Science*, pages 93–114. Springer-Verlag, 1997.

[Arn92] André Arnold. *Systèmes de transitions finis et sémantique des processus communicants.* Etudes et recherches en informatique. Masson, 1992.

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

[Bid82] Michel Bidoit. Types abstraits algébriques : spécifications structurées et présentations gracieuses. In *Colloque AFCET, Les mathématiques de l'informatique*, pages 347–357, Mars 1982.

[Cla92] R.G. Clark. Using LOTOS in the Object-Based Development of Embedded Systems. In C.M.I. Rattray and R.G. Clark, editors, *The Unified Computation Laboratory*, pages 307–319, Department of Computing Science and Mathematics, University of Stirling, Scotland, 1992. Oxford University Press.

[Der95] N. Dershowitz. Hierarchical Termination. *Lecture Notes in Computer Science*, 968:89–105, 1995.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification*, volume 1. 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.

[Gar89] Hubert Garavel. *Compilation et vérification de programmes LOTOS.* Thèse de doctorat (PhD Thesis), Université Joseph Fourier, Grenoble, Novembre 1989.

[Gar90] Hubert Garavel. Introduction au langage LOTOS. Technical report, VERILOG, Centre d'Etudes Rhône-Alpes, Forum du Pré Milliet, Montbonnot, 38330 Saint-Ismier, 1990.

[GHD98] Wolfgang Grieskamp, Maritta Heisel, and Heiko Dörr. Specifying Embedded Systems with Statecharts and Z: An Agenda for Cyclic Software Components. In Egidio Astesiano, editor, *FASE'98*, volume 1382 of *Lecture Notes in Computer Science*, pages 88–106. Springer-Verlag, 1998.

[GLO91] Souheil Gallouzi, Luigi Logrippo, and Abdellatif Obaid. Le LOTOS : théorie, outils, applications. In *CFIP'91*, pages 385–404, 1991.

[GR90] Hubert Garavel and Carlos Rodriguez. An example of LOTOS Specification : The Matrix Switch Problem. Rapport SPECTRE C22, Laboratoire de Génie Informatique — Institut IMAG, Grenoble, June 1990. http://www.inrialpes.fr/vasy/Publications/Garavel-Rodriguez-90.html.

[HL97] Maritta Heisel and Nicole Lévy. Using LOTOS Patterns to Characterize Architectural Styles. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT'97 (FASE'97)*, volume 1214 of *Lecture Notes in Computer Science*, pages 818–832, 1997.

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

[ISO89a] ISO/IEC. ESTELLE: A Formal Description Technique based on an Extended State Transition Model. ISO/IEC 9074, International Organization for Standardization, 1989.

[ISO89b] ISO/IEC. LOTOS: A Formal Description Technique based on the Temporal Ordering of Observational Behaviour. ISO/IEC 8807, International Organization for Standardization, 1989.

[Led87] G. J. Leduc. LOTOS, un outil utile ou un autre langage académique? In *Actes des Neuvièmes Journées Francophones sur l'Informatique — Les réseaux de communication — Nouveaux outils et tendances actuelles (Liège)*, Janvier 1987.

[LFHH92] L. Logrippo, M. Faci, and M. Haj-Hussein. An Introduction to LOTOS: Learning by Examples. *Computer Networks and ISDN Systems*, 23:325–342, 1992. improved version available by ftp at lotos.csi.uottawa.ca.

[LLS97] Thomas Lambolais, Nicole Lévy, and Jeanine Souquières. Assistance au développement de spécifications de protocoles de communication. In *AFADL'97 Approches Formelles dans l'Assistance au Développement de Logiciel*, pages 73–84, 1997.

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

[Poi91] Thierry Poidevin. LOTOS. Rapport de stage de DESS, DESS Informatique, Université d'Orsay, 1991.

[Tur93] Kenneth J. Turner, editor. *Using Formal Description Techniques, an introduction to Estelle, Lotos and SDL*. Wiley, 1993.

[Tur97] K. Turner. Relating architecture and specification. *Computer Networks and ISDN Systems*, 29(4):437–456, 1997.

[VSVSB91] C.A. Vissers, G. Scollo, M. Van Sinderen, and E. Brinksma. Specification styles in distributed systems design and verification. *Theoretical Computer Science*, (89):179–206, 1991.

A Spécification complète

HOPITAL.lotos

```
specification HOPITAL [PRESENT, EMERGENCY, ADMIT, EXITT] : noexit

library
HOPITAL_NATURAL
endlib

behavior HOPITAL [PRESENT, EMERGENCY, ADMIT, EXITT] (init(10))

where

process HOPITAL [PRESENT, EMERGENCY, ADMIT, EXITT] (h:HopitalNAT) : noexit :=

PRESENT ?p:Patient !present(h,p);
HOPITAL [PRESENT, EMERGENCY, ADMIT, EXITT] (h)
[]
EMERGENCY ?p:Patient;
HOPITAL [PRESENT, EMERGENCY, ADMIT, EXITT] (emergency(h,p))
[]
[cNonPlein(h)] -> ADMIT ?p:Patient;
HOPITAL [PRESENT, EMERGENCY, ADMIT, EXITT] (admit(h,p))
[]
[cNonVide(h)] -> EXITT !first(h);
HOPITAL [PRESENT, EMERGENCY, ADMIT, EXITT] (cure(h))

endproc

endspec
```

HOPITAL.lib

```
library
BOOLEAN
endlib

library
NATURAL
endlib

type HOPITAL is

BOOLEAN, NATURAL

formalsorts Patient

formalopns
   eq : Patient, Patient -> BOOL
   ne : Patient, Patient -> BOOL

sorts Hopital
```

```
opns

(* GENERATEURS *)

init (*! constructor *) : NAT -> Hopital
admit (*! constructor *) : Hopital, Patient -> Hopital
emergency (*! constructor *) : Hopital, Patient -> Hopital

(* autres operations internes *)

cure : Hopital -> Hopital

(* operations externes *)

first : Hopital -> Patient
present : Hopital, Patient -> BOOL

(* observateurs d'etat *)

vide : Hopital -> BOOL
interSansUrgence : Hopital -> BOOL
interAvecUrgence : Hopital -> BOOL
pleinAvecUrgence : Hopital -> BOOL
pleinSansUrgence : Hopital -> BOOL

(* observateurs conditions *)

cNonVide : Hopital -> BOOL
cNonPlein : Hopital -> BOOL
cExisteUrgence : Hopital -> BOOL

(* observateurs associés aux ``gardes'' des transitions *)

cPresquePlein : Hopital -> BOOL
cUnSeulPatient : Hopital -> BOOL
cUneSeuleUrgence : Hopital -> BOOL

(* operations externes auxiliaires *)

nbNormaux : Hopital -> NAT
nbUrgences : Hopital -> NAT
nbTotal : Hopital -> NAT
nbPlaces : Hopital -> NAT

eqns

forall n : NAT,
         p, q : Patient,
         h : Hopital

ofsort NAT

(* axiomes des opérations auxiliaires *)
```

```
nbPlaces(init(n)) = n ;
nbPlaces(admit(h,p)) = nbPlaces(h) ;
nbPlaces(emergency(h,p)) = nbPlaces(h) ;

nbNormaux(init(n)) = 0 ;
nbNormaux(admit(h,p)) = 1 + nbNormaux(h) ;
nbNormaux(emergency(h,p)) = nbNormaux(h) ;

nbUrgences(init(n)) = 0 ;
nbUrgences(admit(h,p)) = nbUrgences(h) ;
nbUrgences(emergency(h,p)) = 1 + nbUrgences(h) ;

nbTotal(h) = nbNormaux(h) + nbUrgences(h) ;

ofsort BOOL

(* axiomes des observateurs de conditions *)

cNonVide(h) = nbTotal(h) > 0 ;

cNonPlein(h) = nbNormaux(h) < nbPlaces(h) ;

cExisteUrgence(h) = nbUrgences(h) > 0 ;

cPresquePlein(h) = nbNormaux(h) eq ( nbPlaces(h) - 1 ) ;

cUnSeulPatient(h) = nbTotal(h) eq 1 ;

cUneSeuleUrgence(h) = nbUrgences(h) eq 1 ;

(* axiomes des observateurs d'état *)

vide(h) = not(cNonVide(h)) and not(cExisteUrgence(h)) and cNonPlein(h) ;

interSansUrgence(h) = cNonVide(h) and not(cExisteUrgence(h)) and cNonPlein(h) ;

interAvecUrgence(h) = cNonVide(h) and cExisteUrgence(h) and cNonPlein(h) ;

pleinAvecUrgence(h) = cNonVide(h) and cExisteUrgence(h) and not(cNonPlein(h)) ;

pleinSansUrgence(h) = cNonVide(h) and not(cExisteUrgence(h)) and not(cNonPlein(h)) ;

(* axiomes de présent *)

present(init(n),p) = false ;
p ne q => present(admit(h,p),q) = present(h,q) ;
p eq q => present(admit(h,p),q) = true ;
p ne q => present(emergency(h,p),q) = present(h,q) ;
p eq q => present(emergency(h,p),q) = true;

ofsort Hopital

(* axiomes de cure *)
```

```
vide(h) => cure(admit(h,p)) = h ;
interAvecUrgence(h) => cure(admit(h,p)) = admit(cure(h),p) ;
interSansUrgence(h) => cure(admit(h,p)) = admit(cure(h),p) ;
not(cExisteUrgence(h)) => cure(emergency(h,p)) = h ;
cExisteUrgence(h) => cure(emergency(h,p)) = emergency(cure(h),p) ;

ofsort Patient

(* axiomes de first *)

vide(h) => first(admit(h,p)) = p ;
(* en fait inutile : peut tjs se ramener a cNonVide(h) and cNonPlein(h) *)
interAvecUrgence(h) => first(admit(h,p)) = first(h) ;
interSansUrgence(h) => first(admit(h,p)) = first(h) ;
not(cExisteUrgence(h)) => first(emergency(h,p)) = p ;
cExisteUrgence(h) => first(emergency(h,p)) = first(h) ;

endtype
```

PATIENT.lib

```
library
NATURAL
endlib

type PATIENT is NATURAL
renamedby
sortnames Patient for NAT
endtype
```

HOPITAL NATURAL.lib

```
library
HOPITAL
endlib

type HOPITAL_NATURAL is HOPITAL

actualizedby NATURAL using
sortnames NAT for Patient
opnnames eq for eq
ne for ne

endtype
```