République Algérienne Démocratique et Populaire

Université Mentouri de Constantine
Faculté des Sciences de l'Ingénieur
Département d'Informatique

MODELES DU PARALLELISME

5ème Année Ingénieur en Informatique
Option : Systèmes Parallèles et Distribués

Année universitaire 2004/2005

Présenté par : Dr. Djamel –Eddine SAIDOUNI

Laboratoire LIRE
Equipe Vision et Infographie

Table des matières

Chapitre 1 : Introduction

1) PROBLEMATIQUE :

Au cours des dernières années, les progrès technologiques dans les domaines des réseaux et des télécommunications peuvent être considérés comme une véritable révolution dont l'impact est direct sur les systèmes distribués en général et les systèmes concurrents en particulier.

En particulier, les systèmes distribués offrent une solution simple aux besoins croissants de performance par l'exploitation des ressources disponibles dans le système. La division de tache complexe en processus simples permet d'en simplifier tant la conception que la validation des composantes individuelles. L'arrivée récente de systèmes d'exploitation multi-tâches et multiprocesseurs disponibles pour le grand public ne fait que confirmer cette tendance.

Cependant, le développement d'applications distribuées ou concurrentes est une tache très difficile qui nécessite la prise en compte de plusieurs paramètres à savoir les contraintes de coopération inter-processus coopératifs qui exigent la prise en considération de l'indéterminisme, la synchronisation ainsi que certaines propriétés qualitatives que doit avoir ces applications telle que l'absence de l'interblocage et de la famine. La difficulté dans la conception des applications réparties est posée principalement par l'absence d'une perception de l'état global de telles applications.

On peut dire que sans méthodes de validation appropriées, l'utilisation des systèmes concurrents et distribués devient non fiable du fait que le comportement de tels systèmes serait imprévisible.

Afin de valider avec certitude la spécification de systèmes concurrents, des méthodes d'analyse formelle apportant une certitude mathématique ont été récemment développées. Elles permettent de décrire sans ambiguïté le comportement de ces systèmes.

Le succès relatif remporté par l'utilisation des méthodes formelles fait qu'elles commencent purement universitaires. Depuis, de grandes compagnies oeuvrant dans le domaine des systèmes concurrents commencent également à réaliser l'importance de ces méthodes et tentent d'en imposer l'utilisation.

Ces méthodes formelles sont basées sur les modèles de spécification et les modèles sémantiques du parallélisme ainsi que les approches de vérification.

2) Modèles du Parallélisme :

Durant les vingt dernières années, plusieurs modèles du parallélisme ainsi que leurs sémantiques ont été étudiés dans le cadre de la théorie de la concurrence. Ils ont été utilisés pour donner une sémantique aux langages de description des processus et pour fournir une

base pour différentes notions et définitions d'équivalences de comportements. Ces modèles peuvent être classés en deux catégories :

- Les modèles de spécification du parallélisme.
- Les modèles sémantiques du parallélisme.

2.1) Modèles de spécification du parallélisme :
ces modèles offrent les moyens formels pour la spécification et l'analyse d'applications concurrentes et réparties. Parmi ces modèles, nous pouvons citer les algèbres de processus (ACP, CCS, CSP, …) [BERK85, Mil80, Mil83, Mil89,Hoa85] et les techniques de description formelle (ESTELLE, LOTOS, SDL) [ISO88b, BOLB87, ISO88a, CCI88].

L'efficacité des modèles formels tel que LOTOS, a été largement montrée et démontrée dans la littérature. L'utilisation de spécifications formelles a plusieurs intérêts qui peuvent être résumés dans les points suivants :

Permettent une compréhension approfondie du logiciel à développer. Elles mettent en évidence la plupart des ambiguïtés laissées dans l'ombre par les spécifications informelles. Le coût de telles imprécisions est de plus en plus important lorsque ces imprécisions sont découvertes tardivement.

Les spécifications formelles sont basées sur des représentations mathématiques, ce qui permet la validation de leurs propriétés.

Bien que la spécification formelle ne soit pas nécessairement directement exploitable, on peut, dans la plupart du temps, en dériver un prototype qui réalise partiellement les fonctionnalités spécifiées, et les tester durant la phase de développement.

Dans ce qui suit nous rappelons quelques modèles de spécification du parallélisme.

Les réseaux de Petri :

Les réseaux de Petri sont un outil graphique et mathématique qui s'applique à un grand nombre d'applications où les notions d'événements et d'évolutions simultanées sont importantes.

Un réseau de Petri est un quadruplet R = < **P**, **T**, **Pre**, **Post** > où :

- **P** : est un ensemble fini de places.
- **T** : est un ensemble fini de transitions.
- **Pre** : $P \times T \rightarrow N$ est l'application incidence avant (place précédente).
- **Post** : $P \times T \rightarrow N$ est l'application incidence arrière (place suivante).

Il permet de représenter les aspects de *choix*, de *séquencement* et de *parallélisme* entre comportements. Donc, c'est un modèle élégant pour décrire le parallélisme.

Les techniques de description formelle :

Les langages formels de description de protocoles qui rencontrent le plus de succès empruntent beaucoup de notions de *logique* et d'*algèbre* qu'à celles de l'informatique traditionnelle, sur les trois qui sont normalisés à l'UIT et à l'ISO, LOTOS est d'origine algébrique, les deux autres ; ESTELLE et LDS, sont basés sur la notion d'automates communicants et sont en fait assez proches des langages de programmation et assez proches l'un de l'autre. Ce sont ces deux derniers qui sont les plus employés, LDS est notamment assez soutenu dans le monde des télécommunications.

ESTELLE :

Le langage ESTELLE (**E**xtended **S**tate **T**ransition **L**anguage) est fondé sur le modèle des automates d'états finis étendu avec le langage 'PASCAL'. Il inclut tout le langage PASCAL et l'encapsule dans des éléments qui en font un véritable langage pour l'expression de comportements parallèles. Trois caractéristiques principales sont à noter :

- Une spécification est composée de plusieurs modules. Le langage ESTELLE permet de bien spécifier les interfaces de chacun d'entre eux.
- La description de comportement de chaque module est assez fine et précise, pour que le comportement de la spécification ne soit pas ambigu.
- La notion de typage (définition de type, puis la création d'exemplaires) du langage est étendu aux objets parallèles (modules, canaux,…etc.). ESTELLE utilise la notion de module d'interconnexion et de la structuration comme concepts de base pour spécifier les systèmes.

LDS :

Développé dès 1974, a connu plusieurs versions successives dont la plus récente, datant de 1988, et est standardisée par la recommandation Z.100 du CCITT. Comme Estelle, SDL est basé sur une extension du modèle des automates d'états finis. SDL permet de spécifier des automates structurés de manière arborescente, qui fonctionnent de manière asynchrone et qui communiquent par échanges de signaux. Chaque automate possède des variables locales et une file non bornée pour stocker les messages reçus. Les données sont décrites par des types algébriques (un type comprend des domaines de valeurs), des opérations sur ces valeurs et des équations qui définissent la sémantique de ces opérations. De nouveaux types peuvent être créés à partir des types existants par enrichissement, renommage, … Enfin, SDL possède deux syntaxes : la forme texte, analogique à un langage de programmation classique, et la forme graphique, dont les éléments de base sont des boites connectées par des flèches.

Les Algèbres de Processus :

Les algèbres de processus, comme CCS [Mil80], CSP [Hoa85], ACP [BERK85] ou la technique de description formelle LOTOS [ISO87], constituent un cadre mathématique pour spécifier les systèmes communicants, par composition et transformation de comportements élémentaires appelés *agents* dans CCS([Mil89]) ou *processus* dans la plupart des autres langages ([Hoa85, BERK85, Zui90, ISO87]).

2.2) Les modèles sémantiques du parallélisme :

comme leur nom l'indique, ces modèles sont utilisés pour exprimer la sémantique du parallélisme des modèles de spécification. Ces modèles peuvent etre classés en deux grandes catégories :

les modèles d'entrelacement :

pour ces modèles, l'exécution en parallèle de deux actions (une action est une activité élémentaire au niveau d'abstraction considéré) est représentée par l'entrelacement arbitraire de ces deux actions, c'est-à-dire par leur exécution alternée. Pour utiliser ces modèles à bon escient, il est indispensable de faire l'hypothèse d'atomicité des actions. Par ailleurs, dans de nombreuses applications, l'atomicité des actions est une abstraction acceptable, ce qui rend la

simplicité de ces modèles très attrayante. Parmi les modèles d'entrelacement, nous pouvons citer les systèmes de transitions étiquetées [Mil80, Arn92] et les arbres de synchronisation :

Système de Transitions Etiquetées :

La structure fondamentale du modèle d'Arnold-Nivat est l'automate fini. Celui-ci permet de représenter le comportement des systèmes concurrents à modéliser à l'aide de graphes orientés.

Dans le modèle, les *sommets* et les *arcs* de ce graphe orienté sont appelés respectivement *états* et *transitions*. Ces états et transitions sont associés avec une chaîne de caractères de manière à pouvoir les distinguer entre eux. Ces chaînes de caractères s'appellent *noms* lorsqu'elles sont associées aux états, et *étiquettes* lorsqu'elles sont associées aux transitions.

Le graphe orienté lui-même, avec ses chaînes de caractères associées, est appelé *Système de Transitions Etiquetées*.

Les états d'un système de transitions sont directement associés aux états du système concurrent qu'il modélise. Les transitions quant à elles, jouent un rôle fondamental dans un système de transitions car ce sont elles qui servent à modéliser les actions faites par le système concurrent, l'étiquette d'une transition représentant le type d'action qui est effectuée.

Cependant, et contrairement aux noms d'états, plusieurs transitions peuvent avoir les mêmes étiquettes, ceci est même nécessaire pour exprimer certaines propriétés importantes, telle le non-déterminisme. On ne peut donc pas se référer uniquement aux étiquettes pour identifier une transition.

En tout temps, dans le système de transitions, il n'y aura qu'un seul état qui sera actif. Celui-ci est nommé état courant et les étiquettes des transitions ayant comme état source l'état courant représenteront l'ensemble des actions que le système de transitions peut effectuer à cet instant précis.

Formellement, un système de transitions (étiquetées) est un quadruplait :

$S = (S, \sum, \Delta, s0)$ où :

- S est un ensemble (dénombrable) d'états.
- $\sum$ est un ensemble (dénombrable) d'actions (dites *observables*).
- $\Delta \subseteq S \times (\sum \cup \{\tau\}) \times S$: est l'ensemble des transitions, $\tau \notin \sum$ est appelée action invisible (interne ou non-observable). Un élément $(x, \mu, y) \in \Delta$ est noté $x \xrightarrow{\mu} y$.
- $s0 \in S$ est l'état initial de S.

Un système de transitions est communément considéré ([Mil89, Abr87, NM90, Lar87]…) comme un ensemble de processus S exécutant les actions de $\sum \cup \{\tau\}$ selon les règles de transition spécifiées par Δ. On peut en effet identifier chaque état de S au processus représenté par le système de transitions $(S, \sum, \Delta, s)$. s0 est un processus particulier auquel est identifié le comportement initial du système de transitions.

Un exemple de système de transitions étiquetées est donné par la ***Figure 1*** et pour lequel :

$S = \{ s0, s1, s2, s3, s4, s5\}$

$\Sigma = \{ a, b, c\}$
$\Delta = \{(s0, a, s1), (s0, \tau, s4), (s4, \tau, s1), (s1, b, s2), (s1, c, s3), (s4, a, s5)\}$.

D'après la notation $\xrightarrow{\mu}$ on peut écrire :

$\Delta = \{s0 \xrightarrow{a} s1, s0 \xrightarrow{\tau} s4, s4 \xrightarrow{\tau} s1, s1 \xrightarrow{b} s2, s1 \xrightarrow{c} s3, s4 \xrightarrow{a} s5 \}$.

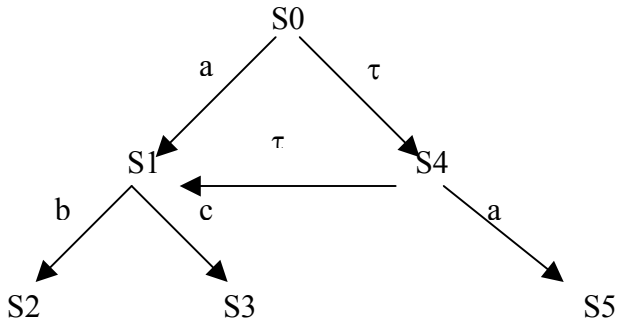


Figure1 : Exemple d'un système de transitions étiquetées

Un système de transitions S est dit fini lorsque l'ensemble de ses états S et l'ensemble des actions observables Σ sont finis.

Une autre relation de transition, $\{ \stackrel{\mu}{\Rightarrow} \}(\mu \in \Sigma \cup \{\varepsilon\})$ est habituellement définie par :

- $s \stackrel{\varepsilon}{\Rightarrow} s` : s = s` \text{ ou } s \xrightarrow{\tau} s1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} sn \xrightarrow{\tau} s`$.
- $s \stackrel{\mu}{\Rightarrow} s` : s \stackrel{\varepsilon}{\Rightarrow} s1 \xrightarrow{\mu} s2 \stackrel{\varepsilon}{\Rightarrow} s`$.

Les notations suivantes sont aussi utilisées :

- $s \stackrel{\mu}{\Rightarrow}$ signifie que s admet une dérivation par μ : $\exists s` \; s \stackrel{\mu}{\Rightarrow} s`$.
- $s \stackrel{\mu}{=/\Rightarrow}$ signifie $\neg (s \stackrel{\mu}{\Rightarrow})$.
- $Out(s) = \{\mu \in \Sigma \mid s \stackrel{\mu}{\Rightarrow} \}$ dénote l'ensemble des actions visibles que le système peut exécuter, à partir de l'état s.

Cette relation est étendue aux séquences (i.e. mots de Σ : $\sigma \in \Sigma^*$) comme suit :

- Si σ est la séquence $\mu 1 \ldots \mu n$, on écrit $s \stackrel{\sigma}{\Rightarrow} s`$ quand : $s \stackrel{\mu 1}{\Rightarrow} s1 \stackrel{\mu 2}{\Rightarrow} \ldots \stackrel{\mu n-1}{\Rightarrow} sn \stackrel{\mu n}{\Rightarrow} s`$.

La séquence vide est notée ε. De façon similaire à la définition de l'ensemble de sortie ***out***, on appelle « *traces* d'un état », l'ensemble des séquences d'actions, $\sigma \in \Sigma^*$, qui

peuvent être exécutées à partir de cet état. Soit $Tr(s) = \{ \sigma \in \sum^* \mid s \overset{\sigma}{\Rightarrow} \}$. Les traces d'un STE désignent celles de son état initial.

Définition : **(STE déterministe)**

Un STE est déterministe ssi aucun état n'admet plus d'un successeur par action. Formellement :

S est déterministe ssi $\forall a \in \sum, \forall s \in S, (s \overset{a}{\Rightarrow} s1 \wedge s \overset{a}{\Rightarrow} s2)$ implique $s1 = s2$.

Les systèmes de transitions permettent de représenter aisément et avec concision des concepts fondamentaux des systèmes concurrents, tel le choix, le non-déterminisme et séquencement.

Les Arbres de Synchronisation :

Avant de définir les arbres de synchronisation nous définissons les arbres de synchronisation rigides (RST).

Un **RST** de sort L est un arbre raciné non ordonné et a des branchements finis, dont les arcs sont étiquetés par des membres de L [Milner 80]. Cet arbre est rigide parce qu'il représente les comportements d'un processus rigide (actions atomiques).

Définition : **(Arbre de Synchronisation)**

n arbre de synchronisation (**ST**) de sort L est un arbre raciné non ordonné, et a des branchements finis, chaque arc est étiqueté par un membre de $L \cup \{\tau\}$, où τ est l'action non observable.

D'après cette définition, un arbre de synchronisation est un arbre de synchronisation rigide qui contient des arcs étiquetés par τ.

Les modèles de non-entrelacement :

appelés parfois modèles du vrai parallélisme ou encore modèles d'ordre partiel, ces modèles ont la capacité de décrire le comportement d'un système en représentant les relations de dépendance et, par dualité, le parallélisme entre occurrences d'actions. L'intérêt de ces modèles est leur aptitude à considérer des actions non atomiques. L'utilisation de ces modèles en tant que domaines sémantiques pour les modèles de spécification permet la conception de systèmes par raffinement successif. Le raffinement consiste à remplacer une action par un processus et cette opération peut etre interprétée dans le modèle sémantique sous-jacent. Ils se divisent en deux catégories de modèles, ceux dits d'ordre partiel et ceux du vrai parallélisme :

Modèle d'Ordre Partiel :

C'est un modèle sémantique dont le principe consiste à considérer les relations de dépendance entre les occurrences d'actions. Cependant, dans [vanGlabbeek89], il a été

montré que ces sémantiques ne sont ni nécessaires ni suffisantes pour la levée de l'hypothèse d'atomicité des actions. Parmi les modèles qui adoptent ce principe :

les arbres causaux [DD89], les arbres causaux dynamiques [Sai96] et les systèmes de transitions étiquetées causaux [Coe93].

Tout d'abord, nous présentons la sémantique de causalité qui est la base de ces modèles. Dans l'approche basée sur la causalité, les transitions sont des événements qui correspondent aux occurrences d'actions. A chaque transition sont associés :

- Le nom d'événement.
- Le nom d'une action.
- L'ensemble des noms des événements qui ont conditionné l'occurrence de cette transition.

- Arbres Causaux :

Les arbres causaux ont été définis par Darondeau et Degano[DD89], il peuvent être considérés comme un enrichissement des arbres de synchronisation de Milner [Mil80]. Dans un arbre causal, à chaque transition observable (i.e. arc dont l'action correspondante est visible), est associé un ensemble de références. Une référence est un entier naturel correspondant au nombre de transitions observables qui séparent cette transition de la transition qui l'a causée.

- Arbres Causaux Dynamique :

On peut considérer que les arbres causaux dynamiques sont un enrichissement des arbres causaux. En effet, on associe à chaque transition de l'arbre causal le nom d'un événement qui identifie son occurrence, nous pouvons alors utiliser les noms de ces événements pour faire référence aux causes des transitions.

- Systèmes de Transitions Etiquetées Causaux :

Le modèle des systèmes de transitions étiquetées causaux a été introduit par Rosvelter Coelho da Costa [Coe93]. Il peut être considéré comme un modèle de vrai parallélisme très proche du modèle des arbres causaux [DD89]. Il a été utilisé pour exprimer la sémantique de CCS [CC92a, Coe93], d'un sous ensemble de LOTOS [CC91, CC93], des réseaux de Petri [CC92b] et pour l'étude de raffinement d'action dans LOTOS [CS92a, CS94a].

Modèle du Vrai Parallélisme :

- Modèle des Structures d'Evénements :

Ces modèles sont définis à partir d'un ensemble d'événements et de relations entre ces événements.

Evénement : chaque événement est étiqueté par le nom d'une action, et représente une occurrence de cette action à un instant donné. De ce fait, un événement apparaît au plus une fois dans une structure d'événements. Les événements sont notés par e1, e2,…, et les noms des actions (étiquettes) sont notés par leurs noms écrits juste à coté.

Relation entre événements : on distingue trois relations qui sont respectivement la relation de précédence, la relation de conflit et la relation d'indépendance.

- Systèmes de Transitions Asynchrones :

Les systèmes de transitions asynchrones, en abrégé **STA** ont été introduits de manière indépendante par Shields [Shi85] et Bednarczyk [Bed87]. Le modèle des STAs peut être considéré comme extension du modèle des STs.

Dans ce modèle, le parallélisme entre deux événements est représenté par un losange. Cependant, la présence d'un losange dans un STA n'implique pas que les événements qui le constituent soient indépendants.

- Les Arbres Maximaux :

La structure des *arbres maximaux* se base sur le principe de la sémantique de maximalité telle qu'elle est donnée dans [Sai96]. La sémantique de maximalité peut être considérée comme étant la réconciliation des sémantiques d'ordre partiel et de la ST-sémantique.

Le principe de *maximalité* consiste à utiliser les relations de dépendance entre les occurrences d'actions pour associer à chaque état du système les actions qui sont potentiellement en cours d'exécution.

Le résultat attendu du modèle des arbres maximaux est de pouvoir exprimer le comportement des systèmes concurrents, c'est-à-dire pour chaque système, d'exprimer ses différents états et de déterminer pour chacun de ces états les actions qui sont entrain de s'exécuter en parallèle.

3) Vérification :

En général, les techniques de vérification traduisent les spécifications, écrites dans un modèle de spécification, en une structure explicite, décrite souvent sous la forme d'un graphe qui représente les différents états du système spécifié. Cette structure interne doit bien entendu respecter la sémantique du modèle de spécification. Il est également souhaitable qu'elle soit la plus proche possible du modèle sémantique utilisé; par exemple, pour les sémantiques d'entrelacement, on utilise un système de transitions étiquetées, qui correspond à une représentation finie d'un arbre de synchronisation qui peut lui etre infini.

Dans ce mémoire, nous nous intéressons à la technique de description formelle LOTOS (que nous exposons au *chapitre 2*), en tant que modèle (langage) de spécification [BOLB87, ISO88a]. L'utilisation d'une telle technique, lors de la conception d'un système de communication, conduit à la manipulation, pour un même système, de deux objets : *la Spécification* et *l'Implémentation*.

La validation du système conçu consiste à montrer que son implémentation est *conforme* à sa spécification. La procédure de validation est connue en milieu industriel sous le nom de *qualification* ou *certification*.

Les approches de validation peuvent etre différentes selon le type des objets appelés : *Spécification* et *Implémentation*. Deux approches de validation peuvent etre distinguées : ***l'approche boite blanche*** et ***l 'approche boite noire***.

Dans les deux approches, *la conformité* est une relation de satisfaction qui doit lier l'implémentation et la spécification. Néanmoins, ces deux approches se distinguent par la façon de vérifier que cette relation est satisfaite.

L'approche dite *boite blanche*, peut être appliquée dès lors que la spécification et l'implémentation sont deux structures entièrement et librement explorables. Cette approche est communément appelée « ***vérification*** ».

La seconde approche dite boite noire est celle traditionnellement appelée « ***Test*** » des implémentations. Elle se substitue à la première approche lorsque l'implémentation n'est accessible que via ses points d'interaction avec l'extérieur appelés ports de communication.

Vu les contraintes supplémentaires qu'impose sa réalisation, La validation par le test qui est applicable pour tout type de représentation de l'implémentation, est connue pour être moins efficace qu'une vérification directe de la conformité. Il est alors important de fournir des techniques de vérification (des relations de conformité) qui servent à réaliser la procédure de validation lorsqu'il est possible de manipuler une implémentation au même titre que sa spécification, notamment au cours d'un cycle de développement d'un système communicant par raffinements successifs.

De ce point de vue apparaît, l'utilité de caractériser les relations de conformité par des définitions automatiquement vérifiables.

D'un autre coté, il est aussi utile de pouvoir générer automatiquement les tests que l'on doit appliquer pour valider les implémentations lorsque celles-ci ne peuvent être validées que par le test.

Chapitre 2 : La Technique de Description Formelle LOTOS

1- INTRODUCTION

Le sujet de spécification formelle est devenu de plus en plus important dans les domaines de communication et des systèmes distribués. Comme ces sujets mûrissent, il y a une demande croissante pour les techniques qui peuvent capturer sans ambiguïté le comportement des composants du système. Ceci est particulièrement important dans le monde des standards. Pendant les quelques années passées, un nombre de techniques de description formelle (*FDT*) sont apparues pour satisfaire cette demande, à savoir SDL, ESTELLE et LOTOS (Language Of Temporal Ordering Spécification), qui a été acceptée comme un standard international par l'ISO en 1988. Depuis, cette technique de description formelle a été largement appliquée dans les domaines de spécification des protocoles de communication, et plus récemment dans les travaux portant sur le traitement distribué ouvert.

LOTOS, le langage parallèle qui s'inspire des algèbres de processus, notamment CCS [Mil80] et CSP [Hoa85], est l'une des deux techniques de description formelle développées dans l'ISO (Organisation Internationale pour la Standardisation) pour la spécification formelle des systèmes distribués ouverts en général et ceux apparentés à l'architecture des réseaux d'ordinateurs de l'OSI (Open Systems interconnection) en particulier. Il a été développé par des experts en FDT du groupe de l'ISO/ TC97/ SC21/ WG1 FDT /subgroupC au cours des années 1981-1986.

L'idée fondamentale à partir de laquelle LOTOS a été développé considère que les systèmes peuvent être spécifiés en définissant la relation temporelle entre les interactions qui constituent le comportement d'un système observable de l'extérieur. Contrairement à ce que le nom semble dénoter, cette technique de description formelle n'est pas apparentée à la logique temporelle, mais elle est basée sur les méthodes algébriques de processus. De telles méthodes ont été introduites en premier par le travail de Milner sur CCS [Mil80], suivi par plusieurs théories proposant d'autres algèbres de processus ; chacune de ces algèbres se distinguent par la sémantique des opérateurs les constituant.

LOTOS inclut un composant qui se charge de la description des comportements des processus et des interactions. Il inclut aussi un second composant qui s'occupe de la description des structures de données et des expressions de valeurs.

Cette partie de LOTOS est basée sur la théorie formelle des types de données abstraits, et en particulier, sur l'approche de spécification des équations des types de données, avec une sémantique d'algèbre initiale. La plupart des concepts dans ce composant sont inspirés du langage de description des types de données abstraits ACT ONE [Rei85].

LOTOS est une FDT généralement applicable aux systèmes de traitement d'information concurrents et distribués. Cependant, il a été développé particulièrement pour l'OSI. Les objectifs principaux pour une telle technique est qu'elle doit permettre la production des spécifications standards de l'OSI qui sont :

- Description d'implémentations indépendantes, complètes, précises et non-ambiguës des standards.

- Documents faciles à lire pour les utilisateurs de l'OSI, les implémenteurs et les testeurs de conformité.

- Une base formellement bien définie pour la vérification et la validation des standards, et pour le test de conformité de leurs implémentations.

Il est clair que ces objectifs sont particulièrement importants pour une architecture standard, distribuée, tel que l'OSI. Les machines doivent communiquer et coopérer l'une avec l'autre, et les spécifications ambiguës, informelles pouvaient aisément mener à des implémentations incomparables. En outre, la possibilité d'exécuter des analyses rigoureuses d'un protocole au niveau de conception, qui est dans un stade anticipé de prolifération des erreurs dans le grand nombre attendu de ses implémentations.

La considération des besoins ci-dessus a conduit à un nombre de critères de conception pour le langage lui même.

Les critères généraux qui ont déterminé la définition présente de LOTOS sont :

Puissance d'expression : une FDT doit être capable d'exprimer le large rang de propriétés qui sont appropriées pour la description des services, des protocoles et des interfaces OSI.

Définition formelle : la syntaxe et la sémantique d'une FDT doivent avoir une définition formelle et complète. En particulier, le modèle formel sur lequel la sémantique du langage est basée doit supporter le développement d'une théorie analytique pour la vérification, la validation et ***le test de conformité.***

Abstraction : les constructeurs du langage doivent représenter les concepts architecturaux appropriés à un niveau suffisamment élevé d'abstraction, où les détails orientés implémentation ne sont pas exprimés. Ce qui est en faveur d'une représentation précise des besoins.

Structure : une FDT doit offrir des moyens pour structurer une spécification en une construction significative et intuitivement plaisante. Une bonne structuration suppose la lisibilité, la facilité de la maintenance, et peut simplifier l'analyse. S'il est désirable, une structure peut être aussi utilisée pour refléter l'organisation logique ou même physique d'une implémentation.

2- PROCESSUS :

En LOTOS, un système concurrent distribué est vu comme un processus consistant éventuellement, en plusieurs sous-processus. Un sous-processus est un processus en lui-même, de telle sorte qu'en général une spécification LOTOS décrit un système via une hiérarchie de définitions de processus.

Un processus est une entité capable d'exécuter des actions internes (non observables), et d'interagir avec les autres processus, qui forment son environnement. Les interactions complexes entre les processus sont accumulées en dehors des unités élémentaires de synchronisation que nous appelons *événements*, ou *interactions*, ou *actions* simples.

Les événements supposent la synchronisation de processus, parce que les processus qui interagissent dans un événement (ils peuvent être deux ou plus) participent à son exécution au même moment dans le temps. De telles synchronisations peuvent entraîner l'échange de données. Les événements sont atomiques dans le sens qu'ils se présentent instantanément, sans consommer de temps.

Un événement se présente à un point d'interaction, ou une porte, et dans le cas de synchronisation sans échange de données, le nom de l'élément et le nom de la porte coïncident.

L'environnement d'un processus P, dans un système S, est formé par l'ensemble des processus de S avec lesquels P interagit, plus un processus observateur non-spécifié, peut être un humain, qui est supposé être toujours prés à observer tout ce qui est observable de ce que le système peut faire. Et, pour être compatible avec le modèle, l'observation n'est qu'une interaction. D'où la terminologie, dire que le processus P exécute une action observable signifie une interaction de P avec son environnement.

La représentation la plus abstraite du processus P, capable d'interagir avec son environnement via des portes, est illustrée par la boite noire dans la suivante (***Figure1)*** :

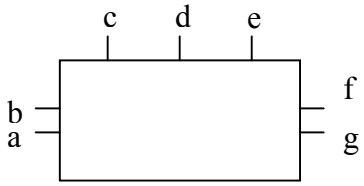


Figure 1 : Représentation du processus P comme une boite noire de portes a, b, c, d, e, f et g

La définition du processus P spécifiera donc son comportement, en définissant les séquences d'actions observables qui peuvent se présenter (être observées) aux sept portes du processus. Les boites noires sont la représentation intuitive traditionnelle pour les processus.

Etant donné que LOTOS a été principalement désigné pour spécifier les protocoles de communication pour les réseaux d'ordinateurs, il comprend des caractéristiques comme le Hiding. Cette caractéristique est mieux introduite en revenant au monde plus abstrait des boites noires, où un processus est représenté comme dans la ***Figure1***.

En considérant la ***Figure 2***, l'interprétation voulue du système décrit est comme suit : le processus Max3 est défini en composant en parallèle deux instances du processus Max2. Chacun des processus composants peut interagir avec son propre environnement, qui consiste en l'autre instance de Max2 et l'environnement extérieur, via trois portes; mais la seule porte de synchronisation entre les deux processus est : mid . Remarquez que cette porte est incluse dans la boite externe qui représente le processus Max3.

Du fait que cette structure est une boite noire, la porte mid n'est pas visible par l'environnement externe : elle a été cachée (hidden).

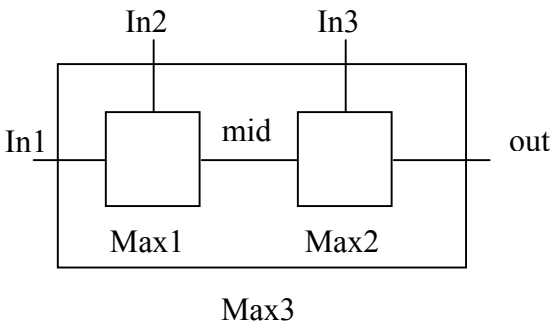


Figure2 : Représentation détaillée du processus Max3

Les deux instances de processus sont ainsi autorisées à interagir indépendamment avec l'environnement sur toutes les portes excepté mid. A cette porte, elles sont obligées de se synchroniser l'une avec l'autre, sans la participation de l'environnement extérieur. Ces interactions : dues au cache, sont devenues des actions internes du système. Cette description informelle est formalisée comme suit :

```
process Max3[in1, in2 , in3, out] :=
        hide mid in
                (Max2[in1, in2, mid]
                 |[mid]|
                 Max2[mid, in3, out]
        where…
endproc ( *max3* )
```

Le fait que le système puisse interagir avec son environnement via des actions (à des portes) in1, in2, in3 et out, est explicitement indiqué dans la première ligne de la spécification. Puisque la porte mid est cachée, par l'opérateur **hide**, elle n'apparaît pas dans la liste. La spécification partielle ci-dessus sera complémentée ultérieurement.

3- BASIC LOTOS :

Basic LOTOS est une version simplifiée du langage employant un alphabet fini d'actions observables. Ceci est ainsi, parce que les actions observables en Basic LOTOS sont uniquement identifiées par le nom de la porte où elles sont offertes, et que les processus LOTOS ne peuvent avoir qu'un nombre fini de portes.

La structure des actions sera enrichie dans LOTOS Complet « *FULL LOTOS* » en permettant l'association des valeurs de données aux noms des portes, et ainsi l'alphabet d'actions observables peut être infini.

Basic LOTOS décrit uniquement la synchronisation de processus, alors que Full LOTOS décrit aussi les valeurs de communications inter-processus.

L'introduction de Basic LOTOS permettra de dégager les trois points suivants :

- Mise en lumière de la signification intuitive des constructeurs (opérateurs) du langage.

- Expression de la sémantique formelle de chacun des opérateurs sans que cette sémantique soit alourdie par les communications inter-processus.
- Les équivalences de comportement sont introduites de manière plus commode.

4- PRESENTATION DES STRUCTURES DE CONTROLE :

Le contrôle des programmes est décrit par des expressions algébriques appelées : ***comportements***. La synchronisation et la communication s'effectuent exclusivement par rendez-vous, sans partage de mémoire.

4.1 ELEMENTS LEXICOGRAPHIQUES & SYNTAXIQUES :

4.1.1 EXPRESSIONS DE COMPORTEMENT :

On appelle expression de comportement (**behaviour expression**), ou plus simplement comportement, un terme syntaxique obtenu par combinaison des opérateurs de comportement.

La structure typique de la définition du processus Basic LOTOS est donnée dans la ***Figure3***, qui complète la définition du processus Max3 commentée précédemment.

Comme une convention, nous utilisons l'italique pour les catégories syntaxiques, c'est à dire, les symboles non-terminaux (ex : expression de comportement), et le gras pour les mots clés LOTOS réservés (ex : process).

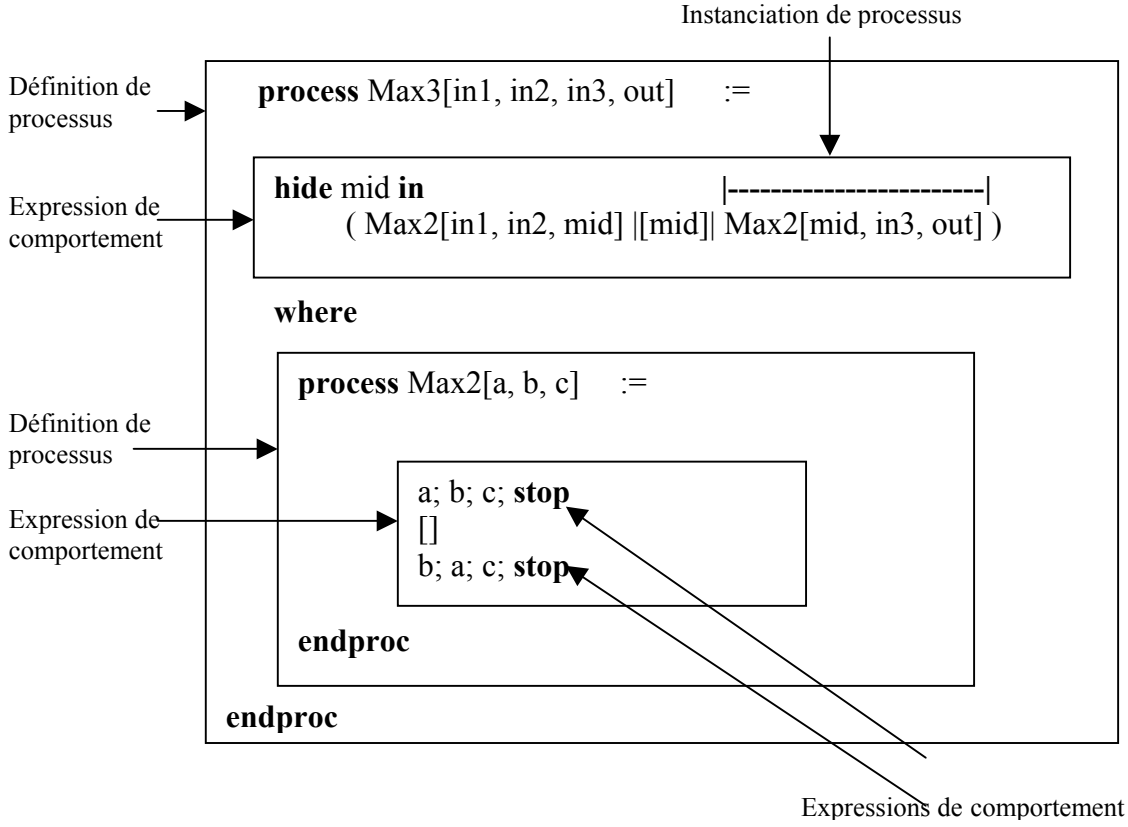


Figure3 : Définition du processus Max3

Un composant essentiel d'une définition de processus est son expression de comportement. Une expression de comportement est construite en appliquant un opérateur (ex : '[]') à d'autres expressions de comportement. Une expression de comportement peut inclure aussi des instanciations des autres processus (ex : Max2), dont les définitions sont fournies dans la clause '**where**' selon l'expression.

4.1.2 EXPRESSIONS DE VALEUR :

On appelle *expression de valeur* (**value expression**), ou plus simplement valeur, un terme algébrique construit à partir de variables et d'opérateurs. Les expressions de valeur apparaissent dans les équations algébriques ; elles sont aussi utilisées dans les expressions de comportement. Elles sont dénotées par le non-terminal V.

LOTOS est fortement typé : chaque valeur ne peut avoir qu'une seule sorte, qu'il est possible de déterminer statiquement.

Les *sortes* et les *opérations* qui sont utilisées dans les comportements LOTOS ne doivent pas être formelles ; elles doivent appartenir à des types complètement instanciés.

Si S est une sorte, on note domain(S) l'ensemble quotient de tous les termes de sorte S par la relation de congruence définie par les équations associées à S. on fait l'hypothèse que le domaine de chaque sorte n'est pas vide.

Si V1 et V2 sont deux valeurs de sorte S, on note V1= V2 soient congrues modulo la relation de congruence définie par les opérations associées à S.

4.1.3 VARIABLES :

Une variable est un nom donné à une valeur. LOTOS est un langage fonctionnel : chaque variable est initialisée dès sa déclaration et sa valeur ne peut pas être modifiée.

4.1.4 PORTES :

En LOTOS, on appelle porte (*gate*) un canal de communication permettant la synchronisation par rendez-vous et l'échange de valeurs entre plusieurs taches qui se déroulent en parallèle. On note Γ l'ensemble de tous les identificateurs de portes, définis par l'utilisateur, qui figurent dans une spécification LOTOS.

Deux portes spéciales sont prédéfinies, qui n'appartiennent pas à Γ :

- La porte invisible, notée « $\imath$ ». cette porte peut apparaître dans les programmes LOTOS, mais uniquement dans le contexte d'un opérateur « ; ».
- La porte de terminaison, notée « δ ». cette porte ne peut jamais être employée explicitement dans un programme LOTOS mais elle est utilisée dans la définition sémantique du langage.

4.1.5 IDENTIFICATEURS :

Chaque classe d'identificateurs est dénotée par un symbole non-terminal défini comme suit :

- G : portes.
- P : processus.

- X : variables.
- T : types.
- S : sortes.
- F : opérations.

On emploie en outre les abréviations suivantes :

- Ĝ : liste non vide de portes G0,…Gn.
- X : liste non vide de variables X0…Xn.

On introduit à présent tous les opérateurs de contrôle du langage LOTOS. (Un même exemple, celui d'un distributeur de boissons, est conservé tout au long de ce cours).

- Opérateur « Stop » :

La structure : **Stop**

Dénote un comportement inactif, qui ne propose aucun rendez-vous avec l'environnement ni aucune transition « i » interne.

- Opérateur « ; » :

L'opérateur « **;** » permet de spécifier le rendez-vous. Si G est une porte et B0 un comportement, la construction suivante :

G ; B0

Dénote le comportement qui propose un rendez-vous sur la porte G et une fois qu'il a eu lieu, exécute B0. la notation « **;** » a une signification séquentielle.

On dit que le comportement B est préfixé par la porte G. les termes événement et interaction seront utilisés comme synonymes de rendez-vous.

- Exemple1 :

Le comportement suivant effectue une interaction MONEY (acquisition de pièces de monnaie) puis une interaction TEA (distribution d'une tasse de thé), après quoi il s'arrête :

MONEY ;
TEA ;
Stop

En fait, cet exemple décrit également le comportement d'un utilisateur qui, après avoir payé, reçoit une tasse de thé.

Cette forme simple de rendez-vous, qui ne comporte pas d'émission ni de réception de valeurs, ne permet que la synchronisation pure. Il existe une construction générale de rendez-vous qui prend en compte l'échange de valeurs :

G O_0, …On ; B0

Où : O_0,…On sont des offres, définies comme suit :

O ≡ !V
| ? X0,…Xn : S.

Une offre de la forme « **!V** » correspond à l'émission sur la porte G de la valeur de l'expression **V**.

Une offre de la forme « **? X0,…Xn : S** » correspond à la réception sur la porte G de n+1 valeurs **v0,…vn** de sorte S, chacune de ces valeurs **vi** est ensuite affectée à la variable **Xi** correspondante.

Le rendez-vous est bloquant aussi bien pour l'émission que pour la réception : L'exécution d'un comportement qui attend un rendez-vous est suspendue et ne reprend qu'après que le rendez-vous a eu lieu.

Le rendez-vous LOTOS est absolument symétrique, aucune distinction n'est faite entre émetteur et récepteur.

Un seul et même rendez-vous peut comporter plusieurs émissions et réceptions qui se déroulent simultanément. De plus, une même porte peut être successivement utilisée dans plusieurs rendez-vous, tantôt avec des émissions, tantôt avec des réceptions.

LOTOS permet de conditionner le rendez-vous par une garde qui est soit une expression booléenne « **[V0]** », soit une équation simple « **[V1=V2]** ». le rendez-vous n'a pas lieu si la condition définie par la garde n'est pas satisfaite.

- Exemple2 :

Le comportement suivant modélise un distributaire qui effectue successivement trois interactions :

- Acquisition d'une somme d'argent (interactions **MONEY**) en dollars et en cents, au moyen d'une garde on interdit le rendez-vous si cette somme est inférieure au prix attendu.
- Distribution d'une tasse de thé (interaction **TEA**).
- Restriction de la monnaie (interaction **CHANGE**).

```
MONEY ? DOLLARDS: NAT ?CENTS /:NAT (TOTAL (DOLLARDS , CENTS) GE COST];
    TEA;
      CHANGE ! CHG_DOLLARDS( DOLLARDS, CENTS ) !CHG_CENTS (DOLLARDS,
         CENTS);
            STOP
```

Les opérations comptables sont décrites à l'aide d'un type abstrait :

```
type CHANGE is NATURALNUMBER, BOOLEAN

   opns COST : -> NAT
        TOTAL : NAT, NAT -> NAT
         CHG_DOLLARDS : NAT, NAT ->NAT
         CHG_CENTS : NAT, NAT -> NAT

  eqns forall DOLLARDS, CENTS : NAT of sort NAT
       COST = 25;  (* prix  d'une  boisson, exprime en cents *)
        TOTAL (DOLLARDS, CENTS) = (100 = DOLLARDS) + CENTS ;
        CHG_DOLLARDS ( DOLLARDS, CENTS ) = ( TOTAL ( DOLLARDS, CENTS)-  COST)
          div 100;
        CHG_CENTS (DOLLARDS, CENTS) = (TOTAL ( DOLLARDS, CENTS) – COST) mod 100;

endtype.
```

On peut faire figurer la porte « **i** » à gauche de l'opérateur « **;** », mais elle ne doit comporter ni offre ni garde. Le préfixage par la porte « **i** » spécifie une évolution interne qui n'est jamais bloquante.

En résumé, la syntaxe générale de l'opération de préfixage est donc :

```
  i ,B0
|  G [O0…On [V0]] ;B0
|  G [O0…On [[V1=V2]] ;B0
```

Les variables éventuellement définies dans les offres O0,…On ne sont visibles que dans V0, V1, V2 et B0.

- Opérateur « [ ] » :
L'opérateur « **[ ]** » permet de spécifier le choix non-déterministe. Si **B1** et **B2** sont deux comportements, la construction suivante :

B1 [ ] B2

Dénote le comportement qui peut exécuter soit B1 ou B2. (la signification intuitive de cet opérateur est identique à celle de la barre carrée « **[ ]** » de DJIKSTRA).

Remarque1 :

Il n'est pas permis d'écrire en LOTOS des comportements de la forme :

(G1 [ ] G2) ;G3 ;STOP

Il s'agit d'une erreur syntaxique car les opérandes de « **[ ]** » doivent être des comportements et non des portes ; de même l'opérande gauche de « **;** » doit être une porte et non un comportement. La manière correcte d'écrire le comportement ci-dessus est :

(G1 ;G3 ;STOP) [ ] (G2 ;G3 ;STOP)

- Exemple3 :
Le comportement suivant modélise un distributeur qui après avoir accepté le paiement (interaction **MONEY**) peut délivrer du thé, du café ou du chocolat chaud (interaction **TEA**, **COFFEE** et **CHOCOLATE**) :

```
 MONEY ;
(
TEA ;
     STOP
 [ ]
 COFFEE;
       STOP
 [ ]
 CHOCOLATE;
         STOP
 ).
```

Le fait que le choix soit non-déterministe ne signifie pas que le distributeur décide arbitrairement de la boisson qu'il fournit. Le distributeur doit respecter les contraintes imposées par l'environnement, c'est-à-dire, les rendez-vous proposés par les autres comportements avec lesquels il communique et se synchronise :

Après avoir payé (interaction **MONEY**), chaque consommateur sélectionne la boisson qu'il désire, par exemple en appuyant sur un bouton. S'il choisit le café, l'interaction **COFFEE** est imposée au distributeur.

Le non-déterminisme intervient effectivement lorsque les contraintes de l'environnement ne déterminent pas une possibilité unique.

Si le consommateur pourrait indiquer qu'il désire soit du café, soit du chocolat, le choix du distributeur (interaction **COFFEE** ou **CHOCOLATE**) serait imprévisible.

Il existe des comportements pour lesquels aucune interprétation déterministe ne peut être trouvée, quel que soit l'environnement.

L'exemple le plus simple est de la forme :

(G ;B1) [ ] (G ;B2)

Dans ce cas, le choix entre l'exécution de **B1** ou de **B2** est purement arbitraire. Cependant, une erreur fréquente consiste à croire que « **;** » est distributif sur « **[ ]** ». l'emploi de cette forme au lieu de : **G ;(B1 [ ] B2])** peut provoquer un blocage.

Donc, le distributeur de boisson présenté dans cet exemple ne doit pas être décrit ainsi :

```
  MONEY
      TEA ;
           STOP
[ ]
MONEY
      COFFEE;
                STOP
[ ]
  MONEY
       CHOCOLAT;
                  STOP
```

Au moment où le consommateur paie (interaction **MONEY**), le distributeur se trouve confronté à un choix non-déterminisme, qu'il résout en sélectionnant arbitrairement une branche, au détriment des autres.

Le choix proposé à l'utilisateur après paiement est restreint à une seule des trois interactions **TEA**, **COFFEE** et **CHOCOLATE**.

- Opérateur « CHOICE » sur les portes :

Soit B0 un comportement qui contient des occurrences d'utilisation d'une porte G. soient G1…Gn des portes et soient B1,…Bn les comportements définis de la manière suivante :

Bi est obtenu à partir de B en remplaçant G par Gi pour exprimer le comportement

B **[ ]** …Bn

LOTOS permet d'utiliser une notation abrégée :

CHOICE G **in** [G1;… Gn] **[ ]** B0.

La porte G sert d'indice à cette itération. Il n'est pas indispensable que les portes G1,...Gn soient deux à deux distinctes.

- Exemple4 :

L'***Exemple3*** peut être décrit de manière plus concise :

```
MONEY
    (
    choice DRINK in [ TEA, COFFEE, CHOCOLATE ] [ ]
        DRINK;
            Stop
     )
```

L'opérateur « **choice** » possède une forme plus générale permettant d'itérer sur plusieurs portes :

choice Ĝ0 **in** [Ĝ0],…Ĝn **in** [Ĝn] **[ ]** B0

Les portes définies dans les listes Ĝ0, …Ĝn ne sont visibles que dans B0.

- Opérateur « | | », « | | | » et « |[…]| » :

Les opérateurs qui ont été présentés jusqu'ici sont strictement séquentiels, LOTOS comprend aussi des opérateurs parallèles, si **B1** et **B2** sont deux comportements et **G0,…Gn** une liste de portes, la construction suivante :

B1 |[G0,…Gn]| B2.

Dénote le comportement qui exécute B1 et B2 en parallèle. La synchronisation et la communication entre les opérandes B1 et B2 s'effectuent uniquement par rendez-vous sur les portes de l'ensemble {G0,…Gn, δ}.

Lorsqu'un des opérandes veut effectuer une transition étiquetée par une porte G de {G0,…Gn,δ}, il doit attendre que l'autre opérande puisse en faire autant. Lorsque le rendez-vous est possible, les deux opérandes effectuent simultanément une même transition synchrone étiquetée G, puis ils reprennent chacun leur exécution.

En revanche si l'un des opérandes veut effectuer une transition étiquetée par une porte G qui n'appartient pas à {G0,…Gn, δ}, il le fait indépendamment de l'autre opérande, de manière asynchrone.

- Exemple5 :

Pour composer en parallèle le distributeur de boissons et un consommateur de thé il faut les synchroniser sur les quatre interactions **MONEY**, **TEA**, **COFFEE** et **CHOCOLATE**. Comme le client n'effectue pas les interactions **COFFEE** et **CHOCOLATE,** le distributeur, qui doit se synchroniser avec lui, ne le peut pas non plus.

```
MONEY
    (
       CHOICE DRINK in [TEA, COFFEE, CHOCOLATE] []
          DRINK;
             STOP
     )
     |[MONEY, TEA, COFFEE, CHOCOLATE]|
          MONEY;
             TEA;
                STOP.
```

Outre l'opérateur de synchronisation général « **|[G0…Gn]|** » qui traduit la synchronisation sur les portes **G0,…Gn** et « **δ** », LOTOS possède deux autres opérateurs de composition parallèle :

- Le premier opérateur exprime la synchronisation sur aucune porte, sauf « **δ** » (interleaving). Sa syntaxe est :

B1 ||| B2

Les deux comportements **B1** et **B2** sont exécutés de manière totalement indépendante (terminaison sur « **δ** » exceptée) : ils ne se synchronisent ni ne communiquent l'un avec l'autre. En revanche ils sont capables d'interagir avec leur environnement commun : « **B1 |||B2** » peut participer à un rendez-vous si et seulement si B1 ou B2 le peut.

- Le second opérateur exprime la synchronisation sur toutes les portes, y compris « **δ** » (full synchronisation). Sa syntaxe est :

B1 || B2.

Les deux comportements B1 et B2 sont exécutés en parallèle de manière entièrement synchrone : ils doivent se synchroniser sur toutes leurs interactions. Ils peuvent interagir avec leur environnement commun : « B1 || B2 » peut participer à un rendez-vous si et seulement si B1 et B2 le peuvent.

Lorsque les portes sont accompagnées d'offres, c'est à dire quand la composition parallèle a la forme suivante :

(G1 O0…On[V1=V2] ;B1) op (G2 O0…Op[V1=V2];B2).

Le rendez-vous n'a pas lieu que si les conditions suivantes sont vérifiées:

- Les portes G1 et G2 sont égales et leur synchronisation est permise par l'opérateur **op**.
- Le nombre d'offres de part et d'autre est le même (**n=p**).
- Les sortes des offres Oi et Oi sont deux à deux identiques.
- Les deux gardes sont vérifiées.

Lorsque il y a confrontation entre une émission (offre ' **!**') et une réception

(offre ' **?**') la valeur émise est affectée à la variable de réception (value passing).

- Exemple6 :

C'est le cas lorsqu'on compose un distributeur qui rend la monnaie et un buveur de thé qui fournit $1.00 à la machine et reprend sa monnaie.

```
MONEY ? DOLLARDS : NAT ? CENTS : NAT [ TOTAL (DOLLARDS, CENTS ) ge COST ];
    (
   choice DRINK in [ TEA, COFFEE, CHOCOLATE ] [ ]
      DRINK;
       CHANGE!CHG_DOLLARDS(DOLLARDS, CENTS)!CHG_CENTS(DOLLARDS, CENTS);
           stop
    )
|[MONEY, TEA, COFFEE, CHOCALATE, CHANGE]|
MONEY !1 !0;
     TEA;
```

```
CHANGE ? DOLLARDS : NAT ? CENTS : NAT ;
    stop.
```

LOTOS autorise également les confrontations entre deux émissions (‘ **!**’ et ‘ **!**’) ou deux réceptions (‘ **?**’ et ‘ **?**’). dans le premier cas (value maching), le rendez-vous n’a pas lieu que si les deux valeurs émises sont égales. Dans le second cas (value generation), les deux variables de réceptions reçoivent une valeur identique, choisie de manière non-déterministe.

- Opérateur « hide » :

LOTOS possède un opérateur qui permet de cacher certaines portes d’un comportement. Si **G0,…Gn** sont des portes et **B0** un comportement, la construction suivante :

hide G0,…Gn in B0.

Dénote le comportement **B0** dont les portes **G0,…Gn** sont renommées en « **i** ». les portes G0,…Gn ne sont visibles que dans B0. elles deviennent inaptes à la synchronisation pour l’environnement de B0 avec lequel elles n’interfèrent plus. Vu de l’extérieur ces interactions sont invisibles (puisqu ’étiquetées « **i** ») et ont lieu spontanément sans aucune participation de l’environnement de B0 : le rendez-vous sur une porte cachée n’est jamais bloquant.

- Exemple7 :

Bien souvent, un comportement est décrit comme la mise en parallèle de plusieurs sous comportements qui se synchronisent sur un ensemble de portes qu’il convient de dissimuler vis-à-vis de l’environnement. Dans cet esprit, on peut décomposer le distributeur décrit dans l’exemple précédent en deux sous-systèmes :

- Le premier reçoit une somme d’argent, s’assure que le montant est suffisant, calcule la monnaie à rendre et envoie une autorisation à l’autre sous-système via une porte **GRANT**.

- Le second, lorsque l’ autorisation est accordée, délivre une boisson (thé, café, chocolat, au choix du client) et rend la monnaie (la somme qu’il faut restituer lui a été communiquée via la porte GRANT).

On cache la porte GRANT au moyen de l’opérateur « **hide** » car il s’agit d’un détail d’implémentation qui n’est pas pertinent pour un observateur extérieur.

Le distributeur est composé en parallèle avec un consommateur de thé qui fournit $1.00 pour payer.

Noter que l’opérateur « || » impose la synchronisation sur les portes **MONEY**, **TEA**, **COFFEE**, **CHOCOLATE** et **CHANGE** mais pas **GRANT**, qui est cachée.

```
hide GRANT in
    (
    MONEY ?DOLLARDS :NAT? CENTS: NAT [TOTAL (DOLLARDS, CENTS) ge COST];
       GRANT !CHG_DOLLARDS (DOLLARDS, CENTS) !CHG_CENTS (DOLLARDS, CENTS);
          stop
   ||[GRANT]||
    GRANT ?DOLLARDS :NAT ? CENTS :NAT;
          (
          choice DRINK IN [TEA, COFFEE, CHOCOLATE] [ ]
               DRINK;
                    CHANGE !DOLLARDS !CENTS;
                         stop
```

```
            )
      )
||
    MONEY !1 !0;
          TEA;
               CHANGE ?DOLLARDS :NAT ?CENTS /NAT;
                    stop.
```

Le rendez-vous “un-aire” est correct et non bloquant : par exemple :

hide G **in** G ; **stop**.

est équivalent, vu de l’extérieur, à :

i ;**stop**.

- Opérateur « exit » :

L’opérateur « **stop** » permet de spécifier explicitement l’arrêt d’un comportement . mais « stop » peut aussi apparaître de manière implicite, lorsqu’un comportement se bloque. Pour distinguer ces deux formes de terminaison, normale et anormale, on introduit un nouvel opérateur. La construction suivante :

exit

Dénote un comportement qui se termine normalement. La terminaison avec succès s’exprime par le franchissement d’une transition « **δ** ». De fait « **exit** » est équivalent à un rendez-vous sur la porte « **δ** » :

δ ; stop.

Un comportement peut, lorsqu’ il se termine par « exit », transmettre des résultats. Cette possibilité correspond à l’ajout d’une liste d’offres au rendez-vous sur la porte « **δ** », mais la syntaxe est différente :

exit (R0,…Rn)

Où les résultats R0,…Rn sont définis comme suit :

R ≡ V
| **any** S

Un résultat dénote donc, soit une valeur V déterminée, soit une valeur choisie de façon non-déterministe dans le domaine d’une sorte S.

La définition de l’opérateur de composition parallèle et de la synchronisation sur la porte « **δ** » implique que la composition parallèle de n comportements B1,…Bn ne se termine par « **exit** » que si B1,…Bn se terminent aussi par « **exit** », de manière synchrone(join), en proposant des offres compatibles.

Certaines règles interdisent les constructions susceptibles de conduire à des blocages sur la porte « **δ** ». Savoir si un comportement se termine ou non étant un problème indécidable dans le cas générale.

LOTOS se contente d’imposer des contraintes « de bon sens » , qu’il est possible de vérifier statiquement et qui protègent l’utilisateur contre certaines erreurs. Chaque comportement

possède une fonctionnalité qui spécifie si le comportement se termine et précise, dans ce cas, les sortes des valeurs qu' il renvoie par l'opérateur « **exit** ». il faut respecter certaines règles quand on compose les comportements ; c'est ainsi qu' il est interdit d'écrire :

exit (V1,…Vn) | | | **exit** (V'1,… V'n') .

Lorsque n et n' ne sont pas égaux.

- Opérateur « >> » :

L'opérateur de préfixage « **;** » est asymétrique : son opérande gauche est une porte (éventuellement accompagnée d'offres) alors que son opérande droit est un comportement. LOTOS possède un autre opérateur de composition séquentielle dont les deux opérandes sont des comportements. Si **B1** et **B2** sont deux comportements, la construction suivante :

B1 >> B2.

Dénote le comportement qui exécute séquentiellement **B1** puis **B2**.

Intuitivement, la composition séquentielle est modélisée en LOTOS comme un cas particulier de composition parallèle. Les comportements B1 et B2 sont exécutés en parallèle mais B2 ne peut pas commencer avant que B1 ne se soit terminé. L'attente de B2 s'obtient par un rendez-vous sur la porte « **δ** », rendez-vous qui devient possible dès que B1 exécute « **exit** ». si B1 boucle indéfiniment ou il se bloque sans atteindre « **exit** » B2 ne sera jamais exécuté.

L'opérateur « **>>** » est souvent appelé **enabling opérator** ou **enable** puisque la terminaison avec succès de B1 autorise l'exécution de B2.

L'opérateur « >> » possède une forme plus générale permettant au processus qui commence de récupérer les résultats renvoyés par le processus qui se termine. Si B1 et B2 sont deux comportements, la construction suivante :

B1 >> **accept** X0 : S0,…Xn : Sn **in** B2.

Dénote le comportement formé par la composition séquentielle de B1 de B2 : lorsque B1 exécute une instruction « **exit** », les résultats qu'il revoie sont affectés aux variables X0,…Xn respectivement. Les variables définies dans les listes X0,…Xn ne sont visibles que dans B2. chaque variable de la liste Xi est de sorte Si (les contraintes portant sur la fonctionnalité des comportements imposent que les résultats renvoyés par B1 correspondent, par leur nombre et par leurs sortes, aux variables déclarées après le mot « **accept** »).

- Exemple8 :

On peut réécrire le distributeur en factorisant la restitution de monnaie (interaction **CHANGE**) grâce à l'opérateur de composition séquentielle :

```
MONEY ?DOLLARDS :NAT ? CENTS : NAT;
   (
  l e t  OK : BOOL =(TOTAL (DOLLARDS, CENTS ) ge COST ) in
      (
     [ not (OK) ] ->
           exit (DOLLARDS, CENTS )
     [ ]
     [OK ]  ->
         (
        choice DRINK in [TEA,  COFFEE, CHOCOLATE ]  [ ]
           DRINK;
```

```
                    Exit (CHG_DOLLARDS (DOLLARDS, CENTS) , CHG_CENTS  (DOLLARDS, CENTS))
            )
        )
    )
>> accept DOLLARDS, CENTS : NAT in
    CHANGE ! DOLLARDS ! CENTS ;
        stop.
```

L'opérateur « >> », avec ou sans « **accept** », est associatif.

- Opérateur « [> » :

LOTOS dispose d'un opérateur permettant de spécifier l'interruption d'un comportement par un autre :

Si **B1** et **B2** sont deux comportements , la construction suivante :

B1 [> B2.

Dénote le comportement qui exécute **B1** mais qui peut abandonner à tout instant l'exécution de B1 pour commencer celle de **B2**.

Intuitivement, il s'agit d'un mécanisme d'interruption avec terminaison : B1 joue le rôle d'un traitement normal et B2 celui d'un traitement d'exception. Initialement, B1 est exécuté seul mais, tant qu' il n'a pas effectuée de transition « **δ** », son exécution peut être interrompue au profit de celle de B2.

Si B1 se bloque avant d'atteindre une instruction « exit » alors B2 est inévitablement exécuté. Dans le cas contraire B2 peut très bien ne jamais être exécuté.

L'opérateur « **[>** » est souvent appelé **disbling opérator** ou **disable** puisque la terminaison avec succès de B1 interdit l'exécution de B2.

- Exemple9 :

L'opérateur « **[>** » peut etre utiliser pour décrire le comportement d'un consommateur de thé lorsque le distributeur comporte un bouton d'annulation (interaction **CANCEL**).le consommateur a le même comportement mais, à chaque instant, il peut s'interrompre et presser sur le bouton d'annulation pour tenter de se faire rembourser par le distributeur (qui n'est pas obligé d'accepter).

```
MONEY !1 !0 ;
      TEA ;
        CHANGE ? DOLLARDS : NAT ? CENTS : NAT ;
            exit
       [>
        CANCEL;
            CHANGE ? DOLLADS : NAT ? CENTS : NAT;
              exit.
```

L'opérateur « **[>** » est associatif.

- Processus et Instanciation :

Dans les langages algorithmiques, il est possible de donner un nom à un bloc d'instructions, en définissant une procédure qui peut éventuellement être paramétrée. De manière analogue, LOTOS permet de nommer un comportement au moyen d'une définition de processus (**process**). Un processus est un objet qui dénote un comportement : il peut etre paramétré par une liste de portes formelles et / ou une liste de variables formelles.

Remarque2 :

Ce mécanisme est limité au premier ordre : il n'existe pas d'objet qui dénote un processus. On n'a donc pas de variables ou de paramètres formels « de type processus ».

Remarque3 :

En LOTOS, le mot « **process** » n'a pas le même sens que dans d'autres langages ; il ne dénote pas forcément une activité concurrente. La création des taches parallèles est dévolue à l'opérateur de composition parallèle et non à l'instanciation.

La construction suivante :

process P [[G0,……Gm]] [(X0: S0,……Xn : Sn)] : *func* :=
B
[where block0……..blockp]
endproc

définit un processus P (éventuellement paramétré par les portes formelles **G1,…Gm** et les listes de variables formelles **X1…Xn** de sortes respectives **S1,…Sn**) dont le corps est le comportement B. le non-terminal ***func*** spécifie la fonctionnalité de B afin de permettre une vérification statique des contraintes de fonctionnalité ; sa définition syntaxique est :

func ≡ **noexit**
| **exit** [(S0,…Sn)].

Le premier cas indique que **B** ne s'achève jamais par « **exit** » (ce qui signifie que B se bloque ou boucle indéfiniment) ; le second cas exprime que **B** se termine en renvoyant des résultats de sortes respectives **S0…Sn**, chaque non-terminal ***block*** dénote une définition de processus ou de type ; dans les deux cas il s'agit d'une définition locale dont la visibilité est limitée à la définition du processus **P**.

L'instanciation d'un processus s'effectue en substituant des paramètres effectifs aux paramètres formels. Si **P** est un processus, **G0…Gm** des portes et **V0,…Vn** des expressions de valeur, la construction suivante :

P([G0…Gm]) [(V0,…Vn)].

Dénote le corps de P dans lequel les portes formelles sont renommées par **G0,…Gm** et les variables formelles sont instanciées avec les valeurs **V0…Vn**.

L'emploie de la récursion est autorisé ; en LOTOS, la récursion est d'ailleurs le seul moyen pour créer des comportements cycliques.

- Exemple10 :

L'exemple suivant décrit un distributeur relativement élaboré. Le consommateur peut effectuer autant de paiements qu'il le souhaite (plusieurs interactions **MONEY** successives jusqu'à ce que le prix d'une boisson **COST** soit atteint ; il peut même dépasser ce montant.

Dés que la somme versée est suffisante, le consommateur peut choisir une boisson et récupérer sa monnaie, l'utilisateur a aussi la possibilité de récupérer l'argent versé en appuyant sur le bouton **CANCEL**.

Le fonctionnement du distributeur est cyclique : il retourne dans son état initial et peut donc servir successivement plusieurs clients.

```
SELL [MONEY, TEA, COFFEE, CHOCOLATE, CANCEL, CHANGE ] (0,0)
where
process SELL [MONEY,  TEA, COFFEE, CHOCOLATE, CANCEL, CHANGE]  (D, C: NAT) : noexit :=
    MONEY ? DOLLARDS :NAT ? CENTS : NAT;
        (
        let DO : NAT =(D + DOLLARDS ), CO : NAT = ( C + CENTS ) in
            (
            [ TOTAL ( DO, CO)  ge  COST ] ->
              (
              choice DRINK  in [TEA, COFFEE, CHOCOLATE ]  [ ]
                  DRINK;
                      CHANGE ! CHG_DOLLARDS ( DO, CO)  !CHG_CENTS (DO,  CO);
                           SELL [ MONEY, TEA, COFFEE, CHOCOLATE, CANCEL, CHANGE ] (0, 0)
              )
             [ ]
              CANCEL;
                CHANGE ! DO ! CO;
                      SELL [MONEY, TEA, COFFEE, CHOCOLATE, CANCEL,  CHANGE ]  (0, 0)
             [ ]
              SELL [ MONEY, TEA, COFFEE, CHOCOLATE, CANCEL, CHANGE]  (DO, CO)
            )
          )
    endproc.
```

Nous proposons dans ce qui suit, de compléter la partie contrôle par les structures de données, permettant ainsi l'association des valeurs de données aux noms des portes et par voie de conséquence, la description de la valeur de communication inter-processus.

5- TYPES DE DONNEES :

Les représentations des valeurs, les expressions de valeur et les structures de données dans LOTOS sont dérivées du langage de spécification pour les types de données abstraits (ADT) **ACT ONE**. Le choix des types de données abstraits pour LOTOS, par opposition aux types de données concrets, est compatible avec le besoin de l'abstraction des détails de l'implémentation qui a été aussi un principe de guide dans la conception de l'autre composant du langage (les définitions de processus).

Un type de données concret indique une description de la manière dont les valeurs de données sont représentées dans la mémoire, et comment certaines procédures associées opèrent sur eux. En d'autres termes, le type de données est défini en donnant formellement son implémentation. Par exemple, une queue (file) en Pascal, peut etre définie comme une liste d'enregistrements et une paire de procédures qui la manipulent pour réaliser les opérations 'Add' et 'Remove'.

Un type de données abstrait peut etre vu comme la spécification formelle d'une classe de types données concrets. Il n'indique pas comment les valeurs de données sont actuellement représentées et manipulées dans la mémoire, mais définit uniquement les propriétés essentielles de données et les opérations que n'importe quelle implémentation (type de données concret) est obligée de satisfaire. Finalement, une définition ADT identifie un objet mathématique, à savoir une algèbre; formé par des ensembles de valeurs de données, appelés : Porteurs de données, et un ensemble d'opérations associées.

ACT ONE est une méthode de spécification algébrique pour écrire des spécifications ADT non-paramétrées aussi bien que des spécifications ADT paramétrées. ACT ONE, et ainsi LOTOS, inclut les caractéristiques suivantes pour la production de spécifications structurées :

1- l'utilisation d'une librairie de types de données prédéfinies ;

2- extension et combinaison des spécifications déjà actuelles (existantes) ;

3- paramétrisation des spécifications, et actualisation des spécifications paramétrées,

4- renommage de spécifications.

La forme la plus fondamentale de spécification de type de données dans LOTOS consiste en une **signature** et, peut être, une liste d'équations.

- ***Signature :***

La première étape dans la spécification d'un type de données consiste en la définition des noms des porteurs de données et des opérations. Les noms des porteurs de données sont soumis comme étant des sortes. La déclaration de chaque opération comprendra son domaine, qui consiste exactement en une sorte. Les sortes et les opérations d'un type de données sont soumises comme étant la signature de ce type de données.

Ci-dessous, nous listons une définition de type des nombres naturels, qui consiste uniquement en une signature. La définition est appelée :'Nat_numbers', par conséquent, elle peut etre soumise par d'autres définitions, et combinée avec eux. La signature de 'Nat_numbers' consiste en l'unique sorte 'nat', et les opérations '0' et 'succ'. L'opération 'succ' peut etre appliquée à un élément unique de la sorte 'nat', et produit aussi un élément de 'nat' comme un résultat, comme indiqué par la notation 'nat->nat'. l'opération '0' est une opération qui n'a pas d'arguments, pourtant, elle produit un élément de 'nat', comme indiqué par la notation '->nat' :

```
type Nat_numbers is
     Sorts  nat
     Opns   0    :    ->nat
            succ :   nat->nat
endtype
```

Nous exprimons le fait qu'une opération ait **n** arguments en disant qu'elle est une opération n-aire. Ainsi, 'succ' est une opération un-aire, alors que '0' est une opération nul-aire. Les opérations nul-aires sont appelées : **Constantes**.

Un exemple supplémentaire d'une définition de type de données complète qui consiste uniquement en une signature est la définition d'un ensemble de caractères {a1, …, an}, où chaque caractère est défini comme une constante :

```
type  Character is
      sorts   char
      opns    a1, …, an, e: ->char
endtype
```

Noter qu'il y a un symbole spécial 'e', qui est utilisé pour représenter une erreur qui est de sorte 'char'.

La signature d'un type donne toutes les informations requises pour construire syntaxiquement des termes corrects, ou des expressions de valeurs, qui représentent des

valeurs de données de (certaines sortes de) ce type. Un terme est le résultat de l'application d'une opération n-aire à n termes. En particulier, une constante est clairement un terme. Plus précisément, si une signature contient la déclaration constante : **c : -> S** où S est le nom d'une sorte, alors nous disons que c est une constante (ou un terme) de sorte S. De la même façon, si une opération est déclarée comme :

op : S1, …, Sn -> S

Alors, op (t1, …, tn) est un terme de sorte S, ou un S-terme, tout court, où t, est un Si-terme, pour i=1,…,n.

Par exemple, étant donnée la signature de type 'Nat_numbers' ci-dessus, nous pouvons construire les termes suivants, tous ceux de la sorte nat :

0, succ(0), succ(succ(0)), …

qui sont destinés à dénoter, respectivement, les éléments 0, 1, 2, …de l'algèbre des nombres naturels.

- ***Equations :***

Supposer maintenant que nous voulons définir une opération 'plus', qui combine deux nat-termes en un nouveau nat-terme :

_+_ : nat, nat -> nat

Les deux symboles '_' marquent la position des opérandes avec respect à l'opérateur, qui est ainsi défini comme un opérateur infixe.

Nous avons maintenant la possibilité d'écrire de nouveaux nat-termes, tel que '0+succ(0)'.

Pour interpréter ces nat-termes correctement, nous avons besoin d'un nouveau constructeur pour exprimer les propriétés des opérations. Ce constructeur est l'équation.

Le but d'une équation est d'affirmer que deux termes syntaxiquement différents dénotent la même valeur. Par exemple, nous voulons exprimer le fait que les termes 'succ(0)' et 'succ(0)+0' dénotent la même valeur, ou, plus généralement, que pour n'importe quel nat-terme x, les termes 'x' et 'x+0' dénotent la même valeur. Une définition correcte des propriétés de l'opérateur '+' est :

```
eqns
    forall  x, y :nat
    ofsort  nat
      x + 0          =x ;
      x + succ(y) = succ(x +y) ;
```

Où les équations identifient les nat-termes (de sorte nat), et sont valides chaque fois que les variables x et y sont remplacées par n'importe quelle paire de nat-termes (**forall** x ,y ; nat).

La première équation exprime le comportement de l'opérateur 'plus' quand il est combiné avec la constante '0'.

L'addition avec un nombre non-zéro est couverte par la seconde équation (noter que le terme 'succ(x)' dénote toujours un nombre non-zéro). Par induction sur la structure des termes, et par l'utilisation de ces équations, il peut etre facilement prouvé que n'importe quel terme contenant une ou plus d'opérations 'plus' est égale à un terme contenant uniquement '0' et 'succ'.

Ceci signifie qu'en introduisant l'opérateur 'plus', nous n'avons pas introduit les termes qui dénotent des valeurs 'new' qui ne pouvaient pas etre exprimées avant. Dans ce cas, nous disons que les équations de '+' sont complètes par la définition de 'Nat_numbers'.

La spécification des nombres naturels étendue avec l'opération 'plus' est :

```
type  Extended_nat_numbers  is
      sorts   nat
      opns   0        : -> nat
              Succ    : nat-> nat
              _+_     :  nat, nat -> nat
       eqns
            forall  x ,y : nat
            ofsort  nat
              x + 0          = x ;
              x + succ(y)  =  succ(x + y) ;
endtype
```

- ***Extensions et combinaisons des spécifications de type :***

Pour spécifier les types de données avec un large nombre d'opérations, nous avons besoin de constructeurs de langage pour combiner les spécifications déjà actuelles, et /ou pour les étendre en ajoutant des sortes, des opérations et des équations supplémentaires. Cette manière de spécification volumineuse peut etre donnée en une seule étape, et un même type de données simple peut etre utilisé comme une base pour plusieurs définitions plus complexes.

Comme un exemple d'enrichissement d'un type, nous redéfinissons le type 'Extended_nat_numbers' sur la base du type 'Natural_numbers' (toutes les définitions sont données précédemment).

```
type  Extended_nat_numbrs is Nat_numbers
    opns   _+_:   nat, nat-> nat
    eqns
        forall x, y:  nat
        ofsort  nat
            x + 0          = x ;
            x + succ(y)   = succ(x + y) ;
endtype
```

Dans 'Extended_nat_numbers' nous avons importé la définition entière de 'Nat_numbers' en faisant allusion à lui dans le titre, et nous l'avons enrichi avec une opération et deux équations. En général, nous pouvons combiner plusieurs définitions de type, et ensuite ajouter de nouveaux éléments spécifiques.

```
type  T is  T1, …, Tn
      sorts  …
      opns  …
      eqns   …
endtype
```

- ***Types paramétrés :***

Les spécifications de types de données paramétrées peuvent etre considérées comme des spécifications partielles où uniquement certaines caractéristiques générales du type sont décrites, et les trous sont remplis ultérieurement avec des détails supplémentaires. Une queue, par exemple, peut être décrite comme un type paramétré, qui peut ultérieurement etre actualisé à devenir une queue d'entiers ou une queue de caractères.

En l'absence des caractéristiques de paramétrisation, nous ne pouvions pas définir la queue des nombres naturels et la queue des caractères comme des enrichissements respectifs du type 'Nat_numbers' et 'Characters' :

```
type Nat_numbers_queue is Nat_numbers
    sorts queue
    opns  create:    ->queue
          add :      nat, queue->queue
          first :    queue->nat
    eqns  forall x, y : nat, z : queue
          ofsort nat
               first (create) = 0;
               first (add(x, create)) = x;
               first (add(x, add(y, z)) =
               first (add(y, z));
endtype

type Character_queue is Characters
    sorts queue
    opns  create:    ->queue
          add :      char, queue->queue
          first :    queue->char
    eqns  forall x, y : char, z : queue
          ofsort char
               first (create) = e;
               first (add(x, create)) = x;
               first (add(x, add(y, z)) =
               first (add(y, z));
endtype
```

Dans ces nouveaux types, l'enrichissement consiste en une nouvelle sorte 'queue'; et en deux nouvelles opérations 'first' et 'add'. 'first' produit le premier élément à une fin de la queue, et 'add' ajoute un élément à son autre fin. Les constantes '0' et 'e' étaient déjà introduites respectivement dans les définitions de type 'Nat_numbers' et 'Characters'. Elles sont utilisées pour indiquer une erreur quand l'opération 'first' est appliquée à une queue vide.

- ***Renommage de type :***

Le renommage des spécifications de types de données est utile au cours du développement d'une spécification dans le cas où un type de données déjà défini est nécessaire dans un environnement spécifique, mais sans aucun changement dans les sémantiques intentionnelles. Par conséquent, le renommage peut être fait explicitement en réécrivant la définition du type de données avec de nouvelles sortes et opérations. Les changements dans la signature, indiquent des changements dans la déclaration des variables et dans les équations. Particulièrement, pour les définitions longues, ceci peut etre une tache encombrante.

L'opération de renommage évite ce désavantage. Laisser nous assumer que la définition du type de données 'queue' vue précédemment, est utilisée dans l'environnement du service de transport de l'OSI, qui s'occupe des canaux et des objets à transférer. Ensuite, la définition 'queue' peut être renommée avec commodité comme suit :

```
type Connection is
     Queue renamedby
          sortnames   channel  for  queue
                      object   for  element
          opnnames    send     for  add
                      receive  for  first
endtype
```

Chapitre 3 : Vérification Formelle des Systèmes Réactifs

Ce chapitre est une présentation de la vérification formelle des systèmes de transitions. Il introduit les deux approches de vérification utilisées, l'approche comportementale et l'approche test. Les relations entre ces deux approches sont précisées.

1- INTRODUCTION :

Les techniques de spécification formelle de façon générale et les algèbres de processus plus particulièrement, ont considérablement contribué par leurs méthodologies de conception et de vérification à la maîtrise de la complexité des systèmes de communication.

L'utilisation des techniques de description formelle lors de la conception d'un système de communication, conduit à la manipulation, pour un même système, de deux objets : ***la spécification*** et ***l'implémentation***.

En effet, il est difficile de spécifier en une seule étape et dans tous les détails, tout ce que doit réaliser le système. Il est par contre plus naturel de procéder par *raffinements successifs*, en commençant par le plus abstrait ou le moins détaillé : la spécification initiale, pour aboutir au plus détaillé ou le moins abstrait : l'implémentation ; en passant par des spécifications intermédiaires.

Autrement dit, nous pouvons décrire les systèmes à différents niveaux d'abstraction; par exemple, il est possible de décrire comment ils sont structurés internellement en termes de sous composants prédéfinis, ou comment ils se comportent du point de vue d'un utilisateur ou d'un observateur externe.

LOTOS est un langage de spécification qui permet la spécification des systèmes à différents niveaux descriptifs. En LOTOS, les mots '***Spécification***' et '***Implémentation***' ont une signification relative (non absolue). Soient deux spécifications Lotos (syntaxiquement homogènes), *S1* et *S2*, nous dirons que *S2* est une implémentation de la spécification *S1*, quand informellement, *S2* donne une description plus structurée et plus détaillée du système spécifié dans *S1*.

Le processus de conception (***Figure 1***), mené dans le cadre d'une technique de description formelle, peut aboutir suite à des *raffinements successifs* à une spécification d'un niveau

détaillé que l'on considère comme l'implémentation de la spécification à l'origine de ce processus. La spécification est ainsi vue comme une abstraction de sa réalisation.

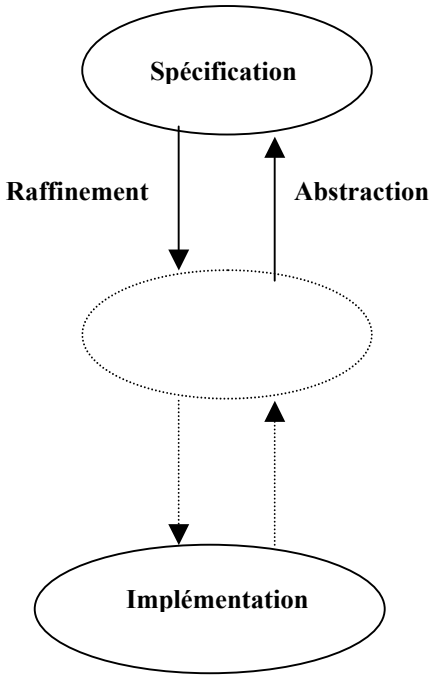


Figure1 : Conception hiérarchique

Les relations entre les différentes descriptions LOTOS d'un système donné et, en particulier, entre les spécifications et les implémentations, peuvent être utilisées en utilisant une notion d'***équivalence.*** Les théories des équivalences s'avèrent très utiles. En fait, elles permettent non seulement de prouver qu'une implémentation est correcte avec respect à une spécification donnée, mais aussi de remplacer les sous-systèmes complexes avec d'autres systèmes équivalents plus simples, à l'intérieur d'un grand système, simplifiant ainsi l'analyse de ce dernier.

Cependant, la notion d'équivalence représente le problème majeur de la sémantique des systèmes concurrents. Deux systèmes donnés, décrits dans un formalisme particulier tel que LOTOS, peuvent-ils ou non être considérés comme deux écritures différentes d'un même système ? Pour répondre à cette question, on doit satisfaire les deux préoccupations suivantes :

- Quels sont les critères à prendre en considération pour affirmer que deux systèmes sont équivalents ?.
- Comment définir formellement cette relation d'équivalence entre les systèmes ?.

La première de ces deux questions est la question fondamentale de la sémantique des systèmes de processus. Car en effet, deux systèmes sont équivalents s'ils ont la même représentation sémantique, c'est-à-dire, si du « *texte* » qui les décrit (*la syntaxe*) on peut extraire la même information, estimée pertinente, sur leur « *comportement* » (*la sémantique*). La définition d'une relation d'équivalence présuppose donc la définition de la sémantique. On peut alors définir la sémantique d'un système comme sa *classe d'équivalence*.

Nous présentons dans cette partie, les principes généraux de la vérification formelle. Par *vérification formelle*, nous entendons toute technique permettant de confronter un système

(sa description opérationnelle) à ses spécifications (aux propriétés que l'on attend de lui). Voir (***Figure 2***). Ce type d'approches nécessite trois ingrédients :

- Une description opérationnelle du système, son graphe de comportement.
- Un langage de spécification permettant d'exprimer les propriétés du système que l'on souhaite vérifier.
- Une procédure de décision qui permet de contrôler la conformité entre la description opérationnelle du système et sa spécification, c'est-à-dire, une procédure qui permet de vérifier que le système satisfait effectivement les propriétés que l'on attend de lui.

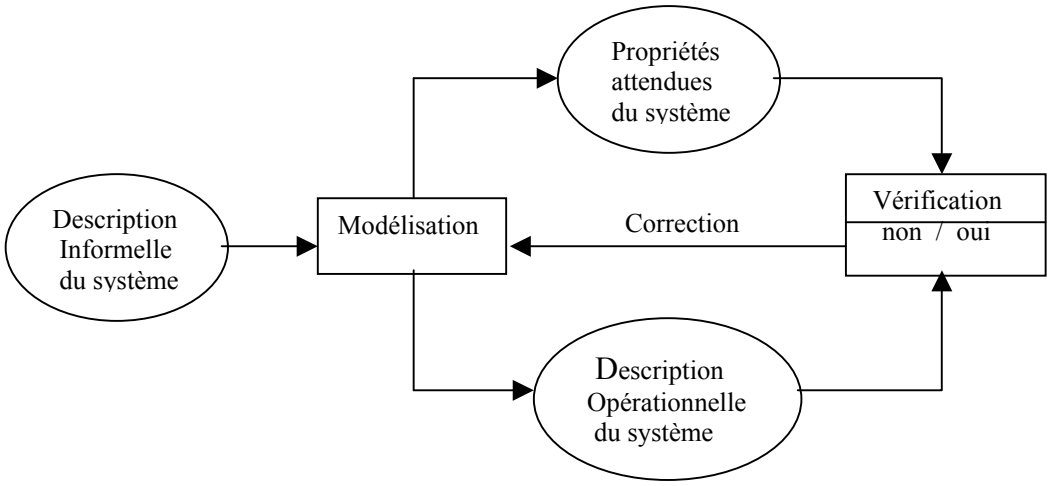


Figure 2 : Vérification

De toute évidence, les deux derniers ingrédients sont fortement couplés : la procédure de décision étant liée au langage de spécification retenu.

En ce qui concerne la définition formelle des équivalences des systèmes concurrents, la syntaxe dans laquelle est décrit un système, joue un rôle capital.

A chaque langage de spécification sont associées des définitions formelles d'équivalence. Cependant, comme nous considérons que les systèmes de processus dont la sémantique est décrite par des systèmes de transitions étiquetées, notre étude peut être ramenée à l'étude de ces relations d'équivalence sur les systèmes de transitions étiquetées.

Différentes relations d'équivalence entre les systèmes de transitions étiquetées qui se distinguent les unes par rapport aux autres par les propriétés jugées pertinentes (on peut dire que l'information pertinente du comportement d'un système est précisément celle qui est commune à tous les systèmes équivalents), peuvent être considérées, chaque relation permet de capturer des aspects particuliers du système à vérifier.

Deux grandes approches sont classiquement distinguées :

L'Approche « Comportementale » et l'Approche « Logique » ou Assertionnelle.

Ce texte est une brève introduction aux principes généraux de ces deux approches :

2- APPROCHE COMPORTEMENTALE :

Pour cette approche, les propriétés attendues du système sont exprimées sous forme de comportement. Dans l'approche Comportementale, la description opérationnelle du système (son graphe de comportement) et sa spécification sont exprimées toutes deux par les

comportements. La procédure de décision revient alors à décider de l'équivalence entre deux graphes.

L'approche Comportementale comprend deux classes de sémantiques sous lesquelles de nombreuses relations d'équivalence ont été proposées pour la comparaison et l'analyse de systèmes concurrents depuis l'équivalence langage (de traces) à l'équivalence Observationnelle en passant par les modèles de Refus et les équivalences de Test. Voir [Gla89] pour un panorama des relations d'équivalences existantes. Ces deux classes de sémantiques sont :

2.1 SEMANTIQUE LINEAIRE :

2.1.1 Equivalence de trace (langage) :

La caractéristique principale de l'*équivalence de trace* c'est qu'elle concerne les couples (*état*, *système de transitions*). En effet, dans la plupart des cas, on peut supposer que tout système a un état initial unique, ce qui nous ramène à supposer que le comportement de ce système est représenté par l'état initial du système de transitions étiquetées qui lui est associé. La comparaison de deux systèmes revient à comparer leurs diverses évolutions à partir de leurs états initiaux. L'évolution de chaque système est représentée par l'ensemble de tous les chemins issus de son état initial, chaque chemin est appelé : ***trace***. Deux systèmes sont liés par cette relation s'ils ont le même ensemble de traces [Arn92].

Formellement :

- *Définition 1* : soit **A** = <S,T,α,β,λ> un système de transitions étiquetées sur un alphabet de Σ .

- *Définition 2* : à tout état s de S, on associe l'ensemble des chemins finis :

$$C\mathbf{A}(s) = \{c \in T^* \setminus \alpha(c) = s\}$$

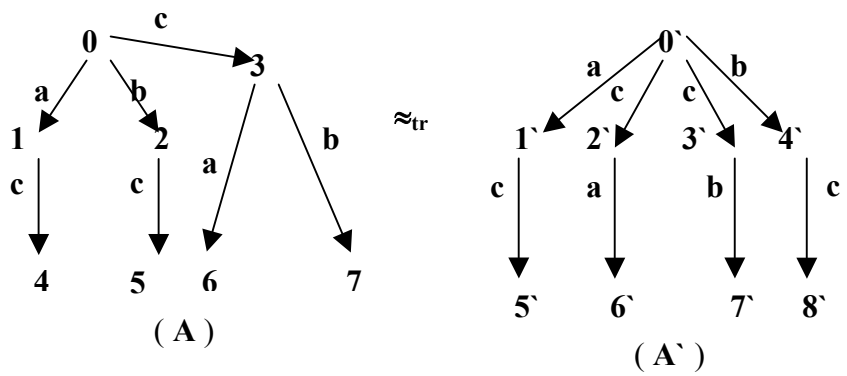
Et le langage L**A**(s) = λ(C**A**(s)) ; inclus dans **A***. L**A**(s) est donc l'ensemble de toutes les suites d'actions qui peuvent être exécutées à partir de s.

On dira que deux états s et s`de deux systèmes de transitions **A** et **A`** étiquetés par le même alphabet sont *trace-équivalents* (noté '**≈tr**') si L**A**(s) = L**A`**(s`).

- *Exemple 1* : soient les expressions de comportement suivantes :

A := ((a; stop [] b; stop) ||| c; stop)
A`:= (a ; stop ||| c; stop) [] (b; stop ||| c; stop)

D'après la ***Figure(3)***, on constate que **A ≈tr A`**, parce qu'ils peuvent exécuter exactement les mêmes séquences d'actions.



Figure(3) : Graphes associés aux expressions **A** et **A`**

2.2 SEMANTIQUE DE BRANCHEMENT :

Dans certains cas, l'étude des traces d'exécutions possibles peut être considérée suffisante pour l'analyse du comportement d'un système ; dans d'autres cas, l'équivalence de trace peut être jugée trop faible pour l'analyse de comportement où *l'indéterminisme* au niveau de tous *les branchements* possibles est d'importance capitale, ce qui est le cas pour les systèmes réactifs. En effet, un système réactif se distingue par rapport aux systèmes de production par son comportement interactif avec l'environnement à travers les points d'interaction observables.

Autrement dit, l'indéterminisme au niveau du comportement de tels systèmes rend l'utilisation de l'équivalence de trace inadéquate. Pour cela, d'autres relations sont utilisées pour la prise en compte de cet indéterminisme d'où leur appellation « *Equivalences de branchement* ». Cette sémantique comprend deux classes d'équivalence : « *l'Equivalence Observationnelle* » et « *l'Equivalence de Test* ».

Nous présentons dans cette section, l'équivalence Observationnelle. L'équivalence de Test sera détaillée ultérieurement.

2.2.1 Equivalence Observationnelle :

L'intuition derrière *l'équivalence observationnelle* est basée sur l'hypothèse que deux systèmes sont équivalents si, un observateur externe ne peut pas les distinguer au moyen d'expériences répétées à travers les points d'interaction. L'équivalence observationnelle entre deux processus est démontrée par l'existence d'une bissimulation entre l'ensemble des états de ces processus.

- BISSIMULATION :

La notion de *bissimulation*, définie initialement par Park [Par81], est à la base des techniques de vérification (dites souvent techniques de vérification par abstraction) utilisées dans le cadre des spécifications algébriques, et sans doute, la façon la plus simple pour définir l'équivalence observationnelle de deux systèmes de transitions. L'équivalence observationnelle définie par Milner dans [Mil80] est redéfinie dans [Mil89] comme une relation de bissimulation.

Intuitivement, deux systèmes sont bissimilaires (ou en bissimulation) lorsqu'ils ont les mêmes possibilités d'évolution et que chaque état atteint par une évolution de l'un est bissimilaire à un état atteint dans l'autre système par la même évolution. Ce qui se traduit par la définition suivante :

- *Définition3 :* Une relation R (sur les états d'un STE) est une bissimulation ssi elle vérifie : $\forall (p,q) \in R$, et $\forall \mu \in \sum \cup \{\tau\}$

si $p \xrightarrow{\mu} p'$ **alors** $q \xrightarrow{\mu} q'$ et $(p',q') \in \boldsymbol{R}$

si $q \xrightarrow{\mu} q'$ **alors** $p \xrightarrow{\mu} p'$ et $(p',q') \in \boldsymbol{R}$

Soient **A1** = <S1, T1, α1, β1, λ1> et **A2** = <S2, T2, α2, β2, λ2> deux systèmes de transitions étiquetées sur le même alphabet **A**. Une bissimulation entre **A1** et **A2** est une relation binaire ***R*** entre S1 et S2 tel que :

1. (a) Pour tout état s1 appartenant à S1, il existe s2 appartenant à S2 tel que s1***R***s2 ;

(b) Pour tout état s2 appartenant à S2, il existe s1 appartenant à S1 tel que s1***R***s2 ;

2. (a) Pour toute transition t1 appartenant à T1, et pour tout état s2 appartenant à S2 tel que $\alpha 1(t1)\boldsymbol{R}s2$, il existe t2 appartenant à S2 tel que $s2 = \alpha 2(t2)$, $\lambda 1(t1) = \lambda 2\ (t2)$ et $\beta 1\ (t1)\boldsymbol{R}\beta 2\ (t2)$;

(b) Pour toute transition t2 appartenant à T2, et pour tout état s1 appartenant à S1 tel que $s1\boldsymbol{R}\alpha 2(t2)$, il existe t1 appartenant à S1 tel que $s1 = \alpha 1(t1)$, $\lambda 1\ (t1) = \lambda 2(t2)$ et $\beta 1\ (t1)\boldsymbol{R}\beta 2\ (t2)$;

Une relation ***R*** qui ne satisfait que les conditions **2.**a et **2.**b est une *simulation* de **A1** par **A2**.En effet, on peut dire que **si** s1***R***s2 **alors** s2 est un état qui simule s1 car pour toute transition t1 d'étiquette **a** issue de s1 il existe une transition t2 d'étiquette **a** issue de s2 dont le but simule aussi le but de t1. Si ***R*** est une *bissimulation*, s1***R***s2 signifie donc que s1 simule s2 et réciproquement [Arn92].

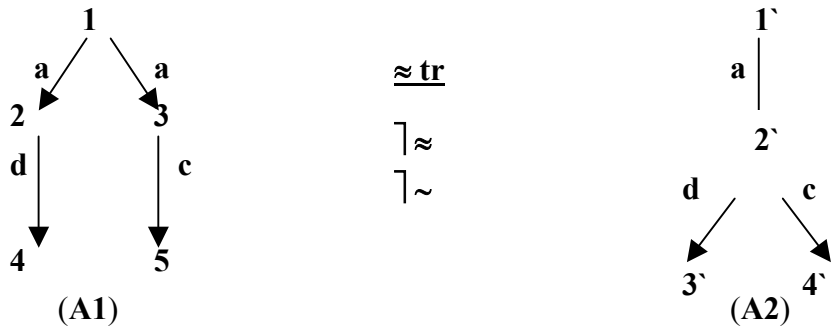
- <u>Exemple</u> :

Considérons les deux systèmes de transitions étiquetées **A1**, **A2** représentés graphiquement dans la ***Figure(4)***.

Bien que ces deux systèmes peuvent exécuter les mêmes séquences d'actions, et sont par conséquent, *trace équivalents*, Milner [Mil80], [Mil89] considère qu'ils ne sont pas équivalents puisque dans l'état **2`** du système **A2** on a le choix d'exécuter l'une des actions **d** ou **c**, alors que dans aucun état du système **A1** il n'est possible d'avoir une possibilité pareille. C'est cette idée intuitive que formalise la bissimulation. De ce fait, les systèmes **A1** et **A2** ne sont pas *bissimilaires*.

En effet, si une bissimulation devrait exister entre ces deux systèmes de transitions, intuitivement elle devrait être la relation ***R*** = { (1, 1`); (2, 2`); (3, 3`); (4, 3`); (5, 4`) }.

Or, cette relation n'est pas une bissimulation, car elle vérifie les conditions **1.**a, **2.**a et **1.**b mais pas la condition **2.**b.



<u>Figure(4) :</u> <u>Deux systèmes de transitions non bissimilaires</u>

- <u>Propriétés des bissimulations</u> :

- La relation inverse d'une bissimulation est aussi une bissimulation.
- La composée de deux bissimulations est une bissimulation.
- La réunion de deux bissimulations est une bissimulation.

- Bissimulation Forte :

Les propriétés précédentes permettent de définir *l'équivalence forte* notée '**~**'.

Définie par : p~q ⇔df il existe une bissimulation ***R*** telle que (p, q) ∈ ***R***.

'**~**' est réflexive car l'identité est une bissimulation. La symétrie et la transitivité proviennent respectivement du fait que l'ensemble des bissimulations est stable respectivement par inversion et par composition.

- Propriété : l'équivalence forte est la plus large *bissimulation*.

Une façon de définir la bissimulation forte, si elle existe entre deux systèmes de transitions étiquetées **A** et **A`** est de définir l'application :

E : φ (S × S`) ⟶ φ (S × S`) par (s, s`) ∈ ***E***(***R***) si et seulement si les trois conditions suivantes sont vérifiées :

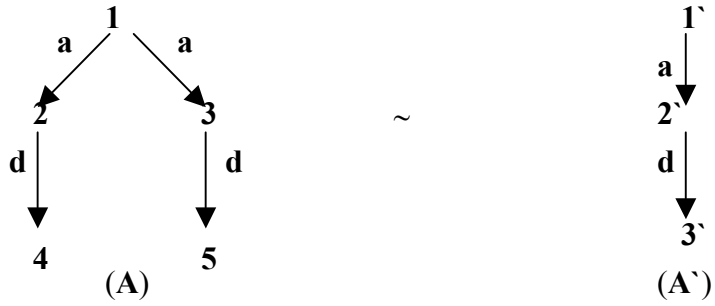
1- (s, s`) ∈ ***R***.

2- $\forall$ t1 = s $\xrightarrow{a}$ s1`∈ T, $\exists$ t2 = s` $\xrightarrow{a}$ s2` ∈ T`, tel que (s1, s2`) ∈ ***R***.

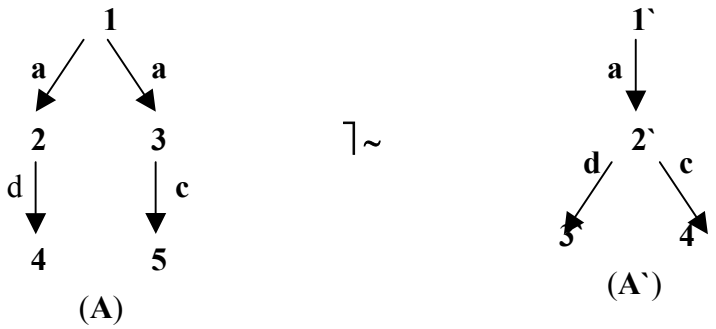
3- $\forall$ t2 = s` $\xrightarrow{a}$ s2`∈ T`, $\exists$ t1 = s $\xrightarrow{a}$ s1`∈ T, tel que (s1, s2`) ∈ ***R***.

- <u>Remarque</u> : la bissimulation forte (**~**) traite les actions internes (non observables) comme étant des actions observables.

- <u>Exemple</u> : les deux systèmes **A** et **A`** de la ***Figure(5)*** sont équivalents selon la bissimulation forte. Par contre, les deux systèmes **A** et **A`** de la ***Figure(6)*** ne sont pas équivalents selon la bissimulation forte.



<u>Figure(5) :</u> <u>Deux systèmes de transitions équivalents selon la bissimulation forte</u>

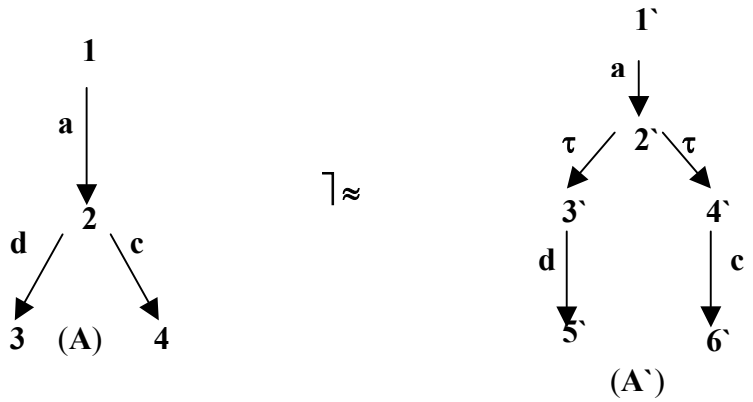


Figure(6) : Deux systèmes de transitions non équivalents selon la bissimulation forte

- Bissimulation Faible :

Jusqu'à présent, toutes les interactions appelées aussi occurrences d'actions ou encore événements apparaissant dans un système de transitons ont été traitées de la même façon. Nous avons seulement fait allusion, à l'emploi d'une action **τ** dite non observable. Milner explique dans [Mil80], [Mil89], les raisons d'être d'une telle action : tel que son nom l'indique, l'action non observable τ permet d'exprimer un changement d'état du système qui, d'une part, ne fait pas intervenir l'environnement et d'autre part, ce changement n'est pas observé par cet environnement. Ce qui permet d'exprimer un comportement spontané du système spécifié. Ceci se produit par exemple, lorsqu'un système est constitué de deux processus qui peuvent communiquer entre eux ou avec un observateur externe avec échange de messages ; lorsque des messages seront échangés entre ces deux processus, le système changera d'état sans que l'action qui a provoqué ce changement d'état soit visible par l'observateur.

Soient les deux systèmes de transitions **A** et **A`** de la ***Figure(7)***. Lorsque le système de transitions **A`** est dans l'état **1`** et que l'action **a** est exécutée, il passe dans l'état **2`** ; puis, spontanément et de façon non déterministe, il passe dans l'état **3`** ou dans l'état **4`**. Selon le cas, il peut alors exécuter soit l'action **d** soit l'action **c**. Tout ce passe donc comme si l'action **a** faisait passer de façon non déterministe le système de l'état **1`** à l'état **3`** ou à l'état **4`**. Le système de transitions **A`** est donc équivalent en un certain sens au système de transitions **A**.



Figure(7) : Deux systèmes de transitions non équivalents selon la bissimulation faible

Toutefois, ces deux systèmes de transitions ne sont pas équivalents par bissimulation. Il faut donc définir une nouvelle relation d'équivalence, qui prend en compte l'existence de transitions invisibles, et qui ne fait intervenir que les comportements « *observables* » des

systèmes de transitions à comparer. Une telle relation est appelée *équivalence observationnelle* (*bissimulation faible*), elle est définie de façon analogue à la bissimulation, mais au lieu de ne considérer que les transitions, on considère aussi les « *chemins* ».

Dans un système de transitions étiquetées **A** = <S, T, α, β, λ> ; dont l'alphabet d'actions A contient une action invisible, notée **τ**, on peut généraliser la notion de transition en définissant un autre ensemble de transitions Tt, inclus dans S×A×S, défini par :

(s, a, s`) si et seulement si :

- il existe dans **A** un chemin allant de s à s` dont la trace est de la forme **τnaτm** avec n et m ≥ 0.

Deux systèmes de transitions sont observationnellement équivalents s'il existe une bissimulation entre les systèmes de transitions obtenus en remplaçant leur ensembles de transitions par les ensembles Tt définis ci-dessus. Cette définition peut aussi

$$a$$

s'écrire de la façon suivante, en notant par $s \Rightarrow s`$ le fait que (s, a, s`) appartient à Tt. Une équivalence observationnelle entre deux systèmes de transitions **A1**, **A2** est une relation ***R*** entre leurs ensembles S1 et S2 d'états qui vérifie :

1. (a) pour tout état s1 appartenant à S1, il existe s2 appartenant à S2 tel que s1***R***s2 ;
(b) pour tout état s2 appartenant à S2, il existe s1 appartenant à S1 tel que s1***R***s2 ;

2. (a) pour tout état s1 appartenant à S1, il existe s2 appartenant à S2 tel que

a a

s1***R***s2, **si** $s2 \Rightarrow s1`$ **alors** il existe s2` appartenant à S2 tel que s1`***R***s2` et $s2 \Rightarrow s2`$.

(b) pour tout état s1 appartenant à S1, il existe s2 appartenant à S2 tel que

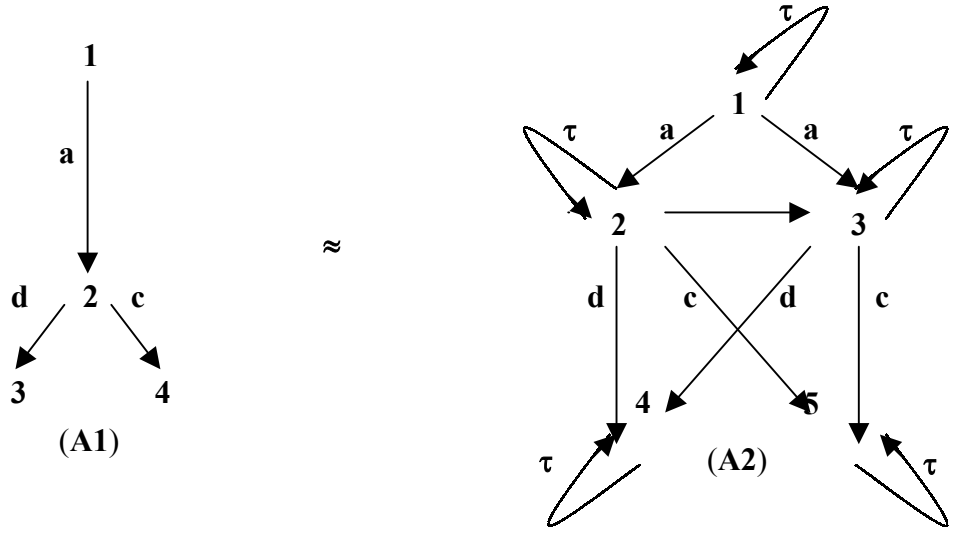
(c)

a a

s1***R***s2, **si** $s2 \Rightarrow s2`$ **alors** il existe s1 appartenant à S1 tel que s1`***R***s2` et $s1 \Rightarrow s1`$.

- *Exemple* : Considérons les deux systèmes de transitions de la ***Figure(8)*** :

Ces deux systèmes sont observationnellement équivalents.



Figure(8) : Deux systèmes de transitions équivalents selon la bissimulation faible

En considérant une évolution comme une transition $\overset{\mu}{\Rightarrow}$, une autre définition de la bissimulation faible est :

- *Définition4* : Une relation ***R*** est dite une bissimulation ssi elle vérifie :

$$\forall (p,q) \in \boldsymbol{R}, \text{ et } \forall \mu \in \Sigma \cup \{\varepsilon\}$$
$$\textbf{si } p \overset{\mu}{\Rightarrow} p` \textbf{ alors } q \overset{\mu}{\Rightarrow} q` \text{ et } (p`, q`) \in \boldsymbol{R}$$
$$\textbf{si } q \overset{\mu}{\Rightarrow} q` \textbf{ alors } p \overset{\mu}{\Rightarrow} p` \text{ et } (p`, q`) \in \boldsymbol{R}$$

- *Remarque* : En l'absence de transitions internes (τ), les deux notions de bissimulation (Forte et Faible) coïncident.

- *Définition5* : (**Equivalence de bissimulation**) Deux états (processus), p et q, sont dits bissimilaires ou (observationnellement équivalents), noté $p \approx q$, lorsqu'il existe une bissimulation ***R*** telle que p ***R*** q.

Cette définition caractérise l'équivalence observationnelle [Mil89] (forte resp. faible) comme la plus large bissimulation (forte resp. faible) :

$$\sim = \bigcup_{\boldsymbol{R} \subseteq S\times S \setminus \boldsymbol{R} \text{ est une bissimulation}} \boldsymbol{R}$$

$$\approx = \bigcup_{\boldsymbol{R} \subseteq S\times S \setminus \boldsymbol{R} \text{ est une bissimulation faible}} \boldsymbol{R}$$

Deux STE sont dits observationnellement équivalents lorsque leurs états initiaux le sont.

La bissimulation peut s'utiliser pour montrer que deux spécifications d'un même système sont obsevationnellement équivalentes. D'un autre coté, elle peut aussi être utilisée pour obtenir le comportement de référence par réduction de systèmes de transitions.

3- APPROCHE LOGIQUE :

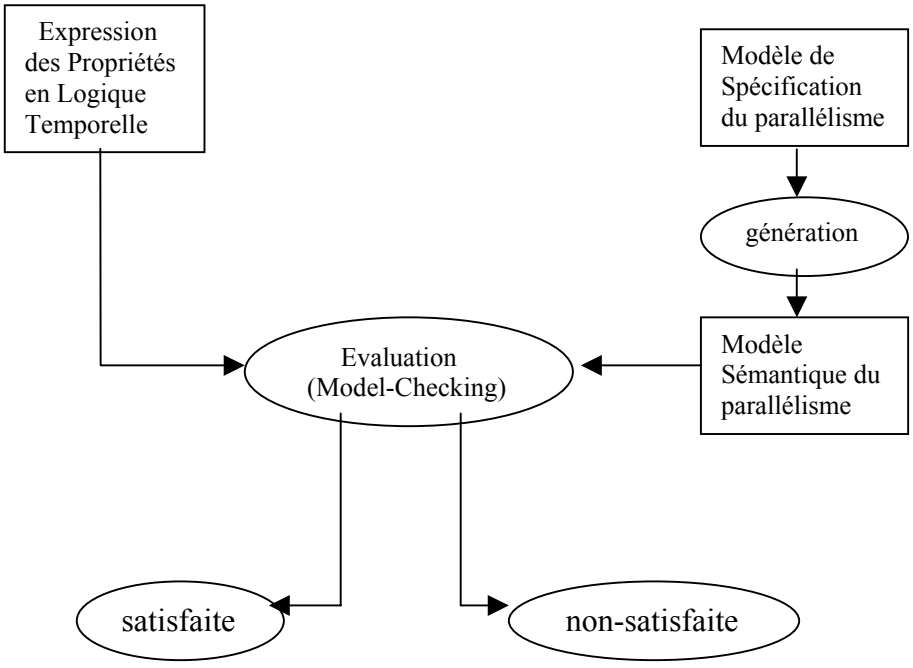
Pour cette approche, la spécification d'un système n'est autre que l'ensemble des *propriétés logiques* que doit vérifier toute réalisation de ce système.

Deux systèmes sont équivalents s'ils satisfont les mêmes propositions logiques. Une réalisation est conforme si elle vérifie toutes les propositions logiques de sa spécification. La vérification de la conformité de la réalisation avec la spécification de ce système est conduite

par une procédure de *contrôle* de satisfaction de ces propriétés qui calcule dans chaque état (la réalisation est vue comme un graphe d'états) la valeur de vérité de chaque proposition logique.

Les propriétés attendues d'un système sont exprimées par des *assertions* d'un langage spécifique, par exemple les *logiques temporelles* [Sif82, CE81, Pnu86a, Pnu86b] qui ont été introduites et utilisées dans le cadre de cette approche. Dans ce cas, la description opérationnelle du système (le graphe de comportement) est considéré comme un modèle de cette logique. La procédure de décision revient alors à effectuer du « *contrôle de modèle* » (*Model-Checking*) permettant d'associer à chaque formule l'ensemble des états du modèle qui la satisfont. Un système vérifie une propriété si cette propriété est satisfaite par tous les états du comportement du système. Cette idée est illustrée par la ***Figure(9)*** ci-dessous.

Notons ici, que le travail de M.Hennessy et R.Milner [24], permet de mieux comprendre les liens entre l'approche *Comportementale* et l'approche *Assertionnelle* de la vérification et de mieux cerner ce qui les unit. Il fournit en particulier, une définition « *logique* » des équivalences de comportement (en particulier, l'équivalence de bissimulation dans [HENM85]) qui permet de préciser le type de propriétés que l'on peut vérifier.



Figure(9) : Représentation de la vérification logique

3.1 *HML* :LOGIQUE DE HENNESSY-MILNER :

- *Syntaxe* : *HML* est le plus petit ensemble vérifiant :
True $\in$ *HML*, A, B $\in$ *HML* $\Rightarrow$ A $\wedge$ B, $\neg$ A $\in$ *HML*
A $\in$ *HML*, o $\in$ O $\Rightarrow$ <o> $\in$ *HML*.

- *Sémantique* : La relation (la plus petite) de satisfaction |= , définit la sémantique du langage. Elle permet de définir à quelles conditions une formule est vérifiée dans un état particulier d'une structure donnée comme suit :

Σ , s |= True $\forall$ s $\in$ S
Σ , s |= A $\wedge$ B ssi Σ , s |= A et Σ , s |= B
Σ , s |= $\neg$ A ssi Non (Σ , s |= $\neg$ A)
Σ , s |= <i> A ssi s => s` et $\Sigma$ , s` |= A
- *Abbreviations*: False ≡ $\neg$ True, A V B ≡ $\neg$ ($\neg$ A $\wedge$ $\neg$ B), [i] A ≡ $\neg$ <i> $\neg$ A
<w> A ≡ <i1> <i2> … <in> A pour w = i1, i2, …in

- *Théorème de Hennessy-Milner* :

HML caractérise **Modalement** la bissimulation. Deux états sont bissimilaires s'ils satisfont les mêmes énoncés de la logique *HML*.

s ≡obs **q** ssi TH-*HML*(**s**) = TH-*HML*(**q**)

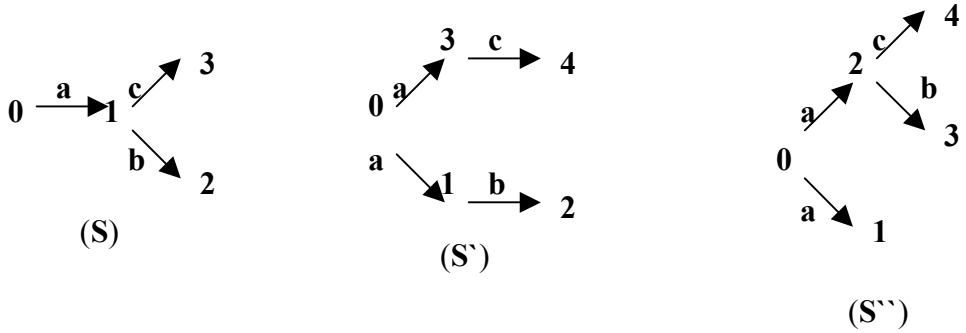
- *Exemple*:

Σ , p |= <s> True Une a-expérimentation est possible à partir de p
Σ , p |= <a> (<b> True $\wedge$ <c> True)

A partir de p, une a-expérimentation est possible menant dans un état pour lequel à la fois sont possibles une b-expérimentation et une c-expérimentation.

Σ , p |= [a] False Aucune a-expérimentation n'est possible partant de p

On considère les STE de la ***Figure(10)*** suivante et les énoncés Fi : i $\in$ [1, 4].



Figure(10): Trois systèmes de transitions différents

La table suivante, indique les couples (*STE*, *formules*) pour lesquels on a :

$$\sum i , 0 \models Fj$$

F1≡ <a> [b] *F*
F2≡ <a> (<b> T ∧ <c> T)
F3≡ [a] (<b> T ∧ <c> T)

F4≡ <a> ((<b> T ∧ [c] *F*) ∨ (<c> T ∧ [b] *F*))

	F1	F2	F3	F4
S	⊥	⊤	⊤	⊥
S`	⊤	⊥	⊥	⊤
S``	⊤	⊤	⊥	⊥

On peut conclure que les STE de la ***Figure(3.2)*** sont deux à deux non équivalents observationnellement :

- **S** est le seul STE satisfaisant *F3* donc **S** n'est équivalent ni à **S`** ni à **S``**.
- **S`** est le seul STE satisfaisant *F4* donc **S`** n'est équivalent ni à **S** ni à **S``**.

- Comparaison entre les approches Logique et Comportementale :

L'approche assertionnelle met en œuvre des propriétés explicitement énoncées dans un langage spécifique possédant une sémantique précise. Vérifier que le système satisfait ces propriétés revient à s'assurer que le système est un modèle de ces propriétés.

L'approche Comportementale ne manipule que des comportements. On étudie l'équivalence entre les comportements du système et de sa spécification. On déduit de cette équivalence que le système satisfait (modulo le critère d'abstraction retenu) ses spécifications sans jamais avoir explicitement énoncé les propriétés associées à la spécification ou préciser la nature des propriétés préservées par l'équivalence utilisée.

Nous présentons dans cette partie, les principes généraux de ***l'approche Test*** ainsi que l'intérêt que nous portons à l'utilisation de cette approche.

4- APPROCHE TEST:

L'approche Test a été récemment développée dans le cadre des techniques de spécification par algèbres de processus, notamment CCS et LOTOS qui est un langage dérivé de cette algèbre.

L'équivalence de test **te** (que l'on peut aussi appeler *équivalence de blocage* ou de refus: *Failure equivalence*) ne distingue les comportements que par leur traces et les possibilités de blocage. Contrairement à l'équivalence observationnelle ≈ (bissimulation faible classique) et à l'équivalence forte ~ (bissimulation forte classique), elle est insensible vis-à-vis de l'endroit où les choix non-déterministes sont réalisés. Autrement dit, elle ne distingue pas les systèmes selon le niveau de non déterminisme ni selon l'origine de ce non déterminisme. Mais distingue des comportements qui bien que possédant les mêmes traces (et donc non distinguables par **tr-eq** , *l'équivalence de traces* ou *langage*), ne présentent pas les mêmes

propriétés de blocage: Intuitivement, la notion de blocage considère que le système est une boite noire munie de boutons représentant les actions que le système peut exécuter. Un bouton bloqué signifie que le système refuse l'action correspondante.

- *Exemple1*: Considérons les deux spécifications LOTOS (*test-équivalentes*) suivantes:

a; (tau; c; stop[]tau; d; stop) **te** a; c; stop[]a; d; stop

Bien que les STE correspondants à ces deux spécifications ne sont pas bissimilaires, il ne sont pas distinguables par *l'équivalence de Test* (**te**) :

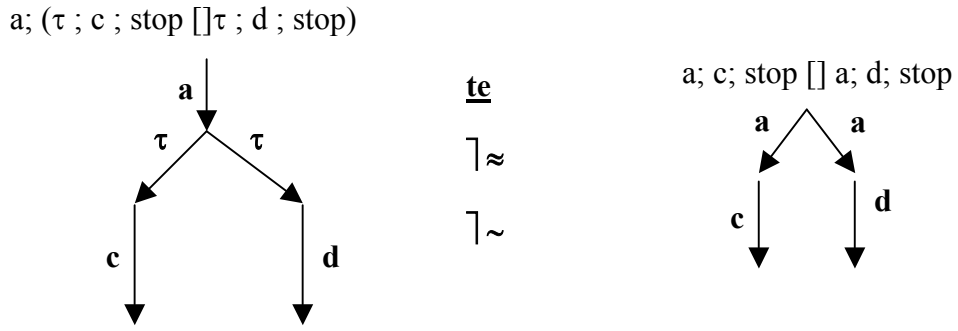


Figure 11 : *Comparaison entre l'équivalence de test et la bissimulation*

- *Exemple2*: Considérons les deux spécifications LOTOS (*trace- équivalentes*) suivantes :

a ; b ; stop[]a; c; stop **t/e** a; (b; stop[]a; c; stop) et les STE associés ci-dessous :

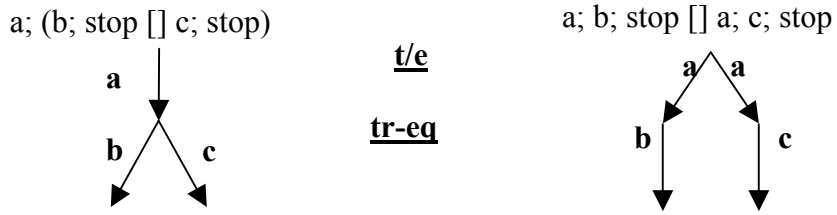


Figure 12 *: Comparaison entre l'équivalence de tests et l'équivalence* de traces

Le système de gauche offre un comportement où il est toujours possible, en tant qu'utilisateur, de choisir entre les actions '**b**' et '**c**'. Le système de droite, bien que capable à priori d'offrir '**b**' ou '**c**' après '**a**', peut offrir uniquement '**b**' (ou '**c**') et *bloquer* par rapport à '**c**' (ou '**b**').

Ainsi, des relations d'équivalences et les préordres qui les génèrent (un préordre est une relation binaire réflexive et transitive. A tout préordre correspond une relation d'équivalence) sont dites *équivalences de test* grâce à une approche qui consiste à *comparer* les systèmes via (uniquement) les résultats de l'application de *tests* (ou observations) à ces systèmes sans se préoccuper de la façon dont ces systèmes sont réalisés.

Vous pouvez constater que bien que l'équivalence observationnelle et l'équivalence de Test soient toutes deux classées sous la même approche, l'approche Comportementale, elles diffèrent dans la manière de comparer les comportements et dans le critère considéré pour

vérifier que deux systèmes sont équivalents. D'où la nécessité d'une autre classification des approches de vérification :

Sachant que la validation du système conçu consiste à montrer que son implémentation est *conforme* à sa spécification, et que les approches de validation peuvent être différentes selon le type des objets appelés : *Spécification* et *Implémentation*, deux approches de validation peuvent être distinguées :

l'approche boite blanche et ***l'approche boite noire***.

Dans les deux approches, *la conformité* est une relation de satisfaction qui doit lier l'implémentation et la spécification.

4.1 LA CONFORMITE EN LOTOS :

Cette relation peut avoir plusieurs définitions selon l'objectif recherché par son utilisation. Nous retenons le même objectif dans les approches *boite blanche* et *boite noire* : c'est la validation de l'implémentation en montrant qu'elle est conforme à sa spécification. Cependant, ces deux approches se distinguent par la façon de vérifier que cette relation est satisfaite :

- Approche boite blanche :

Cette approche peut etre appliquée dès lors que la spécification et l'implémentation sont deux structures entièrement et librement explorables. Elle se distingue par le fait qu'elle ne s'impose pas de contraintes concernant l'accès à l'implémentation (c'est-à-dire, l'implémentation peut être manipulée au même titre que sa spécification). Elle est communément appelée : ***Vérification***. (On peut appliquer en particulier, la vérification par Bissimulation et l'approche Logique).

Dans cette approche, on parle de : *vérification de la conformité*.

- Approche boite noire :

Appelée traditionnellement « ***Test*** » des implémentations, se substitue à la première approche lorsque l'implémentation est considérée comme une boite noire qui n'est accessible que via ses points d'interaction avec l'extérieur, appelés : *ports de communication*. Dans cette approche, on parle de : *test de conformité*.

4.2 MOTIVATIONS :

L'approche Test se distingue par cette contrainte qu'elle impose à la procédure de vérification de la conformité. Ce qui réduit son utilisation à la famille des relations de conformité pouvant être caractérisées selon cette approche et connues sous le nom de relations de test. Mais, en même temps, imposer cette contrainte la rend applicable pour tout type de représentation de l'implémentation.

L'utilisation des spécifications formelles comme référence pour la validation des implémentations a motivé l'introduction des équivalences de test [Abr87, DENH84, Phi87], et a conduit dans le cadre de LOTOS à une définition formelle de la notion de conformité (ou relation d'implémentation) [BRISS87, Led91a] et à la proposition d'une théorie pour la dérivation de tests [Bri88b, Led91b].

L'intérêt que nous portons à l'utilisation des relations de test pour une procédure de vérification lors d'un processus d'implémentation est motivé par les arguments suivants :

- Une relation d'implémentation ne doit pas ignorer la présence du caractère asymétrique entre une implémentation et sa spécification. En effet, la spécification fait souvent abstraction de certains détails que l'implémentation doit réaliser. De plus, les « *styles* » de spécification ne doivent pas influencer le résultat de la vérification de conformité. Il est alors important de fournir des techniques de vérification (des relations de conformité) qui servent à réaliser la procédure de validation lorsqu'il est possible de manipuler une implémentation au même titre que sa spécification, notamment au cours d'un cycle de développement d'un système communicant par raffinements successifs; cycle au cours duquel la spécification et l'implémentation ne sont autres que deux structures comparables qui sont sensées décrire le même système, la seconde étant plus fine (ou détaillée), et la première étant plus abstraite.

De ce point de vue apparaît, l'utilité de caractériser les relations de conformité par des définitions automatiquement vérifiables.

D'un autre coté, il est aussi utile de pouvoir générer automatiquement les tests que l'on doit appliquer pour valider les implémentations lorsque celles-ci ne peuvent être validées que par le test.

L'exemple suivant explique ce problème.

- Exemple :

La spécification de la phase de connexion du service local (coté initiateur) d'un protocole transport (sans requête de déconnexion avant la fin de la phase de connexion) peut être représentée par le système de transitions (non déterministe) suivant :

Après une demande de connexion, l'utilisateur doit s'attendre à ce que sa demande soit refusée (état **2**), et doit la reformuler ; soit elle est acceptée (état **3**) et il peut entamer son transfert de données (état **4**).

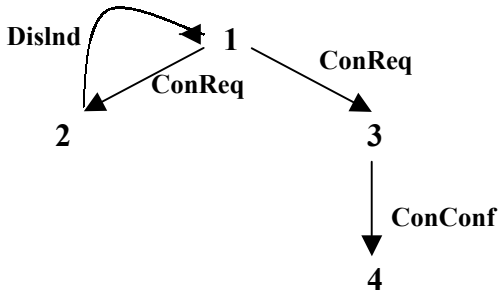
L'implémentation de ce service peut présenter le comportement observable suivant :

Le non déterminisme exprimé par la spécification est retrouvé sous forme d'évolutions internes au protocole de l'entité transport et incontrôlables par son utilisateur. L'origine d'une évolution vers un rejet peut être :

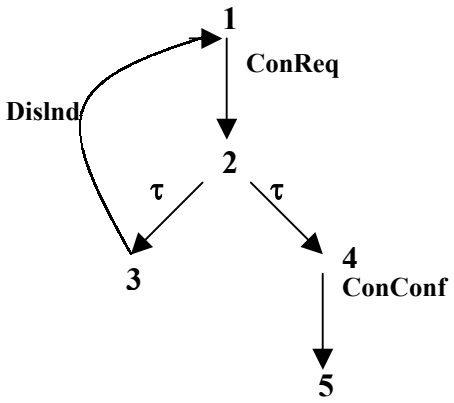
- Soit le résultat d'une décision locale à l'entité : le nombre maximal de connexions supportées est atteint. Elle représente abstraitement une évolution qui dépend d'un calcul interne et non d'une interaction avec l'extérieur.

- Soit le résultat d'une décision de l'entité distante (pour les raisons expliquées ci-dessus) ou de son usager. Dans ces deux cas, les τ-transitions représentent le résultat d'une communication interne à l'ensemble des deux entités.

Les ***Figures*** : ***(13)*** et ***(14)*** représentent, respectivement, *la spécification* et *l'implémentation* du service local transport :



Figure(13) : *Spécification simplifiée d'un service local transport*



Figure(14) : *service local d'une Implémentation transport*

Dans tous ces cas, les τ–transitions représentent des évolutions du système dites internes non-observables et non contrôlables par l'usager. La façon de représenter le non-déterminisme du résultat de la demande de connexion fait que l'implémentation n'est pas bissimilaire (ou observationnellement équivalente) à la spécification. Il est alors inadéquat de retenir cette équivalence comme critère de conformité. Ceci réduirait inutilement les choix d'implémentation et rend le contrôle du processus d'implémentation très sensible aux styles de spécification.

Les relations relevant de la notion de conformité en LOTOS ne distinguent pas ces deux systèmes de transitions et se révèlent de ce point de vue être admissibles pour contrôler la conformité lors du processus d'implémentation.

- La génération de code exécutable à partir du modèle formel d'une implémentation **vérifiée conforme** augmenterait la confiance en la validité du système réalisé. Si l'on admet la fiabilité de la procédure de génération de code, alors il n'est même plus besoin de tester le code généré (la réalisation du système). D'un point de vue « *économique* », ceci est intéressant vu le coût généralement important de l'exécution exhaustive des tests de conformité.

- Définitions :

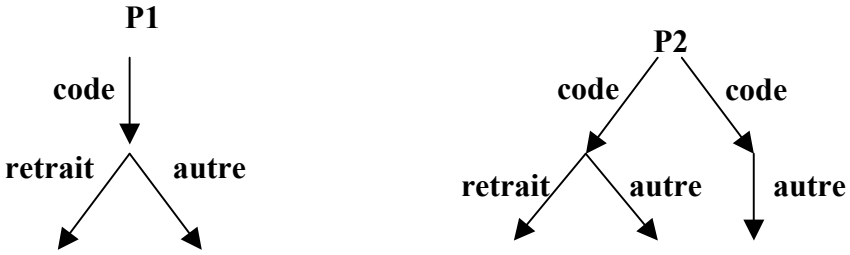
La relation de conformité de base en LOTOS a été initialement définie [BRISS87, Bri88b] comme suit :

$$I \underline{\textbf{conf}} S \Leftrightarrow \forall \sigma \in Tr(S), \forall A \subseteq L:$$
$$\textbf{si} \quad \exists I' \,(I \overset{\sigma}{\Rightarrow} I' \text{ et } \forall a \in A : I' \not\overset{a}{\Rightarrow})$$
$$\textbf{alors} \quad \exists S' \,(S \overset{\sigma}{\Rightarrow} S' \text{ et } \forall a \in A : S' \not\overset{a}{\Rightarrow})$$

Informellement : une implémentation ***I*** est conforme à une spécification ***S*** lorsque : pour toute trace σ du comportement spécifié, si l'implémentation ***I*** peut évoluer par σ alors les ensembles d'actions A qu'elle peut refuser d'exécuter après cette évolution, sont (au plus) ceux que la spécification ***S*** peut refuser après σ.

Notons que cette définition n'exige pas que l'ensemble des traces de l'implémentation soit un sous-ensemble de celui de la spécification, ni le contraire. Ceci est une caractéristique spécifique à cette notion de conformité qui ne veut pas « *dépasser* » les limites pratiques du test de conformité.

- Exemple : Les comportements **P1** et **P2** de la ***Figure(4.3)*** vérifient : **P1** **conf** **P2** mais ⌉(**P2** **conf** **P1**) : voir σ = code et A = {retrait}.



Figure(15) : *Test et Conformité*

Nous nous intéressons ici aux différentes relations définies autour de cette notion de conformité.

Comme dans [Led90], d'autres définitions équivalentes peuvent être données, en utilisant des notations plus compactes. Pour cela, considérons un système de transitions arbitraire S = (S, L, Δ, s0), A ⊆ L un ensemble quelconque d'actions et p ∈ S un état quelconque.

- p **ref** A ≡df $\forall a \in A \quad p \overset{a}{\not\Rightarrow}$
  p n'a aucune dérivation par aucune action a de A.
- p **after** σ = $\{p' : p \overset{\sigma}{\Rightarrow} p'\}$
  ensemble de tous les dérivés de p par la séquence σ. Notons que si σ ∉ Tr(p) alors cet ensemble est vide.
- p **after** σ **ref** A ≡df ∃ p`∈ p **after** σ, p` **ref** A
  p admet au moins une dérivation p` par σ telle que p` **ref** A. Notons que si p n'admet aucune dérivation par σ, alors p **after** σ **ref** A a pour valeur logique FAUX.

Rappelons que tout ce que nous définissons sur les états d'un système de transitions est étendu aux systèmes de transitions en les identifiant à leur état initial, c'est-à-dire S **ref** A signifie simplement que s0 **ref** A, etc.… Rappelons aussi que lorsqu'on compare deux systèmes de transitions, l'alphabet L figurant dans les définitions est la réunion des alphabets des deux systèmes : L = L1 ∪ L2. dans les définitions suivantes ***I*** et ***S*** sont deux systèmes de transitions arbitraires et L est la réunion de leurs alphabets.

Une autre définition de la conformité est :

I **conf** ***S*** ⇔ ∀ σ ∈ Tr(***S***) ∩ Tr(***I***), ∀ A ⊆ L :

Si ***I*** **after** σ **ref** A
alors ***S*** **after** σ **ref** A

La relation de conformité est réflexive mais non transitive, elle n'est donc pas un préordre mais induit trois préordres définis de la façon suivante :

$$I \text{ red } S \Leftrightarrow \{ I \text{ conf } S \text{ et } \{ \mathrm{Tr}(I) \subseteq \mathrm{Tr}(S)$$

$$I \text{ ext } S \Leftrightarrow \{ I \text{ conf } S \text{ et } \{ \mathrm{Tr}(I) \supseteq \mathrm{Tr}(S)$$

$$I \geq S \Leftrightarrow \{ I \text{ conf } S \text{ et } \{ \mathrm{Tr}(I) = \mathrm{Tr}(S)$$

Les relations d'équivalence induites de ces trois préordres coïncident avec la relation appelée ***équivalence de Test*** et notée **te** définie par :

$$I \text{ te } S \Leftrightarrow \{ I \text{ conf } S \text{ et } S \text{ conf } I \; \{ \mathrm{Tr}(I) = \mathrm{Tr}(S)$$

Dans le cadre du langage LOTOS, une théorie de dérivation de tests a été proposée [BRISS87, Bri88b] pour tester la conformité d'une implémentation par rapport à sa spécification.

L'objectif de cette section est de présenter, par une synthèse des approches de Test, un modèle réduit qui résume la problématique. Nous visons de passer progressivement de la problématique relativement abstraite de l'approche de comparaison de systèmes par test, à la problématique pragmatique que constitue le test des systèmes communicants.

4.3 LES MODELES DE TEST :

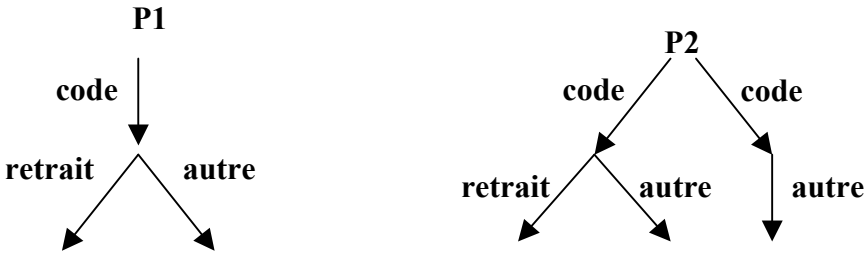
Un modèle (dénotationnel) de test pour un ensemble de processus, P, est défini par la donnée :

- D'un ensemble O de testeurs (ou observateurs), et
- D'une application ***R*** : P × O → P({⊤, ⊥}) \ Ø

Qui détermine les résultats attendus de l'application d'un test à un processus.

La définition de ***R*** peut paraître surprenante dans le sens ou l'application d'un test peut aboutir sur $\{\top, \bot\}$: c'est à dire échec et succès.

- ***Exemple 1 :*** Prenons comme exemple les deux distributeurs d'argent, **P1** et **P2**, de la ***Figure (16).***



Figure(16) : *Test et non déterminisme*

P1 est la machine idéale qui dispose toujours de billets, et qui après avoir identifié le code secret offre au client le choix entre le retrait d'argent ou d'autre opérations concernant son compte. Cette machine « répond » toujours positivement au test :

t = code (puis) retrait. Ceci s'exprime par :

$$\boldsymbol{R}\ (P1, t) = \{\ \top\ \}.$$

La machine P2 non idéale, peut tomber en rupture de stock de billets de banque (n'autorisant plus que les autres opérations) : elle réagit de façon non déterministe (c'est à dire non prévisible par le client) au test précédent. Ce qui s'exprime par :

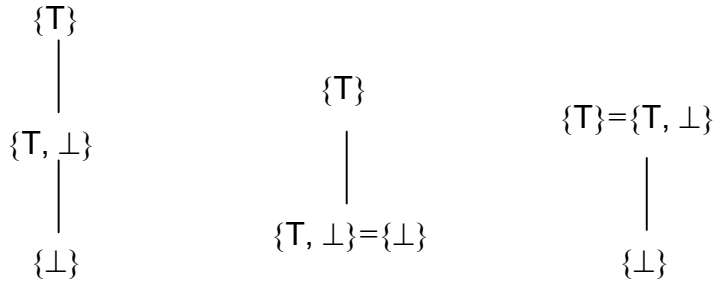
$$\boldsymbol{R}\ (P1, t) = \{\top, \bot\}.$$

Sans avoir à définir l'exécution des tests, on peut noter déjà qu'il peut y avoir plusieurs, au moins trois, façons de comparer deux processus.

En effet, en considèrent l'ensemble $\{\top, \bot\}$ ordonné par échec $\bot \subset \top$:

$$\begin{array}{c} \top \\ | \\ \bot \end{array}$$

Il existe trois façons d'ordonner l'ensemble des parties non vides de $\Theta = \{\top, \bot\}$, qui correspondent respectivement au préordre de Plotkin ($\subseteq$p), de Smith ($\subseteq$s) et de Hoare ($\subseteq$h) :



.

Il en résulte trois préordes standards de test notés respectivement : ≤0, ≤must, ≤may et définis sur P par : ∀p, q ∈ P,

$p \leq 0 \quad q \equiv def \ \forall t \in O \ \ \boldsymbol{R}(p, t) \subseteq p \ \boldsymbol{R}(q, t)$

$p \leq must \ q \equiv def \ \forall t \in O \ \ \boldsymbol{R}(p, t) \subseteq s \ \ \boldsymbol{R}(q, t)$

$p \leq may \ q \equiv def \ \forall t \in O \ \ \boldsymbol{R}(p, t) \subseteq h \ \boldsymbol{R}(q, t).$

Les appellations **must** (i.e. devoir) et **may** (i.e. pouvoir) sont justifiées par le fait que les préordre correspondants peuvent être définis par :

p ≤must q ssi pour tout test t ∈ O, si p **doit** réussir t alors il en est de même pour q.

p ≤may q ssi pour tout test t ∈ O, si p **peut** réussir t alors il en est de même pour q.

Ce qui se formalise par la proposition suivante :

Proposition [Dri92] : caractérisation des préordres de test

(i) p ≤must q ssi ∀t ∈ O p **must** t implique q **must** t.
(ii) p ≤may q ssi ∀t ∈ O p **may** t implique q **may** t.

Les prédicats **must** et **may** étant définis par : pour tout processus p ∈ P, pour tout test

t ∈ O,
p **must** t $\equiv_{def}$ $\perp \notin \boldsymbol{R}(p, t)$: aucun échec n'est signalé par les exécutions du test t sur le processus p.

p **may** t $\equiv_{def}$ $\top \in \boldsymbol{R}(p, t)$: un succès est signalé par une exécution du test t sur le processus p.

Considérant l'ensemble O des tests t, simplement constitué par des séquences (finies mais de longueur arbitraire) d'actions, se terminant par le « *verdict* » succ. Soit l'ensemble :

{σ = a1;a2…;an ;succ | ai∈ ∑ et n >= 0} contenant les expressions définies par la syntaxe suivante :

t ::= succ|a;t.

Où a est une action quelconque dans ∑.

La fonction ***R*** (résultats d'exécutions de test) étant définie par les règles suivantes :

$\boldsymbol{R}(p, succ) = \{\top\}$ (**r1**).

$\boldsymbol{R}(p, a;t) = \cup_{p' \mid p \xRightarrow{a} p'} \boldsymbol{R}(p', t) \cup \cup_{p' \mid (p \xRightarrow{\varepsilon} p') \wedge p' \not\xRightarrow{a}} \{\bot\}$ (**r2**).

(**r1**) signifie qu'un succès est reporté lorsque l'on atteint la fin du test.

(**r2**) signifie que le test a;t reporte le résultat du test t appliqué à p`, si le processus sous test p peut exécuter a et se transformer en p`. dans le cas où ce processus peut évoluer de façon interne, pour atteindre un état à partir duquel, il ne peut pas exécuter a, alors un échec est rajouté au résultat de l'application.

Reprenons l'***exemple 1*** des deux processus décrits dans la ***Figure (4.1).*** D'après la définition de ***R***, il est clair que ces deux processus réagissent de la même manière à tous les tests (par exemple code;succ, code;autre;succ autre ;succ, etc.…) sauf au test code;retrait;succ.

Les résultats concernant ce dernier sont différents selon qu'il est appliqué à P1 ou à P2 :

$\boldsymbol{R}(P1, code;retrait;succ) = \{\top\}$

$\boldsymbol{R}(P2, code;retrait;succ) = \{\top, \bot\}$

On peut alors conclure que P1 et P2 (ne) sont équivalents (que) par le « **may**-test » :

$P1 \leq_{may} P2$ et $P2 \leq_{may} P1$. Ce modèle permet de caractériser, comme préordre de test, l'inclusion des traces. Ce qui se traduit par la proposition suivante :

Proposition : (Caractérisation de l'équivalence des traces)
$Tr(p) \subseteq Tr(q) \Leftrightarrow p \leq_{may} q$.

- ***Exemple 2 :*** Le modèle de test que nous avons présenté précédemment a la particularité de ne considérer que des tests séquentiels. Même par $\leq_{must}$, les deux processus **p** et **q** de la ***Figure (17)*** suivante :

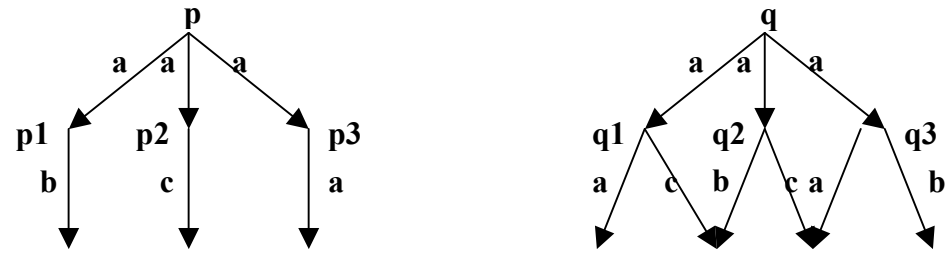


Figure (17) *: processus non distinguables par un test séquentiel.*

Ils ont, tous les deux, {T, ⊥} comme résultat à chacun des tests a;a;succ, a;b;succ, et a;c;succ. Les résultats du test a;succ est {T} pour chacun d'entre eux. Le résultat de tout autre test est {⊥}.

Comme on peut le voir sur la ***Figure (17),*** les processus p et q sont distingués par le fait qu'après l'action a, le processus q ne peut jamais refuser ensemble deux actions parmi {a,b,c} : quelle que soit son évolution après a, il peut toujours exécuter au moins une action parmi {a,b} (idem pour {a, c} et {b, c}). Ceci n'est pas le cas pour le processus p qui –après **a**- peut être dans un état ne pouvant exécuter ni **a** ni **b** (état du milieu qui ne peut exécuter que **c**).

Un test séquentiel ne peut pas vérifier ce genre de comportement. C'est à dire qu'il n'est pas possible d'exprimer le fait que après a, le processus q accepte toujours d'exécuter au moins une des actions **a** ou **b**.

Le modèle précédent peut alors être enrichi par un nouveau type de tests non séquentiels que l'on obtient en ajoutant un nouvel opérateur «**V** » au langage de test.

Un sur ensemble de O, O` ⊇ O, est alors considéré. Les tests t ∈ O` sont définis par la syntaxe :

$$t ::= succ|a\ ;t|t1\ V\ t2.$$

La fonction ***R*** correspondante contient –en plus de (**r1**) et (**r2**)- la règle suivante :

$$\boldsymbol{R}(p,t1\ V\ t2) = \boldsymbol{R}(p,t1)\ V\ \boldsymbol{R}(p,t2). \qquad (\mathbf{r3}).$$

Ceci implique la définition de l'opérateur V sur l'ensemble des parties non vides de

Θ = {T, ⊥} (i.e P(Θ) qui s'obtient (comme dans [Arb87]) par extension du même opérateur défini sur Θ (voir le ***tableau(18)***) :

$\forall X, Y \subseteq \Theta(X, Y \neq \varnothing), X\ V\ Y = \{x\ V\ y, x \in X, y \in Y\}$.

V	⊥	T
⊥	⊥	T
T	T	T

V sur Θ × Θ

V	{⊥}	{T, ⊥}	{T, ⊥}
{⊥}	{⊥}	{T, ⊥}	{T, ⊥}
{T, ⊥}	{T, ⊥}	{T, ⊥}	{T, ⊥}
{T}	{T}	{T}	{T}

V sur P(Θ) × P(Θ).

Tableau(18) *: Définition de l'opérateur V sur Θ × Θ et son exécution à P(Θ) × P(Θ).*

Un des tests qui permettent de distinguer p de q est t = a;(a;succ V b;succ). Le résultat de l'application du test t au processus q se termine toujours par un succès comme le montre le développement suivant :

$\boldsymbol{R}(q, t) = \boldsymbol{R}(q1, a;succ\ V\ b;succ)$
$\quad \cup \boldsymbol{R}(q2, a;succ\ V\ b;succ)$
$\quad \cup \boldsymbol{R}(q3, a;succ\ V\ b;succ)$ [d'après (**r2**)]

$= \boldsymbol{R}(q1, a;succ)\ V\ \boldsymbol{R}(q1, b;succ)$
$\quad \cup \boldsymbol{R}(q2, a;succ)\ V\ \boldsymbol{R}(q2, b;succ)$
$\quad \cup \boldsymbol{R}(q3, a;succ)\ V\ \boldsymbol{R}(q3, b;succ)$ [d'après (**r3**)]

$= (\{\top\}\ V\ \{\bot\}) \cup (\{\bot\}\ V\ \{\top\}) \cup \{\top\}\ V\ \{\top\})$ [d'après(**r1**)].

$= \{\top\} \cup \{\top\} \cup \{\top\}$ [d'après la définition de V]

$= \{\top\}$ [union d'ensembles].

L'application du même test à p peut échouer. Son résultat $\{\top, \bot\}$ est démontré par le développement suivant :

$\boldsymbol{R}(p, t) = \boldsymbol{R}(p1, a;succ\ V\ b;succ)$
$\quad \cup \boldsymbol{R}(p2, a;succ\ V\ b;succ)$
$\quad \cup \boldsymbol{R}(p3, a;succ\ V\ b;succ)$ [d'après (**r2**)]

$= \boldsymbol{R}(p1, a;succ)\ V\ \boldsymbol{R}(p1, b;succ)$
$\quad \cup \boldsymbol{R}(p2, a;succ)\ V\ \boldsymbol{R}(p2, b;succ)$
$\quad \cup \boldsymbol{R}(p3, a;succ)\ V\ \boldsymbol{R}(p3, b;succ)$ [d'après (**r3**)]

$= (\{\top\}\ V\{\top\}) \cup (\{\bot\}\ V\ \{\bot\}) \cup \{\top\}\ V\ \{\bot\})$ [d'après(**r1**)].

$= \{\top\} \cup \{\bot\}\ \cup \{\top\}$ [d'après la définition de V]

$= \{\top, \bot\}$ [union d'ensembles].

D'où comme attendu : $\boldsymbol{R}(p, t) \not\leq_{must} \boldsymbol{R}(q, t)$.

Nous avons essayé par cette présentation (basée en partie sur [Abr87]), illustrée par les exemples, de mettre en évidence les paramètres qui interviennent dans la définition et la comparaison des équivalences de test, à savoir, le langage de test (ensemble O des tests) et le(s) préordre(s) utilisé(s) ($\leq_{must}$ ou $\leq_{may}$). On peut par exemple montrer qu'une équivalence $\approx$**1** est plus forte que $\approx$**2** (noté $\approx 1 \subseteq \approx 2$) en les mettant dans un même modèle (O1,≤), (O2,≤) tel que $O1 \supseteq O2$.

Le pouvoir discriminatoire des préordres de test n'est pas nécessairement augmenté par l'enrichissement du langage de test. Nous avons montré par *l'**Exemple 2*** précédent que le préordre du **must**-test est effectivement plus discriminant en considérant des tests

arborescents. La proposition suivante montre que le **may**-test est aussi discriminant avec des tests séquentiels ($t \in O$) qu'avec les tests « arborescents » ($t \in O`$).

Proposition : (may-test arborescent = may-test séquentiel)
Pour tout couple de processus p,q, on a : $p \leq^{O'}_{may} q \Leftrightarrow p \leq^{O}_{may} q$ (i.e $\leq^{O'}_{may} = \leq^{O}_{may}$).

Propriétés : $\forall t1, t2 \in O`$:

1. p **may** (t1 V t2) $\Leftrightarrow$ (p **may** t1) ou (p **may** t2)
   p peut réussir (t1 V t2) ssi p peut réussir t1 ou p peut réussir t2.
2. p may a;(t1 V t2) $\Leftrightarrow$ (p **may** a;t1) ou (p **may** a;t2) $\Leftrightarrow$ p **may** (a;t1 V a;t2)
   p peut réussir a ;(t1 V t2) ssi p peut réussir a ;t1 ou p peut réussir a ;t2 (distributivité de V pour le **may**-test).

4.3.1 Vérification des relations de test :

D'un point de vue pratique, il est difficile de vérifier si deux comportements sont distinguables.

- *Exemple 3 :* considérons les deux processus **p** et **q**, de la ***Figure (19)***. Ces deux comportements sont différents par la considération suivante : après l'évolution par la séquence a;b, le processus p ne refuse (jamais) d'exécuter l'action c. ce qui n'est pas le cas pour le processus q.

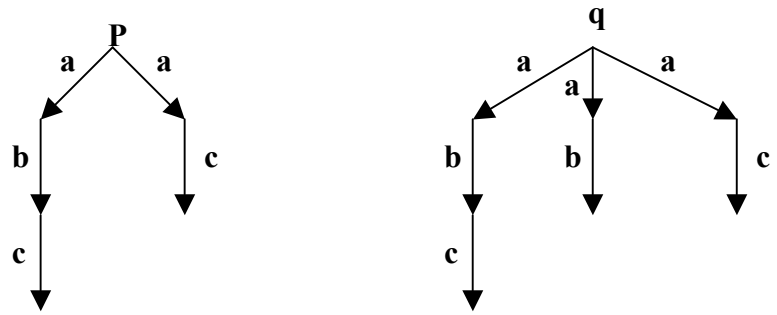


Figure (19): *Exemple*

La traduction de cette considération par le fait que p **must** a;b;c;succ est fausse car ***R***(p, a;b;c;succ) = ***R***(q, a;b;c;succ) = {⊤, ⊥}. C'est à dire d'une part, p ne réussit pas obligatoirement ce test, et d'autre part ce test ne distingue pas p de q puisque son résultat est le même pour les deux.

En considérant le test t = a;(c;succ V b;c;succ), et ne développant selon les règles associées à ***R***, on démontre que ce test permet de distinguer les deux processus p et q :

R(p, t) = {⊥} ≠ {⊤, ⊥} = ***R***(q, t).

Contrairement à la preuve par bissimulation (voir section ***(2.2.1)***), c'est un exercice difficile que de décider si deux processus sont distinguables par le test. Cette difficulté réside dans la recherche du test qui les différencie.

En effet, l'objectif pratique du test est la détection des implémentations non conformes. Ceci est exprimé par la phrase de Dijkstra [Dij72,LS91] : « le *test* peut être utilisé pour montrer la présence de « *bugs* », mais jamais pour montrer leur absence ».

Pour résoudre ce problème De Nicola & Hennessy dans [DENH84] et Phillips dans [Phi87], proposent un système complet de preuve pour les relations de test qu'ils définissent sur CCS. Pour les relations de test considérées par Brinksma, Scollo et Steenbergen dans [BRISS87], nous proposons dans le prochain chapitre, une méthode de vérification qui profite de la commodité de preuve par bissimulation, en considérant un modèle entièrement abstrait appelé *graphe de refus*.

4.3.2 Présentation de modèles de test :

Le langage de test utilisé dans le paragraphe précédent est basé sur la formulation introduite par Abramsky [Abr87] pour définir un langage de test permettant de caractériser l'équivalence d'observation, qui est une variante de l'équivalence observationnelle.

Dans cette section, nous citons les autres théories de tests qui sont celles proposées dans le cadre de CCS [DENH84, Phi87], et celles construites pour le langage LOTOS [Bri88b, BRISS87, Lan90b.

4.3.2.1 Modèles pour LOTOS :

Modèle de Brinksma, Scollo et Steenbergen :

Dans [BRISS87], Brinksma et al considèrent que les systèmes à tester, ainsi que les tests sont les processus *BasicLotos* qui sont finis, et qui se terminent toujours par le processus Lotos *exit.*

L'application d'un test **t** à un processus **p** est conduite par l'opérateur LOTOS de composition parallèle : || . Les deux comportements de ***l'exemple (4.7)*** sont alors écrits selon *la table suivante* :

process p[a,b,c]:exit:=	process q[a,b,c]:exit:=	process t[a,b,c]:exit:=
	a;b;c;exit	a;(
a;b;c;exit	[ ]	c;exit
[ ]	a;b;exit	[ ]
a;c;exit	[ ]	b;c;exit
	a;c;exit	)
endproc	endproc	endproc

Tableau : les processus p et q de *l'exemple (4.7)*, et leur test t, traduits en LOTOS

Le résultat de l'application du test **t** aux processus **p** et **q** est alors donné, respectivement, par **p** || **t** et **q** || **t** dont les comportements (tableau suivant) sont déduits selon les règles de la définition de l'opérateur || .

p || t ≡	q || t ≡
a;b;c;exit	a;b;c;exit
[ ]	[ ]
a;c;exit	a;c;exit
	[ ]
	a;b;stop

Tableau : Résultats de l'application du test t aux processus p et q.

L'application de **t** à **p** conduit toujours à un succès : toute séquence complète (i.e. qui conduit à un arrêt) est terminée par l'action δ (i.e. avec *exit* et non *stop*). Soit

d'après [BRISS87] $\forall \sigma \in Tr(p \parallel t)$, on a $p \parallel t \overset{\sigma}{\Rightarrow} stop$ implique $\sigma = \sigma`\delta$. On en conclut,

d'après la table, que p **must** t. d'autre part, $\neg$(q **must** t) car $q \parallel t \overset{a;b}{\Rightarrow} stop$.

Il existe un autre modèle pour LOTOS appelé : **modèle de Langerak**.

4.3.2.2 Autres modèles :

- Test par refus [Phi87].
- Modèle de De Nicola & Hennessy.

5- CONCLUSION :

Ce chapitre, a introduit les grands principes de la vérification formelle à travers deux des approches les plus représentatives. Nous avons tenté d'en présenter les principaux concepts, leurs différences et d'illustrer ce qui les unit.

En effet, nous avons exposé le problème de l'équivalence entre les systèmes qui peut se ramener à son étude sur le modèle sémantique sous-jacent. Donc, pour prouver l'équivalence entre deux systèmes décrits dans le modèle de spécification, cela revient à montrer que ces deux systèmes peuvent avoir la même représentation sémantique modulo la relation d'équivalence considérée.

Nous avons aussi défini et illustré à travers des schémas et des exemples l'approche logique, ainsi que les équivalences comportementales à savoir, l'équivalence de trace (langage), l'approche par bissimulation (forte et faible) et l'approche test sur laquelle nous avons insistée. Il est important de souligner l'existence d'autres approches à savoir multitraces, step sémantique, …etc.

Une contrainte forte pour qu'une vérification formelle puisse etre conduite est de disposer du graphe de comportement du système à vérifier : que ce graphe soit fini et, dans la pratique qu'il reste de taille raisonnable. Cette contrainte est d'autant plus forte que l'on se heurte très

vite au problème de l'explosion combinatoire du graphe d'états. Les applications les plus simples pouvant donner lieu à des graphes très complexes.

Afin d'établir des réductions aux grands systèmes (graphes) qui permettent d'obtenir des systèmes réduits et qui respectent le même comportement initial, ces approches de vérification sont utilisées au but de faciliter l'analyse des systèmes communicants. Par exemple, la bissimulation forte ne permet pas de faire une grande réduction; car elle traite les actions non observables comme étant des actions observables. Elle est considérée par conséquent, l'approche la moins intéressante.

Par contre, la bissimulation faible élimine quelques actions non observables, ce qui donne des résultats meilleurs que la bissimulation forte.

Nous avons ensuite, présenté une autre classification d'approches de distinction de comportements (représentés par des systèmes de transitions) : l'approche boite blanche et l'approche boite noire, que nous avons appelées respectivement l'approche vérification (et notamment la technique de bissimulation et l'approche logique) et l'approche test (et notamment les tests que doit ou peut réussir une implémentation).

Nous avons motivé, d'une part, l'utilisation de la relation de conformité définie dans le cadre du langage LOTOS pour l'aide à la validation et à la conception des systèmes communicants en utilisant les techniques de description formelle. Et d'une autre part, le choix de *l'approche Test* qui est intéressante ; car elle est la plus utilisée dans la pratique du fait qu'elle ne distingue les comportements que par les traces et les possibilités de blocage ainsi que la considération des systèmes comme des boites noires.

Deux problèmes seront à résoudre pour développer la distinction par le test :

- le premier problème relève de la mise en œuvre d'une procédure de comparaison selon le test, c'est-à-dire une technique de vérification (boite blanche) des relations de test.

- Le second problème concerne l'exploitation de l'approche en générant les tests que doivent réussir les implémentations conformes à une spécification donnée.

Bibliographie

[Abr87] S. Abramsky. Observation equivalence as a testing equivalence. Theoretical Computer Science, 53:225-241, 1987.

[Arn92] A. Arnold. Systèmes de transitions finis et sémantique des processus communicants. Masson, Paris, 1992.

[BerK85] J.A. Berkstra and J.W. Klop. Algebra of communicating processes with abstraction. TCS, 37:77-121, 1985.

[BolB87] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. Computer Networks and ISDN Systems, 14:25-59, 1987.

[BolC89] T. Bolognesi and M. Caneve. Equivalence Verification: Theory, algorithms and a Tool. In Eijk et al. [EVD89], pages 203-326.

[Bol87] T. Bolognesi. Fundamental Results for the Verification of Observational Equivalence. North-Holland, 1987. Elsevier Science Publishers B.V.

[Bri88b] E. Brinksma. A theory for the derivation of tests. In S. Aggrawal and K.Sabani, editors, Protocol Specification Testing and Verification, Volume VIII. Elsevier Science Publishers B.V., North-Holland, 1988. Also in [EVD89].

[BriSS87] E. Brinksma, G. Scollo, and C. Steenbergen. Lotos Specification, their implementations and their tests. In B. Sarikaya and G.V.Bochmann, editors, Protocol Specification Testing and Verification, Volume VI. Elsevier Science Publishers B.V., North-Holland, 1987.

[CCI88] CCITT88. SDL. Recommendation Z.100-Z.104. CCITT, 1988.

[ClePS89] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency Workbench. In J. Sifakis, editors, Automatic Verification Methods for Finite State Systems, number 407 in Lecture Notes in Computer Science, page 11-23, Grenoble-France, June 1989. Springer-Verlag.

[DenH84] R. De Nicola and M.C.B. Hennesy. Testing equivalences for processes. Theoretical Computer Science, 34:83-133, 1984.

[Dij72] E. Dijkstra. Notes on Structures programming. Academic Press, pages 1-82, 1972.

[Dri92] K. Drira. Transformation et Composition de Graphe de Refus: Analyse de la Testabilité. Thèse de Doctorat, Université Paul Sabatier, Toulouse, 1992.

[Fer88] Jean-Claude Fernandez. Un Système de vérification par Réduction de processus Communicants. PHD thesis, Université Joseph Fourier, Grenoble 1, 1988.

[Fer90] J.C. Fernandez. An Implementation of an efficient algorithm for bissimulation equivalence. Science of Computer Programming, 13:219-236, 1989/90.

[FerM90] J.C. Fernandez and L. Mounier. Verifying bissimulation on the fly. In Third International Conference on Formal Description Technique. FORTE '90, Madrid, November 5-8 1990.

[HenM85] M. Hennessy and R. Milner. Algebric Laws For Nondeterminism and Concurrency. Journal of the Association for computing Machinery, 32(1) :137-161, January 1985.

[Hoa85] C.A.R. Hoare. Communicating Sequential Processes. International series in computer science. Prentice-hall, New York, 1985.

[ISO88a] ISO8807. LOTOS, A formal description technique based on the ordering of observational behavior. ISO, November 1988.

[ISO88b] ISO9074. Estelle, a formal description technique based on an extended state transition model. ISO, November 1988.

[Lan90b] R. Langerak. A testing theory for lotos using deadlock detection. In protocol Specification, Testing and Verification, volume IX. International IFIP WG6.1, 1990.

[LarGZ89] K.G. Larsen, J.C. Godskesen, and M. Zeeberg. TAV, tools for automatic verification, user manual. Technical Report 89_19, Department of Mathematics and Computer Science, Aalborg University, 1989.

[Led91a] G. Leduc. A Framework based on implementation relations for implementing LOTOS specifications. Computer Networks & IDDN Systems, 1991.

[Led91b] G. Leduc. Conformance relation, associated equivalence and new canonical tester in LOTOS. In Proc. Of the 11th Symposium on protocol Specification, Testing Specification, Testing and Verification, Stockholm, June 17-20 1991. international IFIP WG 6.1. Voir aussi rapport SART 91/05/13. Université de Liége.

[Mil80] R. Milner. A Calculus of Communicating systems. In Lecture Notes in Computer Science, volume 92. Springer-Verlag, Berlin Heidelberg, 1980.

[Mil89] R. Milner. Communication and Concurrency. International series in computer science. Prentice-Hall, New York, 1989.

[Phi87] I. Phillips. Refusal testing. Theoretical Computer Science, 50:241-284, 1987.

[Pnu86a] Amir Pnueli. Application of temporal logic to the specification and verification of reactive systems : a survey of current trends. Lecture Notes in Computer Science, 224, 1986.

[Pnu86b] Amir Pnueli. Specification and development of reactive systems. In Information Processing, IFIP, pages 845-858. Elsevier Science Publishers B.V (North Holland), 1986.

[Saï96] D.E. Saidouni Sémantique De Maximalité : Application Au Raffinement dans LOTOS .PhD Thesis LAAS-CNRS, 7av. Du colonel roche, 31077 Toulouse Cedex France, 1996.

[Van90] R. J. van Glabbeek. Comparative concurrency semantics and refinement of actions. PhD thesis, Vrije Universiteit Te Amsterdam, 1990.

[Ver89] F. Vernadat. Vérification formelle d'application réparties. Caractérisation logique d'une équivalence de comportement. Thèse de Doctorat, Université Paul Sabatier, Toulouse, 1989.

[Zui90] J. Zuidweg. Concurrent System Verification with Process Algebra. PhD dissertation, PTT Research Leidschendam, Netherlands, 1990.