الجمهورية الجزائرية الديمقراطية الشعبية

République Algérienne Démocratique et Populaire

Ministère de l'enseignement supérieur et de la recherche scientifique

Université de 8 Mai 1945 – Guelma -

Faculté des Mathématiques, d'Informatique et des Sciences de la matière

Département d'Informatique

Mémoire de Fin d'études Master

Filière : Informatique

Option : Informatique Académique

Thème :

Conception et Implémentation d'un Environnement LOTOS ST-Sémantique

Encadré Par :

Mr. Benamira Adel

Présenté par :

Mlle Messaadia Saliha

Mlle Gueroui Hanane

Juin 2013

Remerciements

Nous remercions en premier lieu Dieu le tout

puissant qui nous donné la foi, la force et la patience

de mener ce travail à terme.

Nous exprimons nos remerciements les plus sincères à notre encadreur,

Monsieur Benamira Adel d'avoir eu la gentillesse de nous encadrer,

pour l'assistance qu'il a témoigné envers nous

pour sa disponibilité, ses orientations et ses précieux conseils.

Nous aimons aussi rendre hommage à tous nos professeurs, qui ont

contribué à notre formation durant tout le long de notre cursus.

Nous remercions aussi tous nos ami(e)s et collègues qui ont participé de

près ou de loin à la réalisation de ce mémoire.

A toutes ces personnes, merci du fond de nos cœurs.

Résumé :

Actuellement, les méthodes formelles sont de plus en plus utilisées dans le but d'analyser le comportement des systèmes concurrents distribués. Ces méthodes utilisent des modèles formels de spécification dotés de sémantiques bien définies et des techniques de vérification formelle.

LOTOS (Language Of Temporal Ordering Specification) est une catégorie de formalismes qui permet de faire la description de systèmes parallèles communicants, LOTOS est normalisé par l'ISO et le CCITT afin de permettre la définition formelle des protocoles et des services de télécommunications.

Le but de notre projet était de développer un compilateur pour LOTOS. Nous avons généré un graphe d'état selon les règles de ST-Sémantique via des Noms Statiques, cette dernière c'est un modèle qui représente le non atomicité de l'action par l'exécution séquentielle de deux actions atomiques, qui déterminent respectivement le début et la fin de cette action.

Mots-clés :

Vérification formelle, basic LOTOS, compilation, ST-Sémantique.

Sommaire

Chapitre 1

1.1. Introduction ……7
1.2. Langages FDT ……..…10
1.3. Les modèles du parallélisme ……..…12
1.3.1. Modèles de spécification du parallélisme ……..…..12
1.3.2. Les modèles sémantiques du parallélisme…….……..15
1.3.3. les approches de vérification……..……...16
1.4. Contribution ……16
1.5. Plan du mémoire ……..……16

Chapitre 2

2.1. Processus……19
2.2. Basic LOTOS……..20
2.2.1. Expressions de comportement ……..……..22
2.2.2. Syntaxe formelle de Basic LOTOS……..…….. .21
2.2.3. Opérateurs du langage …….. 24
2.3. ST-Sémantique (State-Transition Semantics) ……..…….……28
2.3.1. ST-Sémantique via des noms statiques …….... 28
2.3.2. Définition de ST-Sémantique Opérationnelle via des Noms Statiques ……..…31
2.4. Conclusion ……..34

Chapitre 3

3.1. Introduction……...…36
3.2. Présentation du projet …...……...……36
3.3. Outil de réalisation du projet……..……..……....36
3.4. Conception globale du projet…… ..37

Sommaire

3.5. Conception du compilateur de Basic LOTOS ……..……37

3.5.1. Les différents composants générés ……..…..41

3.5.2. Compilateur de LOTOS ST-Sémantique.……..…….……..42

3.5.2.1. Compilation LOTOS ...…….……...…42

3.5.2.2. Génération...……...…..50

3.6. Conclusion ……....50

Chapitre 4

4.1. Introduction ……..……..……52

.

4.2. Présentation du système « Le dîner des philosophes » ..……52

4.3. Problème du dîner des philosophes…….……...……53

4.4. Spécification du problème…….……..……..53

4.5. La modélisation du problème …...…….……..…….53

4.5.1. La description informelle du system …..…….……..54

4.5.2. La description formelle du system……...…..……..61

4.6. Conclusion …...…….……61

Conclusion Générale….…….……...……62

Chapitre 1 : Introduction générale

1.1. Introduction :

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.

La complexité de ces systèmes rendant leur analyse classique (prototypage et test) extrêmement ardue, leur fiabilité ne saurait être garantie autrement qu'en employant des méthodes de spécification et vérification formelle, assistées par des outils informatiques performants. Une approche de vérification offrant un bon compromis coût performances est la vérification basée sur les modèles (model-checking), connue aussi sous le nom de la vérification énumérative **[1]**.

Cette approche consiste à traduire le système, préalablement d´écrite dans un langage appropriée **« Description du système»**, vers un modèle **« Sémantique »**, généralement un graphe d'états, sur lequel les propriétés de correction attendues
« Spécification » sont vérifiées au moyen d'algorithmes spécifiques (**Figure1.1).** Bien que limitée aux applications ayant un nombre fini d'états, la présente approche est particulièrement utile dans les premières phases du processus de conception, permettant une détection rapide et économique des erreurs.
La vérification formelle fondée précisément sur les quatre éléments suivants :

- Un langage de description formelle du système, c'est un formalisme du comportement de systèmes de haut niveau, il possède à la fois une syntaxe et une sémantique bien définie (dit formel). Parmi ces langages, nous pouvons citer les réseaux de Pétri [Rei85], les algèbres de processus (**ACP [2]**, **CCS [3] [4] [5]**, **CSP [6]**,…) et les techniques de description formelle (**ESTELLE [7]**, **LOTOS [8] [9]**, **SDL [10]**).

- Un modèle sémantique du parallélisme (modèle de bas niveau), comme son nom l'indique, ce modèle est utilisé pour exprimer la sémantique du parallélisme des langages de description formelle. Nous distinguons deux grandes familles, les modèles d'entrelacement (Arbres de synchronisation **[3]**, STE **[11]**,. . .) et les modèles de non entrelacement (dit vrai parallélisme) (**RDP [12]**, **SEP**, **STA [13] [14] [15]**,…).

- Un langage de spécification, c'est un formalisme dédié à la description de propriétés attendues du système. Nous trouvons dans la littérature deux types de langages de spécification : spécification logique et spécification comportementale.

- La vérification, c'est une relation de satisfaction qui définit la comparaison entre la description du système et sa spécification. Selon le formalisme de la spécification employé.

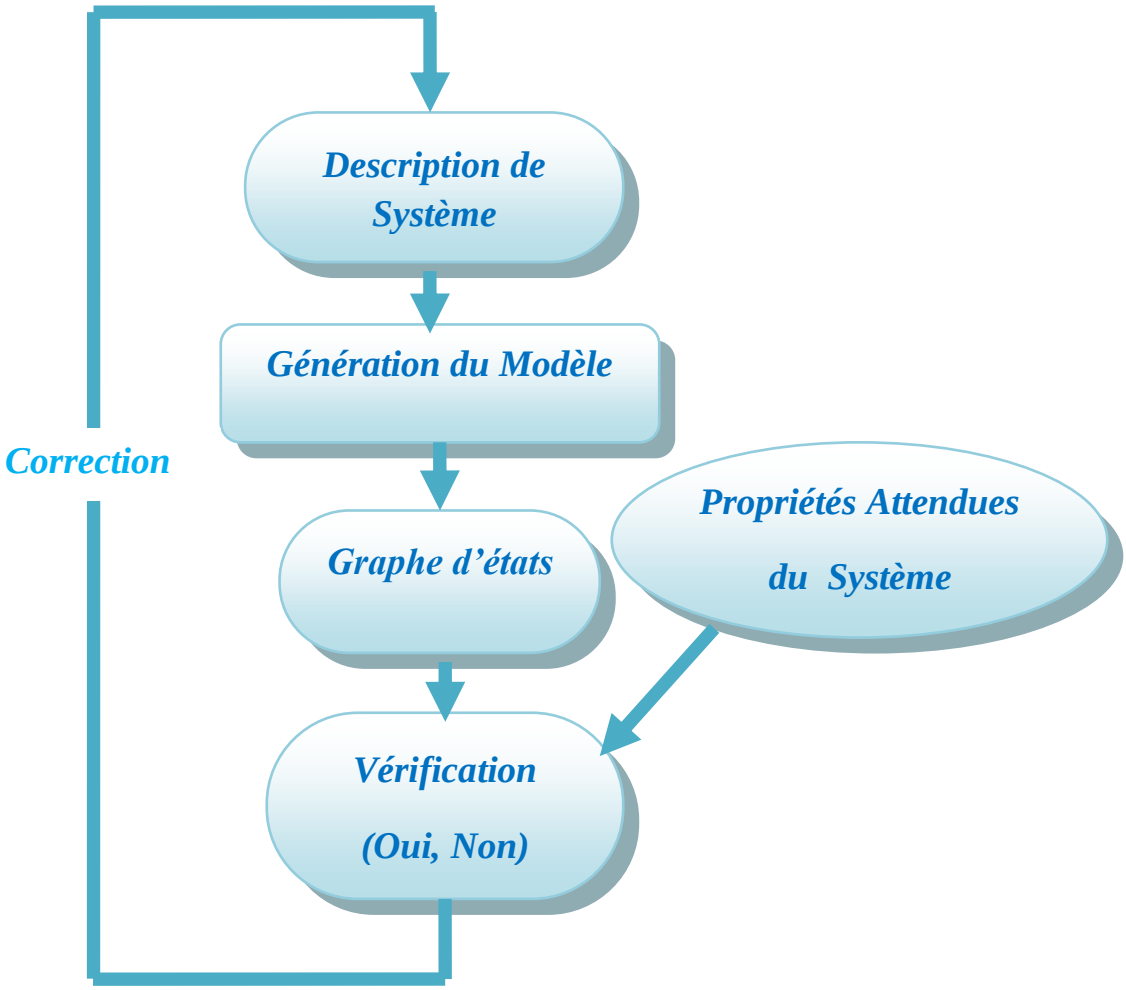

Figure 1.1 : Vérification formelle

Pour permettre à des systèmes informatiques hétérogènes ainsi qu'à des systèmes de commutation téléphonique de communiquer entre eux, deux organismes internationaux, l'**ISO** (Organisation Internationale de Normalisation) et le **CCITT** (Comité Consultatif International pour la Téléphonie et la Télégraphie), ont normalisé le concept d'architecture ouverte **OSI** (Open Systems Interconnections) qui fait désormais l'objet d'une large acceptation. L'architecture **OSI** est basée sur un modèle de référence **[16]** qui établit une décomposition en sept niveaux d'abstraction (couches) pour les aspects communication et application de

systèmes ouverts. Les fonctionnalités de chaque couche (services) sont définies par un ensemble de règles et de formats de dialogue (protocoles).

Le travail des comités de normalisation consiste à éditer des descriptions (normes ou standards) des services et des protocoles, qui constituent la référence à laquelle toute implémentation doit se conformer pour mériter le label **OSI**. De par leur objet même, ces descriptions sont complexes ; ce fait est parfois aggravé par l'existence de nombreuses classes et options **[17]**.

Jusqu'à présent les descriptions étaient données en langage naturel et complétées par des diagrammes de transitions (tables d'états). Il va sans dire que ce mélange de texte informel et de schémas semi-formels conduisait à des définitions volumineuses, imprécises, ambiguës **[18] [19]** et entachées d'erreurs **[20]**.

C'est pourquoi de meilleurs formalismes de description ont été recherchés. Ce travail, qui a duré près de huit années, a abouti à la conception de trois langages, désignés sous le nom générique de **FDT** (Formal Description Techniques).

Les définitions de ces langages sont elles-mêmes normalisées, soit par l'**ISO**, soit par le **CCITT**.

1.2. Langages FDT :

Ils sont principalement destinés à spécifier des propriétés fonctionnelles, c'est-à-dire la description qualitative du comportement des systèmes vis-à-vis de leurs utilisateurs, par opposition aux propriétés opérationnelles, qui expriment des caractéristiques quantitatives comme le temps de réponse, les performances, . . .

De par leur origine, les langages **FDT** ont été conçus pour permettre la spécification des systèmes distribués. C'est pourquoi il s'agit de langages déclaratifs plutôt qu'impératifs. Les langages **FDT** sont néanmoins exécutables ‘ contrairement à d'autres formalismes tels que les logiques temporelles’ et peuvent donc servir également à la programmation, sinon des systèmes, du moins de prototypes.

Les langages **FDT** permettent de décrire la structure interne des systèmes : chaque système est décomposé en sous-systèmes qui sont exécutes en parallèle, se synchronisent et communiquent par envois de messages. Pour les présenter, on distingue le contrôle, qui

spécifié la façon dont un système réagit aux stimuli qu'il reçoit de son environnement, et les données qui concernent les informations mémorisées par le système et les messages qu'il émet ou reçoit **[17]**.

Les trois langages **FDT** différent par les concepts qu'ils mettent en œuvre :

- **ESTELLE** (Extended Finite State Machine Language), développé de 1980 à 1988, fait l'objet de la norme **ISO** 9074 (définition formelle : **[21]**, présentation générale : **[22]**). En Estelle un système est composé d'instances de modules, organisées hiérarchiquement et exécutées simultanément de manière asynchrone. Le contrôle de chaque module est décrit par un automate d'états finis; les données sont décrites en **PASCAL**. Les modules communiquent par des points d'interaction, connectés entre eux par des canaux qui autorisent des échanges bidirectionnels de messages. Les messages sont rangés dans des files d'attente non bornées. La création et la destruction dynamiques de canaux et d'instances de modules sont possibles. Enfin, Estelle permet d'exprimer des contraintes temporelles de type délai.

- **SDL** (Specification and Description Language), développé dès 1974, a connu plusieurs versions successives dont la plus récente, datant de 1988, est standardisée par la recommandation **Z.100** du **CCITT** (définition formelle : **[23]**, présentation générale : **[24]**). 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 abstraits algébriques : un type comprend des domaines de valeurs, des Operations sur ces valeurs et des équations qui définissent la sémantique de ces opérations. De nouveaux types peuvent être créés à partir de types existants par enrichissement, renommage, paramétrisation, . . . Enfin, **SDL** possède deux syntaxes : la forme texte, analogue à un langage de programmation classique, et la forme graphique, dont les éléments de base sont des boites connectées par des flèches.

- **LOTOS** (Language Of Temporal Ordering Specification), conçu entre 1980 et 1988, est défini par la norme ISO 8807 (définition formelle : **[25]**, présentation générale : **[26]**). Il existe en Lotos une nette séparation entre contrôle et données. Comme en **SDL**, les données sont décrites au moyen de types abstraits algébriques **[27] [28]**. Contrairement à **ESTELLE** et **SDL**, le contrôle en Lotos s'inspire, non pas des automates communicants, mais des algèbres de processus dont les plus connues sont **CCS [29]** et **CSP [30]**. Lotos permet de décrire l'ordonnancement temporel d'un ensemble d'actions, tel qu'il serait perçu par un observateur extérieur. Il est possible de spécifier des comportements complexes en combinant des actions élémentaires au moyen d'opérateurs tels que la composition séquentielle, le choix non-déterministe, la composition parallèle avec différents degrés de liberté allant du synchronisme jusqu'à l'asynchronisme . . . Les communications s'effectuent uniquement au moyen de rendez-vous symétriques, qui peuvent être binaires ou n-aires et à partir desquels d'autres mécanismes de synchronisation peuvent être construits.

1.3. Les 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 **[31]** :

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

1.3.1. 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, . . .) [29, 6]** et les techniques de description formelle **(ESTELLE, LOTOS, SDL) [25, 21].**

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
- Les spécifications formelles sont basées sur des représentations mathématiques sont 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 **[31]**.

Les réseaux de Pétri :

Les réseaux de Pétri ont été développés en tant que formalisme opérationnel pour la spécification des systèmes concurrents. Ils donnent une représentation dynamique d'un état du système par l'utilisation de jetons. Les réseaux de Pétri ont été utilisés avec succès pour modéliser une variété de systèmes industriels. Ils peuvent spécifier différents composants du système modélisé dans de différentes étapes d'exécution ou dans de différents instants de temps, ceci rend ce formalisme particulièrement attractif pour modéliser les systèmes embarqués interagissant avec leur environnement extérieur.

Plus précisément, un réseau de Pétri, ou un réseau de Pétri **place / transition**, consiste en quatre composants de base : **places, transitions, arcs et jetons** (voir **Figure 1.2**). Une place est un état dans lequel le système spécifié (ou une de ses parties) peut se situer. Les arcs relient les transitions aux places et les places aux transitions. Si un arc se dirige d'une place à une transition, la place est une entrée (input) de cette transition et l'arc est un arc d'entrée vers cette transition. Si un arc se dirige d'une transition à une place, la place est une sortie (output) de cette transition et l'arc est un arc de sortie à partir de cette transition. Plusieurs arcs peuvent exister d'une place à une transition. Une place peut être vide et peut contenir un ou plusieurs jetons. L'état d'un réseau de Pétri, connu sous le nom de marquage, est défini par le nombre de jetons dans chaque place.

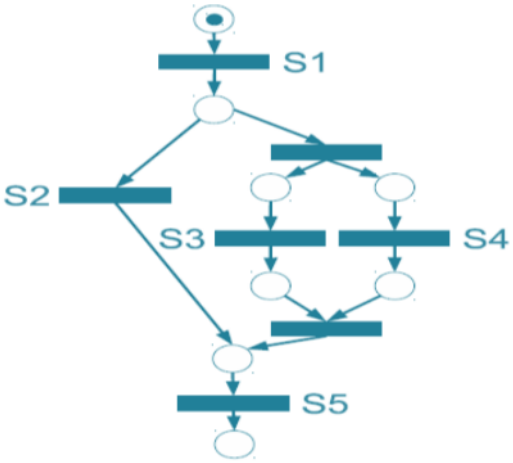

Figure 1.2 : Exemple de Réseaux de Pétri

Algèbres de Processus :

Les algèbres de processus sont des formalismes de description formelle pour la spécification de systèmes logiciels, en particulier pour les systèmes concurrentiels. Elles fournissent des outils pour la description de haut niveau des interactions et des synchronisations entre les processus. Plusieurs algèbres de processus ont été décrites.

Une algèbre de processus est un langage concis pour la description des étapes d'exécution de processus. Ce langage est constitué d'un ensemble d'opérateurs et de règles syntaxiques pour spécifier un processus utilisant de composants simples ou atomiques.

Les algèbres de processus utilisent la notion d'équivalence pour montrer que deux processus ont le même comportement. Plusieurs algèbres de processus ont été utilisées pour spécifier et analyser des processus concurrents qui communiquent entre eux. Nous pouvons citer **CSP** (Communicating Sequential Processes) développé par Hoare **[6]**, **CCS** (Calculus of Communicating Systems) développé par Milner dans les années 1970-1980. Il a développé ultérieurement ce concept dans le π-calculus **[5]**, **ACP** (Algebra of Communicating Processes) de Bergstra et Klop **[2]** et **LOTOS** (Language Of Temporal Ordering Specifications) a été développé dans les années 1980 **[8, 9].**

Ces algèbres de processus citées précédemment sont des algèbres atemporelles du fait qu'elles ne permettent que de raisonner sur l'ordre d'exécution des évènements et étapes d'exécution.

Les algèbres de processus utilisent des structures simples telles que l'émission ou la réception de message par un processus, le séquencement de processus, le choix non déterministe ou encore l'exécution parallèle de processus.

1.3.2. Les modèles sémantiques du parallélisme :

Ces modèles peuvent être considérés comme des extensions des modèles de spécification du parallélisme par l'introduction de mécanismes et d'opérateurs permettant l'expression de contraintes temporelles auxquelles sont soumis les comportements des systèmes critiques. Parmi ces modèles nous citons les réseaux de Pétri temporels, les techniques de descriptions formelles.

Les modèles sémantiques temps réel sont utilisés pour exprimer la sémantique des modèles de spécification sus-indiqués. Deux familles de modèles sémantiques sont à souligner :

- **Les modèles dits d'entrelacement :**

Ils se caractérisent par l'interprétation de l'exécution parallèle de deux actions par leurs exécutions alternées. Cette abstraction peut être acceptable dans le cas où les actions sont atomiques. Cependant, une fois le système est exprimé dans ces modèles, il devient impossible de retrouver toute information relative à l'exécution des actions parallèles.

- **Les modèles dits du vrai parallélisme :**

Ils se caractérisent par la prise en compte de l'hypothèse de non atomicité des actions, et ceci dès les premières étapes de conception. Les retombés de cette approche est double ; elle permet la définition de méthodes de conception formelle basées sur le raffinement d'action ainsi que l'attribution de durées aux actions, ce qui facilite la spécification de système réels.

- **ST-Sémantique (State Transition Semantics) :**

ST-Sémantique c'est un modèle qui représente le non atomicité de l'action par l'exécution séquentielle de deux actions atomiques, qui déterminent respectivement le début et la fin de cette action.

1.3.3. les approches de 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 être infini **[32]**.

1.4. Contribution :

Nos contributions peuvent se résumer en :

La création d'un compilateur pour générer le comportement (via la **ST-Sémantique)** pour la vérification de la spécification textuelle d'un système décrit en **LOTOS**.

1.5. Plan du mémoire :

Le mémoire est organisé de la manière suivante :

- **Le Chapitre 1 :** représente une introduction générale qui permet d'exposer la vérification formelle, puis l'architecture **OSI** et leur techniques de description formelle, en suit nous abordons par ailleurs les langages formels de spécification du parallélisme et les modèles sémantiques du parallélisme et en fin les méthodes de vérification.

- **Le Chapitre 2 :** est consacré à une présentation générale de La Technique de Description Formelle **LOTOS** et nous intéressant par ailleurs de toutes les notions et les opérations du **Basic LOTOS**.

- **Le Chapitre 3 :** va être consacré à la partie pratique de notre travail. Nous allons voir dans ce chapitre les techniques qui ont été utilisées pour l'implémentation du compilateur.

- **Le Chapitre 4 :** représente l'étude du problème classique du dîner des philosophes, où on verra les démarches à suivre pour le modéliser dans notre logiciel Ce chapitre est très important, car il met en valeur le travail réalisé.

Chapitre 2 : La Technique de Description Formelle LOTOS et ST-Sémantique

Nous présentons dans ce chapitre, les principes généraux de la Technique de Description Formelle **LOTOS**.

2.1. 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.

L'environnement d'un processus, dans un système, est formé par l'ensemble des processus de système avec lesquels processus interagit, plus un processus observateur non-spécifié, peut être un humain, qui est supposé être toujours prêts 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 exécute une action observable signifie une interaction de processus avec son environnement. La représentation la plus abstraite du processus, capable d'interagir avec son environnement via des portes, est illustrée par la boite noire dans la **Figure2.1.**

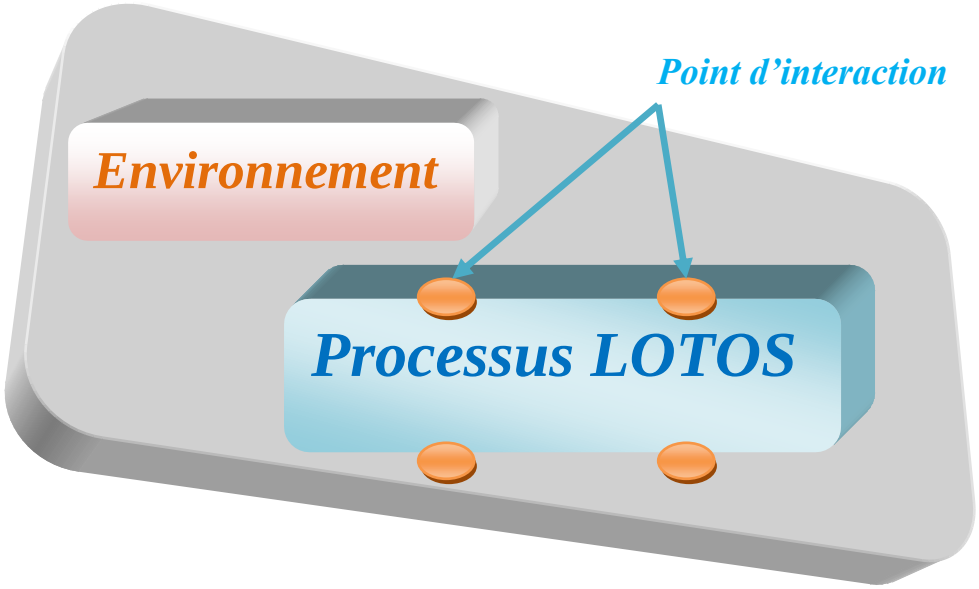

Figure 2.1 : Représentation abstraite d'un Processus LOTOS

La définition du processus spécifiera donc son comportement, en définissant les séquences d'actions observables qui peuvent se présenter (être observées) aux quatre 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 **Figure 2.1**.

2.2. 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 interprocessus.

2.2.1. 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 **Figure 2.2**, qui complète la définition du processus **Max3** commentée précédemment.

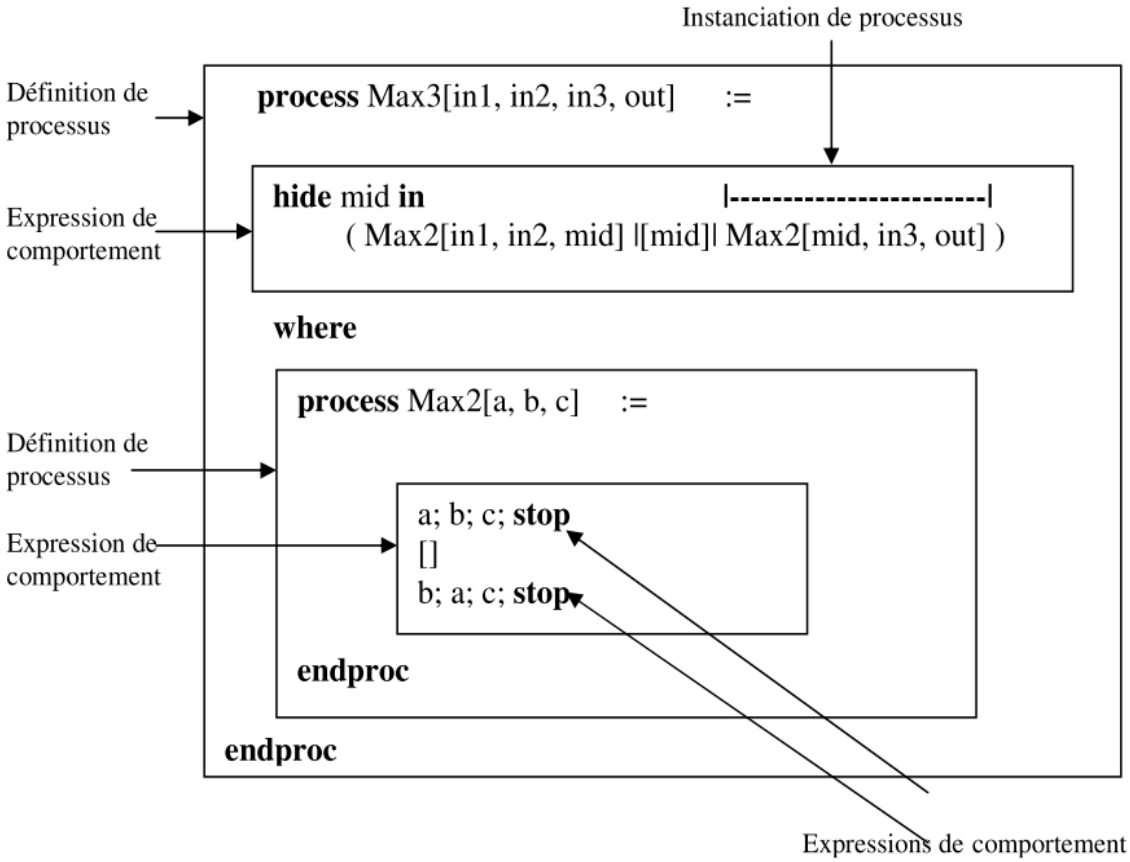

Figure2.2 : Définition du processus Max3

Action interne i (action τ - tau) :

Les comportements peuvent avoir des actions internes, ces actions dénotent un comportement interne de la machine sans vouloir entrer dans les détails, Détails laissés à des raffinements successifs dans la conception ou dans l'implantation,

L'action interne peut être spécifiée directement ou indirectement.

- **Utilisation directe:** On utilise τ comme une action dans le processus
- **Utilisation indirecte :** On utilise un opérateur de masquage des actions

L'action interne ne synchronise pas avec l'environnement (elle est invisible à l'extérieur).

2.2.2. Syntaxe formelle de Basic LOTOS :

Soit **PN** l'ensemble des variables de processus parcouru par **X** et soit **G** l'ensemble des noms de portes définies par l'utilisateur (ensemble des actions observables) parcouru par **a, b,....**Une porte observable particulière $\delta \notin$**G** est utilisée pour noti.er la terminaison avec succès des processus. **L** dénote tout sous ensemble de **G**, l'action interne est désignée par**i**. **B** parcouru par **E, F,...**dénote l'ensemble des expressions de comportement dont la syntaxe est :

E ::= stop Arrêt définitif

|exit Terminaison avec succès

| a ; E Préfixe : une action précède des comportements

|i ; E Préfixage par l'action interne i

|E[]E Choix

|E|[L]|E Parallélisme avec synchro partielle

|hideLinE Masquage

|E >> E Composition séquentielle

|E[> E Interruption

Chapitre 2 : La Technique de Description Formelle LOTOS et ST-Sémantique

Afin de donner une approche intuitive des opérateurs **LOTOS**, nous associons une assertion informelle à chaque configuration de la **Figure 2.3** dans chaque schéma un processus **p [a, b, c, d]** est composé de deux processus **P1 [a, b]** et **P2 [c, d]** (boites grisées) mis en relation avec différents opérateur **LOTOS** et offrant chacun une synchronisation sur les porte a, b, c ou d à leur environnement :

- « **P1 [a, b] ou P2 [c, d]** suivant l'action qui produira en premier » (**Choix**)
- « **P1 [a, b] et P2 [c, d]** s'exécute en parallèle sans synchronisation » (**Parallélisme**)
- « **P1 [a, b] et P2 [c, d]** s'exécutent en parallèle sans synchronisation sauf la porte b sur laquelle les deux processus doivent se synchronise » (**Synchronisation**)
- « **P1 [a, b] et P2 [c, d]** s'exécutent en parallèle sans synchronisation sauf la porte b sur laquelle les deux processus doivent se synchronise, de plus b n'est plus disponible pour une synchronisation potentielle avec les processus appartenant à l'environnement du processus P » (**Intériorisation**)
- « d'bord **P1 [a, b]** suivis de **P2 [c, d]** quand **P1** sera terminer » (**Séquence**)
- « **P1 [a, b]** peut être interrompu à tout instant par **P2 [c, d]** avant sa terminaison » (**Préemption**)

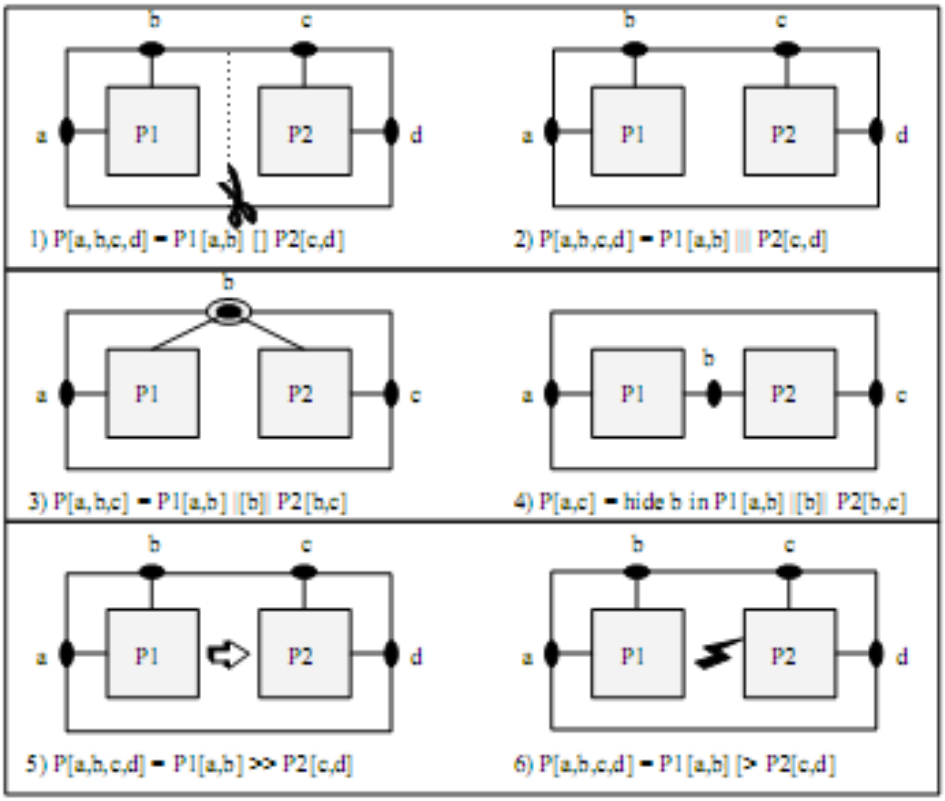

Figure2.3 : Illustration des opérateurs de Basic-LOTOS

2.2.3. Opérateurs du langage :

Soient **E** et **F** deux expressions de comportement et soit **L** ⊆ **G** un sous ensemble de portes

- **Inaction « stop » :**

Est une expression représente un processus inactif (un processus qui ne fait aucun interaction).

- **Terminaison avec succès « exit » :**

L'opérateur « **stop** » permet de spécifier explicitement l'arrêt d'un comportement. Mais « **stop** » peut aussi apparaitre 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 succée s'exprime par le franchissement d'une transition « **δ** ».De fait « **exit** » est équivalent à un rendez-vous sur la porte « **δ** » : **δ ; stop.**

- **Opérateur de préfixage d'action « ; » :**

Soit **E** une expression de comportement donnée, On distingue deux sorte de préfixage d'actions :

- **a ; E : aveca ∈ G**représente le comportement d'un processus qui interagit sur le porte a et qui se comporte ensuite comme **E**
- **i ; E :** représente le préfixage par l'action interne **i**.

- **Opérateur « [ ] » :**

L'opérateur « **[ ]** » parmi 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).

- **Opérateur « hide » :**

Basic LOTOS possède un opérateur qui permet de cacher certaines portes d'un comportement. Si $G_0,\ldots, G_n$ sont des porte et B_0un comportement, La construction suivante : **hide** $G_0,\ldots, G_n$**in** B_0.

Dénote le comportementB_0dont les portes $G_0,\ldots, G_n$ sont renommées en « **i** ». Les portes $G_0,\ldots, G_n$ ne sont pas visible que dans B_0.elles deviennent inaptes à la synchronisation pour l'environnement de B_0avec lequel elles interfèrent plus. Vu de l'extérieur ces instructions sont

invisible (puisqu'étiquetées « **i** ») et ont lieu spontanément sans aucune participation de l'environnement de B_0 : le rendez-vous sur une porte cachée à n'est jamais bloquant.

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

Les opérateurs qui ont été présentés jusqu'ici sont strictement séquentiels, **Basic LOTOS** comprend aussi des opérateurs parallèles, si B_1et B_2 sont deux comportements et $G_0,\ldots, G_n$ une liste de portes la construction suivante :

$$B_1 \,|[G_0,\ldots, G_n]|\, B_2 .$$

Dénote le comportement qui exécute B_1et B_2en parallèle .la synchronisation et la communication entre les opérandes B_1et B_2s'effectuent uniquement par rendez-vous sur les portes de l'ensemble $\{G_0,\ldots, G_n\}$.

Lorsqu'un des opérandes veut effectuer une transition étiquetée par une porte **G**de $\{G_0,\ldots, G_n\}$, 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 reprenant chacun leur exécution.

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

- **Opérateur « >> » :**

L'opérateur de préfixage**« ; »** et asymétrique : son opérande gauche est une porte alors que son opérande droite est un comportement. **Basic LOTOS** possède un autre operateur de composition séquentielle dont les deux opérandes sont des comportements. Si $\mathbf{B_1}$et $\mathbf{B_2}$ sont deux comportements, la construction suivante :

$$\mathbf{B_1} >> \mathbf{B_2}.$$

Dénote le comportement qui exécute séquentiellement $\mathbf{B_1}$puis $\mathbf{B_2}$intuitivement, la composition séquentielle est modélisée en **Basic LOTOS** comme un cas particulier de composition parallèle. Les comportements $\mathbf{B_1}$ et $\mathbf{B_2}$sont exécutés en parallèle mais $\mathbf{B_2}$ne peut pas commencer avant que $\mathbf{B_1}$ ne se soit terminé. L'attente de $\mathbf{B_2}$ s'obtient par un rendez-vous sur la porte « **δ** », rendez|-vous qui devient possible dès que $\mathbf{B_1}$exécute « **exit** ». Si $\mathbf{B_1}$ boucle indéfiniment ou il se bloque sans atteindre « **exit** » $\mathbf{B_2}$ ne sera jamais exécuté. L'opérateur **« >> »** est souvent appelé **enabling opérator** ou **enable** puisque la terminaison avec succès de $\mathbf{B_1}$autoriser l'exécution de $\mathbf{B_2}$.

- **Opérateur « [> » :**

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

Si $\mathbf{B_1}$et$\mathbf{B_2}$ sont deux comportement, la construction suivante : $\mathbf{B_1}$ **[>** $\mathbf{B_2}$.

Dénote le comportement qui exécute $\mathbf{B_1}$mais qui peut abandonner à tout instant l'exécution de $\mathbf{B_1}$ pour commencer celle de $\mathbf{B_2}$intuitivement, il s'agit d'un mécanisme d'interruption avec terminaison: $\mathbf{B_1}$ joue le rôle d'un traitement normal et $\mathbf{B_2}$ celui d'un traitement d'exception. Initialement, $\mathbf{B_1}$ est exécuté seul mais, tant qu'il n'a pas effectué de transition « **δ** », son exécution peut être interrompue profit de celle de $\mathbf{B_2}$ si $\mathbf{B_1}$ se bloque avant d'atteindre une instruction « **exit** » alors $\mathbf{B_2}$ est inévitablement exécuté. Dans le cas contraire $\mathbf{B_2}$ peut très bien ne jamais être exécuté. L'opérateur **« [> »** est souvent appelé

displing opérator ou **disable** puisque la terminaison avec succès de $\mathbf{B_1}$ interdit l'exécution de $\mathbf{B_2}$.

Propriétés algébriques des expressions de comportement :

- Commutativité du choix

  A [] B = B [] A

- Commutativité de la composition parallèle

  A || B = B || A

- Zéro absorption

  A [] STOP = STOP [] A = A

  A || STOP = STOP || A = STOP

- Associativité

  A [] (B [] C) = (A [] B) [] C

  A || (B || C) = (A || B) || C

- Non-distributivité de ''**;**'' par rapport à []

a; (b ; STOP [] c ; STOP) ≠ a ; b ; STOP [] a ; c ; STOP

même si **''a = τ''**

2.3. ST-Sémantique (State-Transition Semantics) :

ST-Sémantique c'est un modèle qui représente le non atomicité de l'action par l'exécution séquentielle de deux actions atomiques, qui déterminent respectivement le début et la fin de cette action **[32]**.

2.3.1. ST-Sémantique Via des Nom Statiques :

La technique de nom statique est basé sur l'idée d'statiquement attribuer un nom à chaque action se produisant dans un processus **P** sur la base de sa position syntaxique dans les noms de **P.** doit être affecté à des actions de telle sorte que nous sommes assurés que les deux mesures du même type a qui peuvent être exécutées simultanément (Il peut exister un état

dans lequel ils sont tous deux commencé mais pas terminé encore) toujours des noms différents. De cette façon, quand une action **a** donné un nom se termine, il est toujours évident que l'action d'un (parmi ceux en cours d'exécution) se termine et nous préserver la relation correcte entre les démarrages d'action terminaisons d'action et tel que requis par **ST-Sémantique**. Nous soulignent que, pour le but de définir une **ST-Sémantique**, il n'est pas nécessaire d'identifier toutes les actions d'un terme comme une action différente.

- ✓ Dans le cas ou deux actions consécutives d'une durée d'une séquence exécutée par le même processus (de la forme $\mathbf{a}.\mathbf{a}.\mathbf{P}'$) ne peut être exécuté simultanément, mais sont causalement liée. Ainsi, les deux actions ne peuvent se chevaucher au cours de leur exécution et une fois action a démarré il doit se terminer avant que l'autre peut commencer. Par conséquent, même si les deux actions ne se distinguent pas par le mécanisme d'identification, le cas d'un blocage d'action est encore unique correspondant à l'événement de son commencer (il n'y a pas d'ambiguïté sur lequel l'action se termine). Aussi alternatif actions d'une durée d'un exécutable par le même procédé séquentiel (avec le formulaire $\mathbf{a}.\mathbf{P}' + \mathbf{a}.\mathbf{P}''$**)** peuvent être identifiés de la même manière depuis la signature de l'un d'eux exclut l'exécution de l'autre.

La méthode que nous adoptons pour l'attribution de noms aux actions est basée sur destinations. Le nom de l'action est donné par son emplacement défini comme la syntaxe position (gauche ou droit) par rapport aux opérateurs parallèles de la durée **P** spécifiant le système. Par conséquent, dans le modèle sémantique d'un processus **P BL**, l'événement de déclenchement d'une action observable **a** est représenté par une transition d'une étiquette par $\mathbf{a}^{+}_{@}$, Où **@** est l'emplacement de l'actiongauche (**l**) ou bien droite (**r)**, tout en cas de cessation de l'action est représentée par une transition marquée par une $\mathbf{a}^{-}_{@}$, Où l'emplacement **@** est un "identifiant" qui détermine de façon unique dont l'action **a** est fin. A conséquence de l'utilisation emplacements d'action que les noms statiques facilement faire face à la récursivité et puisque les emplacements d'action sont "réutilisés" par l'action séquentielle du même processus, nous pouvons obtenir des finis des modèles sémantiques aussi dans la présence de la récursivité, voir la **Figure 2.4**.

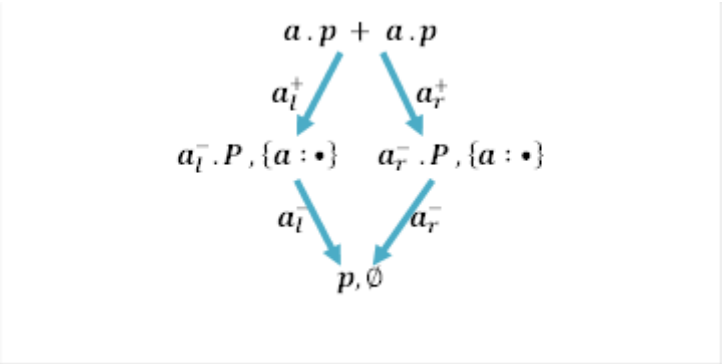

Figure 2.4 : La méthode de l'attribution de noms aux actions

✓ En particulier, puisque les actions visibles a peut être obtenu par la synchronisation, nous avons besoin (comme nous le verrons) pour enregistrer les emplacements des actions synchrones au titre d'exécution (commencé mais pas terminé encore) et il est donc nécessaire de représenter états du système sous forme de paires $(\mathbf{P}, \mathbf{ex})$ composée d'un terme P et un ensemble $\mathbf{ex}$ d'emplacements des actions en exécution.

✓ En ce qui concerne le traitement des actions τ , dans notre approche nous représentons l'exécution d'une action $\boldsymbol{\tau}$ La combinaison des mesures que les deux «événements invisibles" $\mathbf{i}$ de début d'action et la fin des mesures. Tous les deux les événements sont représentés par $\mathbf{i}$ car ils sont invisibles et nous adoptons une notion de bisimulation faible **ST** qui fait abstraction de i transitions (au lieu d'abstraire à partir de transitions, comme d'habitude). D'autre part, le fractionnement de l'actions $\boldsymbol{\tau}$, comme nous le faisons pour les actions visibles, adhère à l'intuition que la sémantique de τ devrait être isomorphe à celle d'un $\mathrm{a} / \{\mathbf{a}\}$, où les deux événements alternative $\mathbf{a}^+$ et $\mathbf{a}^-$ sont confidentiels.

✓ Le problème de l'abstraction nom provient du fait que le rôle de noms statiques est juste de distinguer actions du même type exécutées en parallèle par un seul processus P, mais pas à distinguer les actions du même type réalisé par différents processus $\mathbf{P}$ et $\mathbf{P}'$. Comme le

mécanisme d'attribution des noms aux actions est basé sur les positions syntaxiques, deux termes P et P' qui se comportent de manière équivalente peut obtenir les différents noms des actions correspondantes. Par exemple, dans les deux termes $\mathbf{a} ||_{\emptyset} \mathbf{b}$ et $\mathbf{b} ||_{\emptyset} \mathbf{a}$, un qui devrait être clairement équivalent, l'action a d'un $\mathbf{a} ||_{\emptyset} \mathbf{b}$ des récupère le nom de $\mathbf{a_l}$ (où **l** signifie "gauche"), tandis que l'action **a** d'un $\mathbf{b}||_{\emptyset} \mathbf{a}$, un nom obtient $\mathbf{a_r}$ (où **r** représente " Droit") et de même pour les actions **b**. Pour mettre en œuvre l'abstraction denoms de l'action la définition de la bisimulation faible **ST** doit en quelque sorte associer les noms des actions utilisées par un processus avec des noms de correspondants actions utilisés par l'autre, voir **Figure 2.5.**

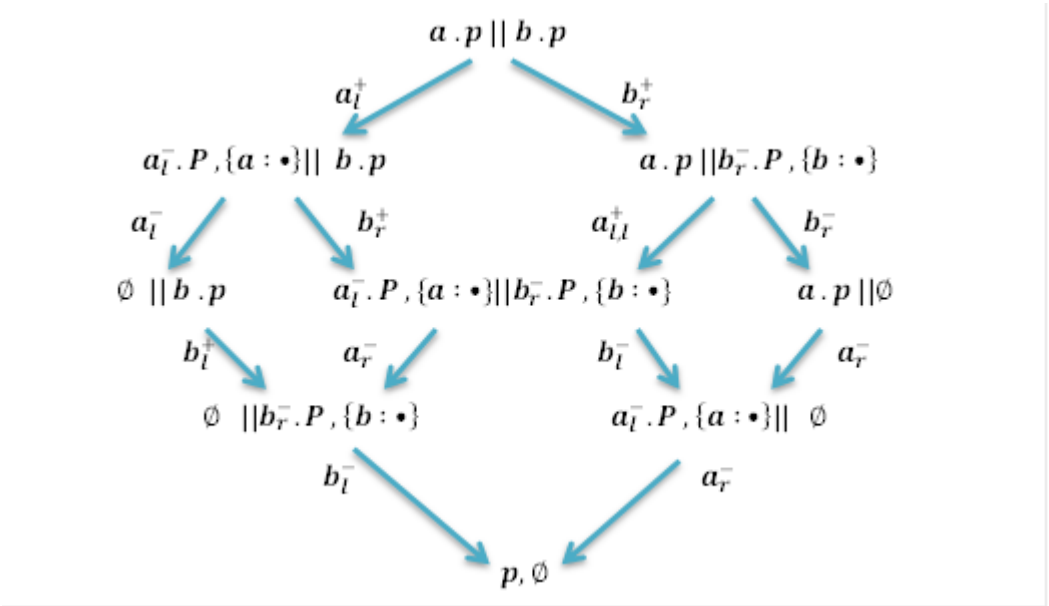

Figure 2.5 : La méthode de la bisimulation faible

2.3.2. Définition de ST-Sémantique Opérationnelle via des Noms Statiques :

Maintenant, nous montrons comment la technique de nom statique peut être exploitée pour donner opérationnelle **ST-Sémantique** de notre langage de base "**BL**. Tout d'abord, nous avons besoin d'étendre la syntaxe **BL** pour représenter les actions engagées, mais encore être résilié dans les conditions de l'état modèles sémantiques. Le préfixe d'un nouveau $\mathbf{a}^-$

représente une action visible **a** qui doit encore pour terminer et le nouveau préfixe**i**représente une action silencieuse **τ**qui doit encore terminer. Par conséquent, la metavariable pour la préfixe action est η , Qui se situeau-dessus de Loi $\mathbf{Act} \cup \{\mathbf{a}^{-} | \mathbf{a} \in \mathbf{A}\} \cup \{\mathbf{i}\}$ **[33]**.

Le $\mathbf{BL_{sn}}$ ensemble de termes d'état de **BL**(où le sn représente le nom statique technique) est engendré par la syntaxe suivante

$$\mathbf{T} ::= \mathbf{0} \mid \mathbf{X} \mid \eta.\mathbf{P} \mid \mathbf{P} + \mathbf{P} \mid \mathbf{T} ||_{\mathbf{S}} \mathbf{T} \mid \mathbf{T} / \mathbf{L} \mid \mathbf{recX.P}$$

Où $\mathbf{P} \in \mathbf{BL}$.

La sémantique des opérateurs est définie de manière précise par des **règles et axiomes d'inférence** Disant:

- étant donnée une expression de comportement et une **action**
- transforme l'expression de comportement dans une autre expression de comportement.

Exécution des règles d'inférence

- L'exécution d'une spécification transforme la spécification en utilisant les règles d'inférence,
- La spécification courante (représentant l'état global courant) peut être transformée en utilisant n'importe quelle règle applicable,
- L'arbre qui représente toutes les transformations possibles est le système de transition étiqueté du système,
- Il est aussi l'arbre d'accessibilité montrant toutes les transitions possibles d'état du système spécifié.

Dans le **Figure 2.6**, nous décrivons les règles de**ST-Sémantique** de la Technique de Nom Statique **[33]**.

$$(a.1)\quad \langle a\,.\,P,\emptyset\rangle \xrightarrow{a^{+}_{\bullet}} \langle a^{-}\,.\,P,\{a:\bullet\}\rangle \qquad (a.3)\quad \langle \tau\,.\,P,\emptyset\rangle \xrightarrow{i} \langle i\,.\,P,\emptyset\rangle$$

$$(a.2)\quad \langle a^{-}\,.\,P,\{a:\bullet\}\rangle \xrightarrow{a^{-}_{\bullet}} \langle P,\emptyset\rangle \qquad (a.4)\quad \langle i\,.\,P,\emptyset\rangle \xrightarrow{i} \langle P,\emptyset\rangle$$

$$(b.1)\quad \frac{\langle P,\emptyset\rangle \xrightarrow{\theta} \langle T,ex\rangle}{\langle P+Q,\ \emptyset\rangle \xrightarrow{\theta} \langle T,ex\rangle} \quad \frac{\langle Q,\emptyset\rangle \xrightarrow{\theta} \langle U,ex\rangle}{\langle P+Q,\ \emptyset\rangle \xrightarrow{\theta} \langle U,ex\rangle}$$

$$(c.1)\quad \frac{\langle T,l(ex)\rangle \xrightarrow{a^{\pm}_{l@}} \langle T',ex'\rangle}{\langle T||_S U,ex\rangle \xrightarrow{a^{+}_{l@}} \langle T'||_S U,ex\cup\ \{a:l@\}\rangle}\ a\notin S \qquad \frac{\langle T,l(ex)\rangle \xrightarrow{i} \langle T',ex'\rangle}{\langle T||_S U,ex\rangle \xrightarrow{i} \langle T'||_S U,ex\rangle}$$

$$(c.2)\quad \frac{\langle T,l(ex)\rangle \xrightarrow{a^{-}_{@}} \langle T',ex'\rangle a:l@\ \in ex}{\langle T||_S U,ex\rangle \xrightarrow{a^{-}_{l@}} \langle T'||_S U,ex-\{a:l@\}\rangle}\ a\notin S$$

$$(c.3)\quad \frac{\langle U,r(ex)\rangle \xrightarrow{a^{\pm}_{@}} \langle U',ex'\rangle}{\langle T||_S U,ex\rangle \xrightarrow{a^{+}_{r@}} \langle T||_S U',ex\cup\ \{a:r@\}\rangle}\ a\notin S \qquad \frac{\langle U,r(ex)\rangle \xrightarrow{i} \langle U',ex'\rangle}{\langle T||_S U,ex\rangle \xrightarrow{i} \langle T||_S U',ex\rangle}$$

$$(c.4)\quad \frac{\langle U,r(ex)\rangle \xrightarrow{a^{-}_{@}} \langle U',ex'\rangle a:r@\in ex}{\langle T||_S U,ex\rangle \xrightarrow{a^{-}_{r@}} \langle T||_S U',ex-\{a:r@\}\rangle}\ a\notin S$$

$$(d.1)\quad \frac{\langle T,l(ex)\rangle \xrightarrow{a_{@'}} \langle T',ex'\rangle \langle U,r(ex)\rangle \xrightarrow{a_{@''}} \langle U',ex''\rangle}{\langle T||_S U,ex\rangle \xrightarrow{a^{+}_{<@'\,|\,@''>}} \langle T'||_S U',ex\cup\{a:<@'\ |\ @''>\}\rangle}\ a\in S$$

$$(d.2)\quad \frac{\langle T, l(ex)\rangle \xrightarrow{a@'} \langle T', ex'\rangle \langle U, r(ex)\rangle \xrightarrow{a@''} \langle U', ex''\rangle \quad a :< @' \mid @'' >\in ex}{\langle T ||_S U, ex\rangle \xrightarrow{a^+_{<@' \mid @''>}} \langle T' ||_S U', ex - \{a :< @' \mid @'' >\}\rangle} \; a \in S$$

$$(d.3)\quad \frac{\langle T, l(ex)\rangle \xrightarrow{a@'} \langle T', ex'\rangle \langle U, r(ex)\rangle \xrightarrow{a@''} \langle U', ex''\rangle \quad a :< @' \mid @'' >\in ex}{\langle T ||_S U, ex\rangle \xrightarrow{a^+_{<@' \mid @''>}} \langle T' ||_S U', ex - \{a :< @' \mid @'' >\}\rangle} \quad a \in S$$

2

$$(e.1)\quad \frac{\langle T, ex\rangle \xrightarrow{\theta} \langle T', ex'\rangle}{\langle T/L, h(ex,L)\rangle \xrightarrow{\theta} \langle T'/L, h(ex', L)\rangle} \quad \text{type}(\theta) \notin L$$

$$(e.2)\quad \frac{\langle T, ex\rangle \xrightarrow{\gamma} \langle T', ex'\rangle}{\langle T/L, h(ex,L)\rangle \xrightarrow{i} \langle T'/L, h(ex', L)\rangle} \quad \text{type}(\gamma) \in L$$

$$(f.1)\quad \frac{\langle P\{recX.P / X\}, \emptyset\rangle \xrightarrow{\theta} \langle T, ex\rangle}{\langle recX.P, \emptyset\rangle \xrightarrow{\theta} \langle T, ex\rangle}$$

Figure2 .6 : Règles pour la Technique Nom Statique

2.4. Conclusion :

Nous avons présenté dans ce chapitre, les principes généraux de la Technique de Description Formelle **LOTOS** et de manière particulière **Basic LOTOS** qu'est une version simplifié de **LOTOS**. Dans la première partie on a expliqué ses expressions de comportement, leur syntaxe formelle et les différentes opérations de **Basic LOTOS** comme le choix, interruption, parallélisme...etc. Ensuite on a parlé de la **ST-Sémantique**

Opérationnelle via des Noms Statiques qui cherche a exprimé les modèles de spécification du parallélisme. En fin on a donné les règles de **ST-Sémantique** pour **La Technique Nom Statique.**

Chapitre 3: Implémentation

3.1. Introduction :

Tout programmeur utilise jour après jour un outil essentiel à la réalisation de programmes informatiques : Le **Compilateur**. Un compilateur est un logiciel particulier qui traduit un programme écrit dans un langage de haut niveau (par le programmeur) en instruction exécutable (par un ordinateur). C'est donc l'instrument fondamental à la base de toute réalisation informatique.

En informatique, l'implémentation désigne mise en œuvre, ou la réalisation, donc dans ce chapitre, nous allons présenter notre projet c'est la création d'un compilateur pour générer le comportement pour la vérification de la spécification textuelle d'un système décrit en **LOTOS**.

3.2. Présentation du projet :

Pour rendre l'usage de **Basic LOTOS** visible et claire sur les plans de la spécification et de l'implémentation, nous avons réalisé un environnement appelé **« LOTOS ST-SEM »**, qui à pour rôle de générer une spécification textuelle "code source de **Basic LOTOS**" qui va être contrôlé par le compilateur.

3.3. Outil de réalisation du projet :

Pour réaliser notre outil nous avons utilisé langage de développement qui sera définis par suite.

❖ **Delphi :**

Delphi est un environnement de développement de type **RAD** (Rapid Application Development) basé sur le langage Pascal. Il permet de réaliser rapidement et simplement des applications Windows. Cette rapidité et cette simplicité de développement sont dues à une conception visuelle de l'application. **Delphi** propose un ensemble très complet de composants visuels prêts à l'emploi incluant la quasi-totalité des composants Windows (boutons, boîtes de dialogue, menus, barres d'outils…) ainsi que des experts permettant de créer facilement divers types d'applications et de librairies.

Pour maîtriser le développement d'une application sous Delphi, il est indispensable d'aborder les trois sujets suivants :

- Le langage Pascal et la programmation orientée objet ;
- L'Environnement de Développement Intégré (**EDI**) de **Delphi** ;
- Les objets de Delphi et la hiérarchie de classe de sa bibliothèque.

Nous complèterons cette approche par la connexion aux bases de données avec Delphi.

3.4. Conception globale du projet :

La **Figure 3.1** représente l'architecture globale de **LOTOS ST-SEM**

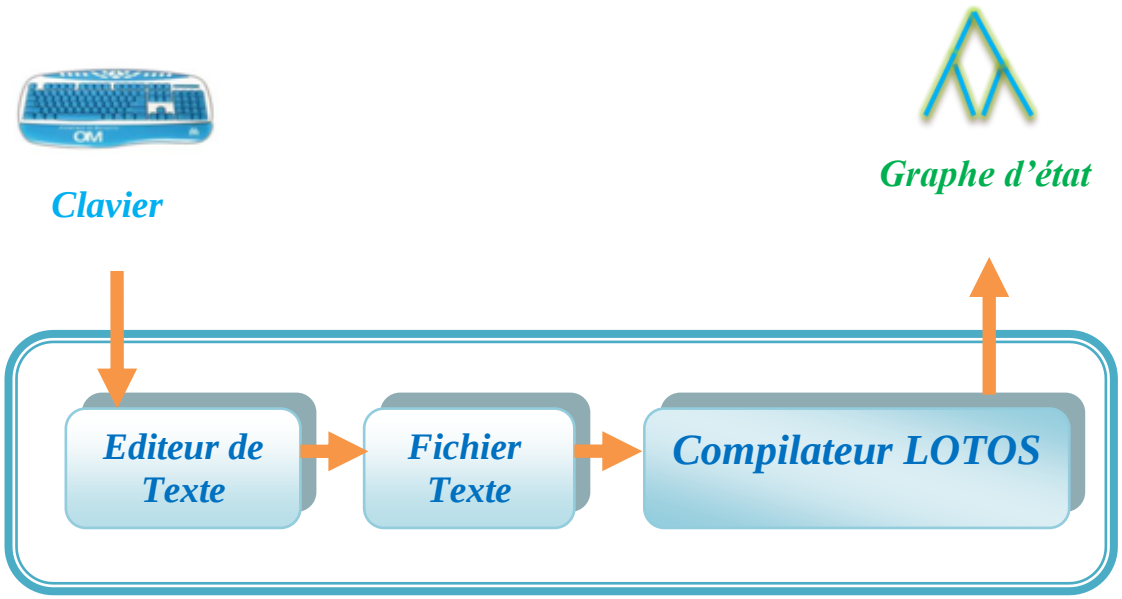

Figure 3.1: L'architecture Globale de LOTOS ST-SEM

3.5. Conception du compilateur de Basic LOTOS :

Ecriture d'un compilateur est maintenant une tâche réalisable pour tout programmeur avec un minimum de connaissance des techniques de compilation. Cette simplicité réside dans l'utilisation de Générateur de compilateurs.

Un Générateur de compilateur est un programme qui lit en entrée un programme écrit dans un premier langage (appelé **Langage Source**) et le traduit en un programme équivalent écrit dans un autre langage (appelé **Langage Cible**). Au cours de ce processus de traduction, le compilateur tentera également de repérer et de signaler les erreurs évidentes commises par le programmeur.

❖ Modèle de base d'un compilateur :

Il y a deux parties dans le processus de compilation :

- L'analyse : partie frontale (**frontend**).
- La synthèse : partie arrière (**backend**).

❖ Conception modèle Analyse et Synthèse :

Traduire un programme source dans un langage cible :

- Analyse le programme conformément à la définition du langage source.
- Produit d'éventuels messages d'erreurs.
- Synthétise un code équivalent au programme source dans le langage cible.

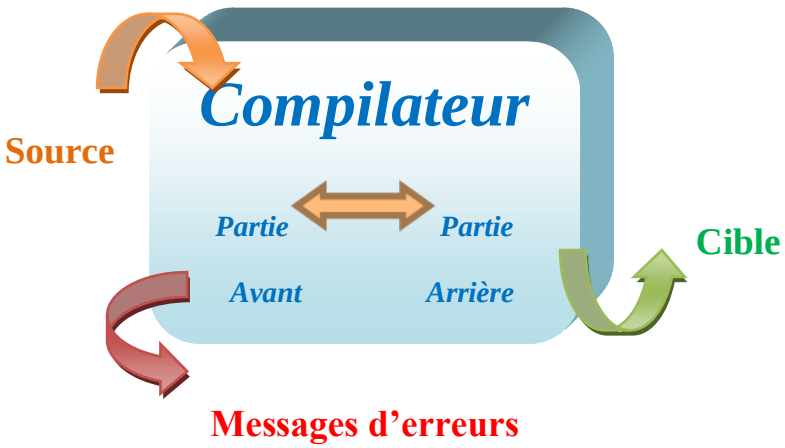

Figure 3.2 : Schéma Fonctionnel d'un Compilateur

La partie analyse :

- ✓ Sa tâche est de scinder le texte du programme source en ses constituants et d'en créer une représentation intermédiaire.

La partie synthèse :

- ✓ Sa tâche est de construire le programme cible à partir de la représentation intermédiaire du programme source.

Chaque partie est divisée en modules. L'entrée d'un module étant le résultat du module précédent.

Le **Compilateur LOTOS** est décrit comme étant constitué des différentes parties représenter dans la **Figure 3.3.**

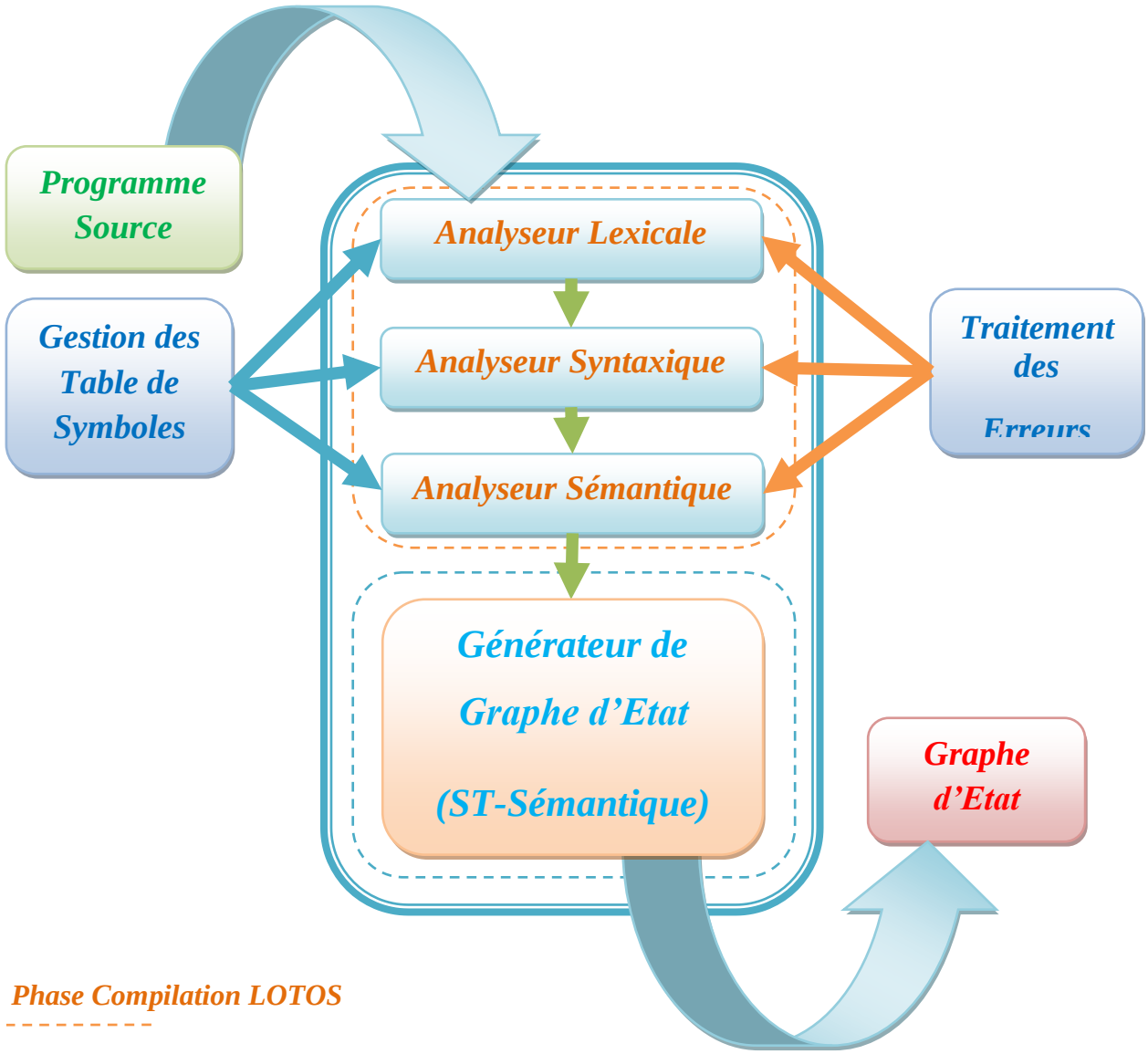

Figure 3.3 : Schéma Global d'un Compilateur d'une envirenement LOTOS ST-Sémantique

3.5.1. Les différents composants générés :

- **L'analyseur lexical :**
  - ✓ L'analyseur lexical prend en entrée le flot de caractères constituant le programme source et délivre les unités lexicales.
  - ✓ Le flot de caractères du programme source est lu de gauche à droite.
  - ✓ Les caractères formant un sens sont groupés en unités lexicales (**lexèmes**, **tokens**, **jetons**).

❖ **Unité lexicale** (**token**) :

C'est une suite de caractères ayant une signification collective.L'analyseur lexical doit aussi déterminer si chaque unité lexicale est un mot correct du vocabulaire du langage.

❖ **Détection des erreurs lexicales :**

- ✓ L'analyseur lexical génère une erreur dès qu'il rencontre un mot non autorisé par le langage.
- ✓ L'analyseur lexical est séquentiel et ne travaille pas sur plusieurs unités lexicales à la fois.
- ✓ L'analyseur lexical travaille sur les mots.

- **L'analyseur syntaxique :**

  - ✓ L'analyseur syntaxique prend en entrée un flot d'unités lexicales fournit par l'analyseur lexical et délivre les unités syntaxiques.
  - ✓ L'analyseur syntaxique doit également déterminer si chaque unité syntaxique est une phrase correcte du langage, c'est-à-dire qu'elle vérifie sa grammaire. Cette dernière définie par un ensemble des règles de production.
  - ✓ Lorsque l'analyseur syntaxique rencontre une structure qui n'est pas définie par les règles syntaxiques (les règles grammaticales), il signale une erreur syntaxique.

3.5.2. Compilateur de LOTOS ST-Sémantique :

La compilation d'une spécification **LOTOS ST-Sémantique** passe par les phases suivantes : compilation **LOTOS** et **génération**.

3.5.2.1. Compilation LOTOS :

Implémentation de l'analyseur lexical de Basic LOTOS :

L'objectif de l'analyseur lexical est de connaître les unités lexicales à partir du code source du programme et de signaler les éventuelles erreurs lexicales.

L'analyse Lexicale consiste à segmenter un texte source en un ensemble de mots que l'on appelle traditionnellement «**tokens**» (leur terme exact est **«lexème»**, ce qui signifie unité lexicale). Il s'agit d'une part de déterminer la suite des caractères comprenant le **token**, et d'autre part d'identifier le type de token (Mot clé, Identificateur, Caractère Spécial, Opérateur).

✓ **Les Unités Lexicales du langage Basic LOTOS :**

- **Mots-clés**: system, endsys, process, endproc, where, nil, exit, stop, hide, in.
- **Les opérateurs**: "[ ]", "||", "|||" , "|[liste_porte]|" , "[>" , ">>" , " ;".
- **Caractères spéciaux**: "(" , ")" , "[" , "]" , "," , " :=" .
- **Identificateur** : le nom du system, le nom du prosecessus, le nom du porte.

Par exemple déterminer, à partir de l'énoncé suivant :

system p [a, b]:= a; b; **stop endsys**

Doit être capable de comprendre qu'il s'agit de la suite de **Tokens** suivants:

Mot clé	**system**	**Caractère spécial**	**:=**
Identificateur	p	**Identificateur**	a
Caractère spécial	[	**Opérateur**	;
Identificateur	a	**Identificateur**	b
Caractère spécial	,	**Opérateur**	;
Identificateur	b	**Mot clé**	stop
Caractère spécial	]	**Mot clé**	endsys

Tableau 3.1: Table des jetons

Et que l'on peut construire **la Table des Symboles** suivante :

Adresse	**Symbole**
1	p
2	a
3	b

Tableau 3.2 : Table des symboles

Implémentation de l'analyseur syntaxique de Basic LOTOS :

L'objectif de l'analyse syntaxique est de reconnaitre les phrases appartenant à la syntaxe du langage. Son entrée est le flot d'unités lexicales construites par l'analyse lexicale.

L'analyse Syntaxique est l'une des opérations majeures d'un compilateur,
Lors de l'analyse syntaxique, on vérifie que l'ordre des **Tokens** correspond à l'ordre définit pour le langage. On dit que l'on vérifie la syntaxe du langage à partir de la définition de sa **Grammaire**, et à en tirer une représentation interne, sous forme d'une **liste** chaque champs de cette liste contient le nom de processus, liste de porte et les expression, il est utilisé ultérieurement pour générer un graphe d'états (**ST-Sémantique**) .

Une **Grammaire** est un ensemble de règles. Les règles qui nous intéressent en informatique sont celles qui donnent une description générative du langage.

✓ Le Grammaire de Basic LOTOS :

A partir des opérations de Basic LOTOS présenté dans Le Chapitre2 nous allons construis cette grammaire :

<grammaire lotos> ::=**SYSTEM**<id> [<liste_porte>] := <expression><corps>
ENDSYS
<corps>: :=<corps1>
<corps1>: :=**WHERE**<liste_processus> | epsilon
<liste_processus>: :=<processus><liste_processus>| <processus>
<processus>: :=**PROCESS**<id>[<liste_porte>] :=<expression><corps>**ENDPROC**
<liste_porte>: :=<porte>| epsilon
<porte>: :=<id>|<id> , <porte>
<expression>: :=<expression1>| epsilon
<expression1>: :=<expression2><op><expression1> |<expression2>
<expression2>: :=<id> ; <expression2>|<id>[<liste_porte>] | **HIDE**<porte>**IN**<expression2> | stop | nil | exit | (<expression1>)
<op> : := || | ||| | |[<liste_porte >]| |>>| [>| [ ]

Implémentation de vérificateur sémantique :

L'analyseur Sémantique L'analyse sémantique contrôle si le programme source contient des erreurs sémantiques . Voici les différentes phases de notre vérificateur sémantique voir **Figure 3.4.**

Figure 3.4: La Vérificateur Sémantique

1. Vérificateur d'action dupliquée :

Cette vérificateur test les doublons dans une liste d'action déclarées dans les paramètres du processus, comme le montre la **Figure 3.5**

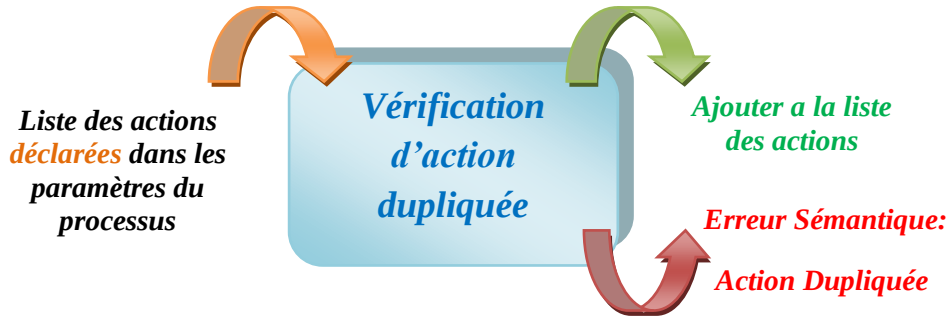

Figure 3.5 : Vérificateur d'action dupliquée

Exemple 3.2 :

system seveur [client1, client2, client1, client3] := client1 ; client2 ; exit

endsys

⇒ Erreur sémantique : action « client1 » dupliquée

2. Vérificateur d'action non déclaré :

Cette vérificateur teste la liste d'action d'appelle avec la liste d'action déclaré d'un processus, voir **Figure 3.7**.

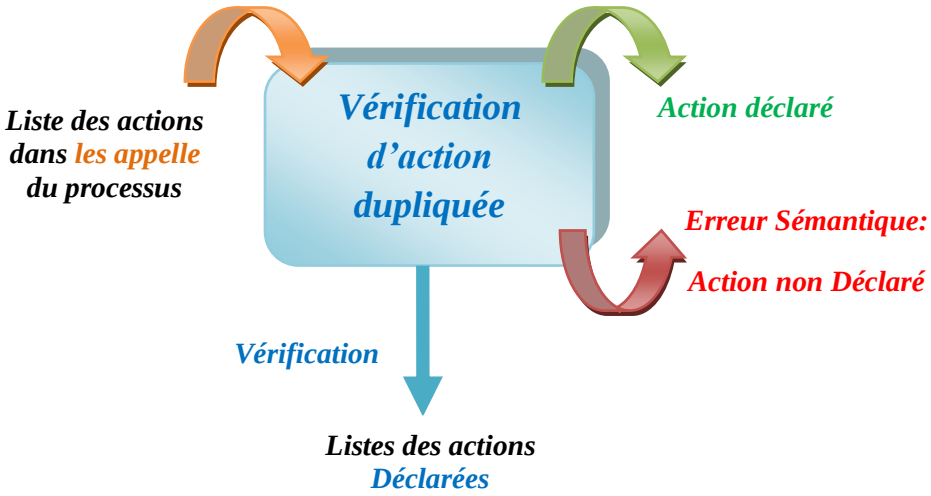

Figure 3.6 : Vérificateur d'action non déclaré

Exemple 3.3 :

system seveur [client1, client2, client3, client2] := client1 ; client5; exit

endsys

⇒ Erreur sémantique : action « client5 » non déclaré.

3. Vérificateur de déclaration de processus :

Cette vérificateur teste la liste d'action d'appelle avec la liste d'action déclaré d'un processus, voir **Figure 3.8.**

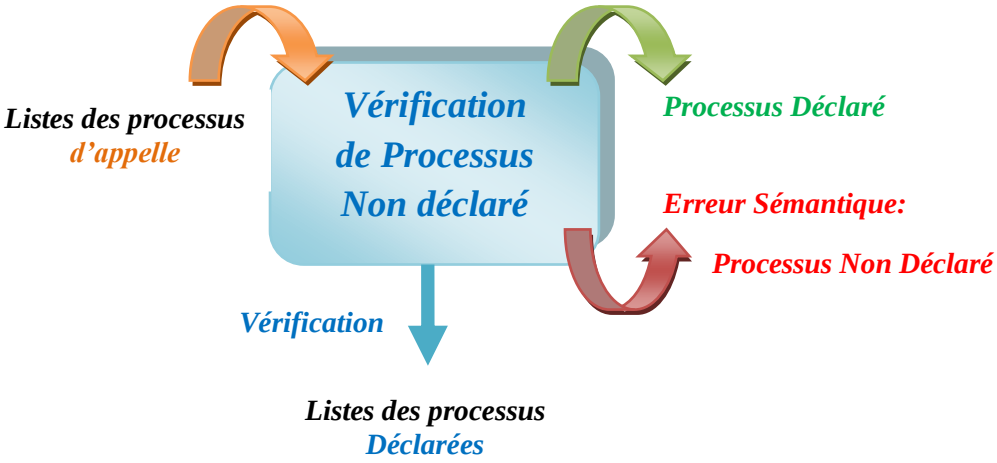

Figure 3.7 : Vérificateur de déclaration de processus

Exemple 3.4 :

system serveur [client1, client2, client3, client2] :=

site1 [client1, client2] >> site2 [client3, client2]

where

process site1 [client1, client2] := client1 ; client2; exit

endproc

endsys

⇒Erreur sémantique : processus « site2 » non déclarée

4. Vérificateur de nombre de paramètre :

Cette vérificateur teste une liste qui contient le non du processus d'appelle et leur nombre de paramètre avec une liste du processus déclarée et leur nombre de paramètre voir **Figure 3.8.**

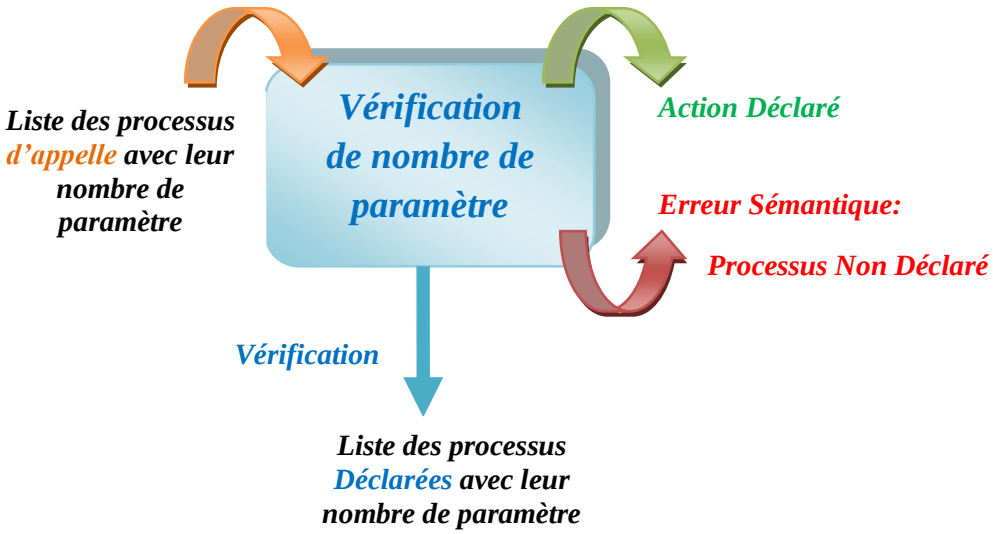

Figure3.8 : Vérificateur de nombre de paramètre

Exemple 3.5 :

system serveur [client1, client2] := client1 ; client2; site1 [client1]

where

process site1 [client1, client2] := client1 ; client2; exit

endproc

endsys

5. Vérificateur d'appel d'action interne « i » :

Cette vérificateur teste l'existence d'action interne « i » dans la liste des actions déclarées dans les paramètres du processus.

Exemple 3.6 :

system serveur [client1, i, client2] := client1 ; client2; exit

⇒ Erreur sémantique : appelle non autorisé

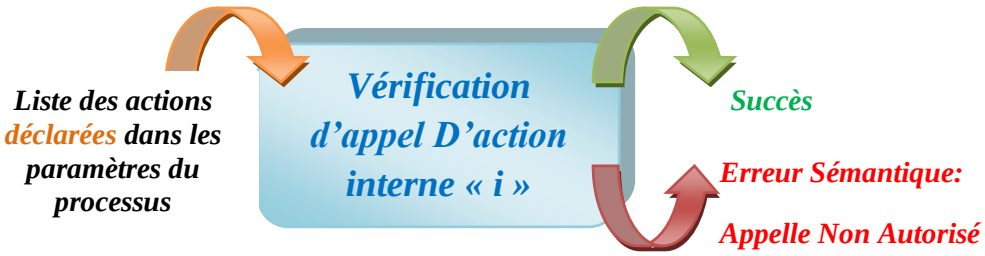

Figure3.9 : Vérificateur d'appel D'action interne « i »

6. Définition de processus dupliquée :

On va expliquer les différent cas ou le processus ne doit pas être dupliqué

1. **première cas les frères :** deux processus frère ne doit pas être dupliquée.

Exemple 3.7 :

system serveur [client1, client2] := site1 [client1] [>site2 [client1]

WHERE

process site1 [client1, client2] := client1 ; client2; exit endproc

process site2 [client1, client2] := client1 ; client2; exit endproc

process site2 [client1, client2] := client1 ; client2; exit endproc

endsys

3.5.2.2. Génération:

Dans cette étape on va produire le graphe d'états (**ST-Sémantique**) correspondant à une spécification textuelle Lotos à partir de la **liste** produise pendant l'analyse syntaxique.

- **Présentation du modèle graphe d'états (ST-Sémantique) :**

  Un graphe est formé d'un ensemble non vide et fini d'états, parmi lesquels on distingue un état initial qui représente le comportement du système lotos, reliées entre eux par des arcs représente l'action qui sera exécuter selon les règles de ST_Sémantique. Si le comportement du système contient des appels de processus sous la forme suivante **id [liste_de_porte]** on va récupérer le graphe d'états (**ST-Sémantique**) de ce processus.

3.6. Conclusion :

L'existence des divers langages et styles de programmation suppose l'existence d'un outil pour convertir n'importe quel langage en code binaire : cet outil s'appelle un **Compilateur**.

Dans ce chapitre nous avons essayé de donner une présentation générale sur notre application.

Notre produit est un système qui permet de développer un **Compilateur** pour générer le graphe d'états (**ST-Sémantique**) d'un système décrit en **LOTOS**.

Chapitre 4 : Etude de Cas

4.1. Introduction :

Tout au long de ce chapitre, nous allons modéliser le problème classique du dîner du philosophe dans notre application **LOTOS ST-SEM**. Et nous allons montrer comment ce problème va être présenté.

4.1. Présentation du système « Le dîner des philosophes » :

Nous souhaitons modéliser le dîner des philosophes. Les caractéristiques du problème sont les suivantes :

- La table est mise avec deux fourchettes lorsque deux philosophes sont présents et avec trois fourchettes lorsque trois philosophes sont présents.
- Les fourchettes sont disposées entre les philosophes.
- Un philosophe mange ou pense.
- Un philosophe peut manger s'il dispose de deux fourchettes.
- Un philosophe ne peut pas prendre une fourchette en cours d'utilisation par un autre philosophe
- Lorsqu'il a fini de manger, le philosophe pose ses fourchettes et se remet à penser.

Par déduction, nous pouvons également formuler les caractéristiques suivantes :

- Les fourchettes sont synchronisées avec les philosophes.
- Les philosophes ne sont pas synchronisés entre eux.
- Les fourchettes ne sont pas synchronisées entre elles.

Nous allons tout d'abord présenter les deux types de processus qui composeront notre système puis nous créerons ensuite ce dernier en utilisation le mécanisme de composition parallèle.

4.3. Problème du dîner des philosophes :

Soit cinq philosophes qui passent leurs temps à penser et à manger. Les philosophes sont assis autour d'une table circulaire, chaque philosophe ayant devant lui un bol de riz, et entre chaque deux bols consécutifs, on a une fourchette (baguette). Quand un philosophe pense, il n'interagit pas avec ses collègues. De temps en temps, un philosophe a faim et tente de s'emparer des deux fourchettes qui sont de part et d'autre de son bol (les fourchettes qui sont entre lui et ses voisins de gauche et de droite). Un philosophe peut prendre seulement une fourchette à la fois. Evidemment, il ne peut prendre une fourchette qui est déjà dans la main d'un voisin. Quand un philosophe affamé possède les deux fourchettes en même temps, il mange sans les libérer. Quand il a fini de manger, il pose les deux fourchettes et recommence à penser.

4.4. Spécification du problème :

Dans ce paragraphe, nous présenterons les démarches que nous suivrons pour modéliser le problème des philosophes par l'utilisation de notre application **LOTOS ST-SEM**. Le system contient deux composants : **Philosophe** et **Fourchette**

La description du system on peut représenter chacun des philosophes et des fourchettes par un processus, les processus correspondant à chacun d'entre eux s'exécutent an parallèle. Chaque instanciation du processus "**Philosophe**" correspond à une arrivée d'un philosophe qui veut partager la table avec ses voisins, et chaque instanciation du processus "**Fourchette**" correspond à une nouvelle ressource partagée entre deux philosophes.

4.5. La modélisation du problème :

La modélisation du modèle passe par 3 phases

- La description informelle du system (en boite noire)
- La description formelle du system (modélisation dans **LOTOS ST-SEM**)
- L'analyse et l'exécution du system (code source généré)

4.5.1. La description informelle du system :

On va définir les différents composants du system en boite noire

Le composant philosophe :

Le philosophe fait des interactions avec six actions : penser, prendre fourchette gauche, prendre fourchette droite, manger, déposer fourchette gauche, déposer fourchette droite, voir **Figure 4.1**.

Boite Noire

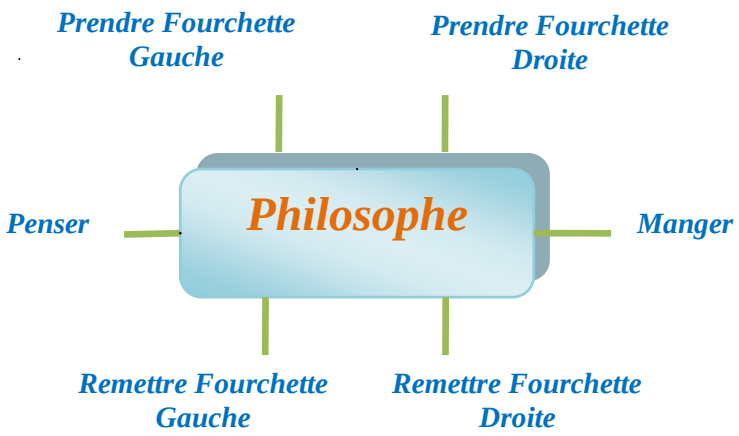

Figure 4.1 : Boite Noire de Philosophe

Les spécifications ci-dessus nous permettent de produire la spécification **LOTOS** suivante :

Algorithme 1 : Spécification LOTOS d'un philosophe

System SysPhi lo [p_pense , p_veut_manger , p_mange , p_prend_fd , p_prend_fg , p_pose_fd , p_pose_fg] :=
philosophe [p_pense , p_veut_manger , p_mange , p_prend_fd , p_prend_fg , p_pose_fd , p_pose_fg]

```
where
process  philosophe [ p_pense , p_veut_manger , p_mange , p_prend_fd , p_prend_fg ,
p_pose_fd , p_pose_fg ] :=
p_pense ; p_veut_manger ; ( p_prend_fd ; e x i t | | | p_prend_fg ; exit )
>>  p_mange ; ( p_pose_fd ; exit | | | p_pose_fg ; exit )
>>  philosophe [ p_pense , p_veut_manger , p_mange , p_prend_fd , p_prend_fg , p_pose_fd ,
p_pose_fg ]
endproc
endsys
```

Le processus philosophe suit le cycle suivant : dans l'état initial, le philosophe pense. Après un certain temps, ce dernier a envie de mange. Il se saisit alors de sa fourchette gauche et de sa fourchette droite (ces deux actions peuvent être effectuées en parallèle). Si il a pu prendre les deux fourchettes (utilisation du symbole >> pour spécifier une composition séquentielle), le philosophe commence à mange. Après cela, il repose ses deux fourchettes (La pose de la fourchette droite et de la fourchette gauche sont exécutés en parallèle). Le philosophe retourne ensuite à l'état initial.

Le composant fourchette :

La fourchette fait des interactions avec quatre action : prendre par philosophe droite, prendre par philosophe gauche, poser par philosophe droite, poser par philosophe gauche, voir **Figure 4.2.**

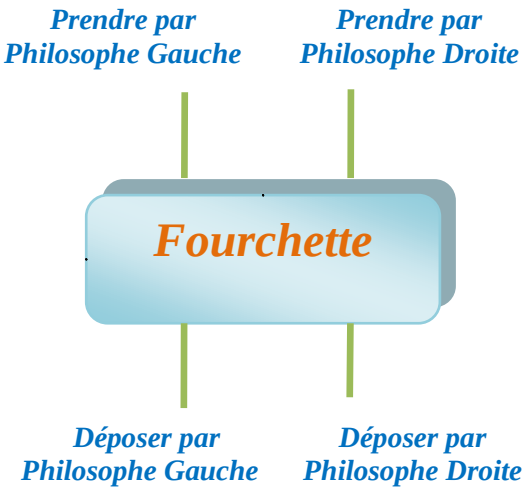

Figure 4.2 : Boite Noire de Fourchette

Les spécifications ci-dessus nous permettent de produire la spécification Lotos suivante :

Algorithme 2 : Spécification Lotos d'une fourchette

```
System SysFour [prise_par_phd, prise_par_phg, posee_par_phd, posee_par_phg] :=
fourchette [prise_par_phd, prise_par_phg, posee_par_phd, posee_par_phg]
where
process  fourchette [prise_par_phd, prise_par_phg, posee_par_phd, posee_par_phg]:=
( prise_par_phd ; posee_par_phd ; exit  [ ]  prise_par_phg ; posee_par_phg ; exit)
>>  fourchette [ prise_par_phd , prise_par_phg , posee_par_phd , posee_par_phg ]
endproc
endsys
```

Le processus fourchette suit le cycle suivant : la fourchette est prise par le philosophe de droite ou par le philosophe de gauche. Lorsque le philosophe a terminé d'utiliser la fourchette, cette dernière est reposée. La fourchette revient alors dans l'état initial.

Les interactions entre les composants :

Dans la **Figure 4.3** on a présenté l'interaction entre deux philosophes et fourchette

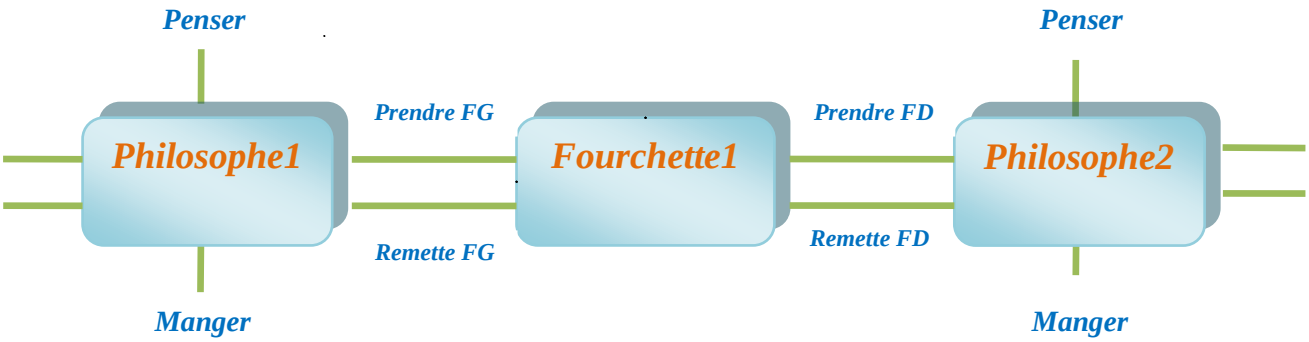

Figure 4.3 : Les interactions entre les Philosophes et les Fourchettes

Ordonnancement des actions :

Dans l'ordonnancement il faut définir l'ordre d'exécution des actions, comme il est montré dans la **Figure 4.4** et **4.5.**

Philosophe:

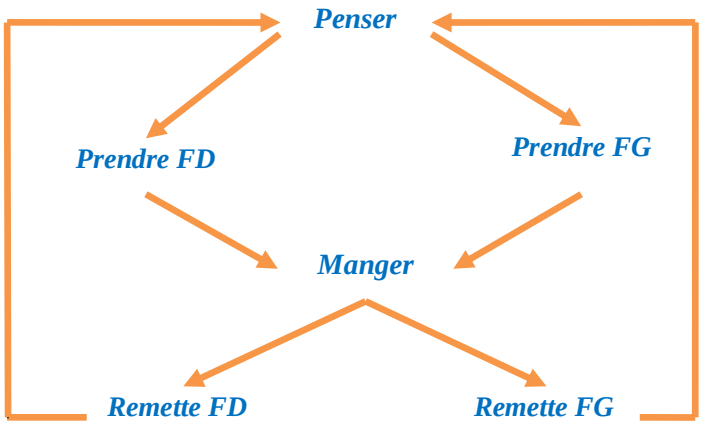

Figure 4.4 : Ordonnancement des actions de Philosophe

Fourchette :

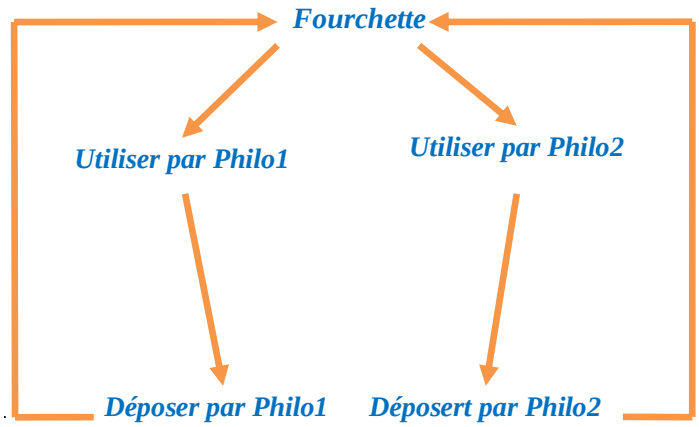

Figure 4.5 : Ordonnancement des actions de Fourchette

Composition parallèle des fourchettes et des philosophes:

- **Composition parallèle de deux philosophes et deux fourchettes (Philosophe1, Philosophe, Fourchette1, Fourchette2):**

Nous allons maintenant mettre en interaction deux philosophes et deux fourchettes pour débuter (**Figure 4.6**). Cette composition est définie par la spécification **LOTOS** suivante :

Algorithme 3 : Philosophe1, Philosophe2, Fourchette1, Fourchette2

```
System Sys2Phi l [ phi1_pense , phi1_veut_manger , phi1_mange , phi2_pense ,
phi2_veut_manger , phi2_mange , four1_prise_phg , four1_posee_phg , four1_prise_phd ,
four1_posee_phd , four2_prise_phg , four2_posee_phg , four2_prise_phd , four2_posee_phd ]
:=
( philosophe [ phi1_pense , phi1_veut_manger , phi1_mange , four1_prise_phg ,
four2_prise_phd , four1_posee_phg , four2_posee_phd ]
| | |
philosophe [ phi2_pense , phi2_veut_manger , phi2_mange , four2_prise_phg ,
four1_prise_phd , four2_posee_phg , four1_posee_phd ] )
| [ four1_prise_phg , four1_posee_phg , four2_prise_phd , four2_posee_phd , four2_prise_phg
, four2_posee_phg , four1_prise_phd , four1_posee_phd ] |
(fourchette [four1_prise_phd, four1_prise_phg , four1_posee_phd ,four1_posee_phg ]
| | |
fourchette [four2_prise_phd, four2_prise_phg, four2_posee_phd , four2_posee_phg] )
where
process philosophe [ p_pense , p_veut_manger , p_mange , p_prend_fd , p_prend_fg ,
p_pose_fd , p_pose_fg ] := p_pense ; p_veut_manger ; ( p_prend_fd ; exit | | | p_prend_fg ;
exit )
>> p_mange ; ( p_pose_fd ; exit | | | p_pose_fg ; exit )
>> philosophe [ p_pense , p_veut_manger , p_mange , p_prend_fd , p_prend_fg , p_pose_fd ,
p_pose_fg ]
endproc
```

```
process fourchette [prise_par_phd, prise_par_phg ,posee_par_phd, posee_par_phg] :=
( prise_par_phd ; posee_par_phd ; exit [ ] prise_par_phg ; posee_par_phg ; exit)
>>  fourchette [ prise_par_phd , prise_par_phg , posee_par_phd , posee_par_phg ]
endproc
endsys
```

On constate que, conformément aux spécifications définies plus tôt, les philosophes vivent de manière asynchrone grâce à l'utilisation du symbole ||| exprimant la composition parallèle sans synchronisation. Il en va de même pour les fourchettes. En revanche, on distingue un ensemble d'actions à synchroniser entre les fourchettes et les philosophes.

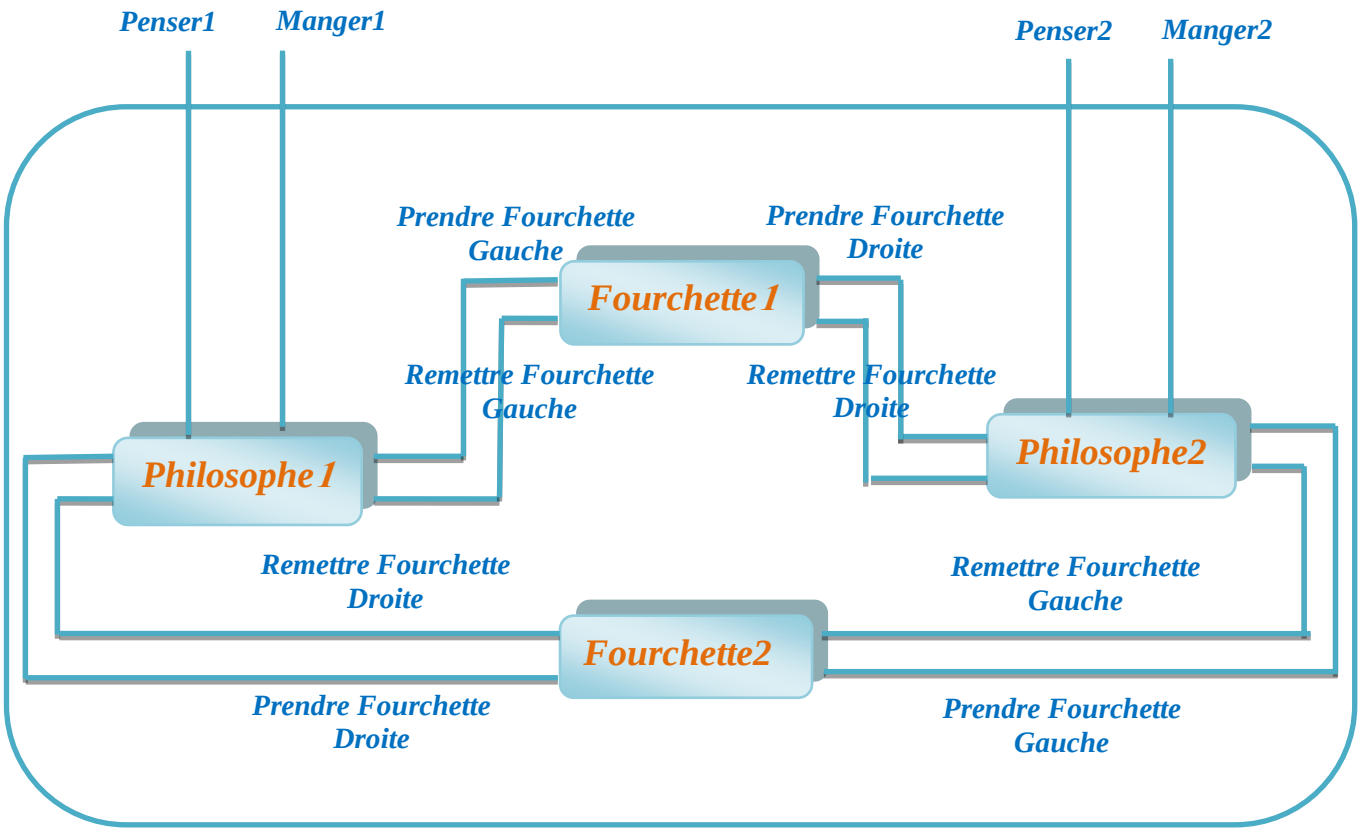

Figure 4.6 : Les interactions entre deux Philosophes et deux Fourchettes

4.5.2. La description formelle du system :

La description formelle du system permet de traduire toute la définition informelle en définition formelle du system, ça veut dire modéliser les différents composants et l'interaction entre eux dans notre application **LOTOS ST-SEM.**

Modélisation du dîner du philosophe dans LOTOS ST-SEM :

Le philosophe et la fourchette sont des processus. Les processus philosophe et fourchette sont exécutés en parallèle et se synchronisent sur des portes de synchronisation.

On suppose que l'initialisation du problème des philosophes impose que ces derniers sont concurrents. Pour exprimer ce comportement, nous utiliserons un opérateur de composition parallèle '|[ ]|'. Offert par **LOTOS** comme un mécanisme de rendez-vous.

4.6. Conclusion :

Les travaux pratiques d'informatique industrielle nous ont permis de conforter les connaissances lors des cours magistraux. Nous avons eu l'occasion de construire plusieurs spécifications de système à partir d'énoncés basés sur la réalité (dîner des philosophes, distributeur de boissons...). Cela nous a permis d'entrevoir l'importance des modèles du parallélisme. Ces derniers permettent, lorsqu'ils sont couplés aux techniques de vérifications formelles de s'assurer de la fiabilité d'un système de manière automatique. Pour finir, cette matière a été l'occasion de nous ouvrir sur un domaine de recherche passionnant et toujours d'actualité.

Conclusion Générale

LOTOS (Langage Of Temporing Ordering Spécification) est un standard utilisé pour spécifier les protocoles de télécommunication, utilisable pour les systèmes distribués en général, permet de décrire des ensembles de processus qui interagissent et échangent des données entre eux et avec leur environnement. **Basic LOTOS** est un sous ensemble de **LOTOS** ou il n'y a pas de prise en compte de type données, **Basic LOTOS** rassemble des opérateurs permet de crée leur comportement qui détermine à tout moment quelle actions sont possible comme action suivant de processus.

Notre produit est un système qui permet de développer un Compilateur pour **LOTOS** qui a pour rôle de générer une spécification textuelle **« Code Source de Basic LOTOS »** a un graphe d'état selon les règles de **ST-Sémantique Via des Noms Statiques**, cette dernière c'est un modèle qui représente le non atomicité de l'action par l'exécution séquentielle de deux actions atomiques, qui déterminent respectivement le début et la fin de cette action.

Après une brève présentation de la technique de vérification formelle basée sur les modèles donnée dans le premier chapitre, nous avons présenté, dans le **chapitre 2**, les principes généraux de la Technique de Description Formelle **LOTOS** et de manière particulière **Basic LOTOS** qu'est une version simplifiée de **LOTOS**. Dans la première partie on a expliqué ses expressions de comportement, leur syntaxe formelle et les différentes opérations de **Basic LOTOS** comme le choix, interruption, parallélisme...etc. Ensuite on a parlé de la **ST-Sémantique** Opérationnelle via des Noms Statiques qui cherche a exprimé les modèles de spécification du parallélisme. En fin on a donné les règles de **ST-Sémantique** pour **La Technique Nom Statique.**

Dans le **chapitre 3**, nous avons essayé de donner une présentation générale sur notre application.et nous allons voir dans ce chapitre les techniques qui ont été utilisées pour l'implémentation du compilateur.

Conclusion Générale

Dans le **chapitre 4**, nous avons eu l'occasion de construire plusieurs spécifications de système à partir d'énoncés basés sur la réalité, dans ce chapitre nous avons représenté l'étude du problème classique du dîner des philosophes, où on verra les démarches à suivre pour le modéliser dans notre logiciel. Ce chapitre est très important, car il met en valeur le travail réalisé.

Bibliographie

[1] Adel BENAMIRA. Vérification des équivalences de comportements des systèmes concurrents. La Boratoire LIRE, Université Mentouri de Constantine .2005-2006.

[2] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. TCS 37, 77—121 (1985).

[3] R. Milner. "Communication and Concurrency", volume 92 of LNCS. Springer Verlag (1980).

[4] R. Milner. Calculus for synchrony and asynchrony. TCS 25, 267—310 (1983).

[5] R. Milner. "Communication and Concurrency". Prentice Hall (1989).

[6] C. A. R. Hoare. "Communicating Sequenctial Processes". Prentice Hall (1985).

[7] ISO 9074. "Estelle, a Formal Description Technique Based on an Extended State Transition Model". ISO (November 1988).

[8] T. Bolognesi and E. Brinksma. "Introduction to the ISO Specification Language LOTOS", volume 14. Computer Networks and ISDN Systems (1987).

[9] ISO8807. LOTOS, a formal description technique based on the ordering of observation behaviour. ISO (November 1988).

[10] CCITT88. SDL, recommendation z.100-z.104. CCITT (1988).

[11] A. Arnold. Systèmes de transitions finis et sémantique des processus communicants. (1992).

[12] W. Reisig. Petri nets. In "EATCS Monofraphs on Theoretical Computer Science", Berlin, Heidelberg, New York, Tokyo (1985). Springer Verlag.

[13] M. A. Bednarczyk. "Categories of Asynchronous Systems". PhD thesis, Univ Sussex (1987 1987). Available as CS R 1/88.

Bibliographie

[14] M. W. Shields. Concurrent machines. The Computer Journal 28(5), 449—465 (1985).

[15] M. W. Shields. Deterministic asynchronous automata. In “Formal Methods in Programming”, North-Holland (1985)

[16] ISO. Basic Reference Model. International Standard 7498, International Organization for Standardization “Information Processing Systems “Open Systems Interconnection, Genève, 1984.

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

[18] ISO. ESTELLE -A Formal Description Technique Based on an Extended State Transition Model. International Standard 9074, International Organization for Standardization “ Information Processing Systems “ Open Systems Interconnection, Genève, septembre 1988.

[19] Marten van Sinderen, Ibrahim Ajubi, and Fausto Caneschi. The Application of LOTOS for the Formal Description of the ISO Session Layer. In Kenneth J. Turner, editor, Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88 (Stirling, Scotland) , Amsterdam, septembre 1988. North-Holland.

[20] Marc Phalippou and Roland Groz. Using ESTELLE for Verification: An Experience with the T.70 Teletex Transport Protocol. In Kenneth J. Turner, editor, Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88 (Stirling, Scotland) , Amsterdam, septembre 1988. North-Holland.

[21] Hubert Garavel : « Compilation et Vérification de Programmes LOTOS ». (1989).

[22] Stanislas Budkowski and Piotr Dembinski. An Introduction to ESTELLE: A Specification Language for Distributed Systems. Computer Networks and ISDN Systems, janvier 1988.

Bibliographie

[23] CCITT. Specification and Description Language. Recommendation Z.100, International Consultative Committee for Telephony and Telegraphy, Genève, mars 1988.

[24] R. Saracco and P. A. J. Tilanus. CCITT SDL: Overview of the Language and its Applications. Computer Networks and ISDN Systems , 1987.

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

[26] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. Computer Networks and ISDN Systems , janvier 1988.

[27] H. Ehrig, W. Fey, and H. Hansen. ACT ONE: An Algebraic Language with two Levels of Semantics. Bericht-Nr 83-03, Technische Universitat Berlin, 1983.

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

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

[30] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A Theory of Communicating Sequential Processes. Journal of the ACM, 31(3):560-599, juillet 1984.

[31] R. J. Van Glabbeek and F. W. Vaandrager. Petri net models for algebraic

theories of concurrency. In "PARLE'87, Vol II (Parallel Language)", volume 259

of LNCS. Spring-Varlag (1987).

[32] N.Belala, A.Delimi et M.N.Seghir. Environnement de vérification formelle pour Basic LOTOS base sur la Sémantique de Maximalité. La Boratoire LIRE, Université Mentouri de Constantine .2001-2002

Bibliographie

[33] MARIO BRAVETTI and ROBERTO GORRIERI. Deciding and Axiomatizing Weak ST Bisimulation for a Process Algebra with Recursion and Action Refinement. University of Bologna. October 2002.