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 :

Implémentation d'un compilateur LOTOS supporte la compilation modulaire

Encadré Par :

Dr. Benamira Adel

Présenté par :

Mlle Selmani Meriem

Mlle Boughazi Aicha

Juin 2013

Remerciement

Au terme de ce travail, nous tenons à remercier toutes les intervenantes et toutes les personnes qui, de prés ou de loin, ont contribué à la réalisation, en particulier :

Nous remercions en premier lieu monsieur «BENAMIRA ADEL» d'avoir accepté de diriger ce travail.ses conseils ainsi que sa gentillesse nous ont permis de mener à bien ce travail. Avec toute notre gratitude et notre reconnaissance. Nous remercions très sincèrement.

Nous remercions également l'ensemble du personnel du département de l'informatique (étudiants, enseignants, techniciens, secrétaire).

Boughazi Aicha

&

Selmani Meriem

Dédicaces

Je dédie ce modeste travail de mes années d'études :

- *A mes très chers parents, en leurs adressant mes plus vifs remercions pour leur amour et soutient qui ils m'ont apporte durant toutes mes années d'étude.*
- *A toutes mes familles plus particulièrement*
- *A mon marie Hichem Aissani*
- *A mes oncles : Nabil Grid , Rachid Ben Hacene*
- *A mes tantes: Chafia, Karima*
- *A mes chers amis*
  - *A mon binôme : Boughazi Aicha*

Selmani Meriem

Dédicaces

Je dédie ce travail de mes années d'études :

- *A mes chers parents «qui sont depuis mon enfance étaient à coté de moi m'encourageaient dans ma scolarité*
- *A toutes mes familles plus particulièrement*
- *A mes chers amis*
- *A mes sœurs et mes frères et leurs enfants*
    - ❖ *A mon binôme : Selmani Meriem*

Boughazi Aicha

SOMMAIRE

Liste des figures

Liste des tableaux

Résumé

Résumé

La vérification formelle des systèmes complexes constitue aujourd'hui un enjeu majeur dans de nombreux domaines de la société humaine. En effet l'employant des méthodes de spécification et vérification formelle, est assisté par des outils informatiques qui rendre l'analyse de ces systèmes fiables et garanties, et assure le bon coût de performance.

Notre grand intérêt est la vérification formelle des systèmes complexes en introduisant une approche modulaire, et nous allons présenter une technique de compilation qui permet de traduire le langage LOTOS, et respecte l'idée de modularité.

La compilation modulaire (séparer) est indispensable pour le développement des applications importantes (en taille !), et la façon pour un programme/projet de taille plus grande, elle suppose un découpage de l'application en modules compilables séparé. Chaque module est composé d'un ensemble des processus regroupés dans un fichier avec les déclarations nécessaires et son programme principal.

Mots clés : LOTOS, compilation modulaire, vérification formelle, modélisation formelle.

Introduction

Introduction générale

Aujourd'hui, les systèmes informatiques deviennent de plus en plus complexes et le risque de les mal concevoir est un problème croissant. Les spécifications de ces systèmes complexes sont la pierre angulaire de la maîtrise de leur fiabilité. Elles jouent à la foi le rôle d'outil de communication entre les différentes équipes impliquées dans le projet de développement et servent de référence pour l'implémentation du système. L'utilisation des méthodes formelles est alors le meilleur moyen d'assister la conception et la validation de ces spécifications, et il est assistées par des outils informatiques performants rendre l'analyse de ces systèmes fiables et garantis, en plus, un bon compromis coût performances est assuré.

La principale limitation de la vérification formelle basée sur les modèles est connue sous le nom de l'explosion combinatoire du graphe d'états. De multiples travaux sont en cours en vue de maîtriser cette explosion.

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 Petri [Rei85], les algèbres de processus (ACP [BK85],CCS [Mil80][Mil83][Mil89], CSP [Hoa85],. . .) et les techniques de description formelle (ESTELLE [90788], LOTOS [BB87][ISO88], SDL [CCI88]).

- 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 [Mil80], STE [Arn92],. . .) Et les modèles de non-entrelacement (dit vrai parallélisme)(RDP [Rei85], SEP ,STA [Bed87][Shi85a][Shi85b],. . .). Leur principale différence réside dans l'adoption ou non de l'hypothèse de l'atomicité temporelle et structurelle des actions (les actions sont de durée nulle et indivisible).Les modèles de vrai parallélisme nous permettent d'éviter les difficultés liées aux modèles entrelacés d'une part, et d'une autre part nous permettre d'introduire une méthode de conception formelle par raffinement successif (remplacer une action par un processus).

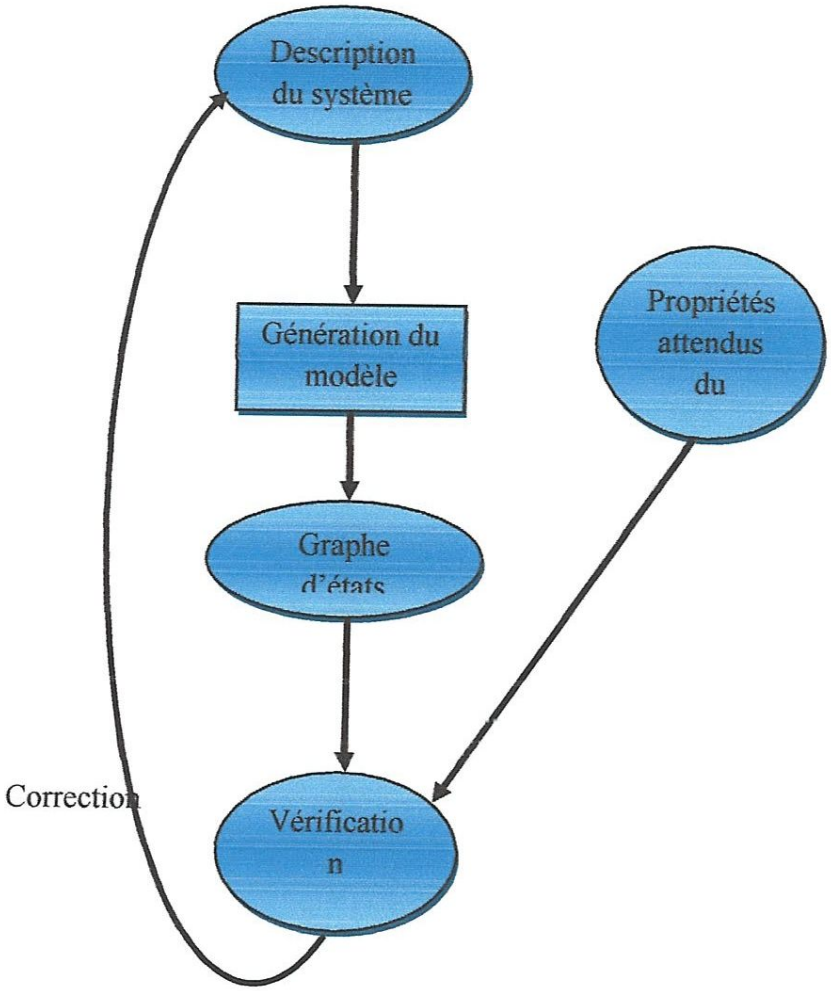

Figure 1.1*.vérification formelle.*

- 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 spécification logique (HML [HM80], CTL [CE81] [CES83], CTL* [EH86],. . .) Est généralement fondée sur des formules d'une logique temporelle modale qui sont interprétées sur le modèle sémantique sous-jacent. Cependant, les propriétés du deuxième type sont exprimées dans le même modèle sémantique de celui de la description du système.
- 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, il existe généralement deux types de vérification ; le premier type consiste de vérifier les formules logiques sur le graphe d'états du

système (dit vérification logique) et l'autre type est une comparaison entre les modèles sémantiques s'effectue au moyen d'une relation d'équivalence ou de préordre [Mil80][Par81] (dit vérification comportementale).

Problématique :

Ce mémoire permet d'intégrer une démarche de modélisation formelle au cours d'un cycle de développement. Cette dernière est basée sur une approche par composition (modularité indispensable lors du développement de systèmes complexes),

Notre approche est à la fois plus souple (en proposant une approche modulaire) et particulièrement efficace pour produire le modèle formel d'un système complexe.

Plan du mémoire

Le mémoire est organisé de la manière suivante :

- Le Chapitre 1 : est consacré à une présentation générale de La Technique de Description Formelle LOTOS, en s'intéressant par ailleurs de toutes les notions et les opérations du Basic LOTOS.
- Le Chapitre 2 : représente la compilation, et on a basé ici sur la compilation modulaire.
- Le Chapitre 3 : est destiné à la partie pratique de notre travail. Nous allons voir dans ce chapitre les techniques qui sont utilisées pour l'implémentation de notre application.
- Le Chapitre 4 : représente l'étude du problème classique du dîner de philosophes, où on verra les démarches à suivre pour le modéliser dans notre application. Ce chapitre est très important, car il met en valeur le travail réalisé.

Chapitre 1 :

La Technique de Description Formelle LOTOS

1. Introduction :

De la définition formelle du langage **LOTOS** ne sont repris que les aspects strictement indispensables à notre mémoire. C'est pourquoi les notions de sémantique des types abstraits algébriques ne sont pas traitées ici. Dans ce chapitre, nous présentons l'aspect contrôle (défini par Basic LOTOS).

LOTOS est une technique de description formelle développée comme une norme internationale, qui pourrait être employée pour la description et l'analyse de systèmes complexes. Son développement était motivé par le fort besoin d'un langage avec une abstraction de haut niveau et une base mathématique solide.

Il existe en **LOTOS** une nette séparation entre données et contrôle, en d'autres termes, une spécification **LOTOS** est constituée de deux parties, la première partie est basée sur la théorie de types abstraits algébriques. La syntaxe et la sémantique employées sont celles du langage ACT ONE. La deuxième partie **(Basic LOTOS)** décrit le comportement de la spécification, la syntaxe et la sémantique utilisées étant inspirées des deux algèbres de processus CCS et CSP. Dans le cadre de ce présent travail, les données ne seront pas prises en compte.

Le comportement d'un système est considéré comme un ensemble de processus en interaction sur des portes de communication. L'interaction ici exprime la synchronisation par rendez-vous symétriques entre plusieurs processus.[AB06]

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 d 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, peu être un humain, qui est supposé être toujours prés à observer tout ce qui est observable de 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 figure (2.1). [TBEB89]

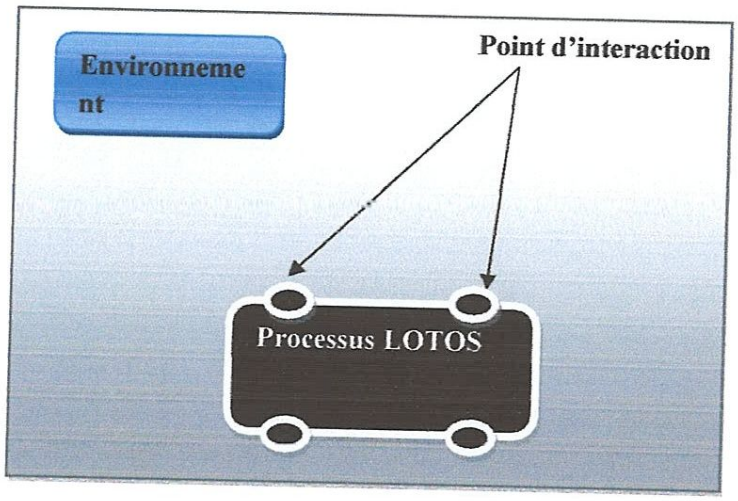

Figure. 1.1 *- Exemple d'un Processus LOTOS.*

La définition du processus spécifiera donc son comportement, en définissant la séquence 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 caractéristiqucs 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 (1.1).

3. Expressions de comportement :

Expression de comportement (behaviour expression), 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 (1.2)

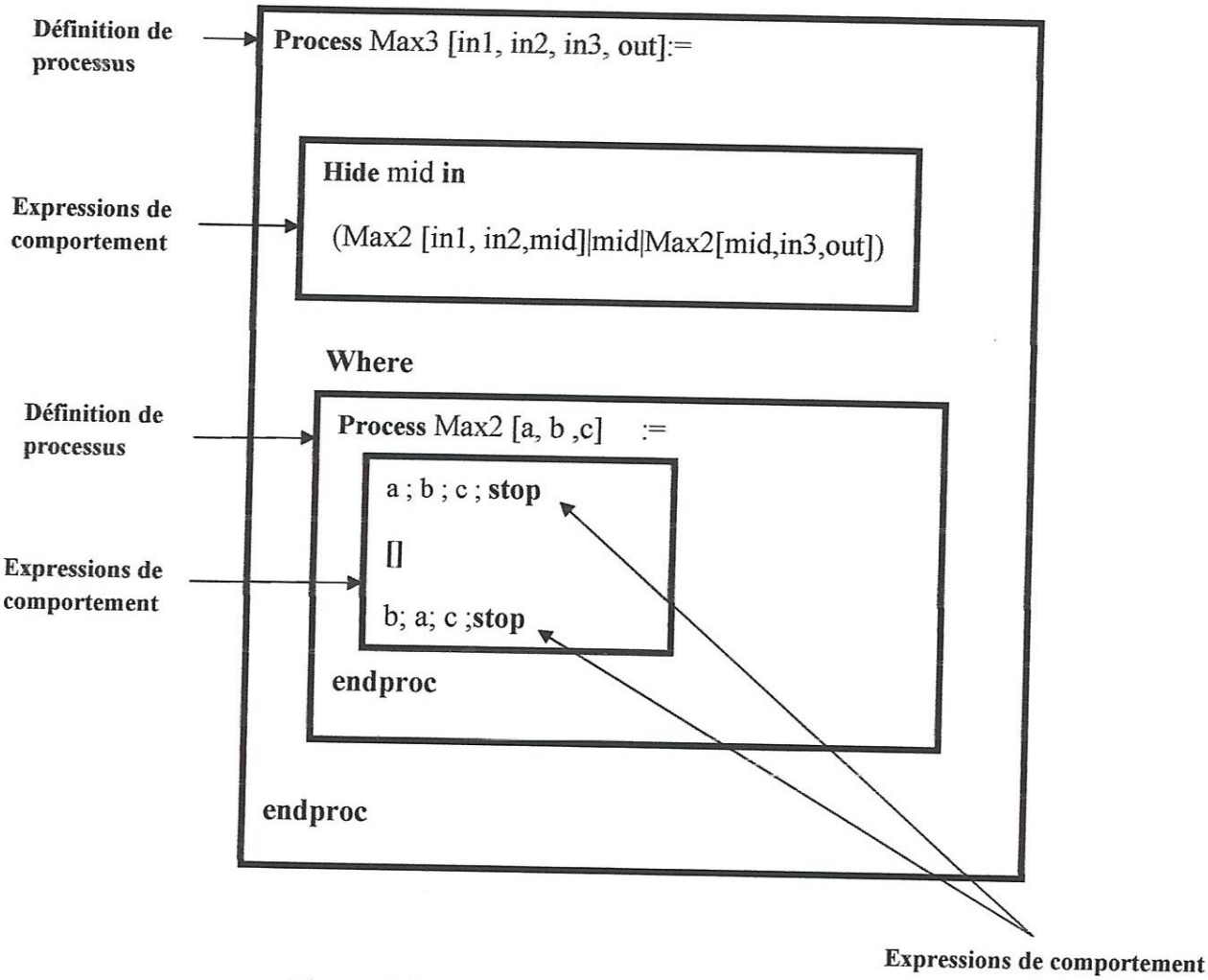

Figure. 1.2 *- Définition du processus Max3.*

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. [TBEB89]

4. 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. [TBEB89]

4.1 Syntaxe formelle de Basic :

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 notifier la terminaison avec succès dcs 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

|exit

|E[L]

|a;E

|i;E

|E[]E

|E|[L]|E

|hide L in E

|E>>E

|E [>E

4.2. Opérateurs du langage :

Soient E et F deux expressions de comportement et soit $L \subseteq G$ un sous-ensemble de portes

- **Inaction** : stop est une expression représente un processus inactif (un processus qui ne fait aucune interaction).

  Exemple 1.1 :

  Lire ; traiter ; écrire ; Stop. [HG93]

❖ **Terminaison avec succès** : exit est un processus de terminaison avec succès, son comportement se résume en une interaction sur la porte avant de se transformer dans le processus inactif.

Exemple 1.2:

MONEY ;(TEA; EXIT [> CANCEL; EXIT). [HG93]

❖ **Préfixage** : a; E représente le comportement d'un processus qui interagit sur le port a et qui se comporte ensuite comme E.

Exemple 1.3 : 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. [HG93]

❖ **Intériorisation** : **hide** L **in** E représente le comportement de E dont lequel toutes les interactions sur les portes de L sont rendues invisibles.

Exemple 1.4 :

On va décomposer le distributeur d'écrit 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 pertinente pour un observateur extérieur. [HG93]

❖ **Choix** : l'expression de comportement E [] F représente le processus qui se comporte soit comme E soit comme F.

Remarque 1-1 :

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).

Exemple 1.5 :

(Lire ; Stop)
[ ]
(Ecrire ; ((traiter ; Stop)
[ ]
(Ouvrir ; Stop))) [CA9900]

- **Composition parallèle** : l'expression de comportement E| [L] |F représente la composition parallèle de E et F avec synchronisation sur les portes qui sont dans L ; c'est-à-dire que E et F se comportent de façon indépendante sauf sur les portes appartenant à L; dans ce cas le premier comportement qui veut interagir sur l'une de ces portes doit attendre l'autre comportement pour qu'ils se synchronisent sur cette action, avant de pouvoir continuer à évoluer. Deux opérateurs sont utilisés pour désigner des cas particuliers de composition parallèle. Ce sont respectivement :

✓ E|||F, qui représente le cas où l'ensemble des portes de synchronisation L est vide (L = ∅)

Exemple 1.6 :

Plusieurs processus clients sont en concurrence pour lacer à une ressource fournie par un processus serveur :

(CLIENT [G]
|||
CLIENT [G]
|||...|||
CLIENT [G])
|[G]|
SERVEUR [G]

✓ E||Γ, qui représente le cas où l'ensemble des portes de synchronisation L est égal à G.

Exemple 1.7 :

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 CHO-COLATE. 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
(
TEA,
STOP
[]
COFFEE,
STOP
[]
CHOCOLATE,
STOP
)
|[MONEY, TEA, COFFEE, CHOCOLATE]|
MONEY ;
TEA ;
STOP.
```

- **Séquencèrent de processus** : l'expression de comportement E >> F dénote un processus qui se comporte d'abord comme E, et qui, dés que E s'est terminé avec succès, se comporte comme F.

Exemple 1.8 :

```
Jeton ; (Thé ; i ; exit [] Café ;i ;exit)
>>
Gobelet ; fin ; exit
```

- **Interruption** : l'expression de comportement E [> F dénote un processus qui se comporte comme E, mais qui peut à tout moment, tant que E ne s'est pas terminé avec succès, être interrompu par F, ce dernier prenant la main et continuant à s'exécuter.

Exemple 1.9:

L'opérateur ´ [>´ peut être utiliser pour d'écrire 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;

TEA;

Exit.

[>

CANCEL;

Exit.

- **Instanciation de processus** : L'ensemble des noms des processus est noté PN, qui est parcouru par P, Q, ... Une instanciation de processus est faite pour les occurrences des noms de processus dans une expression de comportement.
- **Définition de processus** : P := E dénote la définition d'un processus P dont le comportement est décrit par l'expression de comportement E. L'ensemble des définitions de processus est appelé environnement de processus. Une expression de comportement est toujours considérée dans le contexte d'un environnement de processus. [AB06]

4.3. La représentation graphique des Opérations :

Les opérateurs de Basic LOTOS sont schématisés par la figure (1.3).

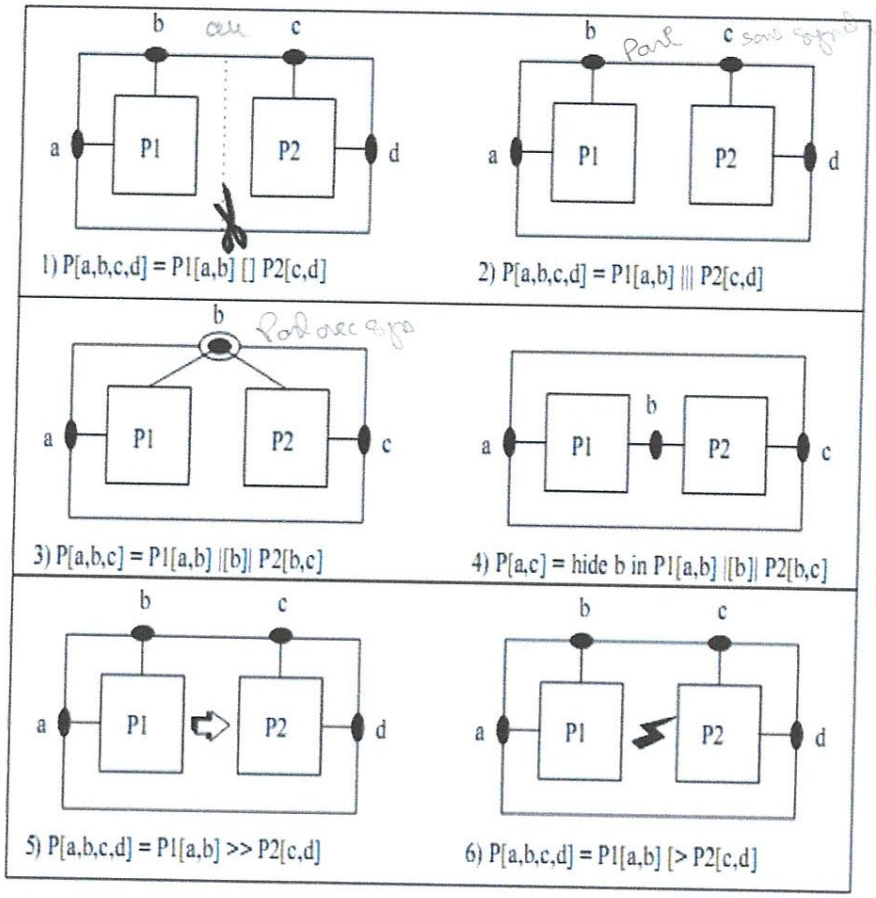

Figure. 1.3 *- Les opérateurs de Basic LOTOS.*

4.4. Sémantique d'entrelacement de Basic LOTOS :

Nous donnons dans ce qui suit la sémantique opérationnelle structurée (SOS) d'entrelacement de Basic LOTOS.

Définition: La sémantique opérationnelle structurée d'entrelacement du langage Basic LOTOS est donnée par l'ensemble des règles d'inférence suivantes *:*

1. $\dfrac{}{exit \xrightarrow{\delta} stop}$

2. $\dfrac{}{a;E \xrightarrow{a} E}$

3. (a) $\dfrac{E \xrightarrow{a} E'}{E \;[]\; F \xrightarrow{a} E'}$

   (b) $\dfrac{E \xrightarrow{a} E'}{F \;[]\; E \xrightarrow{a} E'}$

4. (a) i. $\dfrac{E \xrightarrow{a} E' \quad a \notin L \cup \{\delta\}}{E \;|[L]|\; F \xrightarrow{a} E' \;|[L]|\; F}$

   ii. $\dfrac{E \xrightarrow{a} E' \quad a \notin L \cup \{\delta\}}{F \;|[L]|\; E \xrightarrow{a} F \;|[L]|\; E'}$

   (b) $\dfrac{E \xrightarrow{a} E' \quad F \xrightarrow{a} F' \quad a \in L \cup \{\delta\}}{F \;|[L]|\; E \xrightarrow{a} F' \;|[L]|\; E'}$

[NB10]

5. **Conclusion :**

Dans ce chapitre nous avons introduit de manière générale le Basic LOTOS qui est une version simplifiée de LOTOS "partie contrôle". Et par ailleurs nous avons ses notions syntaxiques et sémantiques. Dans la première partie on a expliqué les différentes opérations de Basic LOTOS comme le choix, interruption, parallélisme... etc., en suite on a parlé de la sémantique opérationnelle qui cherche à exprimer ses modèles de spécification du parallélisme.

Chapitre 2 : La compilation

1. Introduction :

La compilation a été utilisée comme un outil de traitement pour les langages de programmation. Sachant que la génération du code cible soit ; le résultat de plusieurs phases du traitement, ou plusieurs techniques proposées pour chacune de ces phases. Dans ce chapitre, nous allons présenter un aperçu sur les compilateurs, les différentes phases utilisées dans un compilateur ainsi les différentes techniques utilisées dans chaque phase.

Au bout des dernières années les programmeurs utilisent une autre technique de la compilation qui respecte l'idée du la modularité, si pour ça il y a plusieurs compilateurs qui sont plus utilisés actuellement pour la production de logiciels basées sur un principe de compilation séparée. Chaque unité de code est compilée séparément, indépendamment de son utilisation effective, puis des différentes unités sont assemblées pour construire des programmes exécutables. [NWZ05]

2. Historique :

Au début de l'informatique, la programmation des ordinateurs se fait directement en langage machine. Ça se passe vite d'utiliser les possibilités de l'informatique pour faciliter le travail de programmation.

En premier lieu, on a donné des informations symboliques, ce que l'on appelle aussi des mnémoniques, aux instructions machines. Cela a été l'assembleur.

Ensuite, on a utilisé des étiquettes pour exprimer les branchements. Cela a donné des instructions de la forme :

BRANCH boucle ;

à la place d'une instruction de la forme

BRANCH -24, Où **-24** est le déplacement qu'il faut effectuer dans le programme.

Enfin, on a exprimé directement des places mémoires en leurs donnant un nom symbolique à la place de l'adresse en hexadécimal.

L'étape suivante a consisté à s'affranchir complètement de la machine en élaborant ce que l'on appelle des langages de haut niveau. Ces langages ont introduit un certain nombre de constructions qui n'existent pas dans le langage de la machine : expressions arithmétiques, variables locales, procédures et fonctions avec des paramètres qui retournent un résultat, structures de données (tableaux, énumération, record, objet,..), etc. [JF97]

3. Qu'est ce que la compilation :

Au premier temps de l'informatique, les ordinateurs ont été programmés directement par le langage machine. Cela s'est vite avéré fastidieux, Pour cela il a fallu construire des programmes qui traduisent des énoncés exprimés dans le langage de haut niveau utilisé par les programmeurs, ce qu'on appelle le **langage source**, en instructions pour **la machine cible**. Les programmes effectuant ce genre d'opération s'appellent des **compilateurs**

Un traducteur est un programme qui transforme chaque mot d'un langage L1 en un mot d'un langage L2

Un compilateur est un cas particulier de traducteur qui prend en entrée un mot représentant un programme écrit dans un certain langage et traduit le code source de ce programme en instructions pour la machine cible

Exemple 2.1:

- D'un langage de haut niveau « Pascal, Fortran, C… » vers un langage d'assembleur.
- Du langage d'assembleur vers du code binaire.
- D'un langage de haut niveau vers les instructions d'une machine virtuelle (JVM). [SL09]

4. Structure d'un compilateur :

4.1. Interpréteur versus Compilateur :

Un interpréteur est un logiciel qui interprète le programme source, c'est-à-dire qu'il analyse les instructions du programme source, les unes après les autres, et exécute chacune d'elles immédiatement. Dans ce cas, il n'y a pas de création d'un programme objet équivalent.

L'interpréteur ne connait pas a priori la structure du programme qu'il aura à exécuter et ne peut donc effectuer d'optimisations.

Un compilateur effectue un pré calcul sur un programme P pour le transformer en une suite d'instructions P'.

A l'inverse d'un compilateur, il travaille simultanément sur le programme et sur les données.

L'interpréteur doit être présent sur le système à chaque fois que le programme est exécuté, ce qui n'est pas le cas avec le compilateur.

Autre inconvénient, on ne peut pas cacher le code et donc garder des secrets de fabrication.

Les interpréteurs sont généralement plus petits que les compilateurs mais leur principal inconvénient est la lenteur de l'exécution d'un programme interprété par rapport aux autres. [SL09]

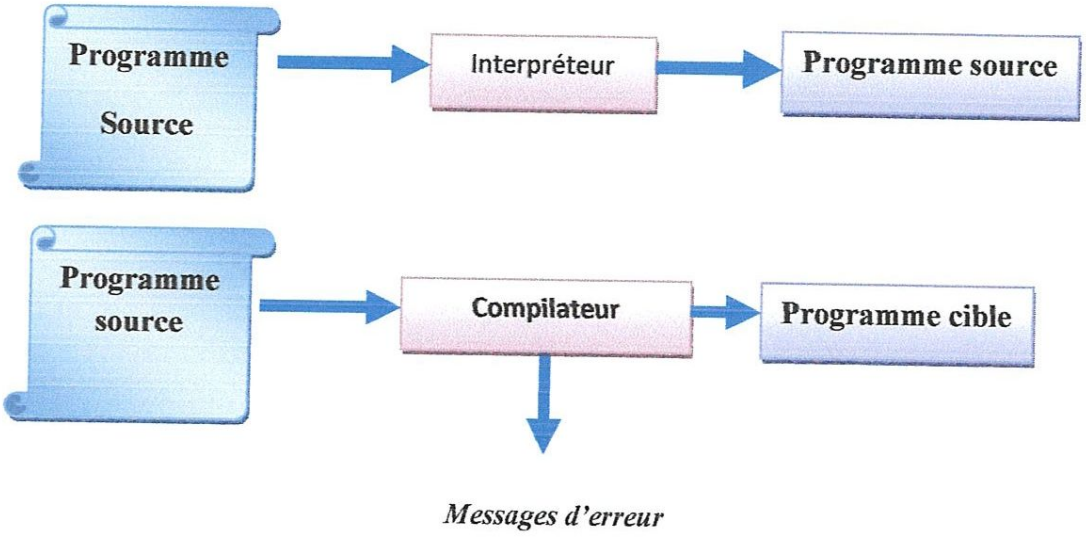

***Figure2.1**-Compilateur.*

4.2. Les Phases d'un compilateur: [FC10]

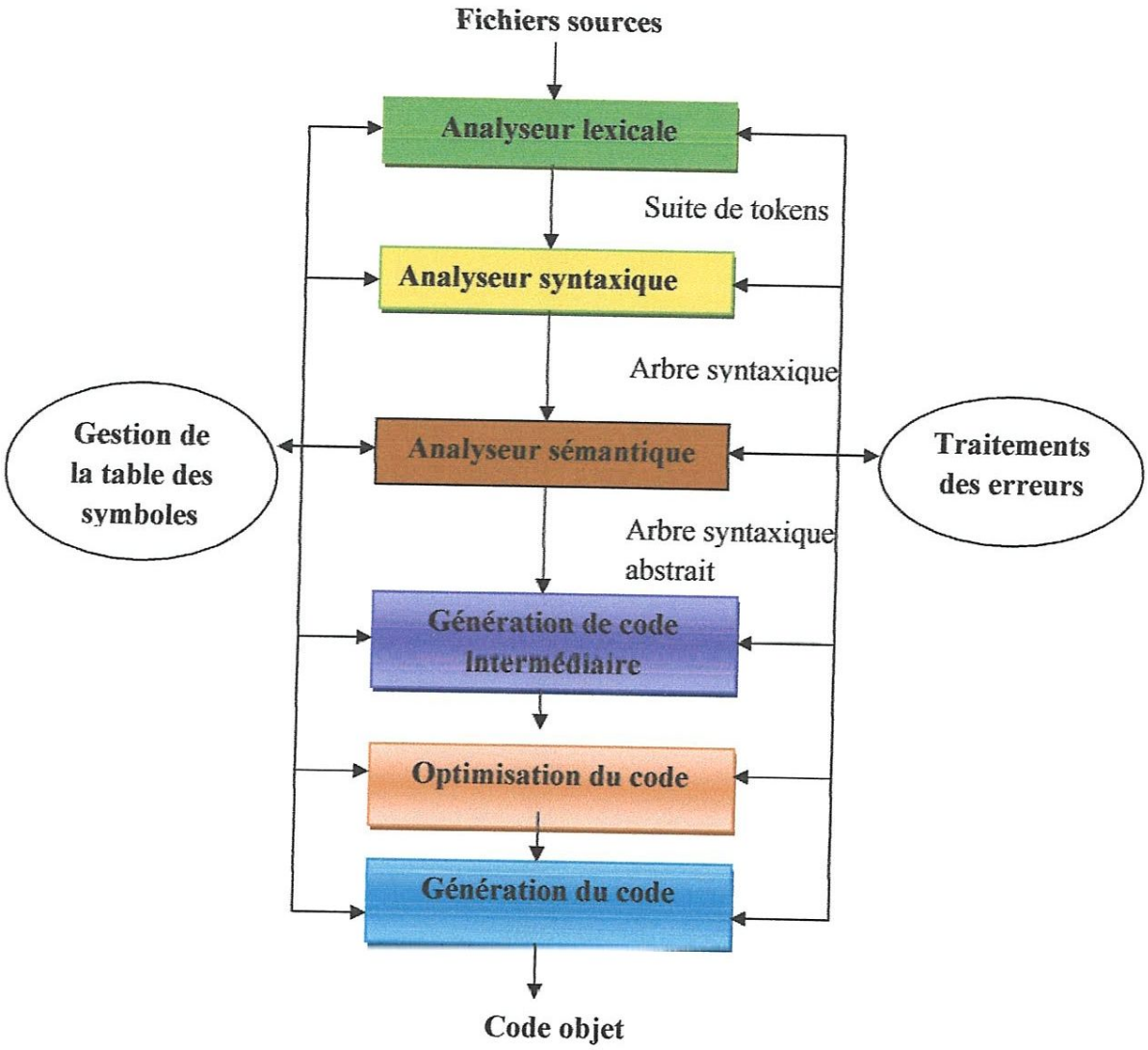

***Figure 2.2**-Structure d'un compilateur.*

4.2.1. La Table des symboles :

La table des symboles enregistre les identifiants et les attributs (emplacement mémoire, type, portée) :

- chaque identifiant (variable) a une entrée dans la table des symboles ;
- l'analyseur lexical crée dans la TS, une entrée à chaque fois qu'il rencontre un nouvel identificateur. Par contre, les attributs seront calculés plus tard.
- L'analyseur sémantique se servira de la table des symboles pour vérifier la concordance des types.

Détection des erreurs à plusieurs niveaux, essentiellement pendant l'analyse :

- erreur lexicale : le flot de caractères n'est pas reconnu ;

- erreur syntaxique : construction non reconnue par le langage ;
- erreur sémantique : problème de typage,... [HF10]

4.2.2. Analyse lexicale :

Définition :

L'analyseur lexical constitue la première étape d'un compilateur. Sa tâche principale est de lire les caractères d'entrée et de produire comme résultat une suite d'unités lexicales que l'analyseur syntaxique aura à traiter.

- ✓ **Une unité lexicale** est une suite de caractères qui a une signification collective.
- ✓ **Un lexème** (jeton) toute suite de caractère du programme source qui concorde avec le modèle d'une unité lexicale.
- ✓ **Un modèle** (Règle lexicale) est une règle associée à une unité lexicale qui décrit l'ensemble des chaînes du programme qui peuvent correspondre à cette unité lexicale.
- ✓ **Attributs** informations concernant le lexème (champs dans la table des symboles).

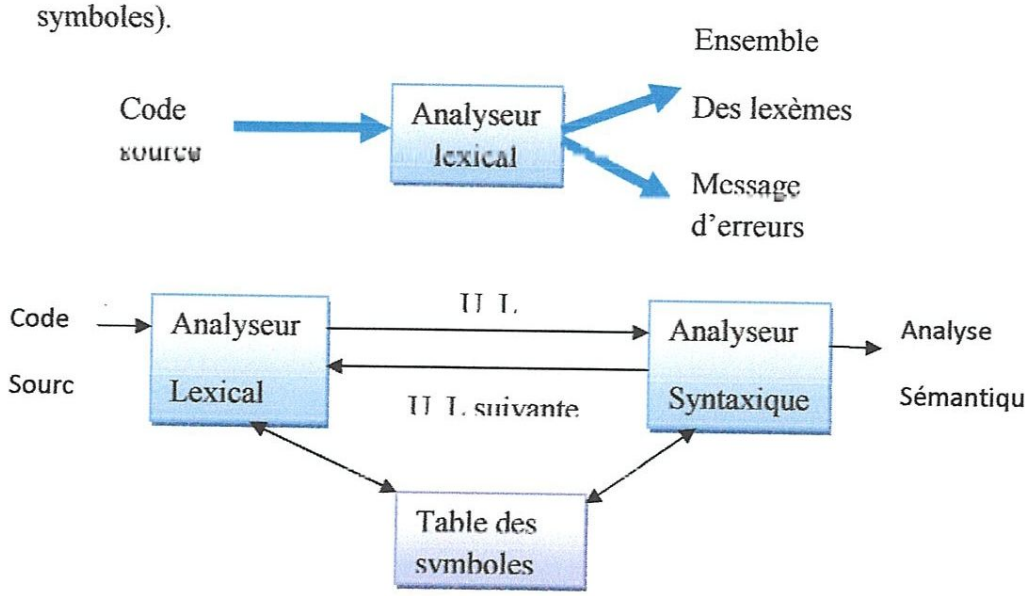

***Figure.2.3**-Analyse lexical.*

En plus, l'analyseur lexical réalise certaines tâches secondaires comme l'élimination de caractères superflus (commentaires, tabulations, fin de lignes, ...), et gère aussi les numéros de

ligne dans le programme source pour pouvoir associer à chaque erreur rencontrée par la suite la ligne dans laquelle elle intervient. [MeNaAL1112]

Expressions régulières :

Une expression régulière est une notation pour décrire un langage régulier.

Soit A un alphabet (un ensemble de lettres), une expression régulière est donc:

1. Les éléments de A, ε et □ sont des expressions régulières.

2. Si α et β sont des expressions régulières, alors (α | β), (α β) et α* sont des expressions régulières. (α | β) représente l'union, (α β) la concaténation et α * la répétition (un nombre quelconque de fois). ε est l'élément neutre par rapport à la concaténation et □ est l'ensemble vide de caractère, neutre par rapport à l'union. [MeNaAL1112]

Exemple 2.2: (a|b)*abb,

Identificateur = alpha (alpha | numer)*

En fait, les expressions régulières sont beaucoup plus puissantes que ce dont on a besoin lorsqu'on fait de l'analyse lexicale.

Automates :

Un automate à états finis (AEF) est défini par :

- Un ensemble fini E d'états.
- Un état e0 distingué comme étant l'état initial.
- Un ensemble fini T d'états distingués comme états finaux (ou états terminaux)
- Un alphabet Σ des symboles d'entrée.
- Une fonction de transition δ:A × E ⟶ E qui à tout couple formé d'un état et d'un symbole de fait correspondre un ensemble (éventuellement vide) d'états : $\delta(e_i, a) = \{ei1,..,e_{in}\}$.

Les automates sont souvent donnés sous la forme d'un graphe: les états sont les nœuds du graphe et Les arcs correspondent à la fonction de transition. [MeNaAL1112]

Exemple 2.3:

Σ= {a, b}, E= {0, 1, 2,3}, e_0=0, T= {3}

δ(0,a)={0,1}, δ(0,b)={0}, δ(1,b)={2}, δ(2,b)={3}.

Représentation graphique :

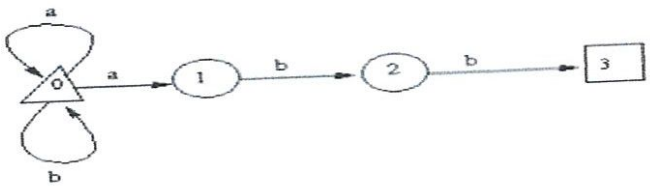

Figure 2.4.*Représentation graphique d'un automate.*

Erreurs Lexicales :

Peu d'erreurs sont détectables au seul niveau Lexical ; Plusieurs stratégies sont possibles :

- **mode panique** : on ignore les caractères qui posent problème et on continue. Cette technique se contente de refiler le problème à l'analyseur syntaxique.
- **transformations du texte source** : insérer un caractère, remplacé, échangé, etc. Elle se fait en calculant le nombre minimum de transformations à apporter au mot qui pose problème pour en obtenir un qui ne pose plus de problèmes. Cette technique de récupération d'erreur est très peu utilisée en pratique car elle est trop coûteuse à implanter. [MeNaAL1112]

Cet exemple explique l'analyseur lexical .

For i: =1 to vmax do a: =a+i;

On peut dégager la suite de **tokens** suivante :

For : mot clé	Do : mot clé	;: séparateur
i : idcntificateur	a : identificateur	
:= : affectation	:– . affectation	
1 : entier	a : identificateur	
To : mot clé	+: opérateur arithmétique	
vmax : identificateur	i : identificateur	

Et que l'on peut construire **la table des symboles** suivante :

Numéro de symbole	**Token**	**Type de token**	**Type de variable**
10	For	Mot clé	
11	To	Mot clé	
12	Do	Mot clé	
13	;	séparateur	
……	……	……	
100	:=	Affectation	
102	+		
……	……	……	
1000	I	ident	
1001	A	ident	
1002	Vmax		
1003	1	Entier	

Tableau 2.1.*Exemple d'une* table des symboles.

Ensuite, l'énoncé précédent peut s'exprimer ainsi : 10, 1000, 100, 5001, 11, 1002, 12, 1001, 100, 1001, 101, 1000, 13.[SL09]

4.2.3. Analyse syntaxique :

Définition :

L'analyseur syntaxique reçoit une suite d'unités lexicales de la part de l'analyseur lexical et doit vérifier que cette suite peut être engendrée par la grammaire du langage.

→Est ce que m appartient au langage généré par G ? Le principe est d'essayer de construire un arbre de dérivation. [MeNaAS1112]

❖**Grammaires:**

Une grammaire est un ensemble de règles décrivant comment former des phrases.

Une grammaire est la donnée de G= (VT, VN, S, P) où

- VT est un ensemble non vide de symboles terminaux (alphabet terminal)
- VN est un ensemble de symboles non-terminaux, avec VT ∩VN

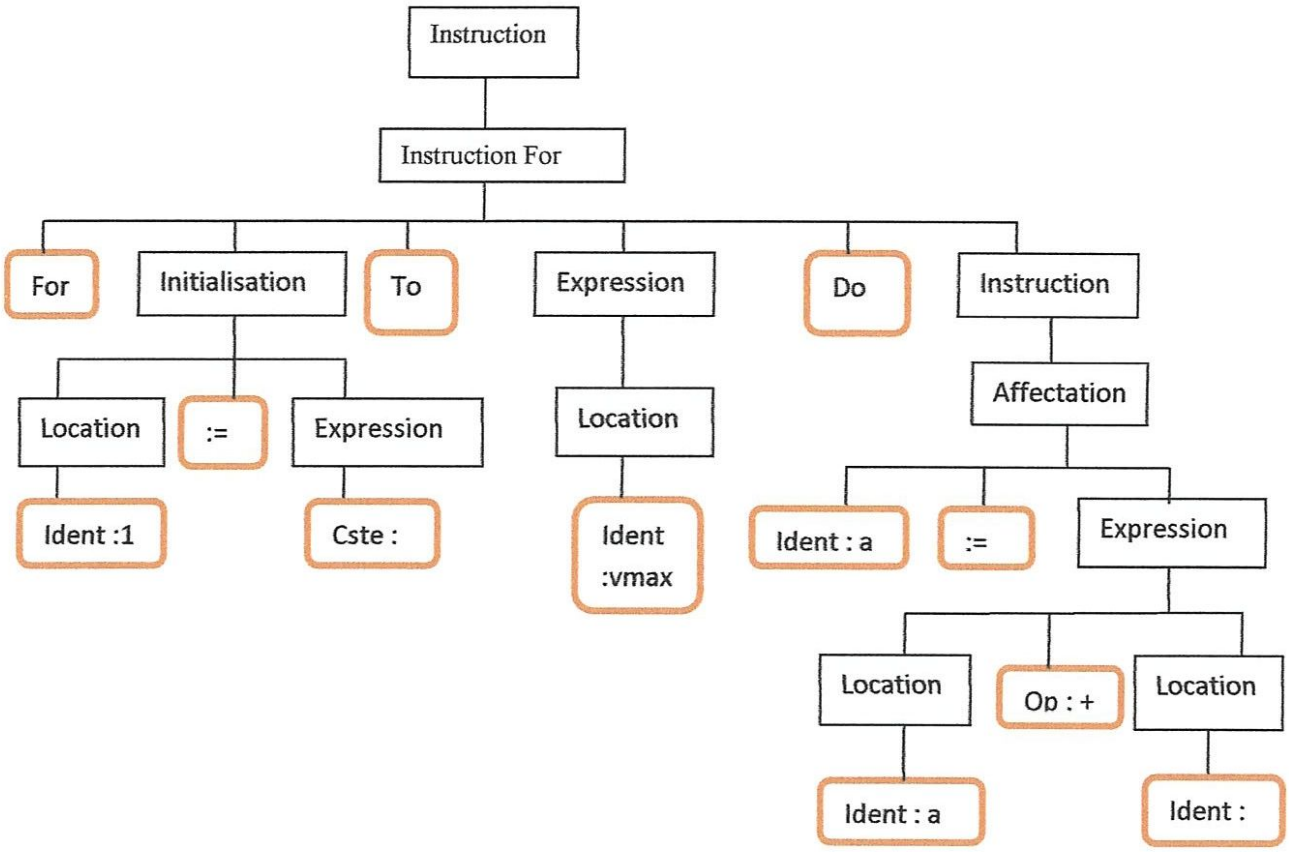

Figure 2.5. *Représentation d'un arbre syntaxique.*

Erreurs syntaxiques :

Les erreurs syntaxiques révélées lorsque les unités lexicales provenant de l'analyseur lexical Contredisent les règles grammaticales.

La nature de l'erreur est très difficile à déduire. La plupart du temps, le gestionnaire d'erreurs doit Deviner ce que le programmeur avait en tête. [MeNaAS1112]

4.2.4. Analyse sémantique :

La phase d'analyse sémantique effectue le contrôle des types: une opération a-t-elle toujours le bon nombre d'arguments avec les bons types ? Le résultat de cette phase est :

- un arbre de syntaxe abstraite qui est un arbre de syntaxe condensé avec éventuellement quelques informations ;
- une table des symboles décrivant des propriétés des symboles du programme. [PPBLI]

- S est un symbole initial ∈VN appelé axiome
- P est un ensemble de règle de productions. [MeNaAS1112]

Arbre de dérivation :

On appelle arbre de dérivation (ou arbre syntaxique) tout arbre tel que :

- la racine est l'axiome.
- les feuilles sont des unités lexicales ou ε
- les nœuds sont des symboles non-terminaux
- les fils d'un nœud α sont β0… βn si et seulement si α → β0… βn est une production (Le parcours préfixe de l'arbre donne le mot). [MeNaAS1112]

Les Types d'analyseur syntaxique :

➢ **Analyseur syntaxique descendant**, ou analyseur prédictif

✓ Construit l'arbre de dérivation à partir de sa racine et en effectuant des dérivations en considérant la tête des règles de production et en faisant des dérivations les plus à gauche.

✓ Famille des **analyseurs LL** (left scanning, leftmost derivation).

➢ **Analyseur syntaxique ascendant** ou par décalage-réduction

✓ Construit l'arbre de dérivation à partir de ses feuilles et en effectuant des dérivations en considérant la partie droite des règles de production et en faisant des dérivations les plus à droite.

✓ Famille des **analyseurs LR** (left scanning, rightmost derivation). [EB1213]

Exemple 2.4:

Soit $G = [V, \sum, P, S]$ avec

$\sum = \{a,b,c,d\}$

$V \setminus \sum = \{S, T\}$

6 règles de production :

$S \rightarrow aSbT|cT|d$

$T \rightarrow aT|bS|c$

Soit w = accbbadbc

Analyse LL :

S ⟶aSbT

S ⟶aSbT ⟶acTbT

S →aSbT →acTbT →accbT

S →aSbT →acTbT →accbT →accbbS

S →aSbT →acTbT →accbT →accbbS →accbbaSbT

S →aSbT →acTbT →accbT →accbbS →accbbaSbT→*accbbadbT*

S →aSbT →acTbT →accbT →accbbS →accbbaSbT→accbbadbT →accbbadbc

Analyse LR:

accbbadbc

accbbadbc

accbbadbc ← acTbbadbc

accbbadbc ← acTbbadbc ←aSbbadbc

accbbadbc ←acTbbadbc ←aSbbadbc ← aSbbaSbc

accbbadbc ← acTbbadbc ←aSbbadbc ←aSbbaSbc ← aSbbaSbT

accbbadbc ←acTbbadbc ← aSbbadbc ← aSbbaSbc ← aSbbaSbT← aSbbS

accbbadbc ← acTbbadbc ←aSbbadbc ← aSbbaSbc ← aSbbaSbT← aSbbS ← aSbT

accbbadbc ← acTbbadbc ← aSbbadbc ← aSbbaSbc ← aSbbaSbT← aSbbS ← aSbT ←S

Exemple 2.5 :

D'après l'exemple d'analyse lexical en fait l'arbre suivant représente la structure de la phrase syntaxique:

For *i*: =1 to vmax do a: =a+*i*

4.2.5. Génération du code intermédiaire :

Il s'agit de produire les instructions en langage cible.

En règle générale, le programmeur dispose d'un calculateur concret (cartes équipées de processeurs, puces de mémoire, ...). Le langage cible est dans ce cas défini par le type de processeur utilisé.

Mais si l'on écrit un compilateur pour un processeur donné, il n'est alors pas évident de porter ce compilateur (ce programme) sur une autre machine cible. C'est pourquoi on introduit des machines dites abstraites qui font abstraction des architectures réelles existantes. Ainsi,

On s'attache plus aux principes de traduction, aux concepts des langages, qu'à l'architecture des machines.

En général, on produira dans un premier temps des instructions pour une machine abstraite (virtuelle). Puis ensuite on fera la traduction de ces instructions en des instructions directement exécutables par la machine réelle sur laquelle on veut que le compilateur s'exécute. Ainsi, le portage du compilateur sera facilité, car la traduction en code cible virtuel sera faite une fois pour toutes, indépendamment de la machine cible réelle. Il ne reste plus ensuite qu'à étudier les problèmes spécifiques à la machine cible, et non plus les problèmes de reconnaissance du programme. [MN1112]

4.2.6. Optimisation du code :

Cette phase tente d'améliorer le code produit de telle sorte que le programme résultant soit plus rapide. Par exemple

- détecter l'inutilité de recalculer des expressions dont la valeur est déjà connue,
- transporter à l'extérieur des boucles des expressions et
- sous-expressions dont les opérandes ont la même valeur à toutes les itérations détecter, et supprimer, les expressions inutiles. [MN1112]

4.2.7. Génération du code objet :

Cette phase nécessite la connaissance de la machine cible (réelle, virtuelle ou abstraite), et notamment de ses possibilités en matière de registres, piles, etc. [MN1112]

5. la compilation modulaire (séparer) :

5.1. Définition :

La compilation séparée est indispensable pour qui veut développer des application importantes(en taille !), lorsque la taille des programmes augmente il devient nécessaire une organisation et un découpage, il existe différent niveau de découpage mais l'idée reste la même comme un programme principale fait appel à des sous programmes qui peuvent eux-mêmes faire appel à des sous programmes du programme principal ou de celui-ci et ainsi de suite. De plus certaines sous programmes pouvant servir dans d'autres programmes c'est le principe de la modularité.

5.2. Principes généraux :

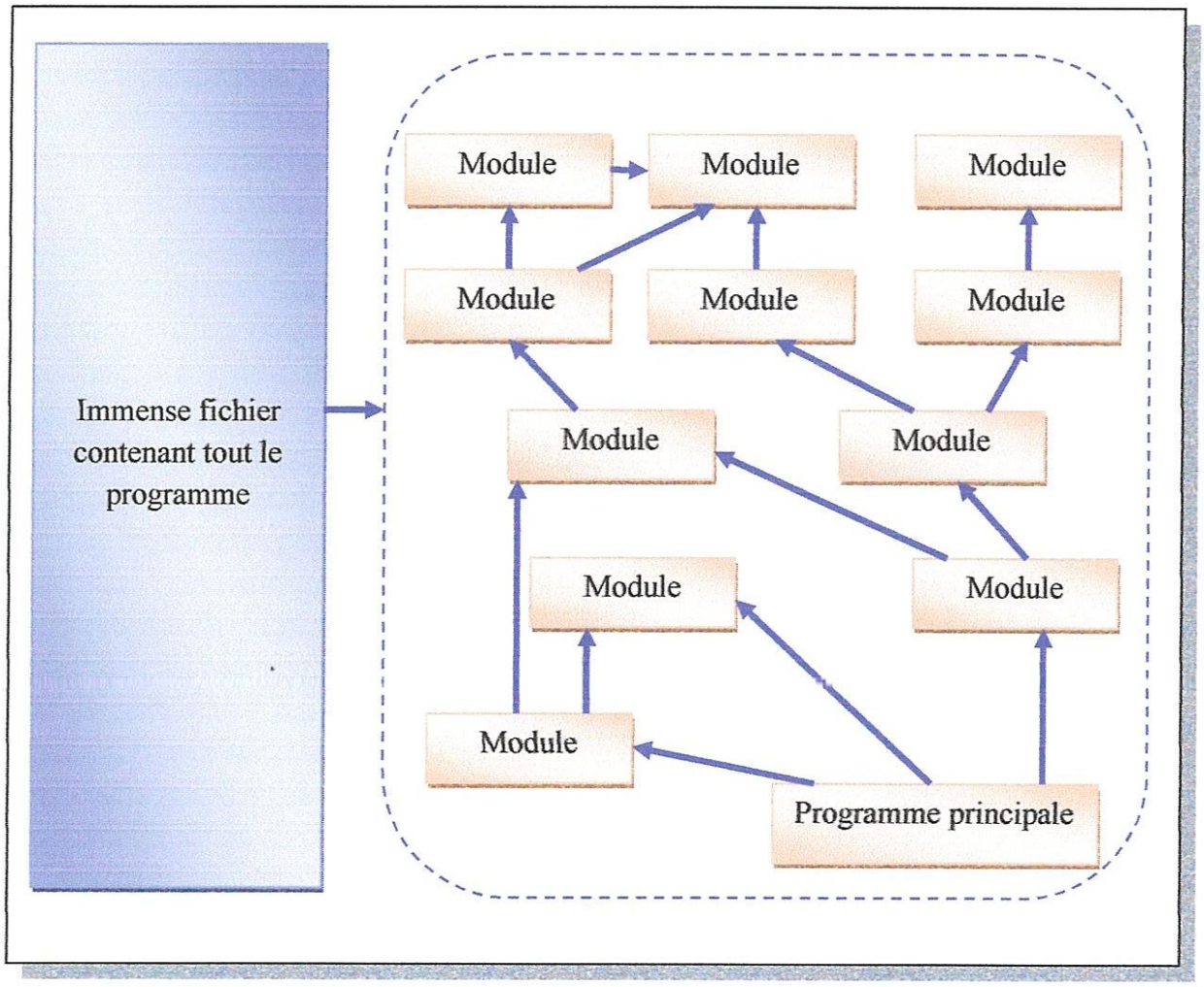

Figure 2.6. *Principes généraux de la compilation séparer.*

Exemple 2.6 :

La chaine de compilation du langage c.

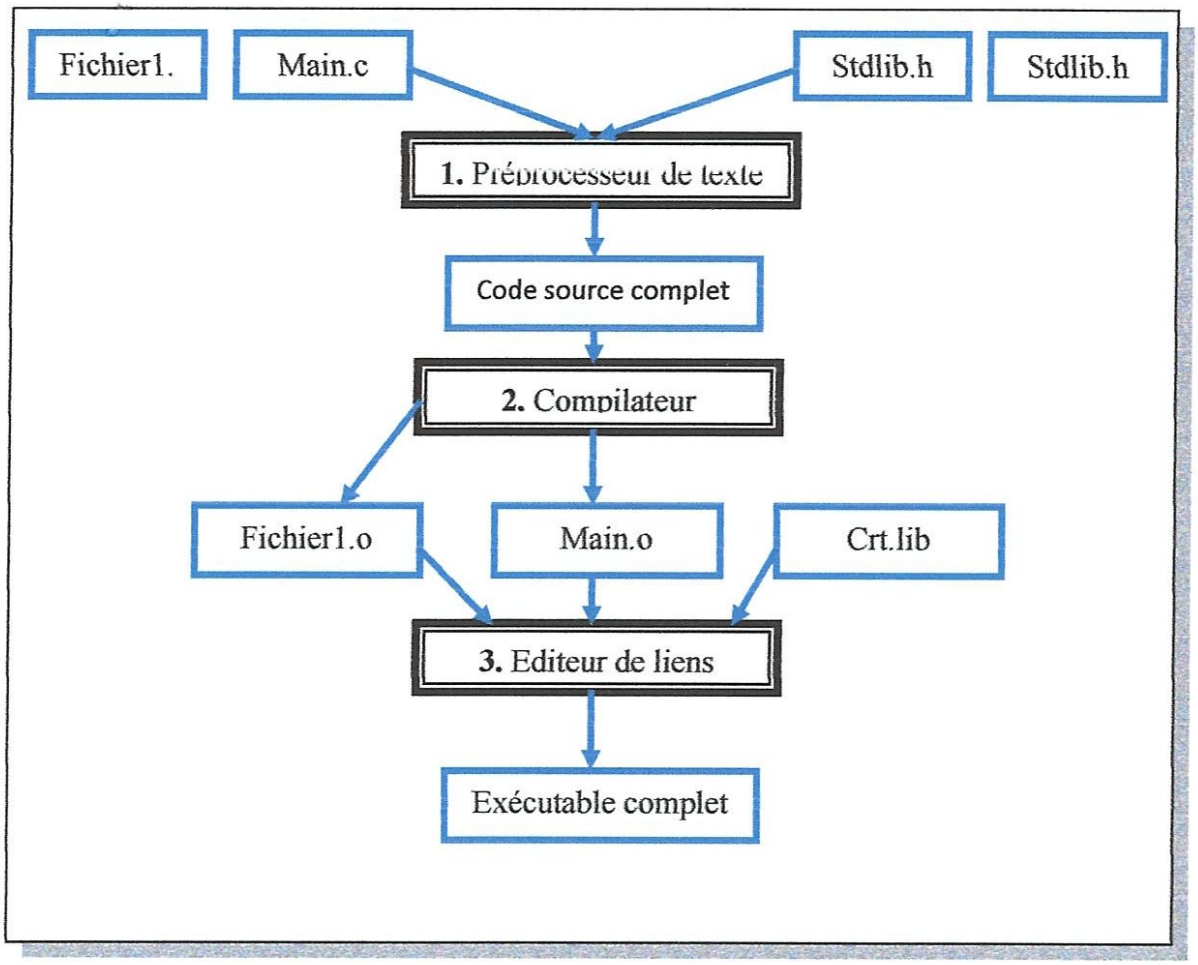

Figure 2.7. *Chaine de compilation du langage c.*

5.3. Définition d'un module :

On utilise généralement les modules pour regrouper un ensemble de déclarations (fonction, types, classes…..) centrées sur le même thème, chaque module est défini par deux fichier :

- Un fichier en tête (fichier.h) contenant toutes les déclarations de types et la déclaration d'entête de fonctions relatives à un type de donnée utilisé dans l'application.
- Un fichier source associe (fichier.c) contenant le corps des fonctions déclarées dans le fichier entêté (fichier.h).

Remarque :

Les modules possibles s'appellent les uns les autres. Si M1, utilise une fonction de M2, il n'a pas besoin de la fonction. Seulement de **l'assignation** de la fonction : nom, nombre de paramètres leur ordre, type du résultat.

Exemple 2.7:

Programme principale qui veut trier un tableau d'entiers par tri a bulles l'afficher.

- Module tab qui manipule les tableaux d'entiers.
- Module tri qui trie les tableaux par plusieurs méthodes.

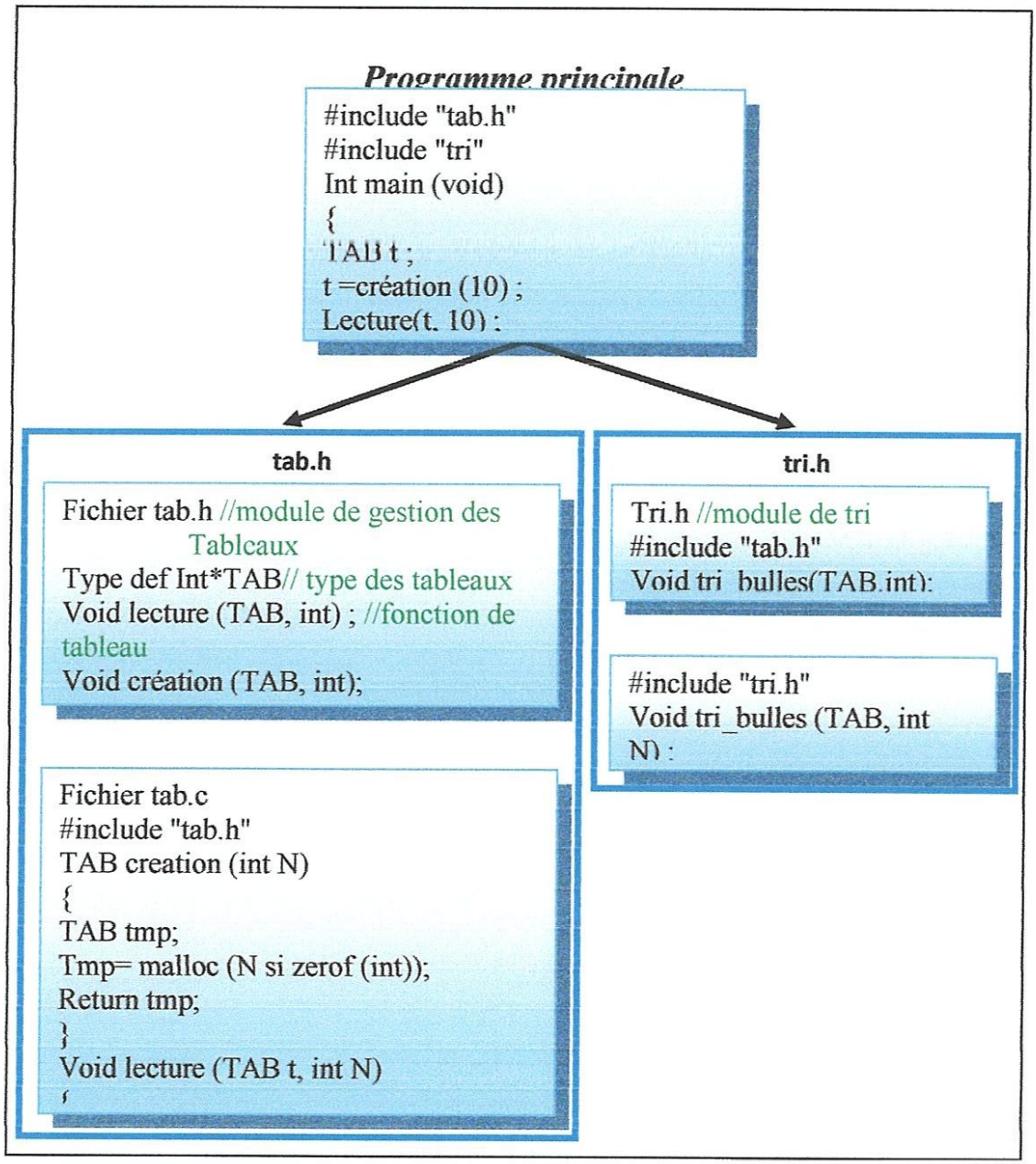

Figure 2.8. *Programme principale qui veut trier un tableau d'entiers par des modules.*

5.4. Critères :

- Décomposabilité : diviser d'une manière descendante les problèmes en sous-problèmes indépendants.
- Composabilité : favoriser le développement d'éléments bien définis qui peuvent librement être composés dans différents contextes. En général ce n'est pas simple avec l'approche descendante.
- Continuité : les changements mineurs dans les spécifications du problème doivent engendrer des changements mineurs des modules.
- Compréhensabilité : documentation du code des modules.
- Protection : support d'une défense contre les erreurs éventuelles lors de l'exécution. [KB12]

5.5. Propriétés :

- Correspondance directe : la structure du problème est reflétée sur la structure de la solution logicielle.
- Cohésion (intra-modulaire) forte : degré de liaison entre les éléments du module.
- Couplage (inter-modulaire) faible : dépendance entre modules
- Encapsulation : séparation entre la spécification du comportement du module (interface) et ses détails d'implantation.
- Principe ouvert/fermé : ouvert à l'extension futures mais fermé à la modification
- Principe du choix unique : dans un système qui supporte une liste d'alternatives, un seul module doit en connaître la logique. [KB12]

5.6. Avantage :

- Mise au point séparée.
- Réutilisabilité.
- Évolutivité.

❖ **Mise au point séparée** :

✓ Les modules sont écrits et testés indépendamment les uns des autres.

✓ Avantage : identification plus rapide des erreurs éventuelles (réduction du temps de développement)

❖ **Réutilisabilité**

- ✓ Un même module peut être utilisé par différents clients ou modules (temps de développement réduit).
- ✓ Aucun risque d'erreurs si la mise au point a été faite correctement ;

Exemple 2.8:

Dans une application universitaire, utilisation du module
Liste pour gérer des listes d'étudiants
Dans une application graphique 3D, utilisation du module
Liste pour gérer des listes d'objets 3D à afficher

❖ **Évolutivité**

- ✓ Un module indépendant peut être compris et amélioré facilement, éventuellement par un autre développeur que l'auteur du module (réduction du temps de développement)

- ✓ Un client peut facilement décider de faire appel à un module plutôt qu'un autre plus performant sans modifier le code de l'application. [KB12]

6. Conclusion :

Dans ce chapitre on a défini en générale le compilateur et son principe de fonctionnement.

Le compilateur est comme un traducteur d'un langage de programmation vers des instructions machine pour l'appliquer à n'importe quel traducteur.

Maintenant en informatique Les compilateurs majoritairement utilisés sont basés sur un principe de compilation séparée, Chaque unité de code est compilée séparément.

La compilation séparée est particulièrement importante dans le développement de grands projets, l'idée de modularité est efficace.

Chapitre 3 :
conception et implémentation

1. Introduction :

Dans ce chapitre Nous allons aborder la partie pratique en présentant la démarche que nous avons suit pour réaliser notre système LOTOS ainsi que les technologies et les outils utilisés, Le but est de présenter la conception et l'implémentation globale de notre projet et la techniques adoptées pour l'implémentation. D'abord, nous commencerons par ; déterminer les objectifs de notre travail, ensuite nous allons présenter son architecture globale et nous terminerons par la description et la présentation du fonctionnent du logiciel.

2. Objectif du travail :

Dans notre projet on a essayé de créer un nouveau compilateur d'un langage de haut niveau(**LOTOS**) qui contient de plus que l'ancien la compilation modulaire.

3. Outils de développement:

Pour réaliser notre logiciel, nous avons utilisé des outils de développement à savoir :

3.1.Le langage de développement (Delphi) :

Delphi est un environnement de programmation visuel orienté objet pour le développement rapide d'applications RAD (Rapid Application Development). En utilisant Delphi, vous pouvez créer des applications Microsoft Windows 95, Windows 9 et Windows NT très efficaces, avec un minimum de codage manuel. Le Delphi fournit tous les outils qui sont nécessaires pour développer, tester, déboguer et déployer des applications, incluant une importante bibliothèque de composants réutilisables, un ensemble d'outils de conception, des modèles d'applications et de fiches, ainsi que des experts de programmation. Ces outils simplifient le prototypage et réduisent la durée du développement. [BA9900]

3.2. Alpha skin :

Skin est un composant qui permet d'améliorer une application Delphi en lui ajoutant une interface graphique dont l'utilisateur peut modifier l'apparence librement. Grâce à ce composant, le développeur n'a pas une ligne de code à écrire pour obtenir cet effet. [JD00]

3.3. SynEdit :

SynEdit est un champ de saisie multi-ligne avancée, pour Borland Delphi (C + + Builder travaille principalement, mais non pris en charge). Il supporte la coloration syntaxique, mot-emballage, la complétion de code, des composants de modèle. [JD00]

4. Conception global:

Le schéma (3.1.) représente l'architecture globale de Basic LOTOS, dans laquelle, vous trouvez des composants de base : Editeur textuel, compilateur LOTOS.

Figure.3.1. *Conception globale.*

Dans ce qui suit, nous allons présenter la conception détaillée de chaque composant constituant notre logiciel.

4.1. Conception de l'éditeur de texte :

L'éditeur de texte est un outil destiné à la création et l'édition des fichiers textes.

Dans notre logiciel nous avons créé éditeur de spécification et traitement de **Basic LOTOS** pour un système complet et paquetage.

Un paquetage se partage en deux parties spécification et traitement de Basic LOTOS, le première pour la spécification (Contenant toutes les déclarations d'entête des processus utilisé dans l'application) et l'autre pour les l'implémentation (Contenant le corps des processus déclarés dans l'entête).

4.2. Conception du compilateur Basic LOTOS :

Dans les années 1950, écrit un compilateur est très difficilement. Il a fallu 18 ans pour mettre en œuvre le premier compilateur Fortran [BBB57].Mais les progrès dans la théorie des compilateurs et le développement de nombreux outils de compilation ont grandement simplifié la tâche.

Un compilateur est composée par huit modules principaux.**la partie analyse** d'un compilateur est composée de trois modules : analyse lexicale, analyse syntaxique, analyse sémantique.**la partie synthèse** est formée de trois modules : génération de code intermédiaire, optimisation de code, et génération de code objet.

Les deux modules restants : gestion de la table des symboles et traitements des erreurs peuvent être utilisés à tout moment par les autres modules.

Dans notre projet nous allons nous intéressé à la partie analyse qui passe par les phases suivantes : analyse lexical, analyse syntaxique, analyse sémantique pour la spécification de **Basic LOTOS.**

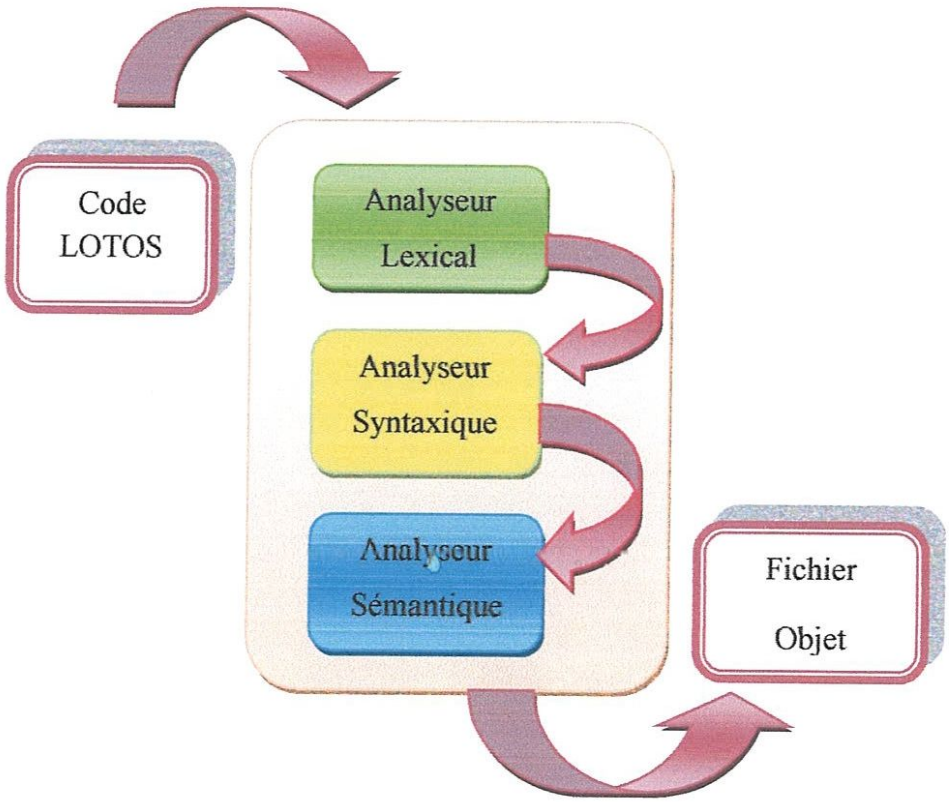

Figure .3.2.*la partie analyse du compilateur.*

4.3.1. 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. Voir la figure (3.3).

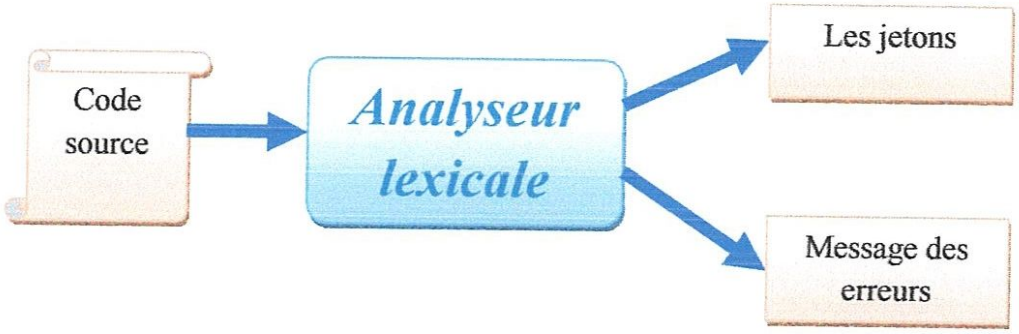

Figure .3. 3. *l'analyse lexicale.*

- **Unités lexicales du langage Basic LOTOS :**

Dans notre compilateur, nous ne distinguons pas entre les lettres minuscules et majuscules au but de simplifier la tâche.

Mots-clés: system, endsys, process, endproc, where, nil, exit, stop, hide, in, package, is, packagebody, end ,with,use.

Les opérateurs : "[ ]", "||", "|||", "| [liste porte]|", "[>", ">>", " ;".

Caractères spéciaux : "(", ")", "[", "]", ",", " :=", "." .

Identificateur : le nom du system, le nom du processus, le nom du porte, le nom de package le nom de package body.

- **Spécification des unités lexicales :**

Les tableaux suivants représentent les sous-jetons et les jetons de **Basic LOTOS** qui seront utilisé dans la programmation.

Les jetons	Description
s y s t e m	mot-clé
e n d s y s	mot-clé
p r o c e s s	mot-clé
e n d p r o c	mot-clé
w h e r e	mot-clé
n i l	mot-clé
s t o p	mot-clé
e x i t	mot-clé
h i d e	mot-clé
i n	mot-clé
p a c k a g e	mot-clé
p a c k a g e b o d y	mot-clé
i s	mot-clé
e n d	mot-clé
"[]"	opérateur
"||"	opérateur
"|||"	opérateur
"|["	opérateur
"]|"	opérateur
">>"	opérateur
" [>"	opérateur
(letter|letter1)(letter|digit|letter1)*	identificateur
" ("	caractère spéciaux
")"	caractère spéciaux
":="	caractère spéciaux
";"	Opérateur
" ["	caractère spéciaux

"]"	caractère spéciaux
"."	caractère spéciaux
","	caractère spéciaux

Tableau 3.2 : *Les unités lexicales "jetons".*

4.3.2. L'analyseur syntaxique de Basic LOTOS :

L'analyseur syntaxique ne lit pas directement le fichier en entrée. Il reçoit un flot d'unités lexicales par l'intermédiaire de l'analyseur lexical on obtient une liste qui contienne des unités appelé unités lexical. L'analyseur lexical nous a permis de détecter les erreurs lier au lexique vérifient ainsi leur valider d'un point lexical mais reste a savoir si cette chaine est valide d'un point syntaxique. Pour ce la on va maintenant la faire passer sur l'analyseur syntaxique. Et c'est à partir de ces unités lexicales que sont appliquées les règles de validation.

On va faire l'analyse en utilisons l'analyseur LR c'est-à-dire une analyse ascendante par décalage et réduction. Dans notre travail on a choisi l'analyseur LR pour sa simplicité et son algorithme de base.

Le principe de l'analyseur LR est très simple sa consiste à lui donner comme entré une chaine lexical celle qui est déjà le résultat de l'analyse lexical et la table d'analyse LR. Le résultat sera un message d'affirmation si l'analyse syntaxique est valide si non sa nous renvoi une erreur tout en indiquant cette erreur .Cette analyse suit un algorithme précis qui continuera a s'exécuter jusqu'à ce qu'il rencontre un état d'acceptation ou bien une erreur.

Figure .3. 4. *l'analyse syntaxique.*

- **Grammaire de Basic LOTOS :**

A partir des opérations de Basic LOTOS présenté dans Le Chapitre1 et nous allons construis deux grammaire un pour le système et deuxième pour paquetage.

<Grammaire lotos> ::= <entête> <corps>

< Entête > ::= <entêtepackage> **SYSTEM** <id> [<listeporte>]

<Corps> ::= := <expression> <corp> **ENDSYS**

< Entêtepackage > ::= <withuse> <listewithuse> | epsilon

<Withuse>::= <with> ;<use>; |<with>;

<With>::= **WITH** <liste>;

<Liste>::= <id> | <listecomposé> | <id> **,** <liste> | <listecomposé> **,** <liste>

<Listecomposé> ::= <id> **.** <id> | <id> **.** <listecomposé>

<Use> ::= **USE** <liste>;

<Listewithuse> ::= <withuse> <listewithuse> | <withuse>

<Listeporte> ::= <porte> | epsilon

<Expression> ::=<expression> <op> <expression> | <Id> **;** <expression> |

<listecomposé> [<listeporte>] | <id> [<listeporte>] | **HIDE** <porte> **IN** <Expression>

|STOP | NIL | EXIT | (<expression>)

<Corp> ::= <copr 1> | epsilon

<Porte> ::= <id> | <porte> **,** <id>

<Copr 1>| ::= WHERE <listeprocessus>

<Listprocessus> ::= <processus> <listeprocessus> | <processus>

<Processus> ::= **PROCESS** <id> [<listeporte>] := <expression> <corp> **ENDPROC**

<Op> ::= || | ||| | |[<listeporte >]| | >> | [> | [ ]

<Grammaire package> ::=<spécification> | <implémentation>

< spécification> ::=**PACKAGE** <id> IS <listeproc> **END** <id>

< implémentation> ::= **PACKAGEBODY**<id> **IS** <listeprocessus> END <id>

<listeproc> ::= <proc> <listeproc> | <proc>

<proc> ::= PROCESS <id> [<listeporte>]

Remarque1: On a remarqué que cette grammaire contient une factorisation qui ne pose pas un problème parce qu'on a utilisé la méthode ascendante LALR.

Remarque2: Cette grammaire est non ambiguë cela veut dire qu'elle ne pose pas un problème de conflit dans la table de LALR.

4.3.3. L'analyse sémantique :

Après l'analyse lexicale et l'analyse syntaxique, l'étape suivante dans la conception d'un compilateur est le vérificateur sémantique qui est associé pour donner un sens aux différentes phrases du texte source.

Et voici les différentes phases de notre vérificateur sémantique :

- **Vérificateur d'action dupliquée :**

Ce vérificateur test les doublons dans une liste d'action déclarées dans les paramètres du processus.

Exemple 1 :

```
SYSTEM serveur [client1, client2, client3, client2] :=
        client1 ; client2 ; exit  ENDSYS
```

=) erreur sémantique : action<<client2>> dupliquée.

- **Vérificateur d'action non déclaré :**

Cette vérificateur teste la liste d'action d'appelle avec la liste d'action déclarée d'un processus.

Exemple 2 :

```
SYSTEM serveur [client1, client2, client3, client2] :=
        client1 ; client6 ; exit ENDSYS
```

=) erreur sémantique : action <<client6>> non déclaré.

- **Vérificateur de déclaration de processus :**

Ce vérificateur test la liste d'action d'appelle avec la liste d'action déclarée d'un processus soit dans système ou bien dans l'appelle de package.

Exemple 3 :

<<Système>>

```
SYSTEM serveur [client1, client2, client3] :=
site1 [client1, client2] >> site2 [client3, client2]
WHERE
PROCESS site1 [c1, c2] :=  c1 ; c2 ; exit
 ENDPROC
ENDSYS
```

=) Erreur sémantique : processus <<site2>> non déclarée.

Exemple 4 :

<<Package>>

- **Code source de système**

```
SYSTEM serveur [client1, client2, client3, client2] :=
  site1 [client1, client2] >> site2 [client3, client2]
ENDSYS
```

- **Code source de package**
  - **Specification:** PACKAGE M IS
    PROCESS site1 [c1, c2];
    END M
  - **Implementation:** PACKAGEBODY M IS
    PROCESS site1 [c1, c2] :– c1 ; c2 ; exit
    ENDPROC ;
    END M

Erreur sémantique : processus <<site2>> non déclarée dans le package M.

➢ **Vérificateur de nombre de paramètre :**

Ce vérificateur test 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.

Exemple 5

<<Système>>

```
SYSTEM serveur [client1, client2] :=
site1 [client1]; exit
WHERE
PROCESS site1 [c1, c2]:= c1; c2; exit
ENDPROC
ENDSYS
```

Exemple 6 :

- **Code source de système**

```
SYSTEM serveur [client1, client2] := site1 [client1] ;exit
ENDSYS
```

- **Code source de package**
  - **Spécification :**

```
PACKAGE M IS
PROCESS site1 [c1, c2] ;
END M
```

  - **Implementation:**

```
PACKAGEBODY M IS
PROCESS site1 [c1, c2] := c1 ; c2 ; exit
ENDPROC ;
END M
```

➢ **Vérificateur le nom de package :**

Ce vérificateur test le nom de package

Exemple 7:

- **Spécification:**

```
PACKAGE M IS
PROCESS site1 [c1, c2] ;
PROCESS site2 [c2, c1] ;
END N
```

Erreur sémantique : N déférente de M.

➢ **Définition de processus dupliqué** :

Nous allons expliquer les différents cas où le processus ne doit pas être dupliqué. **Premier cas les frères** : deux processus frères ne doit pas être dupliqué

Exemple 8 :

<<Système>>

```
SYSTEM serveur [client1, client2] := site1 [client1] [> site2 [client2]
```

WHERE

Process site1 [c1] :=c1; exit endproc

Process site2 [c2] := c2 ; exit endproc

Process site2 [c2] :=c2 ; exit endproc

Endsys

Exemple 9 :

- **Code source de système :**

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

ENDSYS

- **Code source de package :**
  - **Spécification:**

PACKAGE M IS

PROCESS site1 [c1, c2] ;

PROCESS site2 [c2, c1]

END M

  - **Implémentation:**

PACKAGEBODY M IS

PROCESS site1 [c1, c2] := c1 ; c2 ; exit ENDPROC ;

PROCESS site2 [c2, c1] := c2 ; c1 ; exit ENDPROC ;

PROCESS site2 [c2, c1] := c2; c1 ; exit ENDPROC

END

4.4. Conception Fichier (.OI, .OS) produit par la compilation :

Après la compilation de paquetage et système introduit des fichiers stockés aux extensions différente ce forme une liste chainée.

D'après les packages :

- ✓ Un fichier en tête (fichier. OS) contenant toutes les déclarations d'entête de processus utilisé dans le système.
- ✓ Un fichier source associe (fichier. OI) contenant le corps des processus déclarés dans le fichier entête (fichier. OS).
- D'après le système : Un fichier source (fichier objet) contenant les spécifications de système.

5. Présentation de l'application développée :

Dans cette partie, nous présentons notre application développée dans le cadre de développement d'un compilateur LOTOS. Ce compilateur est exécuté dans une fenêtre.

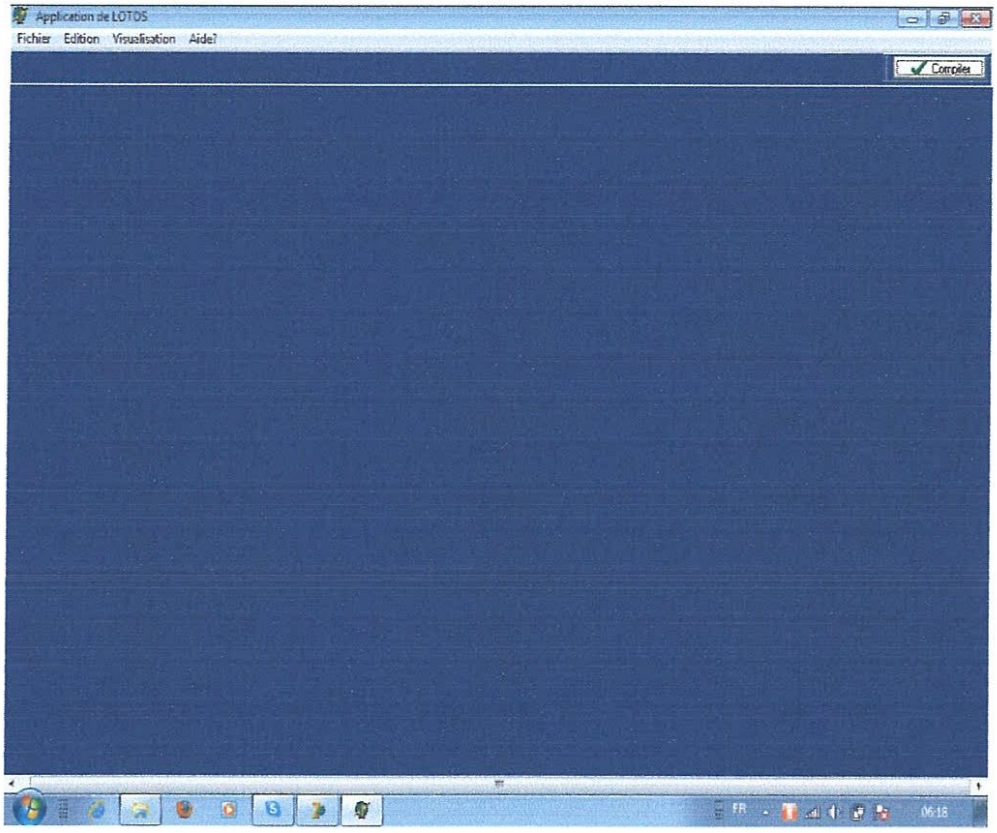

Figure3.5 : *La fenêtre principale.*

La barre de menu est constituée de plusieurs menus : Fichier, Edition, visualisation, et Aide. Le menu Fichier est utilisé pour nouveau système ou paquetage et ouvrir, enregistrer ou enregistrer sous un fichier. Le menu Edition regroupe les fonctions ordinaire de menu Edition comme Copie, Couper et Coller. Et utiliser un buttons pour compiler une spécification dc configuration de LOTOS .les buttons de package utilisés pour faciliter l'accès aux différentes fonctions

L'exécution de la fonction Nouveau à partir de menu Fichier permet l'apparition d'un éditeur de texte de système et paquetage pour l'introduction d'une spécification LOTOS.

6. Conclusion :

Dans ce chapitre, nous avons introduit la partie pratique de notre travail, en d'autres termes, nous avons présenté le schéma de fonctionnement global, et le détaille du logiciel, ainsi que ses différents composants à savoir ; éditeur de texte et compilateur de Basic LOTOS, qui permet de vérifier la spécification qui est réalisé dans l'éditeur.

Le chapitre suivant est l'étude de cas de notre logiciel.

Chapitre 4 :
Etude de cas

1. Introduction :

Tout au long de ce chapitre, nous allons modéliser le problème classique du diner du philosophe via notre application LOTOS. Tout en présentant la manière prévue pour représenter cette séquence d'exécution.

2. Problème du diner 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 plat de spagetti, et entre chaque deux plats consécutifs, on a une fourchette (baguette) (voir figure (4.1)). 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 plat (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.

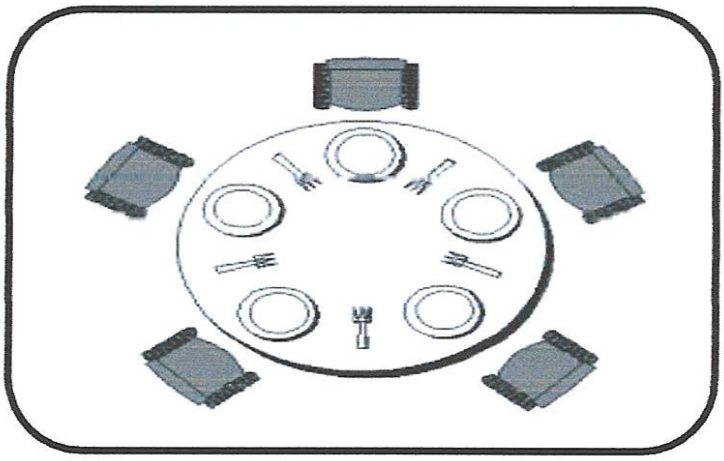

Figure .4.1.*le diné de philosophes.*

3. 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.**

Le system contient deux composants : philosophe et fourchette. Grâce à notre travaille qui permet une description complète 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. Dans la prochaine section on va faire une modalisation du problème dans notre application.

4. La modélisation du problème :

La modélisation du modèle passe par 3 phases :

1. La description informelle du système (en boite noire).
2. La description formelle du système (modélisation dans LOTOS).
3. L'analyse et l'exécution du système (code source généré).
4.

5. La description informelle du système :

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 (4.2)

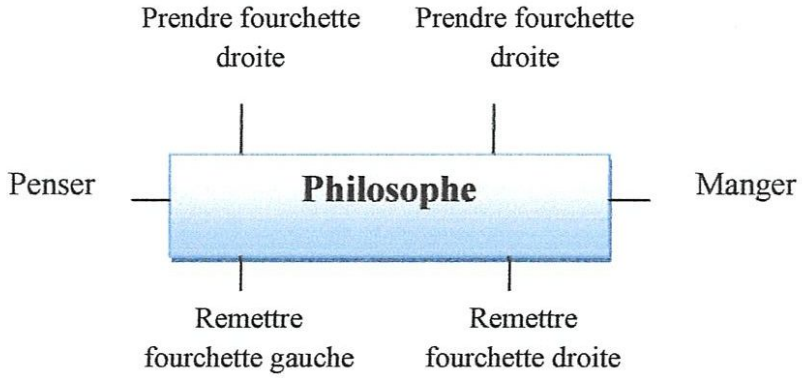

Figure.4.2.*Boite noire de philosophe.*

❖ **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 (4.3)

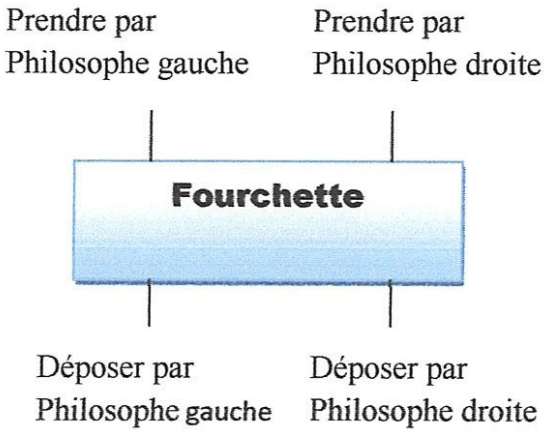

Figure.4.3. *Boite noir de fourchette.*

❖ **Les interactions entre les composants :**

Dans la figure (4.4) on a présenté l'interaction entre deux philosophes et fourchette.

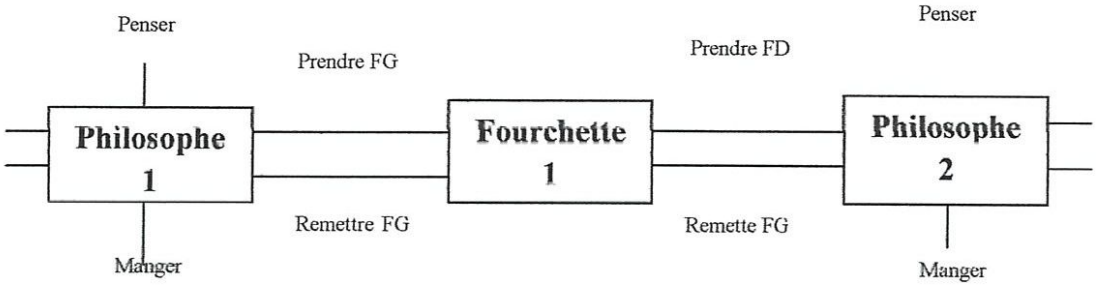

Figure.4.4. Les *interaction entre deux philosophes et une fourchette.*

❖ **Ordonnancement des actions :**

Dans l'ordonnancement il faux définir l'ordre d'exécution des actions, comme il est montré dans la figure (4.4) et (4.5).

- **Philosophe :**

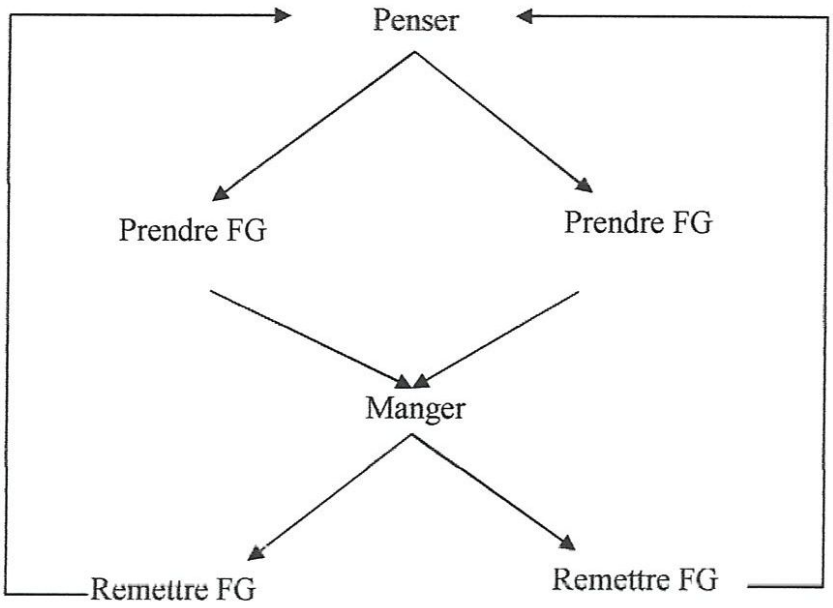

Figure.4.5. *Ordonnancement des actions de Philosophe.*

- **Fourchette :**

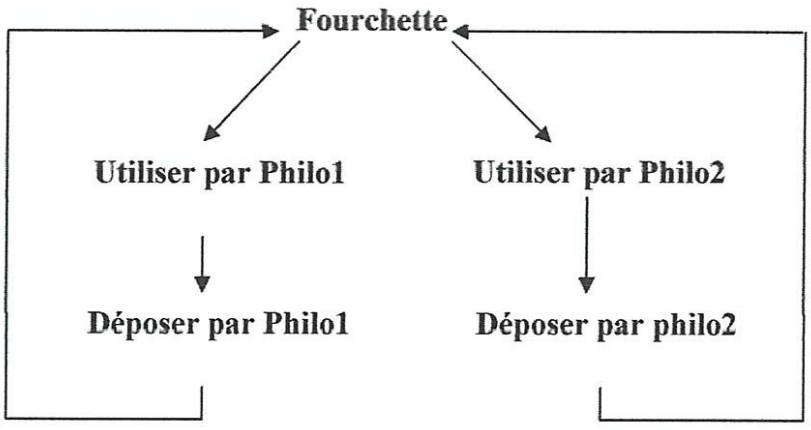

Figure.4.6. *Ordonnancement des actions de Fourchette.*

6. La description formelle du system :

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

❖ **Modélisation du diner du philosophe dans LOTOS :**

Le philosophe et le fourchette sont des processus.

➢ **La modélisation du philosophe :**

✓ **La déclaration :** le processus philosophe fait des interactions avec six actions Philo_Prend_fg, Philo_pose_fg, Philo_Prend_fd, Philo_pose_fd,Philo_pense,Philo_mange.

✓ **Expression de comportement :**

Le philosophe avant qu'il soit affamé, il doit penser tout d'abord, puis il prend sa fourchette gauche puis sa fourchette droite ou contraire. Pour cet effet, on peut constater que le philosophe n'arrivera pas à manger sauf s'il possède ses deux fourchettes en même temps. Dans notre logiciel on doit écrire l'expression de comportement dans la forme de déclaration de la fourchette :

Philo_Pense ;(Philo_Prend_fg ;Philo_manger ;Philo_pose_fg ;exit)

|[Philo_manger]|

(Philo_Prend_fd ;Philomanger ;Philo_pose_fd ;exit)

>>

Philosophe [Philo_Prend_fg, Philo_pose_fg, Philo_Prend_fd, Philo_pose_fd, Philo_Pense, Philo_manger]

➢ **La modélisation de la fourchette :**

✓ **La déclaration :** le processus fourchette fait des interactions avec quatre actions : Prise_par_phD,Pose_par_phD,Prise_par_phG ,Pose_par_phG.

Dans notre logiciel LOTOS on va déclarer ses actions dans la forme de déclarationdu fourchette par les portes.

✓ **Expression de comportement :**

Le comportement d'une fourchette est exprimé de façon que l'un des philosophe se trouvant à droite ou à gauche da la fourchette doit le prendre et posée par lui mémé, puis on fait a chaque alternative (philosophe gauche ou droite) des appelles récursives pour initialiser

l'utilisation de la fourchette lorsqu'un philosophe la pose. C'est-à-dire elle devient disponible et peut être utilisé par un autre philosophe.

Dans notre logiciel on doit écrire l'expression de comportement dans la forme de déclaration de la fourchette :

(Prise_Par_phD ; Poser_par_phD ; Fourchette1 [Prise_par_phD, Poser_par_phD, Prise_par_phG, Poser_par_phG])

[ ]

(Prise_par_phG ; Poser_par_phG ; Fourchette1 [Prise_par_phD, Poser_par_phD, Prise_par_phG, Pose_par_phG])

❖ **Ordonnancement des processus :**

Les processus 'philosophe 'et 'fourchette' sont exécutés en parallèle et se synchronisent sur des portes de synchronisations, pour exprimer ce comportement, nous utiliserons un opérateur de composition parallèle '|[]|' offert par **Basic LOTOS** comme un mécanisme de rendez-vous

Dans cette spécification on va modéliser deux philosophes et deux fourchettes :

La philosophe {1} contient des actions suivantes :

Ph1_Prendre_f2, Phi1_pose_f2, Ph1_Prendre_f1, Ph1_Pose_f1,Ph1_Pense,Ph1_Manger

La fourchette {1} contient des actions suivantes :

Ph1_Prendre_f1, Ph1_pose_f1, Ph2_Prendre_f1, Ph2_pose_f1

La philosophe {2} contient des actions suivantes :

Ph2_Prendre_f2, Phi2_pose_f2, Ph2_Prendre_f1, Ph2_Pose_f1, Ph2_Pense, Ph2_Manger

La fourchette {2} contient des actions suivantes :

Ph2_Prendre_f2, Ph2_pose_f2,Ph1_Prendre_f2,Ph1_pose_f2

Le philosophe {1} est synchronisé avec **la Fourchette {1}** par les portes {Ph1_Prendre_f1, Ph1_pose_f1, Ph1_Prendre_f2, Ph1_pose_f2} et

La Fourchette {1} est synchronisée avec **le philosophe {2}** par les portes {Ph2_Prendre_f1, Ph2_pose_f1, Ph2_Prendre_f2, Ph2_pose_f2} et

La philosophe {2} est synchronisée avec **la Fourchette {2}** par les portes {Ph2_Prendre_f2, Ph2_pose_f2, Ph1_Prendre_f2, Ph1_pose_f2}

❖ **L'exécution de LOTOS :**

L'exécution de **LOTOS** permet de compiler séparément le code du problème de diner du philosophe. Dans cette partie on va spécifier le code par le philosophe et la fourchette et se spécification complète.

❖ **Le code de paquetage philo :**

✓ **Partie spécification :**

```
Package philo is
Process Philosophe [Philo_Prend_fg, Philo_pose_fg, Philo_Prend_fd,
Philo_pose_fd, Philo_Pense, Philo_manger] ;
Processe Fourchette [Prise_par_phD, Pose_par_phD, Prise_par_phG,
Pose_par_phG]
End philo
```

✓ **Partie implémentation :**

```
Packagebody philo is
Process Philosophe [Philo_Prend_fg, Philo_pose_fg, Philo_Prend_fd,
Philo_pose_fd, Philo_Pense, Philo_manger] :=
Philo_Pense ;(Philo_Prend_fg ;Philo_manger ;Philo_pose_fg ;exit)
        |[Philo_manger]|
(Philo_Prend_fd ;Philomanger ;Philo_pose_fd ;exit)
        >>
Philosophe [Philo_Prend_fg, Philo_pose_fg, Philo_Prend_fd,
Philo_pose_fd, Philo_Pense, Philo_manger] endproc ;

Processe Fourchette [Prise_par_phD, Pose_par_phD, Prise_par_phG,
Pose_par_phG] :=
(Prise_Par_phD ; Poser_par_phD ; Fourchette1 [Prise_par_phD,
Poser_par_phD, Prise_par_phG, Poser_par_phG])
        []
(Prise_par_phG ; Poser_par_phG ; Fourchette1 [Prise_par_phD,
Poser_par_phD, Prise_par_phG, Pose_par_phG]) Endproc
End philo
```

➢ **Ordonnancement et les appels des processus**

```
With philo ;

Use philo ;

SYSTEM
philo_spec[Ph1_Pense,Ph1_Manger,Ph2_Pense,Ph2Manger,Ph1_Prendre_f1,Ph1
_pose_f1,Ph1_Prendre_f2,Ph1_pose_f2,Ph2_Prendre_f1,Ph2_pose_f1,Ph2_Pren
dre-f2,Ph2_pose_f2] :=

Philosophe [Ph1_Prendre_f2, Phi1_pose_f2, Ph1_Prendre_f1, Ph1_Pose_f1,
Ph1_Pense,          Ph1_Manger]

            |[ Ph1_Prendre_f1,Ph1_pose_f1,Ph1_Prendre_f2,Ph1_pose_f2]|

Fourchette [Ph1_Prendre_f1, Ph1_pose_f1, Ph2_Prendre_f1, Ph2_pose_f1]

            |[ Ph2_Prendre_f1, Ph2_pose_f1, Ph2_Prendre_f2, Ph2_pose_f2]|

Philosophe [Ph2_Prendre_f2, Phi2_pose_f2, Ph2_Prendre_f1, Ph2_Pose_f1,
Ph2_Pense, Ph2_Manger]

            |[ Ph2_Prendre_f2, Ph2_pose_f2, Ph1_Prendre_f2, Ph1_pose_f2]|

Fourchette [Ph2_Prendre_f2, Ph2_pose_f2, Ph1_Prendre_f2,Ph1_pose_f2]

ENDSYS
```

Dans cette partie nous allons voir vérifier scénarios possible que notre logiciel pourra rencontrer lors de l'exécution. Ces différents scénarios sont appliqués sur la spécification d'un du problème de diner du philosophe.

Après avoir introduit la spécification d'un LOTOS, on peut confronter avec l'une des situations suivantes :

1^er^ cas **:** exécution sans erreurs : dans ce cas l'exécution de compilateur ne détecte aucune erreur dans la spécification LOTOS soit le système ou package. Comme il est montré dans la partie réservé aux erreurs aucune erreur apparu dans notre éditeur.

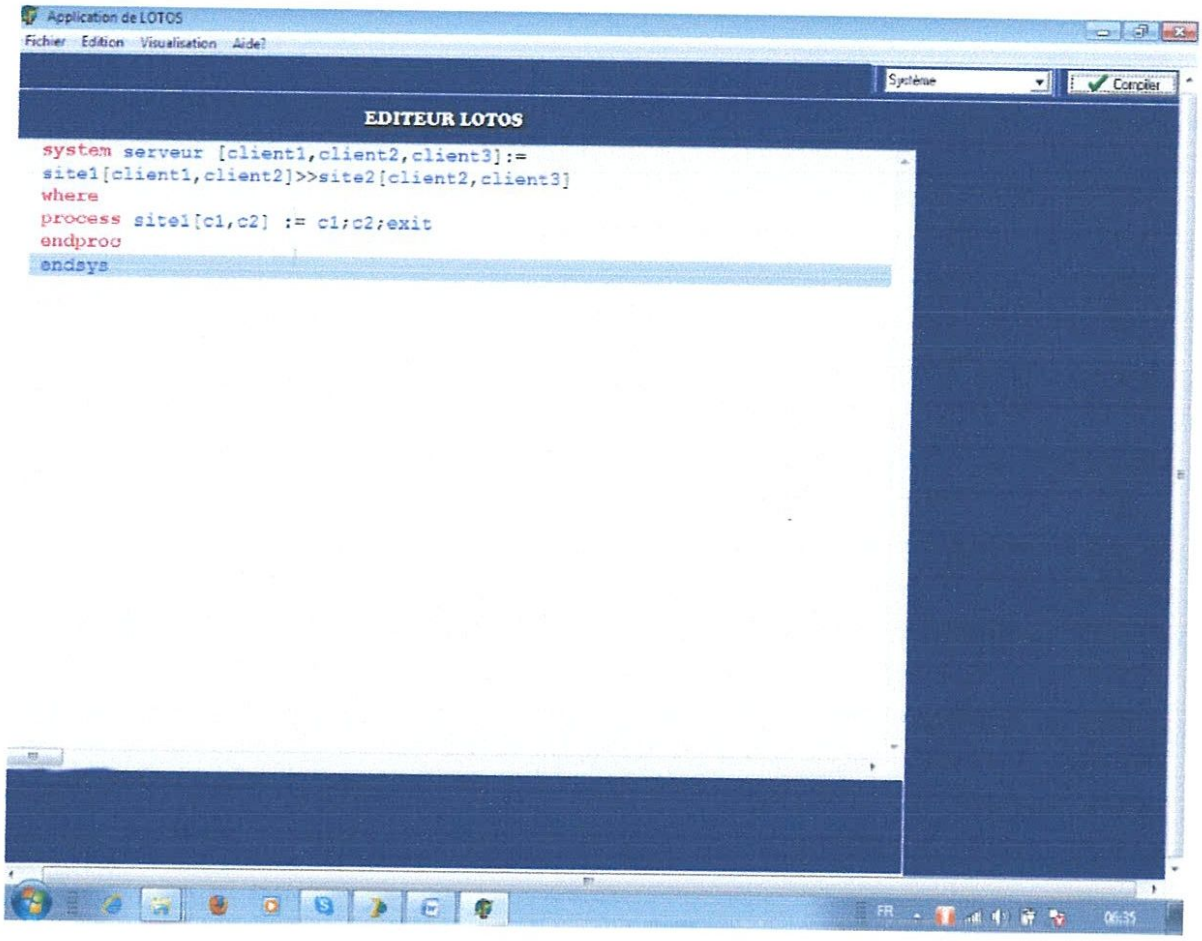

Figure4.7.*le cas d'absence d'erreurs du système.*

2 ème cas : exécution avec erreur lexicale : dans ce cas nous montrons quelques erreurs lexicales. L'erreur lexicale «jetons inconnue » dans le nom du système

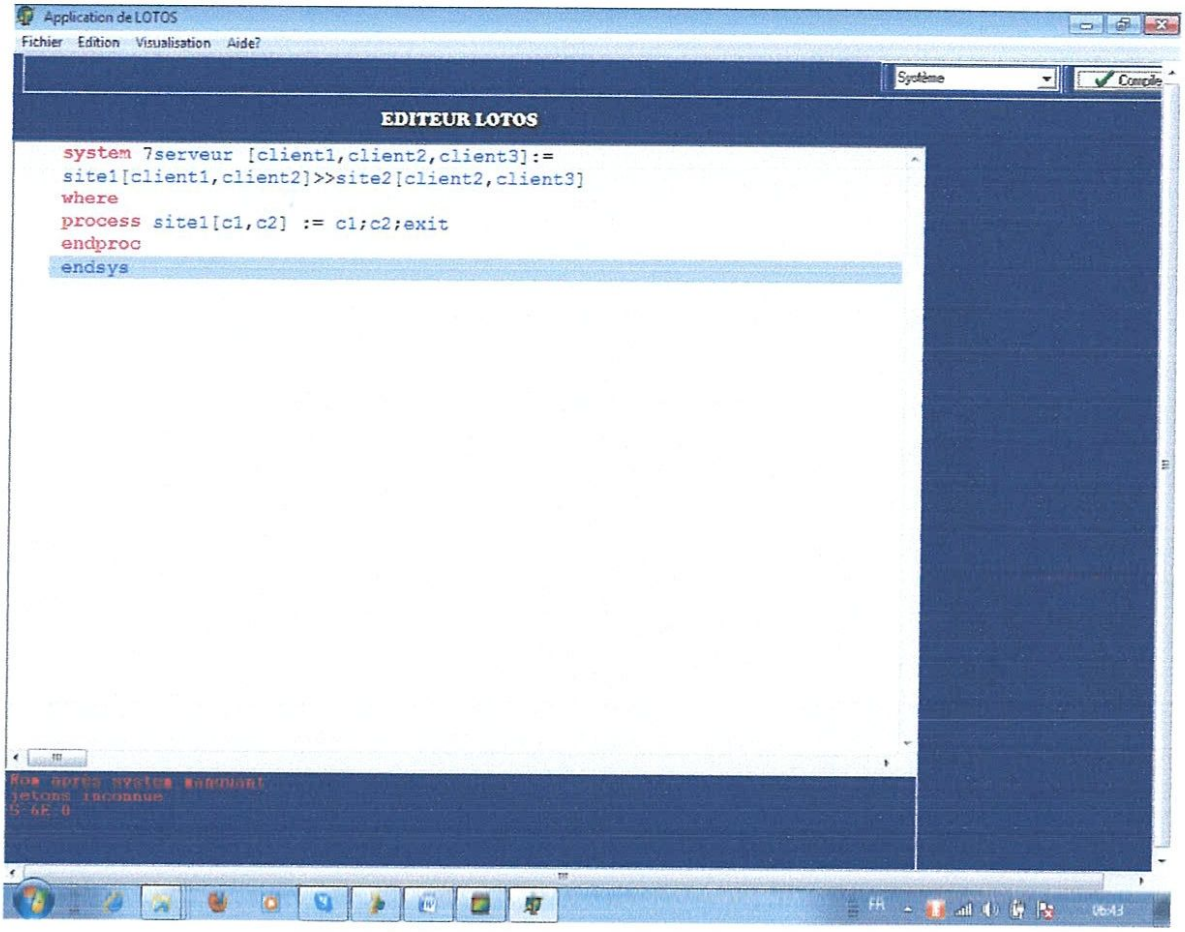

Figure.4.8.*cas d'erreurs lexicales.*

3 ème cas : exécution avec erreur syntaxique : dans ce cas nous confrontons avec une erreur syntaxique sui consiste à la suppression de le nom du système. Malgré que le compilateur détecte que toutes les unités lexicales soient valides. Cependant le manque le nom du système provoque une erreur syntaxique à cause de non respect de grammaire.

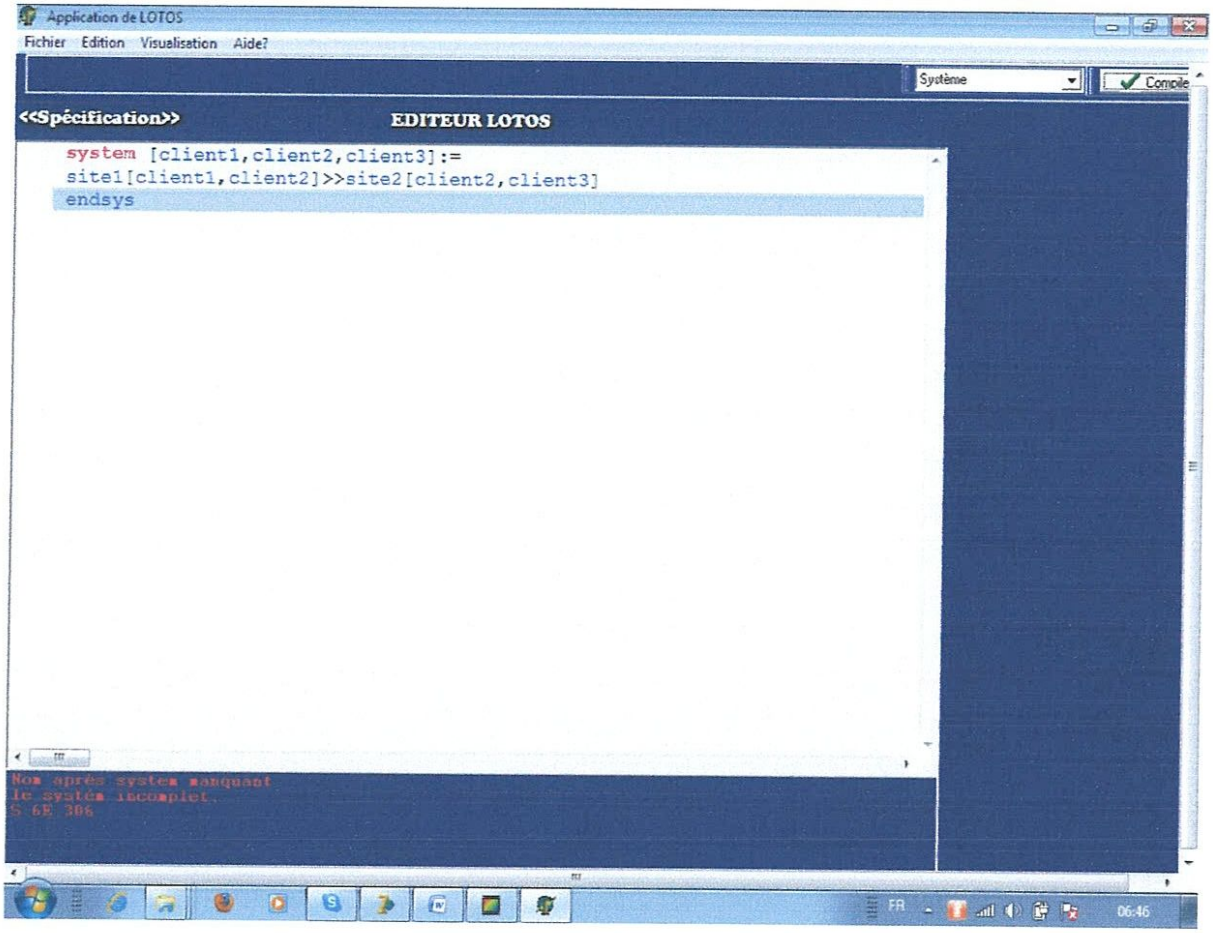

Figure4.9. *Cas d'une erreur syntaxique.*

4 ème cas : exécution avec erreur sémantique : dans ce cas toutes les unités lexicales sont valides. En plus, la spécification respecte la syntaxe. Cependant, on trouve dans cette spécification le nombre de les portes incorrecte.

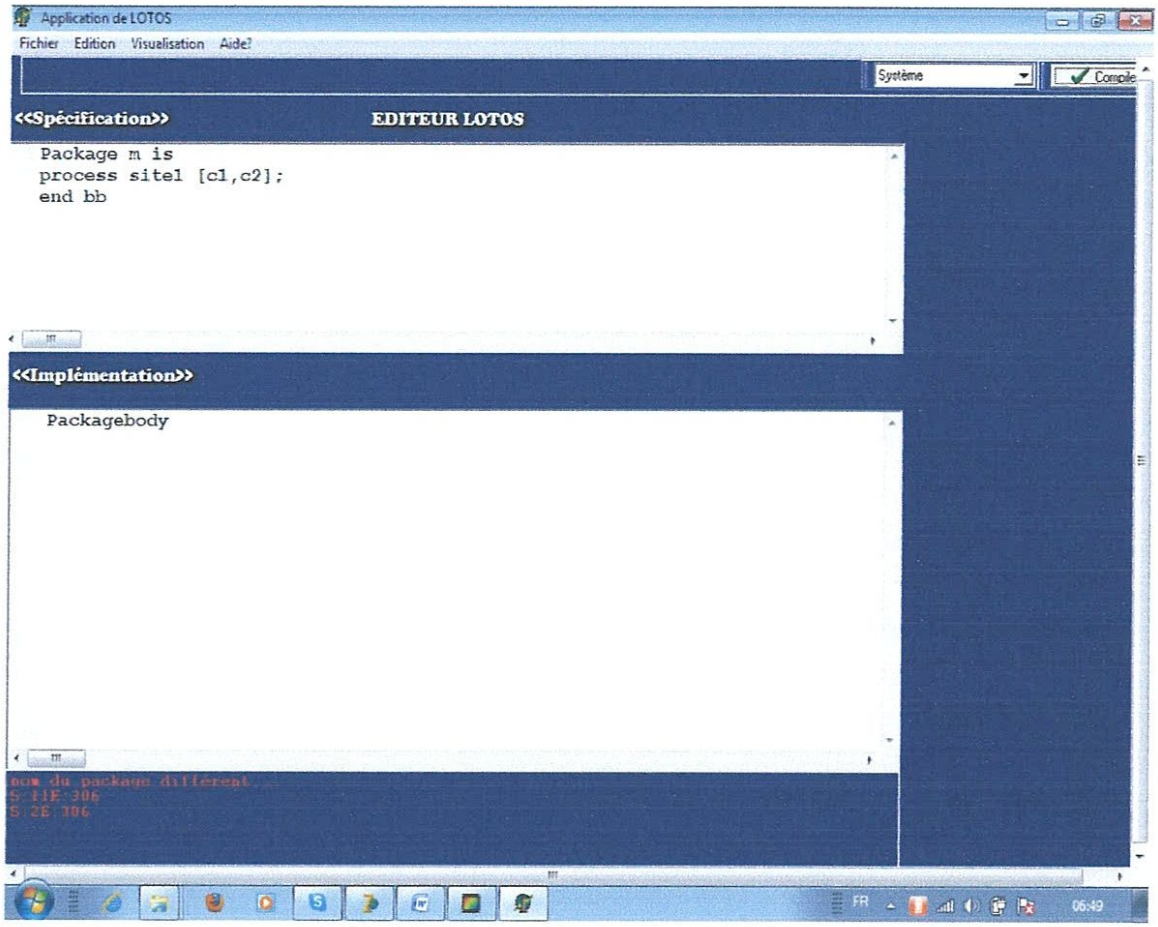

Figure4.10.*cas d'une erreur sémantique*

7.Conclusion :

Dans ce chapitre nous avons basé sur la description de notre application et nous projetons les différents scénarios possibles sur une étude de cas.

Le but est de mettre en valeur le travail fait, en montrant l'importance de la compilation modulaire des systèmes complexes.

conclusion générale

Conclusion générale :

Notre travail se compose d'une étude théorique et une autre pratique, la première concerne le langage de description formelle (basic lotos) et la deuxième concerne une présentation de la compilation modulaire. Dans le premier chapitre, on a spécifié une technique de description formelle qui s'appelle LOTOS qui pourrait être employée pour la description de systèmes complexes, et par la suite nous avons étudié la partie contrôle de LOTOS qui s'appelle Basic LOTOS qui cherche à exprimer les différents comportements entre les processus (parallélisme, choix.... etc.).Dans le chapitre 2, nous avons décrit une technique de compilation modulaire.

Le chapitre 3 a été consacré pour la présentation de la conception et l'implantation de notre outil.

Le chapitre4 a fait le sujet d'une étude de cas, visant à valoriser le travail fait ; on a par l'occasion étudiée le problème des philosophes, où on a vu la démarche à suivre pour le modéliser dans notre application LOTOS et comment exploiter les informations dans ce logiciel

Perspectif :

Comme perspective à notre, nous proposons les points suivants :

- ✓ une méthode formelle peut être définie comme une technique permettant de spécifier un système de manière non ambiguë et permettant de raisonner par le biais de preuves ou de manipulations mathématiques.
- ✓ Les méthodes formelles sont généralement divisées en deux grandes familles la première famille comporte les méthodes dites constructives et la seconde les méthodes par vérification de modèles. La seconde famille va vérifier la correction de l'expression de ce produit sous forme d'automates (par exploration des états atteignables).
- ✓ La modélisation formelle basée sur une approche de la modularité permet séparer les fonctionnalités du projet en unités le moins possible interdépendantes, on simplifie la mise au point du logiciel qui se fait module par module.
- ✓ • La compilation est beaucoup plus rapide, car seuls les modules qui ont été modifiés sont compilés.
- ✓ • Certains modules s'ils sont bien conçus peuvent être réutilisés dans d'autres projets.

Bibliographie

Bibliographie :

[90788] ISO 9074. "Estelle, a Formal Description Technique Based on an extended State Transition Model". ISO (November 1988).

[AB06] Adel BENAMIRA"Thème Vérification des équivalences de comportemen des systèmes concurrents" Université Mentouri — Constantine.12/06/2006

[Arn92] A .Arnold. Systèmes de transitions finis et sémantique des processus commu-nicants. (1992).

[BA9900] Bounceur Ahcène. "Cours Delphi de Bounceur Ahcène". Le nécessaire pour devenir professionnel. Année 1999-2000.

[BB87] T. Bolognesi and E. Brinksma. "Introduction to the ISO Specification Lan-guage LOTOS", volume 14. Computer Networks and ISDN Systems (1987).

[Bed87] M. A. Bednarczyk. "Categories of Asynchronous Systems". PhD thesis, Univ Sussex (1987 1987). Available as CS R 1/88.

[BK85] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction.TCS37, 77—121 (1985).

[CA9900] Christian Attiogbé. "introduction aux algébres de processus et LOTOS ".1999/2000

[CCI88] CCITT88. SDL, recommendation z.100-z.104. CCITT(1988).

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of

synchronization skeletons using branching time temporal logic. In "Logics of Programs Work-shop", volume 131, pages 52—71. LNCS (May 1981).

[CES83] E.M.Clarke,E.A.Emerson,andA.P.Sistla. Automatic verification offinite-state concurrent systems using temporal logic specification: A practical approach. In "10th ACM Symp. Principles of Programming Language", pages.117—126, Austin, Texas (January 1983).

[EB1213] Elise Bonzon. "Théorie des langages et compilation". « Université Paris Descartes »- 2012-2013

[EH86] E. A. Emerson and J. Y. Halpern. "sometims" and "not never" revisited: On branching versus linear time temporal logic.Journal of the ACM33(1), 151—178 (1986).

[FC10] Enseignant : Mr. Faiez CHARFI " Techniques & Outils de compilation". « Université Technologique Privée de Gabés ». « Niveau :- ING2 – Informatique » .Octobre 2010

[GD08] Gwenael Delaval. "Répartition modulaire de programmes synchrones". 1er juillet 2008

[HF10] H. Foucha." Université de Reims Champagne-Ardenne". January 31, 2010

[HG93] Hubert Garavel. "Presentation de langage LOTOS". 31 Août 1993.

[HM80] M. Hennessy and R. Miner. On observing nondeterminism and concurrency.

[JF97] Jacques Ferber . "Cours de compilation ". 1997

[JD00] Jérôme Darmont, "Programmation sous Delphi",Faculté de Sciences Économiques et de Gestion ,Année 1999-2000

[ISO88] ISO8807. LOTOS, a formal description technique based on the ordering of ob-servation behaviour. ISO (November 1988).

[Hoa85] C. A. R. Hoare. "Communicating Sequenctial Processes". Prentice Hall (1985).

[KB12] Prof. Karim Baïna ENSIAS. "Programmation avancée, Techniques de programmation en C, Modularité".6 avril 2012.

[MeNaAL1112] Mr. Meadi M.Nadjib. "Analyse lexicale- Compilation,Université de Biskra ".2011 -2012

[MeNaAS1112] Mr. Meadi M.Nadjib. "Analyse syntaxique-Compilation,Université de Biskra ". 2011 -2012

[Mil80] R. Milner. "Communication and Concurrency", volume 92 of LNCS. Springer Verlag (1980).

[Mil83] R. Milner. Calculus for synchrony and asynchrony. TCS25, 267—310 (1983).

[Mil89] R. Milner. "Communication and Concurrency". Prentice Hall (1989).

[MN1112] Mr. Meadi M.Nadjib. "Compilation". Université de Biskra 2011-2012

[NB10] Nabil Belala. " Modèles de temps et leur intérêt à la vérification formelle des systèmes temps-réel", Université Mentouri de Constantine le 20 octobre 2010

[NWZ05] Niklaus Wirth, Zürich. "Compiler Construction,This is a slightly revised version of the book published by Addison-Wesley in 1996 ,ISBN 0-201-40353-6 ".November 2005

[PPBLI] Prof. Patrick Bellot. "Langages de l'informatique- TELECOM Paris Tech".

[Rei85] W. Reisig. Petri nets. In "EATCS Monofraphs on Theoretical Computer Sci-ence", Berlin, Heidelberg, New York, Tokyo (1985). Springer Verlag.

[Shi85a] M. W. Shields. Concurrent machines. The Computer Journal28(5), 449—465 (1985)

[Shi85b] M. W. Shields. Deterministic asynchronous automata. In "Formal Methods in Programming", North-Holland (1985).

[SL09] SAOUDI Lalia, "cours de compilation Département d'informatique, Centre Universitaire de BBA, 4 ième année Ingénieur, système classique". 28 Sep 2009

[TBEB89] Tommaso Bolognesi & Ed Brinksma. "Introduction to the ISO Specification Language LOTOS", University of Twente .1989.

[ZA08] ZITOUNI Abdelhafid, "Utilisation des design Patterns et des

méthodes formelles dans le développement des systèm d'information". 02/12/2008

ANNEXE A

Grammaire de Basic LOTOS pour le système et le package :

Lorsque vous écrire une spécification Basic LOTOS, vous devez respecter la syntaxe suivante ;

<Grammaire lotos> ::= <entête> <corps>

< Entête > ::= <entêtepackage> ***SYSTEM*** *<id> [<listeporte>]*

<Corps> ::= := <expression> <corp> ***ENDSYS***

< Entêtepackage > ::= <withuse> <listewithuse> | epsilon

<Withuse>::= <with>; <use>; |<with>;

<With>::= ***WITH*** *<liste>;*

<Liste>::= <id> | <listecomposé> | <id> , <liste> | <listecomposé> , <liste>

<Listecomposé> ::= <id> . <id> | <id> . <listecomposé>

<Use> ::= ***USE*** *<liste>;*

<Listewithuse> ::= <withuse> <listewithuse> | <withuse>

<Listeporte> ::= <porte> | epsilon

<Expression> ::=<expression> <op> <expression> | <Id> ; <expression> |

*<listecomposé>[<listeporte>]|<id>[<listeporte>]|****HIDE*** *<porte>* ***IN*** *<Expression>*

|STOP | NIL | EXIT | (<expression>)

<Corp> ::= <copr 1> | epsilon

<Porte> ::= <id> | <porte> , <id>

<Copr 1>| ::= WHERE <listeprocessus>

<Listprocessus> ::= <processus> <listeprocessus> | <processus>

*<Processus>::=****PROCESS****<id>[<listeporte>]:=<expression><corp>****ENDPROC***

<Op> ::= || | ||| | |[<listeporte>]| | >> | [> | [ ]

<Grammaire package> ::=<spécification> | <implémentation>

*< spécification> ::=****PACKAGE*** *<id> IS <listeproc>* ***END*** *<id>*

< implémentation> ::= ***PACKAGEBODY****<id>* ***IS*** *<listeprocessus> END <id>*

<listeproc> ::= <proc> <listeproc> | <proc>

<proc> ::= PROCESS <id> [<listeporte>]

Abréviation :

CCS *Calculus of Communicating Processes.*

CSP *Communicating Sequential Processes.*

CCITT *Comité Consultatif International Téléphonique et Télégraphique ou Consultative Committee for International Telegraph and Telephony*).

OSI *Open Systems Interconnection.*

ISO *Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique based on the Temporal Ordering of Observational Behaviour, ISO 8807, International Organisation for Standardisation, Geneva, CH, 1989.*