Université de NANCY I — UER de Mathématiques

Centre de Recherche en Informatique de NANCY (CRIN)
BP 239 54506 VANDOEUVRE-LES-NANCY Cedex

UN ENVIRONNEMENT D'EDITION EVOLUE, GRAPHIQUE ET SYNTAXIQUE, POUR LA CONCEPTION DES SYSTEMES REPARTIS

THESE

Présentée et soutenue publiquement le 2 novembre 1988
A l'Université de NANCY I
pour l'obtention du titre de

Docteur de 3ème cycle en Informatique

par

Martial LALLIER

devant la commission d'Examen

Président : M **Jean-Claude DERNIAME**, *Université de NANCY I*

Examinateurs : Mme **Marion CREHANGE**, *Université de NANCY II*
MM. **Jacques GUYARD**, *Université de NANCY I*
Jacques LONCHAMP, *Université de METZ*
Guy-René PERRIN, *Université de BESANCON*

Université de NANCY I — UER de Mathématiques

Centre de Recherche en Informatique de NANCY (CRIN)
BP 239 54506 VANDOEUVRE-LES-NANCY Cedex

UN ENVIRONNEMENT D'EDITION EVOLUE, GRAPHIQUE ET SYNTAXIQUE, POUR LA CONCEPTION DES SYSTEMES REPARTIS

THESE

Présentée et soutenue publiquement le 2 novembre 1988
A l'Université de NANCY I
pour l'obtention du titre de

Docteur de 3ème cycle en Informatique

par

Martial LALLIER

devant la commission d'Examen

Président : M **Jean-Claude DERNIAME**, *Université de NANCY I*

Examinateurs : Mme **Marion CREHANGE**, *Université de NANCY II*
MM. **Jacques GUYARD**, *Université de NANCY I*
Jacques LONCHAMP, *Université de METZ*
Guy-René PERRIN, *Université de BESANCON*

Je tiens à remercier tout d'abord les membres du jury.

Ma formation à l'informatique et à la recherche doit beaucoup à Jean-Claude DERNIAME et Guy-René PERRIN. Je souhaitais également la présence de Jacques Guyard, spécialiste des éditeurs syntaxiques. Je les en remercie tous.

Je remercie tout particulièrement Jacques LONCHAMP, pour la confiance qu'il m'a témoignée en proposant ce sujet. L'aboutissement de ce travail est dû en grande partie à ses conseils permanents.

Je remercie sincèrement Marion CREHANGE d'avoir contribué à ce que ce travail se déroule dans les meilleures conditions.

Mes remerciements vont également à Monsieur LIEGEON et à travers lui, à toutes les personnes qui ont contribuées à la réalisation de ce rapport.

Je tiens à associer à ces remerciements mon épouse et tous mes proches; je leur dédie ce travail.

Résumé

La conception des applications informatiques réparties soulève, entre autres problèmes, celui de l'édition et du contrôle des différents niveaux de descriptions. Proposer un outil qui assiste l'utilisateur dans ces tâches fastidieuses de manipulation des spécifications constitue l'objectif principal de ce travail.

Après un bref survol du contexte général de la conception des applications réparties en commande de procédés et du projet global auquel se rattache cette proposition, la définition et la mise en oeuvre d'un environnement d'édition et de contrôle évolué, graphique et syntaxique sont abordés. Un exemple complet illustre ses modalités d'utilisation.

Mots clés

Conception, environnement, outils, structuration, spécification.

Abstract

The design methods for distributed systems raises the problems of editing and control different levels of descriptions. The aim of this thesis is to develop a tool which helps the user manipulate specifications.

After a short survey of the context of the design methods for distributed systems we shall deal with the definition and the creation of an evolved graphic and syntactic editing environment. We will describe in a detailed example how to use it.

Index terms

Design, environment, tools, structuration, specification.

TABLE DES MATIERES

INTRODUCTION

Le domaine de la conception des applications réparties a été abordé de différentes manières. Selon les thèmes vers lesquels se sont focalisées les recherches (communication, parallélisme, répartition, spécification de problèmes, méthode de conception, etc) les travaux ont conduit à l'élaboration de langages plus ou moins spécialisés ou d'outils de développement. Par contre, la conception des applications réparties appliquées à la commande de procédés industriels a été relativement peu étudiée par les informaticiens. En particulier, les travaux relatifs au processus de conception, à ses méthodes et aux outils associés sont rares. Un rapide tour d'horizon est fait au cours du premier chapitre.

Des éléments de méthode de conception de tels types de systèmes sont proposés dans **[LON 87]** et sont ciblés pour des applications de commande de système de production flexible du type atelier ou cellule flexible. On y préconise 3 niveaux d'abstraction pour mener le processus de conception depuis le cahier des charges jusqu'à une description en termes programmatoires. Un niveau logique décrivant les fonctionnalités à assurer, un niveau organique exprimant les contraintes d'organisation et un niveau physique décrivant les choix liés à la mise en oeuvre programmée et la définition de l'architecture matérielle cible. Parmi les moyens de description de l'application on compte les "agents" pour le niveau logique, les modules **[KRA 83]** pour le niveau organique. Ces entités coopèrent par échange de messages typés via des "ports", des liaisons et des attachements prolongeant les liaisons d'un niveau de décomposition à un autre. La méthodologie préconisée et les outils associés font l'objet du chapitre II.

L'objectif de ce travail était donc de concevoir et de réaliser un outil qui assiste le concepteur pour tous les aspects structurels depuis le cahier des charges jusqu'à l'obtention d'un squelette de programme. Cet environnement intégré "colle" à la méthodologie de conception retenue, l'accent ayant été mis sur les contrôles de cohérence des éléments structurants. La mise en oeuvre de cet environnement s'est faite en 2 étapes. La première s'est concrétisée par la réalisation d'un éditeur graphique et contrôleur de hiérarchies de réseaux d'entités communicants. L'interfaçage de cet outil avec un éditeur syntaxique a fait l'objet de la seconde étape. Par compilation de la description structurelle sous forme graphique, on obtient des arbres syntaxiques dans un langage cible mis en oeuvre pour la maquette. On détaille successivement, dans le chapitre III, les objectifs de l'environnement, sa mise en oeuvre et un exemple complet de son utilisation.

CHAPITRE I) CONTEXTE GENERAL DU TRAVAIL : LA CONCEPTION DES APPLICATIONS REPARTIES EN COMMANDE DE PROCEDES

I.1) LE PROBLEME DE LA CONCEPTION DES APPLICATIONS REPARTIES EN COMMANDE DE PROCEDES

De manière générale, un système informatique de commande de procédé industriel est intimement lié au procédé qu'il pilote. Il suit l'évolution du procédé par le biais de capteurs et élabore en sortie des commandes vers des actionneurs agissant sur le procédé pour le contraindre à respecter certaines propriétés ou un comportement donné (cf. Fig. 1.1).

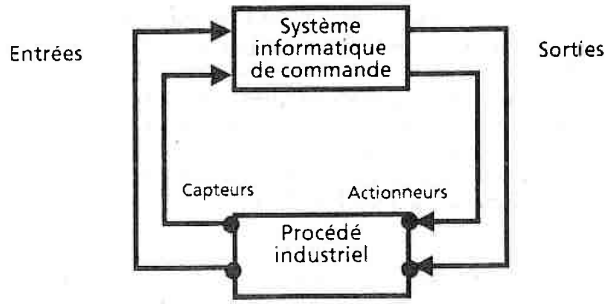


Fig. 1.1 Système industriel informatisé

La conception du système informatique de commande débute par une caractérisation du procédé. Les formes de cette description peuvent être très diverses :

- modélisation du procédé en termes d'objets pouvant prendre un ensemble d'états, et d'ensemble de transformations liées aux lois naturelles d'évolution du procédé **[LEL 83]**,

- description centrée sur les variables caractéristiques du procédé, les actions à déclencher et les lois de commande qui les mettent en relation **[LAD 82]**,

- description à l'interface du système de commande (caractéristiques des entrées, des sorties et des relations entre entrées et sorties) **[LON 87]**.

Quel que soit le mode de description retenu, la conception du système informatique réparti débouche sur un ensemble de composants logiciels s'exécutant de manière asynchrone sur un réseau de processeurs sans mémoire commune et coopérant par échanges de messages. Deux des activités de conception essentielles concernent donc le découpage en composants et leur affectation aux processeurs.

I.2) ETAT DE L'ART DANS LE DOMAINE DE LA CONCEPTION DES APPLICATIONS REPARTIES

Nous présentons d'abord les travaux généraux liés à la conception des applications réparties, quel que soit le domaine d'application, avant de nous centrer sur le domaine qui nous intéresse plus particulièrement.

I.2.1) Travaux généraux sur la conception des applications réparties

Le domaine de la conception des applications réparties est au confluent de multiples courants de recherche. Ils conduisent à l'élaboration de langages dédiés à l'expression de la communication, ou d'environnement complet de conception et de contrôle, en passant par des réflexions plus générales sur la spécification des problèmes ou les méthodes de conception de programmes. Nous essayons, dans la suite, de faire un rapide tour d'horizon de ces différentes approches en utilisant les principaux critères de présentation extraits d'une synthèse sur la programmation parallèle **[PER 87]**.

a) Recherches sur les langages destinées à exprimer la coopération ou la concurrence dans les systèmes parallèles.

On distingue les langages de programmation parallèles de type impératif et les langages de type déclaratif. Dans la première famille, le programme est constitué d'un ensemble de processus qui coopèrent (par communication de messages, signalisation d'événements ou appel de procédures) ou qui sont en concurrence pour l'accès à des données partagées. Dans la seconde famille, on définit les règles d'évolution du système comme un système de transitions d'états ou en définissant un système d'équations (langages applicatifs).

- **langages impératifs** : les langages orientés vers la communication de processus via des messages permettent de définir les chemins de communication par lesquels transitent les messages, et les opérations de communication activées dans le corps des processus. Dans cette catégorie on compte les langages fondés sur les communications synchrones (CSP **[HOA 78]**, OCCAM **[MAY 83]**, COOL **[ADA 86]**) et ceux fondés sur les communications asynchrones (GYPSI **[AMB 77]**, PLITS **[FEL 79]**, LC3 **[LEC 86]**).
ESTEREL **[BERR 85]** est un langage synchrone qui traite la communication par signalisation d'événements aux points de synchronisation.
Dans les langages DP **[BRI 78]**, ADA **[ICH 79]** et SR **[AND 81]** , la coopération s'effectue par appel de procédures. Ceci nécessite la définition de procédures publiques dans les processus, la mémorisation des appels en files d'attente et l'exécution lors de rendez-vous entre le processus appelant et le processus qui active la procédure appelée.
La relation entre processus s'exprime dans certains langages sous la forme d'une concurrence pour l'accès à des données partagées. CPASCAL **[BRI 75]** et ARGUS

[LIS 84], qui sont de ceux-ci, utilisent, respectivement, les notions de "moniteur" et de "gardien" (concepts issus des moniteurs de Hoare **[HOA 74]**) pour exprimer la synchronisation des processus tout en apportant un certain niveau d'abstraction et de modularité de la programmation. De plus, le langage ARGUS étant plus particulièrement destiné à la programmation d'applications réparties, il a été conçu de façon à pouvoir assurer la restauration d'un module en cas de panne et garantir la consistence des données (**[LIS 81]**).

L'exemple de la figure 1.2 décrit une communication synchrone de messages définie dans le langage CSP de HOARE **[HOA 78]** . Pour réaliser le système producteur-consommateur synchrone, les processus communiquent des valeurs de données locales, via un canal, sans mémorisation des messages. Le processus P émet un message vers le processus Q au travers du canal "Q!x" défini implicitement et dynamiquement par le nom du processus interlocuteur (Q) et la donnée communiquée (x). Le symbole "!" précise que le canal est utilisé dans le sens "émission". Lors d'une situation dite de "rendez-vous", le processus Q effectue l'opération complémentaire sur ce canal "P?y", c'est-à-dire réception (symbole ?) dans la variable y d'une donnée émise par le processus P.

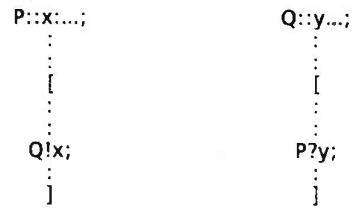


Fig. 1.2 - Rendez-vous CSP

- **les langages déclaratifs :** dans les langages qui décrivent l'évolution d'un système comme un système de transition d'états on construit en général les systèmes par combinaison de processus plus élémentaires. Dans CCS **[MIL 80]**, chaque processus est caractérisé par une liste de ports et un ensemble de comportements. Le parallélisme s'exprime par rendez-vous entre des ports de même nom. LCS **[BERT 85]** utilise le même formalisme pour l'expression du parallélisme mais s'appuie sur le langage fonctionnel ML **[MIL 84]** pour l'expression des fonctions et des types de données. Dans le langage FP2 **[JOR 84]** on définit, pour chaque processus, un ensemble de règles de transitions d'états défini à partir d'opérations sur des termes algébriques. LUSTRE **[HAL 86]** est un langage de type applicatif. Un programme LUSTRE est une fonction sur des suites infinies de valeurs, structurées en réseau de noeuds. Un noeud décrit une relation entre des données et des résultats, par un système d'équations.
Dans le programme LCS de l'exemple de la figure 1.3, les agents récursifs P, Q et séq sont combinés pour définir un agent polymorphe producteur-

consommateur. L'agent intermédiaire séq exprime un choix non déterministe entre la réception d'un objet, alors concaténé à la liste "f" par l'opérateur "@", et, si cette liste est non vide, l'émission de la tête de "f" (opérateur "hd"), alors décapitée (opérateur "tl").

```
agent prod-cons =
    in recagent P = x!u . P
    in recagent Q = y?v . Q
    in recagent séq f =
        ecrire ? t . séq(f@[t])
         + [not(null f) --> lire ! (hd f) . séq(tl f)]
    in (P{écrire / x}|
        séq[]|
        Q {lire/y} ) {/lire ecrire} ;;
```

Fig. 1.3 - Producteur-consommateur asynchrone en LCS

b) Recherches sur la spécification des problèmes.

La spécification d'un problème parallèle (au sens de description formelle du service rendu par le système) passe par une expression de propriétés temporelles. Les fondements de ces spécifications sont les logiques temporelles. Dans **[CHE 83]** la méthode de spécification est basée sur la notion d'événement qui est considéré comme une transition atomique instantanée. **[LAM 83]** spécifie des propriétés de sûreté et de vivacité des systèmes. Certaines propositions incluent la dérivation systématique du programme, comme **[MAN 84]**, qui propose la dérivation de processus à la CSP à partir de spécifications exprimées à l'aide des opérateurs de la logique temporelle. Les spécifications des composantes du système sont transformées en une seule spécification globale qui, elle-même, produit par transformations itérées de ses formules, un graphe d'états fini lequel fournit, par dérivation, un graphe associé à chaque composante du système et une représentation par un processus CSP.

c) Recherches sur les méthodes de conception de programmes.

Ces recherches ont mis en évidence deux démarches principales de conception : descendante et ascendante. La première méthode est utilisée avec les réseaux de Pétri **[YAU 83]** et les processus communicants de Kramer **[KRA 82 a]**. La différence résidant principalement dans le langage cible visé. La seconde est adaptée pour la construction de système par combinaison de processus élémentaires (LUSTRE ou CCS). La définition de nouveaux noeuds dans LUSTRE se

fait à l'aide de fonctions définies dans les noeuds. En CCS, c'est la connexion d'agents qui permet de définir de nouveaux agents.

d) Recherches sur les environnements de développement.

Ces travaux tentent de guider le concepteur tout au long de sa tâche de création de logiciels répartis, c'est-à-dire de proposer des outils recouvrant les problèmes de spécification, de vérification, d'implémentation et de test. Deux types d'environnement ont été patronés par l'ISO. Le premier, LOTOS **[ISO 86a]**, fondé sur la description de l'ordonnancement temporel des interactions, s'inspire du formalisme CCS et des Types Abstraits Algébriques. Le second, SEDOS/ESTELLE **[DIA 85]**, a été obtenu par ajout d'un ensemble cohérent d'outils pour la spécification, la validation et la mise en oeuvre de systèmes répartis et de réseaux au langage ESTELLE **[ISO 86b]** qui combine le langage de programmation Pascal et les réseaux d'automates communicants.

On peut noter également les environnements de conception proposés autour des réseaux de Pétri ainsi que STATEMATE1, outil graphique associé aux STATECHARTS **[HAR 87]** permettant une visualisation de différents niveaux de description.

Les Statecharts correspondent à des diagrammes de transition d'états intégrant les notions de hiérarchie, concurrence et communication. Un diagramme de transition d'états est en fait un graphe composé de noeuds (les états) et d'arcs de transition orientés matérialisant les événements avec éventuellement une condition de déclenchement. La figure 1.4 représente un diagramme de 3 états A,B,C. A l'apparition de l'événement "c" dans l'état A, et sous la condition P, le système passe à l'état C.

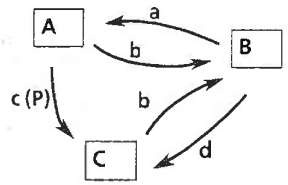


Fig. 1.4

Avec ce mode description "à plat", la prise en compte de nombreux états supplémentaires est pratiquement ingérable et en tout cas incompréhensible. Pour être utilisable, un tel diagramme doit être modulaire, hiérarchisé et bien structuré. En outre, ce formalisme doit favoriser l'abstraction d'un ensemble d'états par un état dont le comportement global est identique, le raffinement d'un état en plusieurs états et prendre en compte la notion d'orthogonalité (indépendance et concurrence). On précise ces notions ci-dessous.

La notion d'abstraction est schématisée dans la figure 1.5. Les états A et C sont "encapsulés" dans l'état D. L'arc "b" se substitue aux 2 arcs "b" de la figure 1.4. Pour être dans l'état D le système doit passer dans l'état A ou (exclusif) dans l'état C.

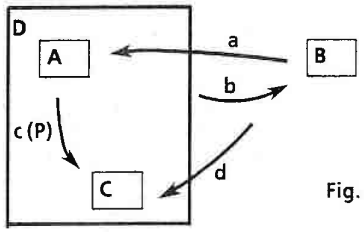
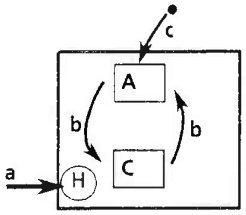


Fig. 1.5

Remarque :
Les demi-arcs (⟶) permettent de désigner un état "par défaut" d'un groupe d'états et les arcs (⟶ (H)) de désigner un état en fonction de l'historique du système, par exemple l'état le plus récemment visité.



Par opposition à la notion de découpage en états accessibles de façon exclusive, la notion d'orthogonalité permet d'exprimer 2 situations. Le graphisme de la figure 1.6 permet d'exprimer le fait que le système est dans l'état A lorsque, simultanément, il se trouve dans les 2 états B et C raffinant l'état A (un trait en pointillés sépare les états B et C). Dans le cas où les états B et C sont, respectivement, composés des états (exclusifs) D,E et F,G,H, le passage dans l'état A signifie passage dans une combinaison des états (D ou E) et (F ou G ou H). Dans le cas de la figure 1.6, en l'absence de toute information complémentaire, si l'on passe dans l'état A alors on entre dans les états D et F (états repérés "par défaut"). Si l'événement "a" survient, il transfert, simultanément D vers E et F vers G. Le système passe à l'état (E,G). Cette première situation illustre une sorte de synchronisation d'événements. Par contre, si dans l'état (D,F) l'événement "b" survient, alors on passe dans l'état (D,H). Il y a, dans ce cas, indépendance des états B et C.

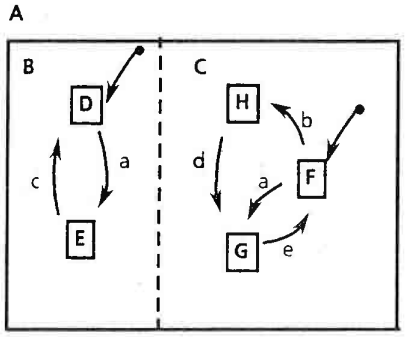


Fig. 1.6

Le graphisme peut être allégé par l'introduction d'arc conditionnel (C) ou sélectif (S). Dans le premier cas l'arc véhicule toujours le même événement mais le destinataire dépend de la condition en vigueur lors du déclenchement (cf. fig. 1.7). L'arc sélectif peut, quant à lui, supporter des événements de nature différentes, la sélection étant faite par le type de l'événement.

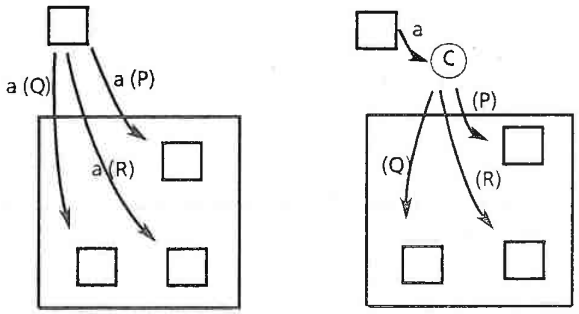


Fig 1.7

Notons, enfin, que les Statecharts permettent l'expression de délais et de time out, la génération d'événements et le changement de conditions de déclenchement.

I.2.2) La conception des applications réparties en commande de procédés

a) Etat de l'art.

La conception des applications réparties appliquée à la commande de procédés industriels est un domaine relativement peu étudié par les informaticiens. Dans la majorité des publications, comme nous l'avons vu au paragraphe précédent, le parallélisme et la répartition sont abordés sous l'angle des langages généraux ou spécialisés, de la sémantique et de la preuve, des architectures de systèmes ou de l'algorithmique. Par contre, les travaux relatifs au processus de conception, à ses méthodes et aux outils associés sont assez rares; on peut les ranger en deux catégories :

- celle des travaux théoriques ponctuels centrés sur la mise en oeuvre d'une démarche précise de découpage et de moyens d'expression formels, comme **[KRA 78]**, **[GOL 79]**, **[CHE 83]**;

- celle des contributions pragmatiques, prenant en considération plus largement le processus de conception, mais souvent coupées des avancées récentes sur la spécification et la validation, comme CONIC **[KRA 83]**, COSY **[LAU 80]**, **[CAL 82]**, **[DER 83]**.

Des auteurs ont déjà souligné la distance séparant ces 2 types d'approches **[KRA 82 b]**. La voie intermédiaire étudiée est fondée sur la sélection d'outils formels selon leur aptitude à s'intégrer à une vision globale de la conception et au contexte de la classe de problèmes étudiée.

b) Caractérisation précise des applications étudiées.

Les applications réparties plus particulièrement étudiées sont les applications de commande de systèmes de production flexibles du type atelier ou cellule flexible. Ces systèmes se caractérisent par la complexité de leur pilotage liée à l'automatisation complète de la manutention et du transport des pièces, et parfois même des outils. Ils permettent d'usiner ou d'assembler simultanément des produits de types différents, grâce à cette automatisation du transport et à l'adaptativité des postes de travail à commande numérique.

Les principales caractéristiques de ces applications, dans l'optique de la conception, sont les suivantes (cf. **[LON 87]**) :

(1) l'environnement de ces applications est massivement parallèle : le procédé comporte de nombreuses entités et activités évoluant en parallèle ;

(2) la structure de cet environnement est stable : les entités et activités du procédé sont en nombre sensiblement constant (à l'opposé d'applications foncièrement dynamiques, comme par exemple en téléphonie) ;

(3) la répartition géographique peut être importante ;

(4) les exigences de promptitude sont le plus souvent peu élevées : on peut parler de "système temps réel lent" pour tout ce qui touche à la coordination des activités ; mais les exceptions à cette règle, comme certaines alarmes, doivent pouvoir être prises en considération ;

(5) les volumes d'informations nécessaires pour la coordination des activités sont le plus souvent faibles ;

(6) la multiplicité des composantes de nature diverses du système industriel global, composantes mécaniques, informatiques, humaines, etc... rend indispensable l'étude de plusieurs modes de marche, mode de fonctionnement normal et divers modes de fonctionnements dégradés ;

(7) le concepteur doit tenir compte de multiples contraintes de réalisation économiques, techniques (ex : contraintes environnementales), informatiques (ex : matériels ou logiciels préexistants) ; peu de systèmes sont construits "ex nihilo".

c) Support et objectifs du travail.

Les travaux de J. LONCHAMP **[LON 87]**, qui constituent le support de notre étude, s'intéressent plus particulièrement à la construction d'une spécification formelle à partir du cahier des charges et donc à la définition de moyens d'expression et d'éléments de méthode pour parvenir à cette description en tenant compte des caractéristiques ci-dessus. Pour situer cette contribution, rappelons les trois niveaux d'expression communément admis pour définir le processus de conception d'applications réparties en commande de procédés :

- une description informelle du système et de ses qualités requises dans les domaines de l'efficacité, de l'évolutivité et de la sûreté de fonctionnement, ou "**cahier des charges**";
- une (ou plusieurs) forme(s) de spécification formelle du problème à traiter pouvant inclure aussi des choix, plus ou moins explicites, relevant de la solution;
- la mise en oeuvre programmée utilisant des langages impératifs variés (de l'assembleur à ADA !).

Les travaux menés durant la thèse ont contribué à la réalisation d'un environnement de conception intégré. Ils se sont concrétisés par la mise en oeuvre d'un outil qui "colle" à la méthodologie de conception retenue. L'outil

graphique de conception propose des fonctionnalités sensiblement équivalentes à celles des Statecharts pour les aspects hiérarchie, mais a l'avantage d'être adapté à une méthode de conception et de faciliter la mise en oeuvre programmatoire (fonction compilation d'une description en squelette de programme). Avant de décrire cet environnement nous précisons, dans le paragraphe suivant, le support sur lequel il se fonde.

BIBLIOGRAPHIE

[ADA 86] ADAMO, BONNEVILLE
Cool : manuel de référence.
Rapport Université Lyon, 1986.

[AMB 77] AMBLER A.L.
Gypsy : a language for specification and implementation of verifiable programs.
Conf. ACM «Language Design for Reliable Software», Sigplan Notices, 12, 3, 1977.

[AND 81] ANDREWS G.R.
Synchronizing Resources.
ACM Toplas, 3, 4, 1981.

[BERR 85] BERRY, COSSERAT
The Esterel synchronous programming language and its mathematical semantics.
"Sem. on Concurrency", LNCS 197, Springer Verlag, 1985.

[BERT 85] BERTHOMIEU B.
Le langage LCS et son interprète : une implémentation expérimentale de OCS.
Actes COLL. Nat. C3, 1985.

[BRI 75] BRINCH HANSEN P.
The programming language Concurrent Pascal.
IEEE Trans. on Software Engineering, SE-1, 2, 1975.

[BRI 78] BRINCH HANSEN P.
Distributed Processes : a concurrent programming concept.
CACM, 21, 11, 1978.

[CAL 82] CALVEZ J.P.
Une méthodologie de conception de systèmes multi-micro-ordinateurs pour les applications de commande en temps réel.
Thèse d'Etat, NANTES, 11/82.

[CHE 83] CHEN, YEH
Formal specification and verification of distributed systems.
IEEE Trans. on Soft. Eng., SE-9, 6, 1983.

[DER 83] DERNIAME, ZAKARI.
Langage de conception pour les applications réparties de conduite de procédés.
BIGRE 83, CAP D'AGDE, 10/83.

[DIA 85] DIAZ, VISSERS, ANSART, SEDOS
Software Environment for the Design of Open distributed Systems.
Proceedings of the Esprit'85 week, North-Holland.

[FEL 79] FELDMAN A.
High level programming language for distributed programming.
CACM, 22, 6, 1979.

[GOL 79] GOLDSACK, KRAMER
The use of invariants in the application-oriented specification of real-time control systems.
Research Report Imperial College, LONDON, 1979.

[HAL 86] HALBWACHS N.
Automatic control system programming using a real time declarative language.
IFAC/IFIP Symposium SOCOCO 86, 1986.

[HAR 87] HAREL D.
STATECHARTS : A visual formalisme for complex systems.
Science of Computer Programming 8, 231-274, 1987.

[HOA 74] HOARE C.A.R.
Monitor : an operating system structuring concept.
CACM,17,12,1974.

[HOA 78] HOARE C.A.R.
Communicating Sequential Processes.
CACM, 21, 8, 1978.

[ICH 79] ICHBIAH J.D.
Ada reference manuel and rationale for the design of the Ada programming language.
Sigplan Notices, 14, 6, 1979.

[ISO 86a] ISO/TC97/SC21/WG16-1 DP8807
Lotos : a formal Description Technique, juillet 1986.

[ISO 86b] ISO/TC97/SC21/WG16-1 DP9074
Estelle : a formal Description Technique based on an Extended State Transition Model, juillet 1986.

[JOR 84] JORRAND P.
FP2 : Functionnal Parallel Programming based on term substitution.
Rapport Recherche LIFIA, 15, 1984.

[KRA 78] KRAMER, CUNNINGHAM
Towards a notation for the functionnal design of distributed processing systems.
Proc. IEEE Int. Conf. on Parallel Proces , 1978, 69-76.

[KRA 82 a] KRAMER MAGEE, SLOMAN
A Sofware architecture for distributed computer control systems.
IFAC Symp. on "Theory and applications of digital control", 1982.

[KRA 82 b] KRAMER J.
Distributed computer systems - two views.
Research Report 82/5, Imperial College, LONDRES, 3/82.

[KRA 83] KRAMER, MAGEE, SLOMAN, LISTER
Conic : an integrated approach to distributed computer control System.
IEE Proc., Vol 130, n°1, 1/1983.

[LAD 82] LADET P.
Contribution à l'étude des systèmes informatiques répartis pour la commande de procédés industriels.
Thèse d'Etat, GRENOBLE, 1982.

[LAM 83] LAMPORT L.
Specifying concurrent program modules.
ACM Toplas, 3, 2, 1983.

[LAU 80] LAUER, SCHIELDS, BEST
Design and analysis of highly parallel and distributed systems.
LNCS 86, Springer Verlag, 1980.

[LEC 86] LE CERTEN P.
Conception et mise en oeuvre d'un langage impératif pour la programmation parallèle.
Thèse Univ. Rennes, 1986.

[LEL 83] LE LANN G.
Sur le traitement réparti temps-réel.
Actes IFIP 83, PARIS, pp 213, 225.

[LIS 81] LISKOV, SCHEIFLER
Guardians and Actions : Linguistic support for Robust Distributed Programs.
Laboratory for Computer Science, MIT, Cambridge, 1981.

[LIS 84] LISKOV B.
The Argus Language and System.
LNCS 190, Springer Verlag, 1984.

[LON 87] LONCHAMP J.
Conception des applications informatiques réparties en commande de procédés industriels : une démarche, des outils.
Thèse d'Etat, NANCY I, 5/87.

[MAN 84] MANNA, WOLPER
Synthesis of Communicating Processes from temporal logic specifications.
ACM Toplas, 6, 1, 1984.

[MAY 83] MAY D.
The Occam language.
Sigplan Notices, 18, 4, 1983.

[MIL 80] MILNER R.
A calculus for Communicating Systems.
LNCS 92, Springer Verlag, 1980.

[MIL 84] MILNER R.
A proposal for standard ML.
Report Edimburgh Univ., 1984.

[PER 87] PERRIN G.R. : Programmation parallèle : point de vue sur les langages et les méthodes.
Synthèse, TSI, Vol. 6, n°2, 1987.

[YAU 83] YAU, CAGLAYAN
Distributed software system design representation using modified Petri nets.
IEEE Trans. on Software Engineering, SE-9, 6, 1983.

CHAPITRE II) SUPPORT DU TRAVAIL : UNE METHODE ET DES OUTILS POUR LA CONCEPTION D'APPLICATIONS REPARTIES EN COMMANDE DE PROCEDES

II.1) CHOIX METHODOLOGIQUES

II.1.1) Choix des niveaux de description

Pour mener le processus de conception depuis le cahier des charges jusqu'à une description en termes programmatoires, **[LON 87]** propose trois niveaux d'abstraction calqués sur ceux mis en oeuvre dans des méthodes opérationnelles de construction de systèmes d'information de gestion **[TAR 83]**.

L'ensemble des propriétés requises à prendre en compte et donc l'ensemble des choix à réaliser, est hiérarchisé et le concepteur progresse en descendant cette hiérarchie. L'idée maîtresse consiste à baser la méthode de conception sur la notion d'invariance : le concepteur progresse du plus stable au plus évolutif.

Les trois niveaux sont adaptés à la classe de problèmes étudiés :

- le niveau "**logique**" décrit les fonctionnalités à assurer. Il est doté de la plus forte invariance;

- le niveau "**organique**" exprime les contraintes d'organisation qui s'imposeront à toute mise en oeuvre. Son invariance est donc relativement forte. Ces contraintes peuvent concerner le processus de découpage en composants comme celui d'affectation des composants.

Exemples de contraintes relevant du niveau organique :

- les contraintes de découpage et d'affectation liées à la répartition géographique du système piloté;
- les contraintes de découpage et d'affectation liées aux nécessités d'intégration de composants préexistants (sites, logiciels, base de données);
- les contraintes de découpage et d'affectation liées aux besoins de redondances structurelles de l'application, pour des considérations de sûreté;
- les contraintes de découpage liées à la granularité des "briques élémentaires de configuration", dans le cas de systèmes à configuration dynamique.

- le niveau "**physique**" décrit les choix essentiels concernant la mise en oeuvre programmée (choix des langages, choix définitif des composants du logiciel, des modes de communication entre composants, etc...) ainsi que la définition de l'architecture matérielle cible et éventuellement l'affectation des composants à cette architecture. Beaucoup de choix de détail peuvent être laissés à l'initiative des programmeurs ; par exemple, les choix de réalisation des structures de données et des algorithmes purement locaux à un site.

La spécification formelle d'une application répartie inclut sous une forme ou une autre :

- une description structurelle en termes de composants inter reliés,
- une description comportementale de ces composants et éventuellement des "systèmes de communication" par lesquels ils coopèrent.

Nous détaillons, successivement, dans les paragraphes suivants, ces deux aspects.

II.1.2) Choix des moyens de description

II.1.2.1) Modèle de structuration retenu

Au niveau logique, les composants baptisés "**agents**" correspondent à des sous-systèmes de granularité quelconque.

Au niveau organique, les composants baptisés "**modules**" correspondent à des sites virtuels : ils ne peuvent être répartis sur plusieurs sites physiques, mais plusieurs d'entre eux peuvent coexister sur le même site physique. D'un point de vue interne, on peut les définir comme un ensemble de processus séquentiels se partageant des ressources communes. Ils permettent donc d'exprimer explicitement les contraintes d'organisation de la solution évoquées précédemment et touchant à la localisation, sans toutefois lier la description à une architecture physique donnée. Ils sont similaires aux modules de [**KRA 83**].

Ces entités (agents ou modules) coopèrent par échange de messages typés via des ports et des "liaisons" (**cf. Fig. 2.1**). La structure de l'application étant stable (cf. propriété (2) précitée), tous les éléments structurants retenus sont statiques (composants et liaisons).

La notion de "liaison" est explicitée plus loin, quant à celle de port, on distingue quatre types :

- les ports d'entrée (**E-PORT**), pour les réceptions de messages d'un type donné,
- les ports de sortie (**S-PORT**), pour les émissions de messages d'un type donné,
- les ports de sortie-entrée (**SE-PORT**), pour les émissions de requêtes d'un type donné avec attente de réponses du type correspondant,
- les ports d'entrée-sortie (**ES-PORT**), pour les réceptions de requêtes d'un type donné avec émission des réponses correspondantes.

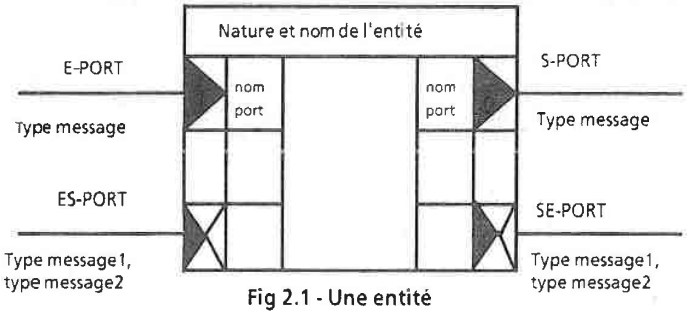


Fig 2.1 - Une entité

Un chemin de communication **monodirectionnel** lie un port de sortie à un port d'entrée; il est étiqueté, sur la représentation graphique, par le type de message qu'il véhicule. Un chemin de communication **bidirectionnel**, lie un port de sortie-entrée à un port d'entrée-sortie; il est étiqueté par le type de message de requête suivi du type de message de réponse(**cf. Fig. 2.2**).

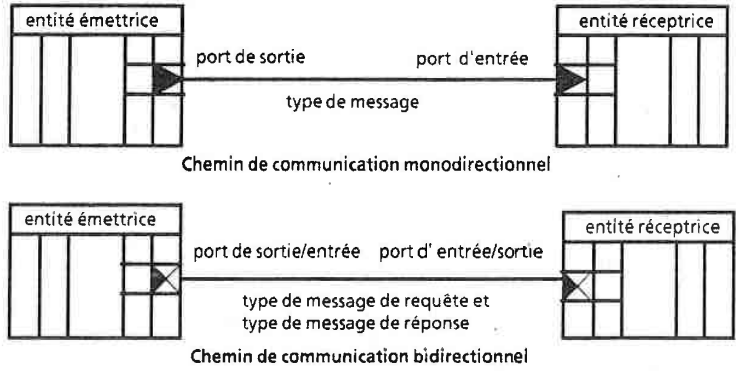


Fig. 2.2 - Interactions entre entités

Un ensemble de chemins de communication ayant un même port de sortie comme origine est appelé "**liaison**". On distingue deux catégories :
- les liaisons bipoints ne comportant qu'un seul chemin,
- les liaisons multipoints comportant p chemins ($p \geqslant 2$) (**cf. fig. 2.3**).

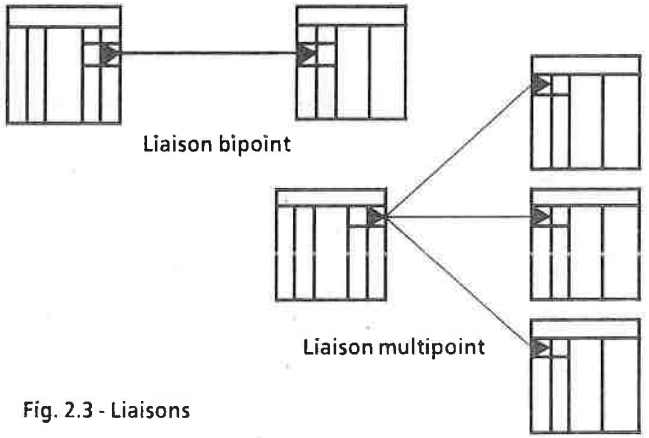


Fig. 2.3 - Liaisons

Une **communication** est une utilisation d'une liaison (tout ou partie de ses chemins) pour un échange d'information. Elle est caractérisée par :

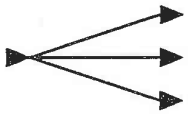
- le type du message échangé,
- le nombre des interlocuteurs concernés ; on distingue les communications "**1 vers 1**" et les communications "**1 vers n**" (ou **diffusions**).

Afin d'obtenir une structuration claire et compréhensible et éviter toute ambiguïté au niveau de la représentation graphique, les règles restrictives suivantes ont été adoptées :

(a) Les ports ne peuvent recevoir ou émettre que des messages d'un seul type. Le nom du type de message admis est porté à côté des chemins.

(b) Les ports de sortie ou de sortie-entrée ne peuvent être l'origine que d'une seule liaison pour éliminer les ambiguïtés d'interprétation :

Exemple :



La figure ci-contre pourrait être interprétée comme une liaison multipoint, ou comme trois liaisons bipoints avec le même type de message, ou comme une liaison multipoint à deux chemins plus une liaison bipoint, etc...

Par contre, les ports d'entrée peuvent faire partie de plusieurs liaisons, ceci ne soulève jamais de problèmes d'interprétation.

(c) Une liaison multipoint véhicule exclusivement des communications "1 vers 1" ou des diffusions vers tous les agents récepteurs. On parle respectivement de liaison multipoint utilisée en **sélection** ou en **diffusion**. Une * près du port de sortie repère graphiquement les diffusions (**cf. fig. 2.4**).

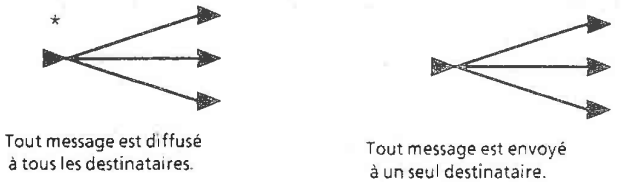


Fig. 2.4 - Diffusion et sélection

(d) Seules les liaisons monodirectionnelles accueillent des diffusions, en raison de l'ambiguïté potentielle des diffusions de requêtes/réponses.

La typologie des liaisons comporte donc cinq catégories (**cf. fig. 2.5**).

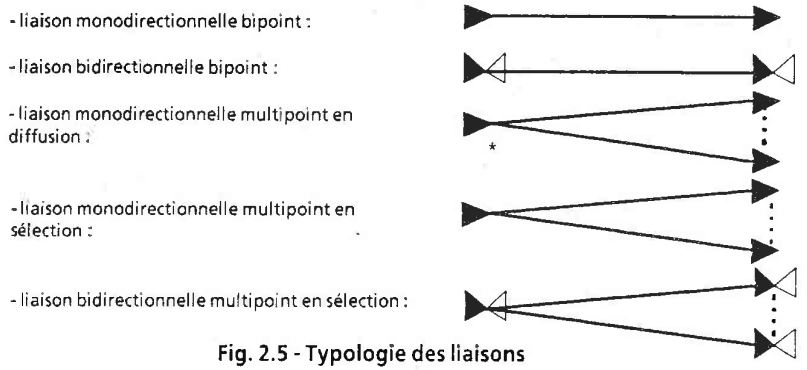


Fig. 2.5 - Typologie des liaisons

II.1.2.2) Langage de spécification comportementale

Considérant qu'une application de commande de procédés est naturellement définie par la relation qu'elle maintient entre ses entrées et ses sorties en vue de contraindre le procédé qu'elle commande à conserver un comportement donné (ce que les automaticiens nomment "lois de commande"), la préférence a été donnée à une spécification en termes d'événements à l'interface (entrées et sorties sur les ports externes) et de relations entre ces événements. Plus précisément, la description des conséquences de certains événements externes ou de certaines mesures de valeurs en fonction de l'état actuel du système, conséquences correspondant pour l'essentiel à des déclenchements d'actions, est fondamentale. Sont donc retenues, la notion de déclenchement (ou précédence causale) entre événements à l'interface et la notion d'état du système ; cet état résulte de tous les événements à l'interface qui ont précédé l'instant considéré. Précédence causale, précédence temporelle et sa traduction en forme de valeurs de variables d'état (mémorisant des histoires d'événements à l'interface) constituent les ingrédients de base de l'outil de spécification comportementale.

Examinons plus en détail ces notions avant d'aborder la manière d'exprimer les propriétés comportementales.

i) Les événements : les ports sont le siège d'événements considérés comme atomiques (ils surviennent complètement ou pas du tout) et instantanés (de durée nulle).

Il s'agit, selon le type de ports, d'émissions de messages (ports de sortie), de réceptions de messages (ports d'entrées), d'émissions de requêtes et de réceptions de réponses (ports de sortie-entrée), de réceptions de requêtes et

d'émissions de réponses (ports d'entrée-sortie). Tous les événements sur un port sont strictement ordonnés.

On note **EV (< nom-port>)**, l'ensemble des événements sur un port ; **<nom-événement > ∈ EV (< nom-port >)**, un événement sur un port ; **< nom-événement >. msg**, le message associé à l'événement et **< nom-événement >. msg. < nom-champ >**, un champ de ce message ; **ord (< nom-événement >)**, l'entier positif qui représente le numéro d'ordre de l'événement sur le port.

Lorsque deux ports sont liés, la liaison est à priori "transparente" : tout événement sur le port origine se confond avec un événement sur le port extrémité. En d'autres termes la communication est sûre et instantanée. On peut spécifier explicitement des liaisons "non transparentes" grâce à une entité intermédiaire ayant les propriétés requises.

ii) La précédence causale : deux événements a et b sont en précédence causale, notée **a ⇒ b**, si l'événement a est la cause directe ou indirecte, par un enchaînement de causalités, de b. La précédence causale est antiréflexive (~ (a ⇒ a)), antisymétrique (si a ⇒ b alors ~ (b ⇒ a)), transitive (si a ⇒ b et b ⇒ c alors a ⇒ c). Les causalités courantes au niveau d'un composant concernent des réceptions qui déclenchent directement des émissions. Plus rarement, on peut trouver des causalités entre émissions. Une réception peut être la cause de plusieurs émissions. Par contre, toute émission a une cause effective unique même si elle possède des causes potentielles multiples.

iii) La précédence temporelle : elle est couramment définie de la manière suivante : sur le même port, deux événements a et b sont en précédence temporelle, notée **a → b**, si a survient avant b selon l'horloge du site, si a et b sont sur des sites différents, on peut assurer que a → b lorsque a est l'émission d'un message et b sa réception, après une communication de durée non nulle. Pour beaucoup d'événements sur des sites différents, il est impossible de dire si l'un précède l'autre ou réciproquement : on dit qu'ils sont **concurrents** (notation : **a‖b**). On note **a ↔ b** la **simultanéité** de a et b : il peut s'agir d'événements simultanés sur le même site ou d'une émission et d'une réception pour une communication de durée nulle. La précédence temporelle est antiréflexive (~(a→a)), antisymétrique (si a → b alors ~ (b → a)) et transitive (si a → b et b → c alors a→c).

La relation de précédence temporelle est parfaitement adaptée à la spécification du comportement des modules (sites virtuels sans horloge commune). Elle l'est moins pour les agents de la phase logique, auxquels la notion de site est étrangère. Au niveau logique, la réflexion s'inscrit naturellement dans un cadre où l'on fait abstraction du temps "lié à l'implantation" pour ne considérer que le "temps réel de l'environnement". Deux événements en relation de causalité sont confondus : a ⇒ b implique a ↔ b sauf si une temporisation selon le temps réel est explicitement indiquée, auquel cas a → b. On retrouve le contexte qui sous-tend les langages de programmation "synchrones" (LUSTRE, ESTEREL,...).

Ce modèle "simplifié", comme nous le verrons plus loin, facilite les validations formelles.

La notion de site apparaît au niveau organique, ce qui invite à utiliser le modèle "complet" de CHEN, en vue de spécifier et de valider d'autres types de propriétés. Les événements sont toujours atomiques et instantanés, mais la relation de causalité n'implique plus que les événements soient confondus. Les délais existent, qui modélisent les temps de traitement et de communication a ⇒ b implique a → b.

iv) Les propriétés comportementales : le comportement des composants est spécifié sous la forme d'un ensemble de propriétés exprimées à l'aide :

- des événements à l'interface,
- des messages associés aux événements,
- d'opérateurs (par priorités décroissantes) :
  - + unaires (quantification et négation) : ∀, ∃, ~
  - + relationnels : ∈, =, ≡, ⇒, →, ↔, ‖
  - + logiques : ∨, ∧
  - + d'implication : #>, <#>.

Ces propriétés peuvent être soit des propriétés invariantes, soit des propriétés d'accessibilité, facilement exprimables grâce à la précédence causale, soit des propriétés sur les données. La forme la plus courante est le déclenchement conditionné de sorties sur des entrées :

∀a (a ∈ EV(A) ∧ P #> ∃b (b ∈ EV(B) ∧ a ⇒ b)) où P est un prédicat quelconque, A un port en entrée et B un port en sortie d'une entité.

A l'aide de ces concepts, on peut spécifier le comportement d'entités de natures très diverses : application toute entière considérée comme une boîte noire unique, agents, modules, "systèmes de communication" liant ces modules.

II.1.3) Choix relatifs à la transition cahier des charges / spécification logique

Le passage cahier des charges / spécification formelle du niveau logique s'effectue en différents niveaux de raffinements. A tout niveau, on adopte la démarche suivante :

- on spécifie, indépendamment, le comportement global attendu de tout système,
- on spécifie, indépendamment, la structure de chaque système en sous-systèmes reliés et le comportement de chaque sous-système,
- on procède à la validation de la cohérence de l'implantation (structuration plus ensemble des comportements des sous-systèmes) par rapport au comportement global attendu du système.

Ce processus de raffinement permet un enrichissement progressif de la description. La démarche permet d'aboutir à la spécification formelle du niveau logique évoquée précédemment.

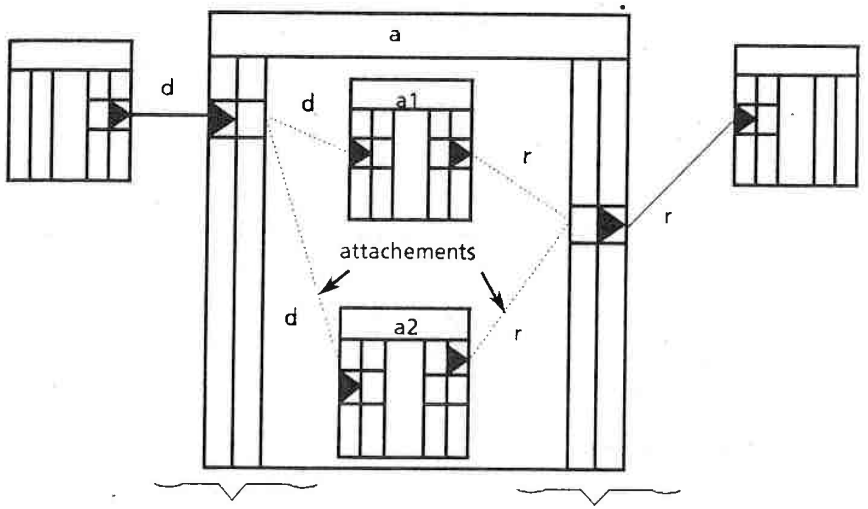
Pour conduire le découpage logique en agents, le concepteur peut travailler de manière descendante, ascendante ou mêler les deux techniques ; il peut s'appuyer sur divers critères : le critère fonctionnel, le critère de correspondance avec des entités caractéristiques du système commandé propre aux approches "orientées objets", **[BOO 86]** etc...

Des conseils généraux peuvent guider ces choix :

- la démarche ascendante est préférable pour les parties bien connues ou partagées par de nombreux sous-systèmes, alors que la démarche descendante est préférable pour analyser les parties moins bien connues **[MEY 78]**;

- un bon agent élémentaire doit être de taille "raisonnable" sa description tenant sur un "feuillet" ; un bon raffinement doit générer un réseau de 3 à 6 agents (cf. les "normes" de SADT).

Lorsqu'une entité est raffinée en un réseau d'entités, il faut assurer l'identité des ports de l'entité englobante et des ports externes du réseau. Le concept d'**attachement** est donc introduit. Tous les ports de l'entité à raffiner sont attachés à un ou plusieurs ports de même type du réseau, avec possibilité de sélection ou de diffusion : ils deviennent de simples points de continuation des chemins de communication ; en entrée, en cas d'attachement multiple, il y a changement de profil de la liaison ; en sortie, en cas d'attachement multiple, il y multiplication de liaisons de mêmes caractéristiques (**cf. fig. 2.6**). Une certaine cohérence doit bien sûr être maintenue en cas d'attachements successifs à plusieurs niveaux, afin de respecter la typologie des liaisons : en aucun cas un chemin de communication "prolongé" par des attachements ne doit mélanger diffusion et sélection, ni comporter plus d'un "tronçon" du type liaison, les autres étant du type attachement. L'outil de manipulation des spécifications devra assurer le respect de ces règles de la même façon qu'il devra contrôler la cohérence des types de ports attachés ou liés, des types de messages sur les chemins d'une liaison multipoint, etc...



la liaison bipoint ainsi "prolongée" équivaut à une liaison multipoint à 2 chemins en sélection.

la liaison bipoint ainsi "prolongée" équivaut à deux liaisons bipoints avec le même port de réception.

Fig 2.6 - Attachements

II.1.4) Choix relatifs à la transition spécification logique / spécification organique

La spécification organique doit rendre compte des contraintes d'organisation (citées au paragraphe II.1.1) qui s'imposent à toutes les solutions. Leur expression se fait en termes de sites virtuels ou modules : on peut donner à titre d'exemples quelques critères de regroupement des agents en modules :

- le couplage entre agents : il peut être de nature temporelle ou informationnelle (cf. les propriétés (4) et (5) du chapite I, caractérisant le contexte des applications étudiées). Un fort couplage informationnel signifie un échange important de messages. Un fort couplage temporel signifie une contrainte de temps stricte sur un type de communication qui apparaît en conséquence des contraintes de temps de réponse à l'interface;
- la cohérence des agents : il est très souhaitable que les modules conservent une certaine unité ;
- les contraintes d'implantation : la localisation de telle fonction ou de telle ressource sur un site peut être imposée (cf. propriété (7)) et l'organisation des modules doit en tenir compte ; ces contraintes peuvent conduire à l'éclatement d'agents élémentaires en plusieurs modules ;

- les contraintes de sûreté : elles peuvent se traduire par l'introduction de nouveaux composants avec des contraintes de localisation physique distinctes ou de nouvelles communications.

L'évaluation des couplages peut être facilitée lorsque le découpage logique est poussé jusqu'à des agents élémentaires à granularité forte, tels que les agents purement séquentiels : toutes les interactions ressortent alors clairement sous la forme des seules communications, qui peuvent être analysées.

II.1.5) Choix relatifs à la transition spécification organique / spécification physique

Il faut en particulier concevoir l'organisation retenue des modules en processus se partageant éventuellement des objets locaux. Le langage de spécification des comportements pourrait permettre de décrire les processus séquentiels, leurs communications et les règles d'accès aux objets partagés par les processus. Par contre, il apparaît moins adapté pour spécifier les structures de données, les interruptions, les exceptions et les aspects temps-réel, aspects fondamentaux pour les applications de commande de procédés. Une description "programmatoire" est donc à ce niveau mieux adaptée. Notons qu'un langage comme ESTELLE **[ISO 86]**, marie des concepts formels (pour la description du contrôle en termes de réseaux d'automates) et des aspects programmatoires en Pascal (pour les structures de données par exemple). Nous présentons succinctement les deux solutions qui consistent à recourir au langage ADA ou à utiliser un pseudo-code "ad-hoc".

Le langage ADA ne s'inscrit pas a priori dans la catégorie des langages de "programmation répartie". Toutefois, diverses approches permettent son utilisation dans un contexte réparti :

- écriture d'un programme ADA par site, l'exécutif réparti se chargeant des interactions entre ces programmes ;
- écriture d'un programme ADA unique, structuré comme un ensemble d'entités (tâches **[SCH 81]**) ou "noeuds virtuels" **[DAR 84]**), interagissant exclusivement par appels d'entrées ; l'exécutif réparti gère les appels d'entrées à distance ;
- utilisation de systèmes automatisés de "répartition du code source" ; le programme ADA est soit modifié par un préprocesseur pour l'adapter à une répartition donnée **[ARM 85]**, soit complété par un programme écrit dans un langage de répartition et un compilateur spécial pour produire un code exécutable sur un environnement spécifique **[COR 83, 84]**.

Le modèle retenu pour les applications visées est beaucoup plus simple que celui qui sous-tend le langage ADA (exemple : réseau de composants défini statiquement). On peut donc juger préférable d'introduire un pseudo-code cohérent avec le modèle retenu et à partir duquel sera généré le squelette du code définitif. Le langage de spécification détaillé (**LSD**), dont une proposition de définition est donnée en annexe 1, a été conçu :

- pour focaliser l'attention sur l'essentiel et laisser de côté l'accessoire (exemple : absence d'arithmétique) ;
- pour faciliter une mise en oeuvre en ADA ; tous les concepts peuvent se traduire en ADA ;
- en reprenant un maximum de constructions linguistiques usuelles dans les langages récents ;
- en prévoyant de manière systématique des paragraphes de description libre (en langue naturelle ou dans des formalismes au choix).

Nous avons utilisé ce LSD comme support de certains développements d'outils de notre environnement..

A l'issue de la spécification organique, le concepteur dispose, pour chaque module, d'une décomposition en agents élémentaires séquentiels obtenue à la phase de structuration logique. Plusieurs facteurs peuvent l'inciter à modifier ce découpage pour définir l'organisation retenue des modules en processus. En particulier :

- le parallélisme de conception : pour des raisons d'efficacité, de traitement (parallélisation) ou de modularité (taille des processus), le concepteur peut décider se scinder un agent élémentaire en un réseau de processus. A l'inverse, le concepteur peut estimer que la fusion d'agents élémentaires permet d'obtenir une structure plus simple et raisonnablement efficace ;
- l'introduction de processus d'interruption qui n'avait pas été envisagée au niveau logique ;
- la prise en compte des contraintes temporelles critiques qui touchent certaines fonctionnalités et qui peuvent entraîner leur isolement dans des processus qui recevront des priorités élevées ;
- le couplage avec des périphériques d'entrée ou de sortie, via des processus spécifiques assurant l'interfaçage ;
- l'introduction de processus de "service" pour réaliser par exemple la gestion de ressources et discipliner les accès à celles-ci.

Les choix des types de communication doivent se faire en parallèle avec le découpage en processus, car la connaissance des interlocuteurs effectifs est un paramètre de choix important.

La validation de la cohérence de la spécification détaillée vis-à-vis des propriétés comportementales peut être pratiquée selon deux approches partielles :

- la technique classique de validation du comportement du réseau de processus au regard du comportement du module peut être utilisée lorsqu'un modèle est implanté exclusivement sous la forme de processus normaux interagissant par des communications. La spécification formelle du comportement de chaque processus peut être rapprochée du schéma de contrôle de ce processus réduit aux actions de communication et aux choix (conditionnelles, choix indéterministes) ;

- dans les cas plus complexes, il est possible de recourir à une technique qui consiste à simuler le fonctionnement de l'application (ou du module) et à enregistrer la trace des événements à l'interface. L'analyse de ces traces permet de détecter d'éventuels comportements ne respectant pas les propriétés requises et d'apporter, en leur absence, un certain degré de confiance dans la solution retenue.

Il faut :

- simuler le comportement de l'environnement de l'application (ou du module) étudié,
- traduire la spécification détaillée dans un langage se prêtant à la simulation.

II.2) ENVIRONNEMENT D'OUTILS

La phase de conception, comme nous venons de la décrire, est complexe avec de nombreuses tâches fastidieuses d'édition et de contrôle. Traverser toutes les étapes logique, organique et physique en gérant les entités (agents et modules) et les liens de communication, tout en veillant à la cohérence de la structuration et à la validation du comportement des différents sous-systèmes par rapport au comportement global attendu sont autant de tâches diversifiées et délicates exigeant un environnement d'outils d'aide à la conception.

Les outils doivent être au service de la méthode et recouvrir les trois niveaux, logique, organique et programmatoire. Leurs domaines d'intervention seraient l'édition de descriptions, les contrôles syntaxiques et sémantiques, et l'automatisation de certaines tâches (passage d'un formalisme à un autre, validations, découpage). Les réalisations doivent se faire avec le souci du contrôle au plus tôt et de la non remise en cause ultérieure des choix déjà validés. L'interaction doit être la plus agréable possible.

Au regard des tâches que le concepteur est amené à effectuer tout au long de la phase de conception, on peut proposer les outils suivants :

a) éditeur/contrôleur graphique des structures,
b) éditeur/contrôleur de formules décrivant le comportement des entités de la structuration,
c) outil d'aide au découpage logique en agents élémentaires séquentiels,
d) outil d'aide à la validation de la cohérence d'un niveau de la décomposition par rapport à un autre,
e) éditeur syntaxique de la forme programmatoire,
f) noyau de simulation d'une application en réseau de modules,
g) vérification de la cohérence entre les événements à l'interface que l'on observe au travers du simulateur (trace) et les comportements spécifiés lors de la description formelle.

Trois de ces outils ont déjà été réalisés :

1) Outil d'aide à la validation de la cohérence, d'un niveau de la décomposition par rapport à un autre (d).

Pour prouver la cohérence du comportement d'une implantation en réseau d'agents vis-à-vis du comportement de l'agent englobant il faut que toutes les liaisons entre entités du réseau aient été définies, qu'elles véhiculent les bonnes informations, que les attachements soient adéquats et que les propriétés des agents du réseau se composent correctement au regard des propriétés globales.

Dans le modèle "simplifié" du niveau logique, cohérence étant synonyme de conservation des propriétés, l'assistance aux preuves de cohérence est obtenue grâce à un outil PROLOG **[LON 87]** permettant la génération de tous les enchaînements de causalités dans une implantation avec leurs conditionnements globaux. Toutes les propriétés décrivant des causalités entre entrées et sorties de l'agent global peuvent être retrouvées à partir des propriétés des agents du réseau.

2) Outil d'aide au découpage logique en agents élémentaires séquentiels (c).

Cet outil vise à capturer le "parallélisme de situation" en termes des simultanéités possibles ou à l'inverse des exclusions temporelles entre entrées et sorties. La commande d'entités parallèles du procédé implique des entrées et sorties simultanées ; inversement l'existence d'entrées ou sorties temporellement exclusives reflète la commande séquentielle d'une même entité élémentaire du procédé quand elle est corroborée par une concentration de relations de causalités.

La méthode proposée est la suivante **[LON 87]** :

- on construit un graphe biparti complet entre entrées et sorties externes avec des poids 0,
- on ajoute des arcs liant entre elles les entrées et les sorties en exclusion temporelle, avec des poids 1,
- on donne des poids particuliers aux arcs liant les entrées et les sorties en relation de causalité : ∞ s'il s'agit d'entrées et sorties sur un même port bidirectionnel, 1 sinon.
- on recherche un ensemble de composantes complètes telles que le poids total des arcs inter composantes soit minimum. Chaque composante correspond à un agent élémentaire muni de ses entrées et sorties externes. Les arcs inter composantes entre entrées et sorties correspondent à des communications du type "déclenchement à distance". L'unicité de la solution n'est pas garantie.

Les justifications sont les suivantes : dans une composante complète, par définition, tous les sommets sont liés; en particulier, toutes les entrées et toutes les sorties; il n'y a donc pas possibilité de parallélisme entre entrées et entre sorties d'un même agent. La minimisation du poids total des arcs inter composantes relève évidemment de la recherche d'un couplage faible entre entités à cohésion forte, caractéristique généralement retenue comme critère de

bon découpage modulaire. Les poids ont été fixés de manière standard pour rendre compte de la cohésion logique entre entrées et sorties :

- les poids 0 correspondent à des entrées et sorties sans relation de causalité mais qui peuvent néanmoins se retrouver sur un même agent en conséquence d'autres relations;
- les poids intermédiaires 1 privilégient le regroupement d'entrées et de sorties en relation de causalité. Ces relations servent à évaluer grossièrement les couplages et la cohésion;
- les poids ∞ correspondent aux entrées et sorties sur un même port bidirectionnel, qui ne peuvent pas être dissociées, par définition.

3) Noyau de simulation d'un réseau de modules (f).

Un travail a été mené au CRIN dont certains aspects pourraient être repris dans cette optique **[GOF 84]**. Il s'agit de la construction en SIMONE **[BEZ 76]** de maquettes de simulation d'applications structurées selon un modèle semblable à celui exposé précédemment essentiellement en vue d'étudier les conséquences des défaillances possibles des sites et des liaisons.

Notre contribution vise à réaliser les outils des points a) et e).

BIBLIOGRAPHIE

[ARM 85] ARMITAGE, CHELINI
ADA software on distributed targets : a survey of approaches.
ADA Letters, Vol 4, n°4, 2/85, pp 32,77.

[BEZ 76] BEZIVIN, KAUBISCH, LEROY, NEBUT, RANNOU
Simone : Manuel de référence.
Publication SFER/IRIA, 1976.

[BOO 86] BOOCH G.
Object oriented development.
IEEE Trans. on Soft. Eng., Vol SE-12, n°2, 2/86, pp 211,221.

[COR 83] CORNHILL D.
A survivable distributed computing system for embedded application programs written in ADA.
ADA Letters, 1983.

[COR 84] CORNHILL D.
Partitioning ADA programs for execution on distributed systems.
Honeywell Syst. and Research Center, MINNEAPOLIS, 1984.

[DAR 84] DARPRA, MADERNA, STAMMERS
Using ADA and APSE to support distributed multi-microprocessor targets.
ADA Letters, Vol. 3, n°6, 5-6/84.

[GOF 84] GOFFIC P.
Architecture de logiciels répartis. Spécification et simulation.
Thèse Doct. Ing., NANCY, 6/84.

[ISO 86] ISO/TC97/SC21/WG16-1 DP9074
Estelle : a formal Description Technique based on an Extended State Transition Model, juillet 1986.

[KRA 83] KRAMER, MAGEE, SLOMAN, LISTER
Conic : an integrated approach to distributed computer control System IEE Proc., Vol 130, n°1, 1/1983.

[LON 87] LONCHAMP J.
Conception des applications informatiques réparties en commande de procédés industriels : une démarche, des outils.
Thèse d'Etat, NANCY I, 5/87.

[MEY 78] MEYER, BAUDOIN
Méthodes de programmation.
Eyrolles, PARIS, 1978.

[SCH 81] SCHUMAN, CLARKE, NIKOLAOU
Programming distributed applications in ADA : a first approach.
Proc. Int. Conf. on Parallel Processing, 1981.

[TAR 83] TARDIEU, ROCHFELD, COLLETTI
La méthode Merise - Principes et outils.
Ed. d'organisation, PARIS, 1983.

CHAPITRE III) REALISATION D'UN ENVIRONNEMENT ORIENTE VERS LA GESTION DES DESCRIPTIONS STRUCTURELLES

III.1) INTRODUCTION

La méthodologie de conception, telle qu'elle a été décrite, amène l'utilisateur à manipuler une hiérarchie de niveaux de décomposition. Chaque niveau correspond à un réseau d'entités communiquantes dont il faut spécifier le comportement qui doit être cohérent avec le comportement du niveau supérieur. La concordance entre la description structurelle en termes de composants interreliés et la description comportementale et programmatoire est mise en évidence sur la **Fig. 3.1**.

L'environnement décrit dans ce chapitre assiste le concepteur pour tous les aspects structurels depuis le cahier des charges jusqu'à l'obtention d'un squelette de programme dans le langage ad hoc "LSD".

Les principes qui ont guidé cette réalisation sont soulignés en premier lieu. Un exemple complet d'utilisation de l'environnement termine le chapitre.

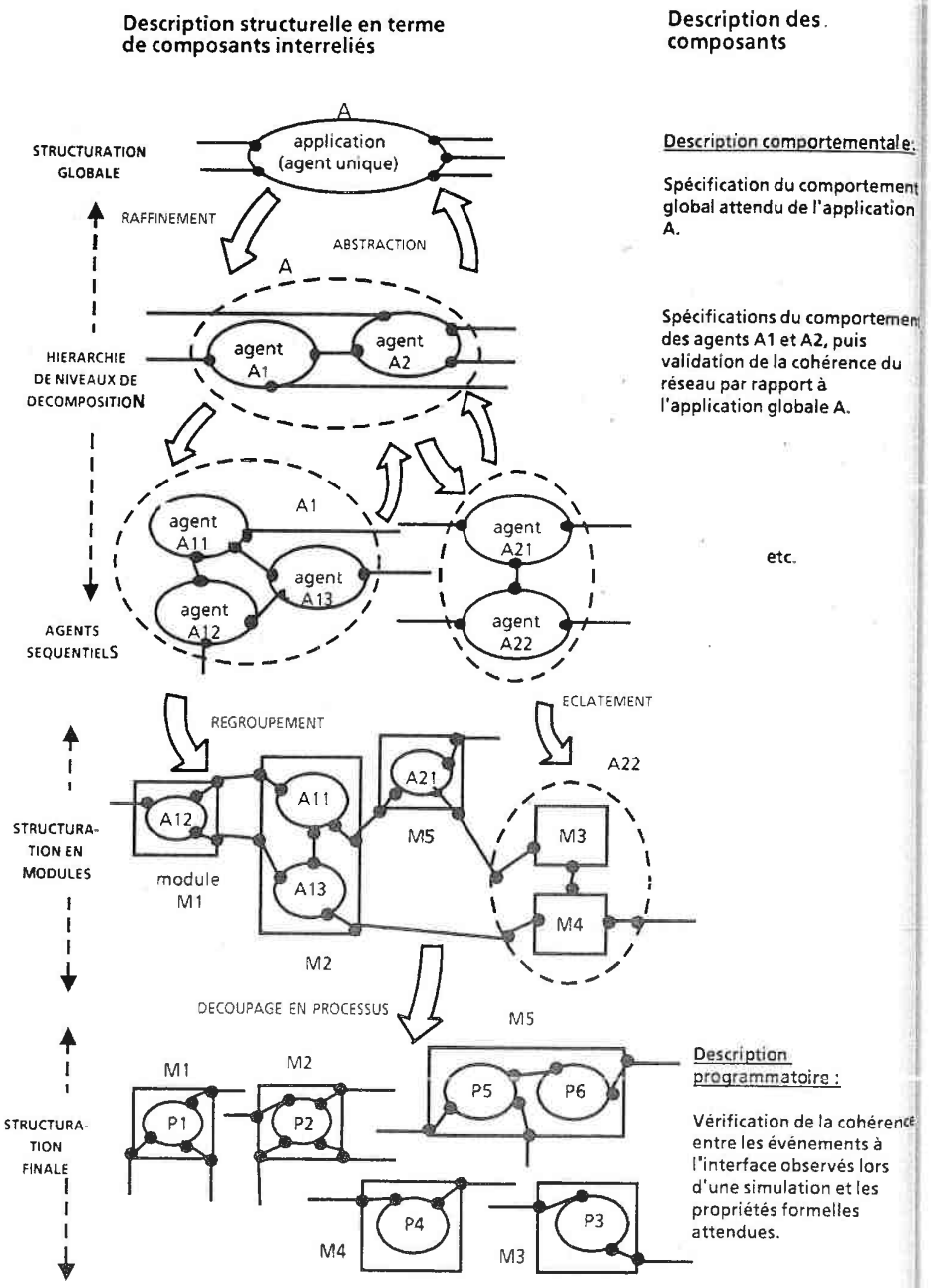


Fig. 3.1 - Méthodologie de conception

III.2) PRINCIPES DE BASE - SOLUTIONS RETENUES

La réalisation de l'environnement de gestion des descriptions structurelles a été guidée par les principes suivants :

- l'environnement doit mettre à la disposition de l'utilisateur un moyen unique pour définir et manipuler les aspects structurels ; la description structurelle ne doit pouvoir être remise en cause lors de la spécification du comportement des composants ou lors de leur description en termes programmatoires ; il faut toujours revenir à l'outil de gestion des structures qui assure les contrôles indispensables.
- l'outil doit procéder à un maximum de contrôles syntaxiques et sémantiques le plus tôt possible afin de maintenir une structure cohérente ; ces contrôles immédiats sont complétés par un contrôle de complétude différé.
- le concepteur doit pouvoir travailler simultanément aux différents niveaux mis en évidence par la figure 3.1 :

  - décomposition de l'application en termes d'agents élémentaires,
  - structuration organique à l'aide de modules,
  - spécification détaillée des modules en processus et objets partagés.

  Ces 3 phases sont étroitement liées. En effet, la structuration en modules se fait sur la base des agents. Pour la spécification détaillée des modules, l'utilisateur peut se baser sur les agents logiques que regroupe le module, lors du choix des processus à créer. L'utilisateur doit pouvoir, par exemple, définir des agents et les modules les contenant, puis spécifier la structuration interne de ces modules en processus, revenir au niveau création d'agents, etc. Ces 3 niveaux de représentation (hiérarchie des agents, réseau des modules, structuration interne des modules) doivent donc cohabiter et l'outil doit veiller à maintenir la cohérence des descriptions.
- la convivialité doit être maximum.

La spécification terminale étant de nature textuelle (programmatoire), nous pourrions également envisager une description textuelle des structures lors des phases de conception logique et organique.
Ces structurations, qui sont hiérarchiques, pourraient être décrites de différentes manières; par exemple:

- par imbrication complète des implantations à profondeur quelconque (cf.**fig.3.2.a**); les raffinements se traduisent alors par des insertions de textes,
- par description séparée des différentes implantations à profondeur 1 (cf.**fig.3.2.b**); les raffinements se traduisant alors par des adjonctions de description

Fig.3.2.a

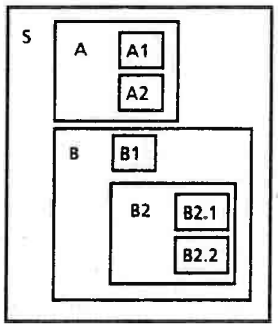
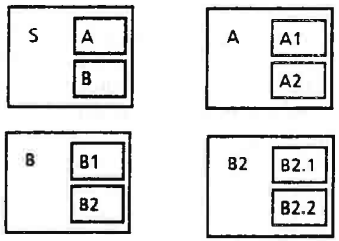


Fig.3.2.b



Dans les 2 cas, la manipulation par un éditeur classique est très pénible et ne permet d'avoir une vue d'ensemble du travail en cours (cas a) ou de parcourir facilement la hiérarchie (cas b). Les contrôles, mêmes syntaxiques, seront difficiles à gérer.

Un éditeur syntaxique, tel MENTOR **[DON 79]**, garantit la correction syntaxique des descriptions et améliore la puissance des manipulations réalisables.
Les déplacements dans les hiérarchies sont aisées et il est possible d'extraire une sous-hiérarchie et de n'en représenter que les implantations à profondeur limitée.
On peut par exemple disposer d'un programme structuré comme dans la figure a, puis extraire la partie B, dont la représentation de profondeur 1 est donnée dans la figure 3.2.b .
Même si cette possibilité apporte une nette amélioration, elle reste toutefois d'un maniement lourd (cf. III.4). En effet, les éditeurs syntaxiques de "première génération" MENTOR **[DON 79]**, Cornell Program Synthetiser **[TEI 81]**, GANDALF **[HAB 79]**, sont dotés d'interfaces utilisateur assez pauvres.
Des tentatives ont été menées pour améliorer cette interface : CEPAGE **[MEY 85]** est un éditeur structurel pleine page doté d'un algorithme qui calcule la manière d'abréger le texte de part et d'autre de la zone d'intérêt pour maintenir une vue

d'ensemble du contexte. On peut douter cependant que pour un langage plus complexe que PASCAL, on puisse apporter suffisamment d'informations textuelles dans une fenêtre-écran de taille très limitée.

La compréhension, à postériori, d'une structure logique ou organique n'étant pas triviale, elle nécessite donc un minimum de réflexion et parfois plusieurs parcours de la description textuelle. A ce titre il est légitime d'exiger une représentation graphique qui est beaucoup plus "parlante" qu'un texte, même structuré.
Des tentatives d'introduction de vues graphiques de programmes existent comme dans le système PECAN **[REI 84]** par exemple.

Si la compréhension et la conception, se trouvent facilitées par un outil graphique, il ne faut pas tomber dans une description trop "fine" de l'application sous peine de surcharger la représentation graphique. Les entités communicantes (agents, modules), les ports, les liaisons et attachements se prêtent bien à une représentation graphique claire. Par contre la structuration interne des modules, avec ses interactions par partage d'objets de natures très diverses, s'y prête beaucoup moins.
Toute la partie spécification détaillée relève donc plutôt de l'éditeur syntaxique classique.

Face à ces contraintes, se pose donc le problème de cohabitation entre les 2 outils (graphique et syntaxique), étant donné la carence actuelle de ce type d'environnement mixte.
Une solution simple par interfaçage de 2 éditeurs a été recherchée dans le but de cacher le plus longtemps possible l'éditeur syntaxique.

Les 2 éditeurs et leur interfaçage sont présentés dans les paragraphes suivants.

III.3) DESCRIPTION DE L'ENVIRONNEMENT

III.3.1) Editeur / contrôleur graphique de structures

a) Objets manipulés

i) présentation informelle :

Les structures sont à base d' "**agents**" et de "**modules**". Les agents matérialisent des sous-systèmes de granularité quelconque et seront déclarés en tant que modules s'ils correspondent à des sites virtuels. Ces entités qui communiquent par échange de messages typés via des ports et des liaisons forment un "**réseau**".

La démarche de conception amène le concepteur à raffiner une entité en un réseau d'entités en "prolongeant" les liaisons aboutissant ou partant de l'entité raffinée. Ces deux réseaux forment une "**hiérarchie**" et ont un point commun : le premier comporte l'entité à l'état "**plein**", le second l'entité à l'état "**creux**" avec à "l'intérieur" le réseau le raffinant (**Cf. fig. 3.3.1**).

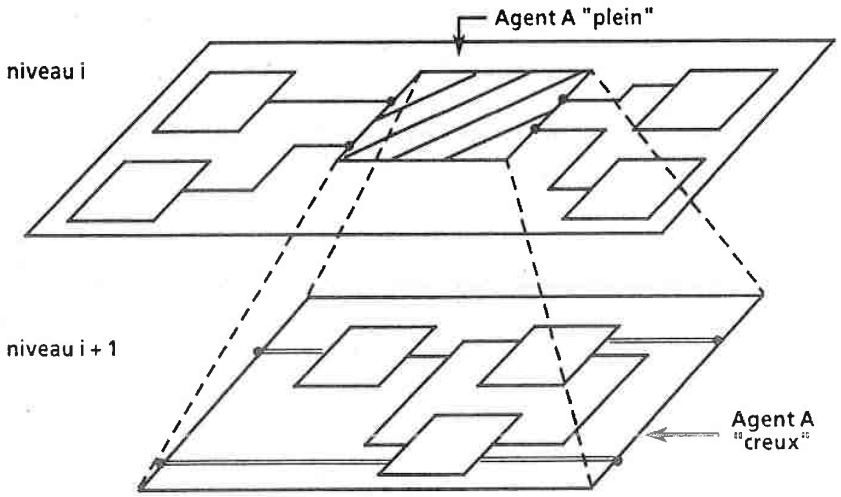


Fig. 3.3.1

Les attachements prolongent les liaisons en reliant les ports de l'agent creux aux ports des agents pleins. Le processus de découpage peut être réitéré au sein de la même hiérarchie, pour obtenir une hiérarchie avec plusieurs niveaux de réseaux, ou à l'extérieur de la hiérarchie, auquel cas plusieurs hiérarchies indépendantes peuvent coexister, momentanément. En particulier, la démarche ascendante permet d'encapsuler un réseau d'agents pleins dans une entité creuse sans créer l'agent plein correspondant.

Toute hiérarchie sera donc composée d'un réseau "**racine**" et d'une suite de réseaux reliés deux à deux par un agent à l'état plein et l'agent, de même nom, à l'état creux. Le réseau "racine" peut ou non contenir un agent creux. Une "**application**" est constituée d'une ou plusieurs hiérarchies.

ii) description plus formelle :

type **application** = (ap : liste de hiérarchies) ;
Une application est une liste de hiérarchies.

type **hiérarchie** = (h : liste de réseaux) ;
Invariant : ($\forall$ r $\in$ h , r $\neq$ premier-élément(h), $\exists$ a : agent | a $\in$ r $\wedge$ état(a) = "creux")
Une hiérarchie est une liste de réseaux. Tout réseau non "racine" d'une hiérarchie, c'est-à-dire non premier élément de la hiérarchie, contient un agent à l'état "creux".

type **réseau** = (agent-creux : agent | $\emptyset$, la : liste agents, ll : liste liaisons, lt : liste attachements) ;
Invariant : (Si agent-creux $\neq \emptyset$ alors état(agent-creux) = "creux") $\wedge$ ($\forall$ b $\in$ la, état(b) = "plein") $\wedge$ ($\forall$ a,b : agent, r, r' : réseau, a $\in$ r, b $\in$ r', nom(a) $\neq$ nom(b))
Un réseau comprend une liste d'agents à l'état plein, une liste de liaisons, une liste d'attachements et éventuellement un agent à l'état creux.
Il y a au plus un agent à l'état "creux" par réseau. Le nom d'un agent doit être unique au sein d'une application.

type **agent** = (nom : caractères, coord : $\mathbb{R}\times\mathbb{R}$, lp : liste ports, taille : $\mathbb{N}$, genre : "agent" | "module", état : "plein" | "creux")
Un agent est caractérisé par son nom, sa position à l'écran, la liste de ses ports, le nombre d'agrandissements qu'il a subis (taille), le fait qu'il soit déclaré ou non en tant que module, ainsi que par son état ("plein" ou "creux").

type **liaison** = (nom-agent-émetteur, nom-port-émetteur, nom-agent-récepteur, nom-port-récepteur, requête, réponse : caractères | $\emptyset$) ;
Invariant : (nom-agent-émetteur = $\emptyset \Leftrightarrow$ nom-port-émetteur = $\emptyset$) $\wedge$
(nom-agent-récepteur = $\emptyset \Leftrightarrow$ nom-port-récepteur = $\emptyset$) $\wedge$
(nom-agent-émetteur $\neq \emptyset \vee$ nom-agent-récepteur $\neq \emptyset$) $\wedge$

(requête ≠ Ø ∨ réponse ≠ Ø) ∧
((∃ a,b : agent, r : réseau | a,b ∈ r ∧ nom(a) = nom-agent-émetteur ∧ état(a) = "plein" ∧ nom(b) = nom-agent-récepteur ∧ état(b) = "plein") ∨ (∃ a : agent, r : réseau | a ∈ r ∧ (nom(a) = nom-agent-émetteur ∨ nom(a) = nom-agent-récepteur) ∧ état(a) = "plein"))

Une liaison comprend les noms d'agent et port émetteurs, les noms d'agent et port récepteurs et les noms des types de messages de requête et de réponse.
Une liaison peut être incomplète momentanément (demi-lien), ou relier deux agents "pleins" d'un même réseau.

type **attachement** = (nom-agent-émetteur, nom-port-émetteur, nom-agent-récepteur, nom-port-récepteur, requête, réponse : caractères | Ø) ;
Invariant : (nom-agent-émetteur = Ø ⇔ nom-port-émetteur = Ø) ∧
(nom-agent-récepteur = Ø ⇔ nom-port-récepteur = Ø) ∧
(nom-agent-émetteur ≠ Ø ∨ nom-agent-récepteur ≠ Ø) ∧
(requête ≠ Ø ∨ réponse ≠ Ø) ∧
((∃ a,b : agent, r : réseau | a,b ∈ r ∧ nom(a) = nom-agent-émetteur ∧ nom(b) = nom-agent-récepteur ∧ ((état(a) = "plein" ∧ état(b) = "creux") ∨ (état(a) = "creux" ∧ état(b) = "plein")) ∨ (∃ a : agent, r : réseau | a ∈ r ∧ (nom(a) = nom-agent-émetteur ∨ nom(a) = nom-agent-récepteur) ∧ (état(a) = "plein") ∨ (état(a) = "creux")))
Un attachement comprend les noms d'agent et port émetteurs, les noms d'agent et port récepteurs et les noms des types de messages de requête et de réponse.
Un attachement peut être incomplet ou relier un agent "plein" à un agent "creux" du réseau.

type **port** = (nom, nom-agent : caractères, type : s-port | e-port | se-port | es-port | sc-port | sd-port | ec-port | ed-port)
Invariant : (∀ a : agent, p1, p2 : port, p1 ∈ a, p2 ∈ a, nom(p1) ≠ nom(p2))
Un port est caractérisé par son nom, le nom de l'agent auquel il appartient et le sens de la communication.
Tous les ports d'un même agent ont des noms différents.

b) Fonctions de manipulation

i) présentation informelle :

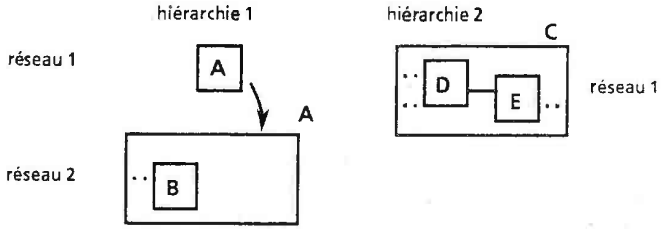
Les différents concepts : application, hiérarchie, réseau, agent, liaison, attachement et port, qui sont à la base de la description structurelle d'une application, se manipulent à l'aide de fonctions que l'on peut classer en cinq catégories : fonctions de création et de suppression de concepts, fonctions de modification des caractéristiques des concepts, fonctions de déplacement pour

naviguer dans les hiérarchies ou déplacer un agent au sein d'un réseau et fonctions de contrôle utilisées pour valider une application.

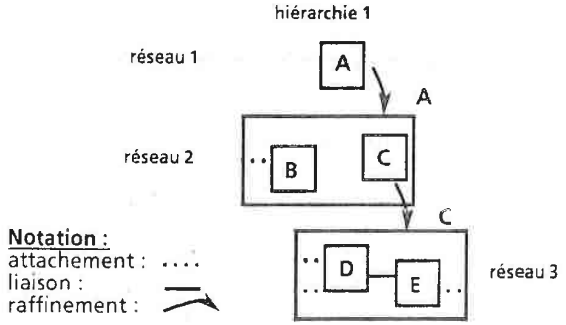
ii) description plus formelle :

fonctions de création

créer-agent : (n : caractères, c : ℝ×ℝ) → a : agent | Ø;
Si (∃ b : agent, r : réseau, b ∈ r | nom(b) = n) alors
Si (état(b) = "plein") alors Ø
sinon %état(b) = "creux" %
Si (hiérarchie(a) = hiérarchie(b)) alors Ø
sinon (nom(a) = n ∧ coord(a) = c ∧ état(a) = "plein" ∧ lp(a) = lp(b) ∧ genre(a) = genre(b) ∧ taille(a) = taille(b) ∧ (∀ r ∈ hiérarchie(b), hiérarchie(r) = hiérarchie(a)))
sinon (nom(a) = n ∧ coord(a) = c ∧ genre(a) = "agent" ∧ état(a) = "plein" ∧ lp(a) = Ø ∧ taille(a) = 1)
La création d'un agent peut avoir pour effet de "recoller" deux hiérarchies si l'agent "creux" existait déjà (cf figure ci-dessus).



Création de l'agent C dans le réseau 2 de la hiérarchie 1



La fonction auxiliaire **hiérarchie** retourne la hiérarchie à laquelle appartient l'agent.

créer-port : (np, na : caractères, t : type) → p : port | ∅;
Si (∃ a : agent, r : réseau, a ∈ r | nom(a) = na) alors
Si (∃ q : port, q ∈ lp(a) ∧ nom(q) = np) alors ∅
sinon (lp(a) = lp(a) ∪ p ∧ nom(p) = np ∧ type(p) = t ∧ (si (∃ b ∈ r, nom(b) = na ∧ état(b) ≠ état(a)) alors (lp(b) = lp(b) ∪ p)))
sinon ∅
La création d'un port se répercute, le cas échéant, sur l'agent "plein" ou "creux" correspondant.

créer-liaison : (na,np1,nb,np2,t1,t2 : caractères | ∅) → l : liaison | ∅;
Si (∃ a,b : agent, p1,p2 : port, r : réseau | a,b ∈ r ∧ nom(a) = na ∧ nom(b) = nb ∧ nom(p1) = np1 ∧ nom(p2) = np2 ∧ état(a) = "plein" ∧ état(b) = "plein" ∧ p1 ∈ lp(a) ∧ p2 ∈ lp(b) ∧ liaison-compatible(p1,p2)) alors
(nom-agent-émetteur(l) = na ∧ nom-agent-récepteur(l) = nb ∧ nom-port-émetteur(l) = np1 ∧ nom-port-récepteur(l) = np2 ∧ requête(l) = t1 ∧ réponse(l) = t2)
sinon ∅

On vérifie la compatibilité des types de ports reliés à l'aide de la fonction auxiliaire :
liaison-compatible : (p1,p2 : port) → booléen;
Si ((type(p1) = s-port ∧ type(p2) = e-port)
∨ (type(p1) = se-port ∧ type(p2) = es-port)
∨ (type(p1) = sc-port ∧ type(p2) = e-port | ec-port)
∨ (type(p1) = sd-port ∧ type(p2) = e-port | ed-port)) alors vrai
sinon faux

créer-attachement : (na,np1,nb,np2,t1,t2 : caractères | ∅) → l : attachement | ∅;
Si (∃ a,b : agent, p1,p2 : port, r : réseau | a,b ∈ r ∧ nom(a) = na ∧ nom(b) = nb ∧ nom(p1) = np1 ∧ nom(p2) = np2 ∧ ((état(a) = "plein" ∧ état(b) = "creux") ∨ (état(a) = "creux" ∧ état(b) = "plein")) ∧ p1 ∈ lp(a) ∧ p2 ∈ lp(b) ∧ attachement-compatible(p1,p2)) alors
(nom-agent-émetteur(l) = na ∧ nom-agent-récepteur(l) = nb ∧ nom-port-émetteur(l) = np1 ∧ nom-port-récepteur(l) = np2 ∧ requête(l) = t1 ∧ réponse(l) = t2)
sinon ∅

On vérifie la compatibilité des types de ports reliés à l'aide de la fonction auxiliaire :
attachement-compatible : (p1,p2 : port) → booléen;
Si ((type(p1) = s-port | sc-port ∧ type(p2) = s-port | sc-port)
∨ (type(p1) = s-port | sd-port ∧ type(p2) = s-port | sd-port)
∨ (type(p1) = e-port | ec-port ∧ type(p2) = e-port | ec-port)
∨ (type(p1) = e-port | ed-port ∧ type(p2) = e-port | ed-port)

∨ (type(p1) = type(p2) ∧ type(p1) = se-port | es-port)) alors vrai
sinon faux

créer-module : (na : caractères) → b : booléen;
Si (∃ a : agent | nom(a) = na ∧ genre(a) = "agent" ∧ état(a) = "plein" ∧ (∀ b ∈ agent-sous(a), genre(b) = "agent") alors (genre(a) = "module" ∧ b = vrai)
sinon b = faux
Avant de déclarer un agent en tant que module, on vérifie qu'il n'y a pas de modules dans les réseaux situés "sous" l'agent dans la hiérarchie.

La fonction suivante donne la liste des agents situés "sous" un agent.
agent-sous : (a : agent) → lm : liste agents;
∀ b : agent, b ∈ lm ⇔ (∃ r(i) i = 1..n,r(i) : réseau | a ∈ r(1) ∧ b ∈ r(n) ∧ (∀ r(i),r(i + 1), i = 1..n-1,∃ a(i),a(i + 1) : agent, a(i) ∈ r(i), a(i + 1) ∈ r(i + 1) ∧ a(i) ∈ la(r(i)) ∧ état(a(i)) = "plein" ∧ a(i + 1) = agent-creux(r(i + 1)) ∧ nom(a(i)) = nom(a(i + 1)))
Un agent b est placé "sous" un agent a s'il existe une "chaîne" de réseaux reliés deux à deux par un agent de même nom, à l'état "plein" dans un réseau et à l'état "creux" dans l'autre. Cette chaîne part du réseau contenant a pour aboutir au réseau contenant b.

fonctions de suppression

supprimer-agent : (na : caractères) → b : booléen;
Si (∃ a : agent, r : réseau | a ∈ r ∧ nom(a) = na ∧ état(a) = "plein") alors (la(r) = la(r)-a ∧ (∀ l ∈ ll(r), (na = nom-agent-émetteur(l) ∨ na = nom-agent-récepteur(l)) ⇒ supprimer-liaison(l)) ∧ (∀ t ∈ lt(r), (na = nom-agent-émetteur(t) ∨ na = nom-agent-récepteur(t)) ⇒ supprimer-attachement(t)) ∧ b = vrai) ∧ (si (∃ b : agent, r' : réseau | b ∈ r' ∧ nom(b) = na ∧ état(b) = "creux") alors supprimer-réseau(r'))
sinon
si (∃ a : agent, r : réseau, h : hiérarchie, ap : application | a ∈ r ∧ hiérarchie(a) = h ∧ nom(a) = na ∧ état(a) = "creux") alors (supprimer-réseau(r) ∧ ap = ap-h ∧ b = vrai)
sinon b = faux
La suppression d'un agent entraîne la suppression des liaisons et attachements y aboutissant ainsi que la hiérarchie de réseaux situés éventuellement "sous" l'agent.

supprimer-réseau : (r : réseau);
(∀ a : agent, a ∈ la(r), supprimer-agent(a)) ∧ r = ∅

supprimer-liaison : (l : liaison) → b : booléen;
Si (∃ r : réseau | l ∈ r) alors (ll(r) = ll(r)-l ∧ b = vrai)
sinon b = faux

supprimer-attachement : (t : attachement) → b : booléen;
Si (∃ r : réseau | t ∈ r) alors (lt(r) = lt(r)-t ∧ b = vrai)
sinon b = faux

supprimer-port : (n,na : caractères) → b : booléen;
Si (∃ r : réseau, a : agent, p : port | nom(a) = na ∧ a ∈ r ∧ nom(p) = n ∧ p ∈ lp(a)) alors (b = vrai
∧ (Si état(a) = "plein" alors
(∀ l ∈ ll(r) | ((n = nom-port-émetteur(l) ∧ na = nom-agent-émetteur(l))
∨ (n = nom-port-récepteur(l) ∧ na = nom-agent-récepteur(l))) ⇒
supprimer-liaison(l)) ∧
(∀ t ∈ lt(r) | ((n = nom-port-émetteur(t) ∧ na = nom-agent-émetteur(t))
∨ (n = nom-port-récepteur(t) ∧ na = nom-agent-récepteur(t))) ⇒
supprimer-attachement(t)) ∧ lp(a) = lp(a)-p)
∧ (Si état(a) = "creux" alors
(∀ t ∈ lt(r) | ((n = nom-port-émetteur(t) ∧ na = nom-agent-émetteur(t))
∨ (n = nom-port-récepteur(t) ∧ na = nom-agent-récepteur(t))) ⇒
supprimer-attachement(t)) ∧ lp(a) = lp(a)-p))
On supprime le port de l'agent désigné avec les liaisons et les attachements y aboutissant, ainsi que celui de l'agent "plein" ou "creux" correspondant.

fonctions de modification

renommer agent : (na,nb : caractères) → b: booléen;
Si (∃ a : agent | nom(a) = na ∧ (∀ b : agent, nom(b) ≠ nb)) alors
nom(a) = nb ∧ b = vrai
sinon b = faux

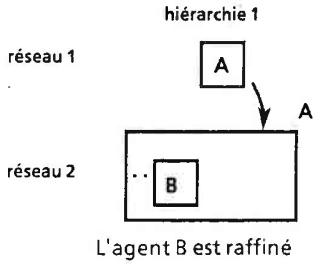
renommer port : (na,np1,np2 : caractères) → b: booléen;
Si (∃ a : agent | nom(a) = na ∧ (∃ p : port | nom(p) = np1 ∧ p ∈ lp(a)) ∧ (∀ q : port, q ∈ lp(a) | nom(q) ≠ np2)) alors
nom(p) = np2 ∧ b = vrai
sinon b = faux

agrandir agent : (na : caractères) → b : booléen;
Si (∃ a : agent | nom(a) = na) alors taille(a) = taille(a) + 1 ∧ b = vrai
sinon b = faux

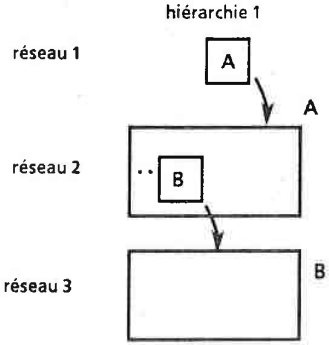
Le renommage d'un agent et d'un port et l'agrandissement d'un agent doivent s'effectuer sur l'agent "plein" et l'agent "creux".

raffiner agent : (na : caractères) → r : réseau, b : agent;
Si (∃ a : agent | nom(a) = na ∧ état(a) = "creux") alors r = ∅ ∧ b = ∅
sinon
Si (∃ a : agent | nom(a) = na ∧ état(a) = "plein") alors
nom(b) = na, lp(b) = lp(a), taille(b) = taille(a), genre(b) = genre(a),
état(b) = "creux", agent-creux(r) = b, la(r) = ∅, ll(r) = ∅, lt(r) = ∅

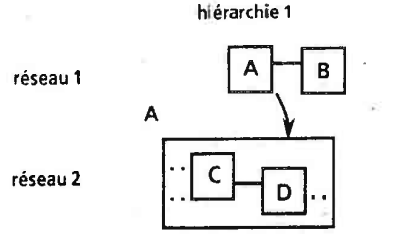
sinon r = ∅ ∧ b = ∅
Cette fonction de base de la démarche descendante crée un réseau "sous" un agent.



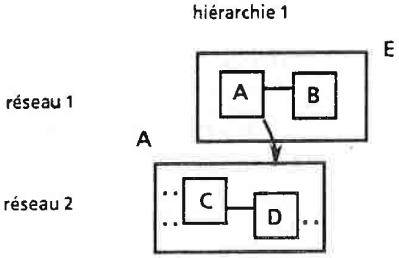
L'agent B est raffiné



encapsuler agent : (na : caractères, r : réseau) → b : agent;
Si (∃ a : agent | nom(a) = na) alors b = ∅
sinon
Si agent-creux(r) ≠ ∅ alors b = ∅
sinon nom(b) = na, lp(b) = ∅, taille(b) = 1, genre(b) = "agent",
état(b) = "creux", agent-creux(r) = b
Cette fonction, qui prépare l'insertion d'une hiérarchie dans une autre, est utilisée lors de la démarche ascendante. L'insertion sera effective dès la création d'un agent "plein" de même nom (cf. fonction créer-agent)



Le réseau 1 est encapsulé dans l'agent E



fonctions de déplacement

monter : (r : réseau) → bool : booléen;
Si (∃ r' : réseau; a,b : agent | a ∈ r, b ∈ r', nom(a) = nom(b), b ∈ la(r), a = agent-creux(r')) alors (réseau courant = r' ∧ bool = vrai)
sinon bool = faux

descendre : (r : réseau) → bool : booléen;
Si (∃ r' : réseau; a,b : agent | a ∈ r, b ∈ r', nom(a) = nom(b), a ∈ la(r), b = agent-creux(r')) alors (réseau courant = r' ∧ bool = vrai)
sinon bool = faux

Ces fonctions permettent de visualiser un autre réseau à l'écran.

déplacer agent : (na : caractères, c : $\mathbb{R} \times \mathbb{R}$) → b : booléen;
Si (∃ a : agent | nom(a) = na ∧ état(a) = "plein") alors
coord(a) = c ∧ b = vrai
sinon b = faux
On peut déplacer un agent au sein d'un même réseau.

fonctions de contrôle

contrôle père : → b : booléen;
Si (∃! h : hiérarchie, h ∈ ap : application ∧ (∃! a : agent, a ∈ premier-élément(h), état(a) = "plein")) alors b = vrai
sinon b = faux
On vérifie l'unicité de la hiérarchie et de l'agent se trouvant dans le réseau "racine".

contrôle module : → bool : booléen;
Si (∀ a : agent, (genre(a) = "module" ∨ (∃ b : agent, genre(b) = "module" ∧ a ∈ agent-sous(b)))) alors bool = vrai
sinon bool = faux
Tout agent doit être déclaré en tant que module ou être inclus dans un module (placé hiérarchiquement sous un module).

contrôle port : → b : booléen;
Si (∀ a : agent, ∃ p ∈ lp(a)) ∧ (∀ a : agent, ∀ p ∈ lp(a), est-utilisé-à-bon-escient(p)) alors b = vrai
sinon b = faux

est-utilisé-à-bon-escient : (p : port) → b : booléen;
Si (type(p) = s-port ∨ type(p) = e-port ∨ type(p) = se-port ∨ type(p) = es-port ∨ type(p) = ec-port ∨ type(p) = ed-port) ∧ (∃ (l : liaison ∨ a : attachement) | (nom-port-émetteur(l) = p ∨ nom-port-récepteur(l) = p ∨ nom-port-émetteur(a) = p ∨ nom-port-récepteur(a) = p)) alors b = vrai
sinon
si (type(p) = sc-port ∨ type(p) = sd-port) ∧ (∃ (l,l' : liaison ∨ a,a' : attachement ∨ (l : liaison ∧ a : attachement)) | ((nom-port-émetteur(l) = p ∧ nom-port-émetteur(l') = p) ∨ (nom-port-récepteur(l) = p ∧ nom-port-récepteur(l') = p) ∨ (nom-port-émetteur(a) = p ∧ nom-port-émetteur(a') = p) ∨ (nom-port-récepteur(a) = p ∧ nom-port-récepteur(a') = p) ∨ (nom-port-émetteur(l) = p ∧ nom-port-émetteur(a) = p) ∨ (nom-port-récepteur(l) = p ∧ nom-port-récepteur(a) = p))) alors b = vrai
sinon b = faux
Tout agent doit posséder au moins un port et tout port doit avoir un nombre de liaisons et/ou d'attachements correspondant à son type.

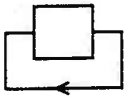
c) Règles de manipulation des objets

Tout au long de la création de l'application, des contrôles s'effectuent le plus tôt possible pour veiller à ce que l'application reste "**bien construite**". Les contrôles sont tant syntaxiques (unicité des noms d'agents et de ports,

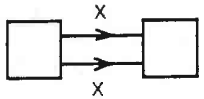
vérification du nombre de messages) que sémantiques (cohérence du réseau de liaisons et d'attachements, absence de modules imbriqués). Lorsque l'utilisateur considère que son application est terminée, le système procède aux contrôles différés chargés d'assurer la **"complétude"** de la structure. Une structure est complète s'il n'y a qu'une hiérarchie avec un agent unique à la racine (cf. fonction "contrôle père"), les modules forment une partition de l'ensemble des agents non raffinés (cf. fonction "contrôle module"), tous les ports sont utilisés à bon escient (cf. fonction "contrôle port") et il n'existe de demi-liens qu'au niveau de l'agent racine.

Outre ces contrôles "obligatoires" on pourrait instaurer des contrôles "facultatifs" dont le rôle consisterait à maintenir une structure "vraisemblable". Des messages d'avertissements seraient émis pour structuration inhabituelle. Par exemple, les situations suivantes peuvent cacher des erreurs :

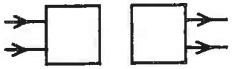
Plusieurs chemins de communication d'une même liaison aboutissant à la même entité destination.



Un chemin de communication ayant même entité émettrice et réceptrice.



Plusieurs liaisons avec le même type de message entre les mêmes entités.



Une entité sans sortie possible ou sans entrée possible.

d) Eléments de réalisation

La réalisation de cet éditeur, commencée en DEA , a été effectuée sur DPS 8 sous Multics en FORTRAN A avec la bibliothèque graphique interactive de Tektronix (IGL). La structure de données décrivant les hiérarchies a été conçue de façon à faciliter la mise en oeuvre de commandes de documentation (liste des agents, des modules, des messages, etc.).

Les difficultés majeures rencontrées ont concerné la gestion des différents niveaux de décomposition et surtout le contrôle de la cohérence du réseau de communication. Le concepteur étant autorisé à travailler de manière ascendante, descendante ou mixte et à modifier à tout moment une partie

créée, beaucoup de ces contrôles ne peuvent se faire que lorsqu'il estime avoir une application complète, de manière différée.

Notre effort a porté d'avantage sur la réalisation de ces logiciels de contrôle que sur les problèmes de convivialité ou de rapidité d'exécution.

III.3.2) Interfaçage de l'éditeur graphique avec un éditeur syntaxique

Un éditeur syntaxique relatif au langage de spécification détaillé facilite toutes les opérations de création et de manipulation des programmes, tout en se chargeant des contrôles syntaxiques. Une représentation interne du programme sous forme d'arbre de syntaxe abstraite autorise des déplacements et des modifications complexes.

L'adéquation entre l'outil de base retenu (**MENTOR [DON 79], [MEL 85]**) et nos préoccupations se mesure au travers des 5 points suivants :

- possibilité de générer un environnement pour le langage LSD, par METAL **[MEL 82]**,
- possibilité de manipulations complexes de la spécification détaillée (langage Mentol),
- possibilité de construction descendante de descriptions en LSD, (les fragments non encore définis sont repérés par des "méta-variables" et instanciés ultérieurement. On retrouve donc une démarche utilisée lors de la structuration globale de l'application au niveau graphique),
- transport de descriptions en LSD en un programme ADA (langage cible par excellence), Pascal, LTR-V2, LTR 3 ou C, ces langages étant déjà disponibles sous MENTOR,
- possibilité de greffer d'autres outils sur la représentation interne normalisée que constituent les arbres de syntaxe abstraite.

Présentons les caractéristiques de MENTOR et la mise en oeuvre de Mentor-LSD avant d'interfacer cet environnement à l'éditeur graphique.

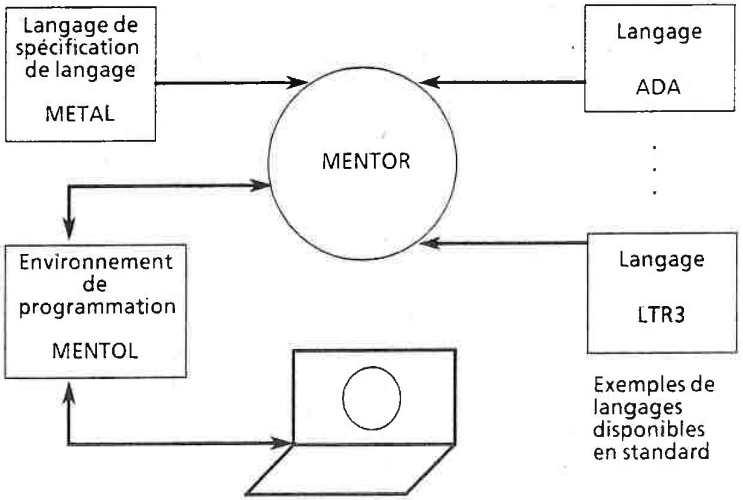
a) mise en oeuvre de l'éditeur syntaxique MENTOR

MENTOR est issu d'un projet de recherche INRIA, démarré en 1975 par l'équipe "Conception et Réalisation d'Outils de Programmation" sous la direction scientifique de Gérard HUET, Gilles KAHN et Bernard LANG.

Ce système assiste les étapes du travail de spécification des concepteurs en maintenant la légalité syntaxique des programmes grâce à l'exploitation d'une représentation formelle de la syntaxe du langage de spécification.
Les principales fonctionnalités sont les suivantes :

- **MENTOL** : un langage de programmation spécialement conçu pour l'écriture de commandes "sur mesure" permettant la manipulation, l'analyse et la transformation de programmes.

- **METAL** : un langage de description formelle de langages permettant d'introduire de nouveaux langages sous MENTOR.



L'intégration d'un nouveau formalisme de langage de programmation, dans l'environnement de programmation MENTOR est faite par l'écriture d'un programme METAL définissant ce formalisme. Le langage METAL existe lui-même sous MENTOR. On dispose donc de l'environnement MENTOR-METAL pour mettre au point le programme METAL qui définit le langage LSD. La commande COMPILE, sous MENTOR-METAL, crée les tables du langage LSD. MENTOR peut ensuite être appelé immédiatement sur ce nouveau formalisme. La session MENTOR-LSD est donc empilée sur MENTOR-METAL et si MENTOR-LSD n'est pas satisfaisant, on revient à la session METAL pour modifier le programme.

Le programme METAL qui définit le formalisme LSD, contient les éléments suivants :

a) La **syntaxe concrète de LSD,** c'est-à-dire un ensemble de règles permettant de décider si une phrase donnée est admise par le formalisme LSD. Les règles sont écrites dans une forme proche de la BNF. L'analyseur syntaxique de LSD sera dérivé à partir de ces règles.

b) La **syntaxe abstraite de LSD**, c'est-à-dire la syntaxe des arbres qui représenteront les formules légales (les programmes légaux) dans le formalisme LSD. La correspondance entre la forme textuelle et la forme arborescente d'un même objet est assurée par deux processeurs : l'**analyseur-constructeur** qui construit l'arbre à partir de la forme textuelle, et le **décompilateur** qui produit une représentation textuelle à partir de l'arbre de syntaxe abstraite.

c) Les **fonctions de construction des arbres de syntaxe abstraite**. Ces fonctions sont le lien logique entre la forme textuelle, ou concrète, d'une phrase de LSD et sa forme arborescente, ou abstraite.
Elles indiquent, pour chaque construction du langage, quel est l'arbre associé. A partir de ces fonctions sera dérivé le constructeur d'arbres pour le formalisme LSD. L'analyseur syntaxique de LSD sera ensuite relié au constructeur d'arbres par une interface indépendante du langage, pour former l'analyseur-constructeur associé à LSD.
A chaque production de la syntaxe concrète de LSD est donc associée une action sémantique ou fonction de construction d'arbre.
Chaque fois qu'une réduction est effectuée, au cours de l'analyse syntaxique d'un programme LSD, l'action sémantique correspondante est exécutée, un arbre est donc construit et est passé à la suite du processus d'analyse.

d) Les **fonctions de décompilation de LSD.** Ces fonctions sont le lien logique entre la forme arborescente et la forme textuelle. Elles indiquent le morceau de texte associé à chaque arbre élémentaire. A partir de ces fonctions sera construit le décompilateur de LSD, c'est-à-dire le processeur qui produit une représentation textuelle à partir d'un arbre abstrait.

La construction du programme LSD.METAL et un exemple de programme de décompilation sont présentés en annexes 2 et 3.
L'environnement MENTOR facilite donc la création d'un éditeur structurel pour un formalisme donné et la manipulation de programmes dans ce formalisme.
Si les avantages théoriques par rapport à un éditeur standard sont évidents (légalité syntaxique et possibilités de manipulations et transformations complexes), on peut tout de même déplorer la lourdeur de MENTOL (cf. l'exemple de constitution de programme LSD à l'aide de MENTOL, développé en annexe 4). L'édition d'un programme en manipulant des pointeurs, des phyla et des opérateurs complique singulièrement le travail.
L'intérêt d'un environnement mixte apparaît ici clairement. La traduction automatique de toute structuration en réseau de modules en un squelette de programme est appréciable. L'intervention de l'utilisateur est repoussée aux branches non détaillées du programme qui ne nécessitent pas une vision "panoramique" sur tout le programme. De plus, cette tâche pourrait être facilitée pour l'utilisateur en lui fournissant une bibliothèque de macro procédures MENTOL plus conviviales.

b) intégration des outils graphique et syntaxique

L'environnement mixte intègre l'éditeur graphique et l'éditeur syntaxique MENTOR-LSD dans le but de fournir à l'utilisateur trois moyens pour développer son application (**cf fig. 3.3.2**) :

(1) les commandes de l'éditeur graphique pour la structure globale,
(2) le langage de manipulation d'arbres MENTOL pour tous les aspects de la spécification détaillée qui ne relèvent pas de l'éditeur graphique,
(3) des procédures MENTOL prédéfinies, visibles de l'utilisateur, pour permettre la définition et la manipulation plus conviviale, des éléments structurés du LSD.

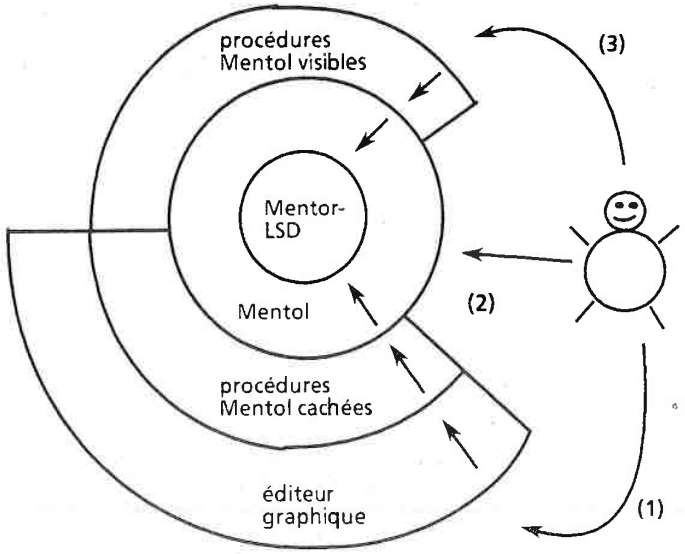


Fig. 3.3.2

Le but de l'interfaçage est de générer, depuis l'éditeur graphique, des appels de procédures MENTOL cachées de l'utilisateur, dans des fichiers de compilation ou d'interprétation.

Ces fichiers seront exécutés par l'éditeur textuel pour créer ou modifier des arbres syntaxiques correspondant à la structuration globale.

Toutes les commandes graphiques ont donc été transformées et permettent trois modes de fonctionnement de l'éditeur graphique:

- le mode **"création"**, lors de la création de l'application. Une commande "compile" permet de traduire la représentation graphique, validée par tous les contrôles, en un arbre MENTOR-LSD équivalent,

- le mode **"mise à jour globale"**, dans lequel on passe dès le chargement d'une application préexistante, en vue de sa modification; l'arbre syntaxique y est mis à jour après chaque modification structurelle **"contrôlable"**, c'est-à-dire après chaque commande dont l'effet sur la spécification détaillée est interprétable.

  exemple: création de module, de port de module, de liaison intermodules,...

- le mode **"mise à jour locale"**, c'est-à-dire purement graphique. Dans ce cas, l'application toute entière devra ultérieurement être recompilée. On passe dans ce mode dès utilisation de la commande "ailleurs", que l'on ne peut interpréter.
  En effet, lorsque l'utilisateur créé une nouvelle hiérarchie, on ne peut prévoir la structure du réseau après "regroupement" et le processus d'interprétation est en défaut. Il s'agit de cas tout à fait exceptionnels correspondant à une refonte fondamentale de l'application.

L'éditeur textuel fonctionne selon trois modes symétriques:

- le mode **"création"**, par exécution du fichier de compilation,

- le mode **"mise à jour globale"**, par exécution du fichier d'interprétation produit par l'éditeur graphique, sur une application préexistante,

- le mode **"mise à jour locale"**, sur une application préexistante, par mise en oeuvre du langage MENTOL ou des procédures visibles. L'utilisateur travaille exclusivement sur les aspects détaillés, non contrôlés par l'éditeur graphique, afin que la représentation graphique reste valide (l'accès aux parties hautes de l'arbre devra être automatiquement interdit).

Mode création

L'utilisateur se trouve implicitement sous ce mode lors de son entrée dans l'éditeur graphique. Les commandes n'ont d'effets qu'au niveau graphique, aucun appel de procédure MENTOL n'étant généré.

A la fin de son développement, le concepteur peut exécuter la commande "compile" qui, après les contrôles "module", "port" et "père", génère le fichier "compilation" et sauvegarde l'application compilée. Cette même application, rappelée ultérieurement instaurera le mode mise à jour globale.

Une application sauvegardée sans compilation ne peut être rappelée qu'en mode création.

Le fichier de compilation contient les appels de procédures MENTOL permettant de créer un arbre abstrait dont la décompilation donne un programme LSD.

Mode mise à jour

L'utilisateur est placé dans le mode mise à jour global à partir du moment où il charge une application compilée précédemment.

Quatre situations peuvent alors se présenter:

1) L'utilisateur exécute des commandes sans effet sur la spécification détaillée.

exemple:
. commandes concernant des agents, ports, liaisons internes aux modules,
. commandes de déplacement.

Aucun appel de fonction MENTOL n'est généré.

2) L'utilisateur exécute la commande "**ailleurs**" pour créer une autre hiérarchie. Un message l'avertira que le mode mise à jour local sera instauré s'il désire poursuivre. Si tel est son choix, à charge pour lui de recompiler toute l'application. S'il sort de l'application sans appeler la commande "compile", aucun fichier MENTOL ne sera créé.

3) L'utilisateur exécute des commandes que l'on ne peut interpréter mais qui seront nécessairement suivies d'une autre commande interprétable. On reste dans le mode mise à jour globale et on ne complète pas immédiatement le fichier d'interprétation. L'interprétation effective se fera lorsque l'utilisateur exécutera la commande interprétable correspondante.

exemple:
La création d'un agent en dehors des modules sera interprétée lorsque cet agent sera déclaré module.

A la fin des modifications l'utilisateur devra lancer la commande "compile" pour que les contrôles "module", "port" et "père" s'effectuent. Si ces contrôles réussissent, le fichier d'interprétation est utilisable pour la spécification

détaillée. Dans le cas contraire, l'utilisateur est invité à compléter l'application. S'il sort de la session sans compléter et recompiler l'application, aucun fichier MENTOL ne sera généré.

Les commandes relatives à cette situation sont les suivantes:

- création d'un agent en dehors des modules,
- création d'un port sur un agent en dehors des modules.

Dans les 2 cas, l'interprétation se fera lors de l'inclusion de l'agent dans un module.

4) L'utilisateur exécute des commandes interprétables immédiatement. Le fichier d'interprétation est créé, mais l'utilisateur devra tout de même lancer la commande "compile" pour que le fichier d'interprétation soit utilisable.

Commandes interprétables:

- création d'un port sur un module,
- création d'un lien entre 2 agents non inclus dans un module,
- création d'un 1/2 lien,
- création d'un attachement,
- création d'un module,
- supprimer un agent correspondant à un module ou à un réseau de modules,
- supprimer un port,
- supprimer un lien,
- supprimer un attachement,
- renommer un agent,
- renommer un port.

Le contenu des fichiers de compilation et d'interprétation sera détaillé au paragraphe suivant, mais de manière générale, chaque liste d'instructions MENTOL générées est composée d'une création d'un ou plusieurs pointeurs qui sont passés en paramètres d'une procédure MENTOL cachée (cf. § III.4.2 et III.4.3). Ces procédures se chargent de la construction/modification d'une partie de l'arbre abstrait.

III.4) UN EXEMPLE COMPLET D'UTILISATION DE L'ENVIRONNEMENT

III.4.1) Structuration à l'aide de l'éditeur graphique

La session de conception graphique s'appuiera sur le problème de la pompe de KRAMER [**KRA 80**].
Nous ne détaillerons pas la démarche qui permet de structurer ce problème en termes d'agents communicants, celle-ci étant largement étudiée dans [**LON 87**].

Définition du problème de la pompe (cf [**LON 87**]):

"Il s'agit du système de commande d'une pompe de drainage située dans une galerie d'une mine de charbon et de contrôle de son environnement. La pompe, après avoir été mise en état de fonctionnement par un ordre de l'opérateur en surface, marche automatiquement : démarrage lorsque l'eau est mesurée au dessus d'un niveau maximum et arrêt lorsque l'eau est mesurée en dessous d'un niveau minimum. Pour des raisons de sécurité, la pompe ne peut pas démarrer ou continuer à fonctionner lorsque le pourcentage de méthane dans l'air dépasse un certain seuil. Des capteurs de méthane, de monoxyde de carbone et de ventilation renseignent périodiquement le système. L'opérateur en surface peut à tout instant interroger le système pour connaître l'état de l'atmosphère. Des alarmes, en cas de dépassement des seuils relatifs aux gaz et à la ventilation, sont émises vers la surface."

Après être entré dans le système, un dialogue s'établit entre l'utilisateur et le système. Les commandes choisies par l'utilisateur sont en **gras**, le menu et la réponse du système en *italique* et les commentaires précédées de %.

Ce problème peut être représenté dans un premier temps par un agent unique APP.

1:créer 2:supprimer 3:déplacer 4: renommer 5:modifier 6:utilitaires 7:arrêt
QUEL EST VOTRE CHOIX ?
1 % création
1:créer agent 2:créer port 3:créer liaison 4:créer attachement 5:créer module 6:menu global
QUEL EST VOTRE CHOIX ?
1 % création d'un agent
NOM DE L'AGENT, 20 lettres maximum
APP
Désignez à l'aide du curseur la position du coin inférieur gauche de l'agent
HAUT->H, BAS->B, NON->N %pour déplacer la fenêtre avant de placer l'agent
N %le curseur est déplacé à l'aide du réticule, la position validée par <CR>
%le gabarit d'un agent est dessiné en pointillés

ETES-VOUS D'ACCORD : O/N ? %on peut choisir une autre position
O

L'agent APP est dessiné avec l'emplacement prévu pour 20 ports et le menu global est proposé ce qui permet au concepteur de poursuivre en créant un port sur l'agent APP.

1:créer 2:supprimer 3:déplacer 4: renommer 5:modifier 6:utilitaires 7:arrêt
QUEL EST VOTRE CHOIX ?
1 % création
1:créer agent 2:créer port 3:créer liaison 4:créer attachement 5:créer module 6:menu global
QUEL EST VOTRE CHOIX ?
2 % création d'un port
NOM DE L'AGENT, 20 lettres maximum
APP
NOM DU PORT, 4 lettres maximum
M
TYPE DU PORT : E, S, ES, SE, SC, SD %type "entrée", "sortie", "entrée-sortie", "sortie-entrée", "sortie-sélection", "sortie-diffusion"
E
POSITION DU PORT SUR L'AGENT PLEIN %on désigne, à l'aide du curseur, l'emplacement qui accueillera le port
%le port **M** de type "entrée" est dessiné sur l'agent **APP**

Le concepteur définit les types de messages externes en entrée et sortie :
-les messages des types EAU,METH, CO, AIR véhiculent les valeurs mesurées par les capteurs,
-les messages des types ALA, ALM, ALC sont les signaux purs d'alarme pour l'air, le méthane et le CO,
-les messages du type REQG précisent la nature du gaz à tester (méthane, co, air),
-les messages du type REPG comportent la nature du gaz et son état (normal, anormal),
-les messages du type CDE donnent la nature de la commande à la pompe (arrêt, marche),
-les messages du type ORD précisent la nature de l'ordre de l'opérateur (prêt, couper, requête),
-les messages du type REPP indiquent l'état de la pompe (coupée, prête, en-marche, arrêtée-eau, arrêtée-méthane).

1:créer 2:supprimer 3:déplacer 4: renommer 5:modifier 6:utilitaires 7:arrêt
QUEL EST VOTRE CHOIX ?
1 % création
1:créer agent 2:créer port 3:créer liaison 4:créer attachement 5:créer module 6:menu global
QUEL EST VOTRE CHOIX ?
3 % création d'une liaison
Voulez-vous créer une liaison complète : O/N ?
N %création d'un demi-lien
NOM DE L'AGENT, 20 lettres maximum
APP

NOM DU PORT, 4 lettres maximum
M
Donner le type du message sans le séparateur "/", 21 lettres maximum
METH
%un trait ayant pour extrémité le port **M** est dessiné
Placez les coins pour la liaison extérieure, en partant du trait issu du port
%Il faut déplacer le curseur sur l'extrémité du trait et valider par une lettre différente de "X" puis désigner de la même manière tous les points de cassures de la liaison. Quand le dernier point de la liaison est validé par la lettre "X", la liaison est dessinée.
ETES-VOUS D'ACCORD : O/N ?
O
Donner la position du message %déplacer le curseur et valider par <CR>

Les autres ports et demi-liens étant créés de la même manière sur l'agent APP, on obtient la figure 3.4.1.

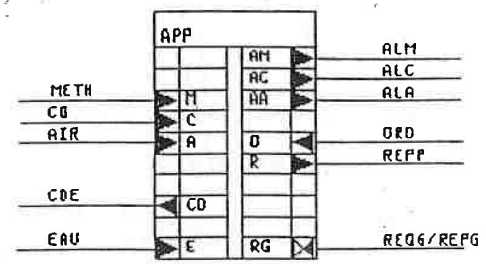


Fig. 3.4.1 - Agent APP

Le concepteur décide ensuite de matérialiser les 2 fonctions principales de l'agent APP par 2 agents PPE (commande de la pompe) et ATM (contrôle de l'atmosphère). Pour cela il doit "raffiner" l'agent APP puis créer les 2 agents "pleins" PPE et ATM "dans l'agent "creux" APP.

1:créer 2:supprimer 3:déplacer 4: renommer 5:modifier 6:utilitaires 7:arrêt
QUEL EST VOTRE CHOIX ?
5 % modification
1:raffiner agent 2:encapsuler agent 3:agrandir agent 4:ailleurs 5:menu global
QUEL EST VOTRE CHOIX ?
1
Nom d'agent à raffiner :
APP

Un réseau avec l'agent APP à l'état "creux" est dessiné. La création des agents ATM et PPE s'effectue de la même manière que la création de l'agent APP. Le concepteur procède ensuite à la création de tous les attachements qui prolongent les demi-liens aboutissant aux ports de l'agent APP à l'état "plein".

1:créer 2:supprimer 3:déplacer 4: renommer 5:modifier 6:utilitaires 7:arrêt

QUEL EST VOTRE CHOIX ?
1
1:créer agent 2:créer port 3:créer liaison 4:créer attachement 5:créer module 6:menu global
QUEL EST VOTRE CHOIX ?
4 % création d'un attachement
Donner le nom de l'agent "creux", 20 lettres maximum
APP
Donner le nom du port de l'agent "creux", 4 lettres maximum
M
Donner le nom de l'agent "plein", 20 lettres maximum
ATM
Donner le nom du port de l'agent "plein", 4 lettres maximum
M
Donner le type du message sans le séparateur "/", 21 lettres maximum
METH

Après vérification de la cohérence des types de messages des liaisons et attachements aboutissant aux ports APP.M et ATM.M, l'utilisateur est invité à définir les points de cassure de l'attachement de la même manière que lors de la création d'une liaison. Le tracé des attachements est dessiné en pointillés.

Pour mettre en évidence quelques tests effectués par le système lors de la création d'attachement, nous donnons les messages d'erreurs fournis dans les cas suivants:

1) création d'attachement dont le type de message n'est pas cohérent

%création de l'attachement APP.E - PPE.E avec le message EAUX
le message ne coïncide pas avec les liaisons !

2) création d'attachement dont les ports ne sont pas compatibles

%création de l'attachement APP.AM - ATM.RG
attachement interdit !

3) création d'attachement incomplet

%les points de cassure ne permettent pas de joindre les 2 ports
Prolongeons l'attachement !
Haut -> H, Bas -> B, Non -> N
N
Relier le bout inachevé au port repéré par un trait

4) type de message où il manque le séparateur "/"

%création de l'attachement APP.RG - ATM.RG avec un message de type monodirectionnel
Il manque le séparateur "/"

La création de 2 communications internes au réseau s'avère nécessaire:
-le message ALM doit être diffusé vers PPE en plus de l'extérieur,
-le démarrage de la pompe nécessite la connaissance d'une variable "ETAT-METHANE" (messages REQM, REPM).

1:créer 2:supprimer 3:déplacer 4: renommer 5:modifier 6:utilitaires 7:arrêt
QUEL EST VOTRE CHOIX ?
1
1:créer agent 2:créer port 3:créer liaison 4:créer attachement 5:créer module 6:menu global
QUEL EST VOTRE CHOIX ?
3 % création d'une liaison
Voulez-vous créer une liaison complète : O/N ?
O %création d'une liaison complète
NOM DE L'AGENT 1, 20 lettres maximum
ATM
NOM DU PORT DE L'AGENT 1, 4 lettres maximum
RM
NOM DE L'AGENT 2, 20 lettres maximum
PPE
NOM DU PORT DE L'AGENT 2, 4 lettres maximum
RM
Donner le type du message avec réponse (le séparateur étant " / "), 21 lettres maximum
REQM / REPM

La figure 3.4.2 représente le réseau situé "sous" l'agent APP, avec les 2 agents, ATM et PPE à l'état "plein", avec leurs ports et les liens y aboutissant.

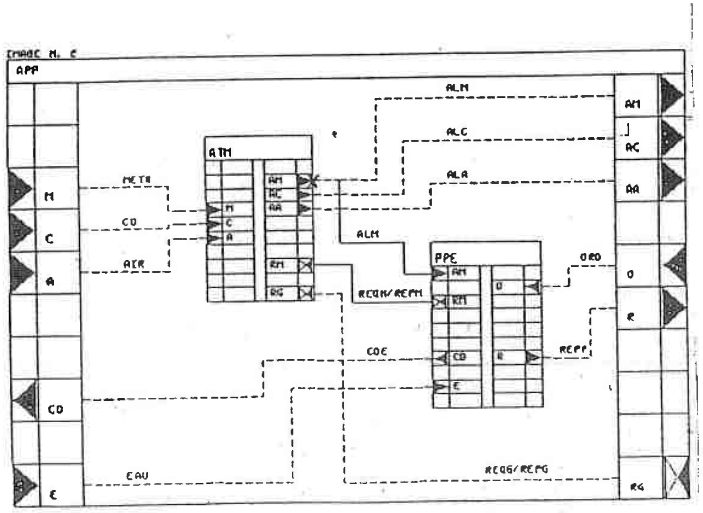


Fig. 3.4.2 - Réseau "sous" l'agent APP

Le concepteur décide ensuite d'exprimer par trois agents élémentaires (CAIR, CCO, CMETH), les trois fonctions similaires de contrôle d'un gaz.

1:créer 2:supprimer 3:déplacer 4: renommer 5:modifier 6:utilitaires 7:arrêt
QUEL EST VOTRE CHOIX ?
5 % modification
1:raffiner agent 2:encapsuler agent 3:agrandir agent 4:ailleurs 5:menu global
QUEL EST VOTRE CHOIX ?
4 %création d'un réseau indépendant de la hiérarchie précédante

La figure 3.4.3 correspond au nouveau réseau contenant les 3 agents élémentaires de contrôle d'un gaz. Si on se déplace "horizontalement" dans l'application (commandes *3:déplacer* puis *5:changer*), on réaffiche à l'écran la figure 3.4.1 contenant APP.

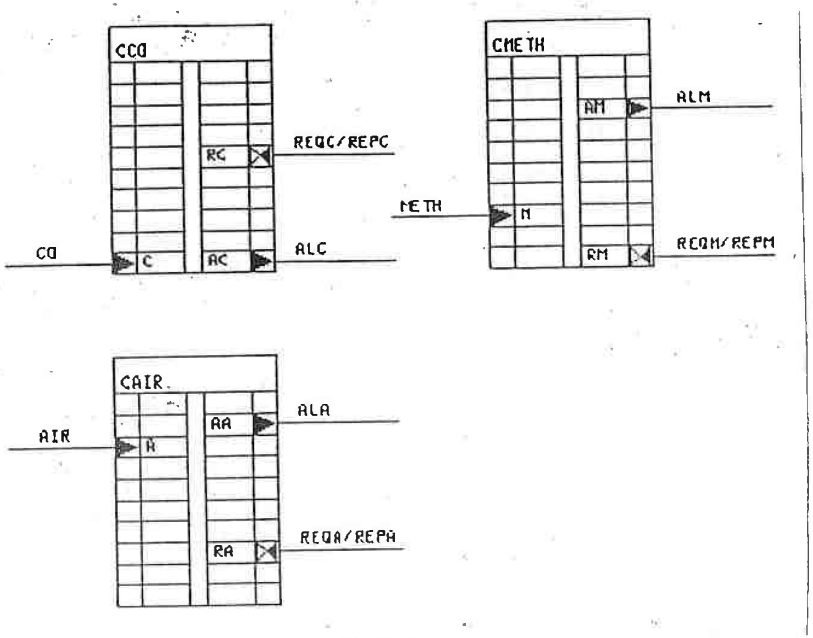


Fig. 3.4.3

Le concepteur note que l'agent ATM. de la figure 3.4.2, abstrait tout ce qui concerne le contrôle des gaz. Il encapsule donc les 3 agents dans ATM.

1:créer 2:supprimer 3:déplacer 4: renommer 5:modifier 6:utilitaires 7:arrêt
QUEL EST VOTRE CHOIX ?
5 % modification
1:raffiner agent 2:encapsuler agent 3:agrandir agent 4:ailleurs 5:menu global
QUEL EST VOTRE CHOIX ?
2 %encapsuler l'image courante si celle-ci ne possède pas d'agent à l'état "creux"

Nom d'agent dans lequel encapsuler :
ATM

Le résultat de cette opération, matérialisée par la figure 3.4.4, n'est pas satisfaisant; les agents élémentaires "débordent" sur l'agent "creux" ATM. Il faut donc les déplacer.

1:créer 2:supprimer 3:déplacer 4: renommer 5:modifier 6:utilitaires 7:arrêt
QUEL EST VOTRE CHOIX ?
3 %déplacer
1:déplacer fenêtre haut 2:déplacer fenêtre bas 3:monter 4:descendre 5:changer 6:déplacer agent 7:menu global
6 %déplacer un agent à l'intérieur du réseau
Nom d'agent à déplacer
CAIR
Donner la nouvelle place de l'agent
HAUT->H, BAS->B, NON->N
N
%un gabarit d'agent est dessiné en pointillés à l'endroit désigné par le curseur
ETES-VOUS D'ACCORD : O/N ?
O
%l'agent est redessiné à la nouvelle place puis l'utilisateur est invité à fournir tous les nouveaux points de cassures de chaque liaison et/ou attachement issu de l'agent déplacé.

Dans la figure 3.4.5 les 3 agents élémentaires sont à leur nouvelle place.

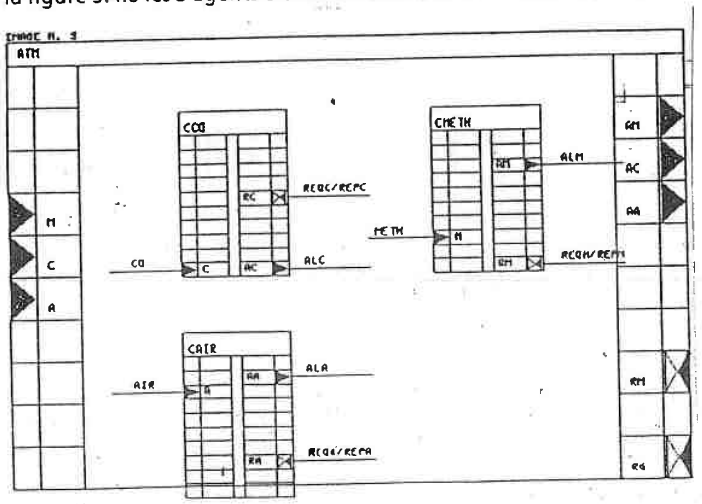


Fig. 3.4.4

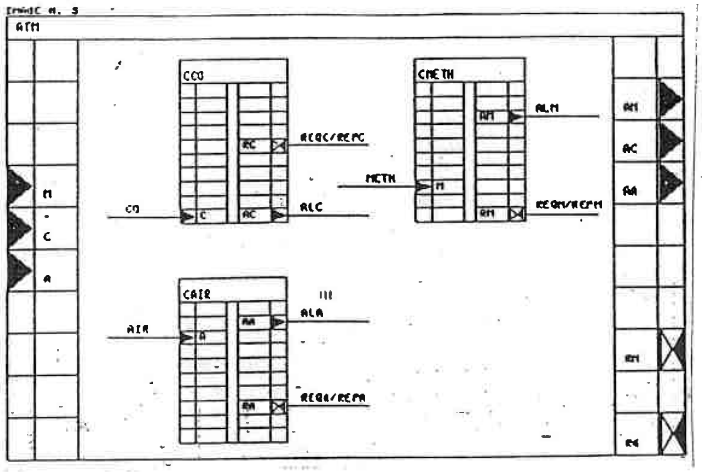


Fig. 3.4.5

Remontons à la racine de la hiérarchie pour vérifier l'unicité de la hiérarchie et donc s'assurer que le réseau de la figure 3.4.3 à bien été intégré "sous" l'agent "plein" ATM.

1:créer 2:supprimer 3:déplacer 4: renommer 5:modifier 6:utilitaires 7:arrêt
QUEL EST VOTRE CHOIX ?
3 %déplacer
1:déplacer fenêtre haut 2:déplacer fenêtre bas 3:monter 4:descendre 5:changer 6:déplacer agent 7:menu global
3 %remonter dans la hiérarchie
%visualisation de la figure 3.4.2
1:créer 2:supprimer 3:déplacer 4: renommer 5:modifier 6:utilitaires 7:arrêt
QUEL EST VOTRE CHOIX ?
3 %déplacer
1:déplacer fenêtre haut 2:déplacer fenêtre bas 3:monter 4:descendre 5:changer 6:déplacer agent 7:menu global
3 %remonter dans la hiérarchie
%visualisation de la figure 3.4.1
1:créer 2:supprimer 3:déplacer 4: renommer 5:modifier 6:utilitaires 7:arrêt
QUEL EST VOTRE CHOIX ?
3 %déplacer
1:déplacer fenêtre haut 2:déplacer fenêtre bas 3:monter 4:descendre 5:changer 6:déplacer agent 7:menu global
5 %se déplacer horizontalement dans la hiérarchie
Il est impossible de changer de hiérarchie !
%revenons à la figure 3.4.5 à l'aide des commandes *3:déplacer* et *4:descendre*

Les demi-liens créés sur les agents CAIR, CCO et CMETH permettaient au concepteur de mettre en évidence les liaisons mises en jeu indépendemment de tout contexte. Cet artifice facilite donc une démarche ascendante de conception et permet au système de vérifier la cohérence des liens et attachements aboutissant à un même port. Après la création de la liaison complète ou de l'attachement, le concepteur doit supprimer le demi-lien.

1:créer 2:supprimer 3:déplacer 4: renommer 5:modifier 6:utilitaires 7:arrêt
QUEL EST VOTRE CHOIX ?
2 %supprimer
1:supprimer agent 2:supprimer port 3:supprimer liaison 4:supprimer attachement 5:menu global
3
Voulez-vous supprimer une liaison complète : O/N ?
N
Donner le nom de l'agent ; 20 lettres maximum
CAIR
Donner le nom du port de l'agent ; 4 lettres maximum
AA

Tous les demi-liens ont été supprimés, par l'utilisateur, après création de la liaison ou de l'attachement adéquat. Il reste toutefois le demi-lien issu du port CAIR.A pour mettre en évidence le contrôle effectué lors de la compilation de l'application. En effet, lorsque l'utilisateur décide de traduire la hiérarchie de réseaux en programme LSD, il ne doit subsister de demi-liens qu'au niveau des ports de l'agent racine. D'autre part, l'agent NOEUD est créé aux côtés des agents qui contrôlent les gaz, afin d'aiguiller les requêtes (Cf.**fig.3.4.6**).

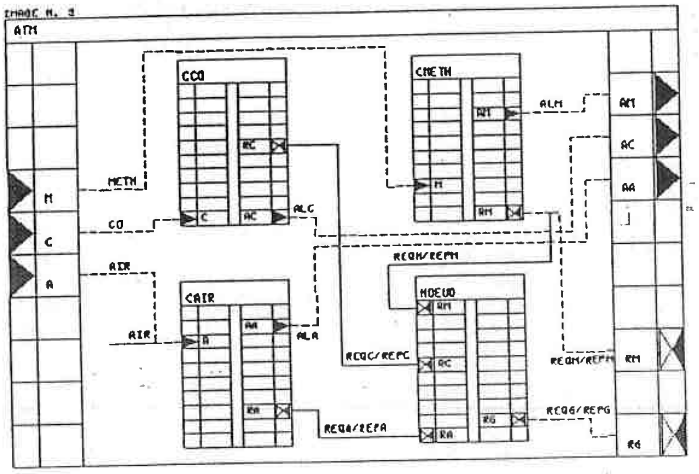


Fig. 3.4.6

L'agent PPE est quant à lui, découpé en un agent TEAU qui teste l'eau vis-à-vis des seuils mini et maxi et un agent POMPE qui commande la pompe.
La figure 3.4.7 est donc obtenue en raffinant l'agent PPE et en créant les agents, ports, liaisons et attachements adéquats.

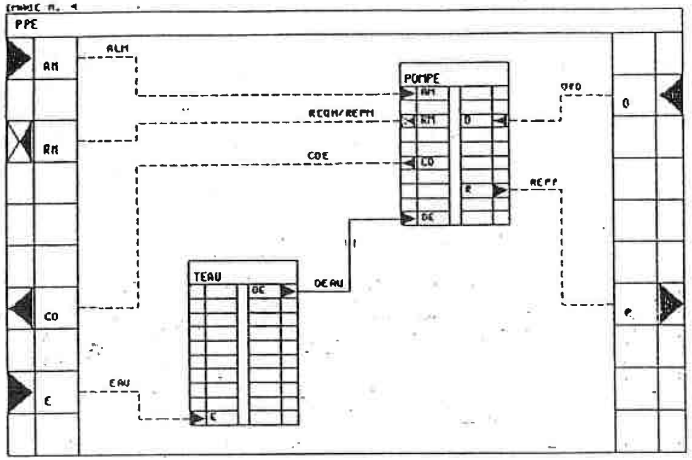


Fig. 3.4.7

La spécification globale de l'application étant terminée, le concepteur décide de procéder à la phase de structuration en termes de modules. Les agents élémentaires sont donc déclarés "modules" de la manière suivante :

1:créer 2:supprimer 3:déplacer 4: renommer 5:modifier 6:utilitaires 7:arrêt
QUEL EST VOTRE CHOIX ?
1
1:créer agent 2:créer port 3:créer liaison 4:créer attachement 5:créer module 6:menu global
QUEL EST VOTRE CHOIX ?
5 %déclaration d'un agent en module
Nom d'agent à déclarer module:
TEAU
Un "M" apparaît à côté du nom de l'agent.

On ne peut déclarer en module qu'un agent présent à l'écran, donc, après déplacement dans la hiérarchie, on déclare de la même manière les agents POMPE, CCO, CAIR et CMETH en tant que modules (cf. Fig.3.4.8).
L'agent NOEUD n'est, volontairement, pas transformé en module. La compilation nous rappellera cet oubli !.

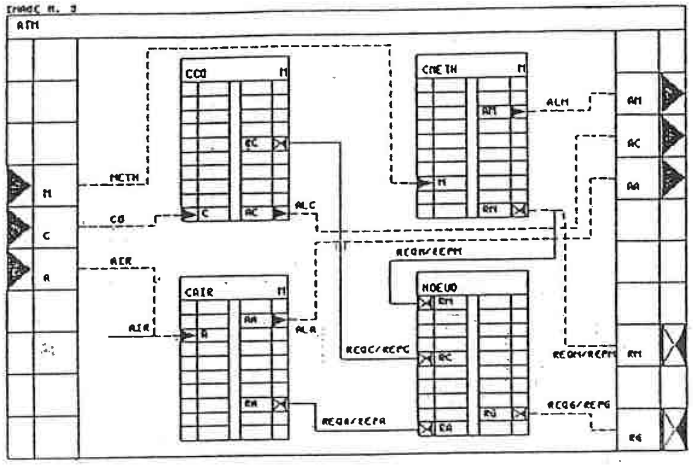


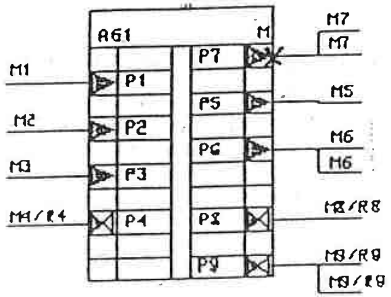
Fig. 3.4.8

Si le concepteur venait à déclarer l'agent ATM en tant que module, le message *L'agent ou un de ses fils est un module ou est dans un module ; opération interdite !* apparaitrait et le système rejèterait à juste titre la commande.

Le réseau de modules qui vient d'être créé sera utilisé au § III.4.3 pour expliciter une session de mise à jour.

III.4.2) Mode compilation

Le résultat commenté de la compilation de l'agent ci-dessus est présenté dans les pages suivantes. Les fonctions MENTOL précédées d'un "." sont des fonctions prédéfinies décrites en annexe 5.




```
@A:&          % création du pointeur repérant le nom de l'application
[NOM]
AG1
.init<@A>     % appel de la fonction prédéfinie permettant de créer un arbre
              abstrait correspondant au programme de nom AG1.

@M:&          % création du pointeur @M repérant le module AG1
[NOM]
AG1
@L C @L_PORTS_M          % création du pointeur @L repérant l'interface du
                         module (c'est-à-dire la liste des ports)
@L S
@L C @se      % traduction du premier port P9 de type SE avec les messages
              M9 et R9
@L S3 C:&
[NOM]
R9
@L S C:&
[NOM]
P9
@L S2 C:&
[NOM]
M9

      % traduction des autres ports
```

```
@L I @se ; @L R ; @L S3 C :& ; [NOM] ; R8 ; @L L ; @L R ; @L S C :& ; [NOM] ; P8 ; @L S2 C :& ;
[NOM] ; M8
@L I @s ; @L R ; @L S C :& ; [NOM] ; P7 ; @L S2 C :& ; [NOM] ; M7
@L I @s ; @L R ; @L S C :& ; [NOM] ; P6 ; @L S2 C :& ; [NOM] ; M6
@L I @s ; @L R ; @L S C :& ; [NOM] ; P5 ; @L S2 C :& ; [NOM] ; M5
@L I @es ; @L R ; @L S3 C :& ; [NOM] ; R4 ; @L L ; @L R ; @L S C :& ; [NOM] ; P4 ; @L S2 C :& ;
[NOM] ; M4
@L I @e ; @L R ; @L S C :& ; [NOM] ; P3 ; @L S2 C :& ; [NOM] ; M3
@L I @e ; @L R ; @L S C :& ; [NOM] ; P2 ; @L S2 C :& ; [NOM] ; M2
@L I @e ; @L R ; @L S C :& ; [NOM] ; P1 ; @L S2 C :& ; [NOM] ; M1
@L U
.Imod<@M,@L>  % appel de la fonction prédéfinie de traduction de module en
                arbre abstrait, avec les pointeurs repérant le nom du module et
                son interface.

@L C @mess % traduction du premier message
@L S C :&
[NOM]
R9
.Imess<@L>

% traduction des autres messages
@L C @mess ; @L S C :& ; [NOM] ; M9 ; .mess<@L>
@L C @mess ; @L S C :& ; [NOM] ; R8 ; .mess<@L>
@L C @mess ; @L S C :& ; [NOM] ; M8 ; .mess<@L>
@L C @mess ; @L S C :& ; [NOM] ; M7 ; .mess<@L>
@L C @mess ; @L S C :& ; [NOM] ; M6 ; .mess<@L>
@L C @mess ; @L S C :& ; [NOM] ; M5 ; .mess<@L>
@L C @mess ; @L S C :& ; [NOM] ; R4 ; .mess<@L>
@L C @mess ; @L S C :& ; [NOM] ; M4 ; .mess<@L>
@L C @mess ; @L S C :& ; [NOM] ; M3 ; .mess<@L>
@L C @mess ; @L S C :& ; [NOM] ; M2 ; .mess<@L>
@L C @mess ; @L S C :& ; [NOM] ; M1 ; .mess<@L>

@L C@multip__diff          % traduction de la première liaison
@L S
@L C@nompointe
@L SC:&
[NOM]
AG1
@L S2C:&
[NOM]
P7
@L R
@L C:&
[NOM]
EXTERIEUR
@L R
@L C@lpor
@L S
@L C:&
[NOM]
EXTERIEUR
@L U2
.Iliaison<@L>

% traduction des autres liaisons
```



```
@L C@multip__sel ; @L S ; @L C@nompointe ; @L SC:& ; [NOM] ; AG1 ; @L S2C:& ; [NOM] ; P9 ;
@L R ; @L C:& ; [NOM] ; EXTERIEUR ; @L R ; @L C@lpor1 ; @L S ; @L C:& ; [NOM] ; EXTERIEUR ;
@L U2 ; .liaison<@L>
@L C@bip ; @L S ; @L C@nompointe ; @L SC:& ; [NOM] ; AG1 ; @L S2C:& ; [NOM] ; P8 ; @L R ;
@L C:& ; [NOM] ; EXTERIEUR ; @L U ; .liaison<@L>
@L C@multip__sel ; @L S ; @L C@nompointe ; @L SC:& ; [NOM] ; AG1 ; @L S2C:& ; [NOM] ; P6 ;
@L R ; @L C:& ; [NOM] ; EXTERIEUR ; @L R ; @L C@lpor1 ; @L S ; @L C:& ; [NOM] ; EXTERIEUR ;
@L U2 ; .liaison<@L>
@L C@bip ; @L S ; @L C@nompointe ; @L SC:& ; [NOM] ; AG1 ; @L S2C:& ; [NOM] ; P5 ; @L R ;
@L C:& ; [NOM] ; EXTERIEUR ; @L U ; .liaison<@L>
@L C@bip ; @L S ; @L C:& ; [NOM] ; EXTERIEUR ; @L R ; @L C@nompointe ; @L SC:& ; [NOM] ;
AG1 ; @L S2C:& ; [NOM] ; P4 ; @L U ; .liaison<@L>
@L C@bip ; @L S ; @L C:& ; [NOM] ; EXTERIEUR ; @L R ; @L C@nompointe ; @L SC:& ; [NOM] ;
AG1 ; @L S2C:& ; [NOM] ; P3 ; @L U ; .liaison<@L>
@L C@bip ; @L S ; @L C:& ; [NOM] ; EXTERIEUR ; @L R ; @L C@nompointe ; @L SC:& ; [NOM] ;
AG1 ; @L S2C:& ; [NOM] ; P1 ; @L U ; .liaison<@L>
@L C@bip ; @L S ; @L C:& ; [NOM] ; EXTERIEUR ; @L R ; @L C@nompointe ; @L SC:& ; [NOM] ;
AG1 ; @L S2C:& ; [NOM] ; P2 ; @L U ; .liaison<@L>

P*          % imprimer à une profondeur quelconque
.XIN        % fin du fichier de compilation
```

L'exécution du fichier de commandes MENTOL produit un arbre de syntaxe abstraite dont la décompilation donne le programme LSD qui suit.

Le nom de l'application correspond au nom de l'agent qui se trouve à la racine, ici "AG1". Le programme contient un module "AG1" avec ses 9 ports constituant l'interface. Les liaisons issues de l'agent racine sont des 1/2 liens dont l'extrémité inconnue est matérialisée par le mot clé "EXTERIEUR". Toutes les parties relevant de la spécification détaillée telle la description des messages, sont repérées par une métavariable constituée du phylum du noeud précédé de "$".

Les indentations résultent du programme de décompilation.

```
application AG1
    presentation
        $TEXTE1
    description
        modules
            module AG1
                presentation
                    $TEXTE1
                description
                    interface
                        se__port P9:M9 reponse R9; se__port P8:M8 reponse R8; s__port P7:M7;
                        s__port P6:M6; s__port P5:M5; es__port P4:M4 reponse R4; e__port
                        P3:M3; e__port P2:M2; e__port P1:M1;
                    $CORPS1
            fin__module;
        liaisons
            AG1.P7 vers EXTERIEUR et EXTERIEUR;
            AG1.P9 vers EXTERIEUR ou EXTERIEUR;
            AG1.P8 vers EXTERIEUR;
            AG1.P6 vers EXTERIEUR ou EXTERIEUR;
```

```
        AG1.P5 vers EXTERIEUR;
        EXTERIEUR vers AG1.P4;
        EXTERIEUR vers AG1.P3;
        EXTERIEUR vers AG1.P1;
        EXTERIEUR vers AG1.P2;
    messages
        message R9
            $DESCR_MESS1
        fin_message;
        message M9
            $DESCR_MESS1
        fin_message;
        message R8
            $DESCR_MESS1
        fin_message;
        message M8
            $DESCR_MESS1
        fin_message;
        message M7
            $DESCR_MESS1
        fin_message;
        message M6
            $DESCR_MESS1
        fin_message;
        message M5
            $DESCR_MESS1
        fin_message;
        message R4
            $DESCR_MESS1
        fin_message;
        message M4
            $DESCR_MESS1
        fin_message;
        message M3
            $DESCR_MESS1
        fin_message;
        message M2
            $DESCR_MESS1
        fin_message;
        message M1
            $DESCR_MESS1
        fin_message;
    types
        $L_TYPES1
fin_application.
```


III.4.3) Mode interprétation

Nous reprenons l'exemple de la pompe de Kramer développé au paragraphe III.4.1.
Le réseau de modules ayant été créé, il nous reste à le charger puis le compiler avant d'effectuer toute modification.

1:creer 2:supprimer 3:déplacer 4:renommer 5:modifier 6:utilitaires 7:arrêt
QUEL EST VOTRE CHOIX ?
6 % utilitaires
1:contrôles 2:sauvegarde 3:chargement 4:lister 5:compiler 6:hardcopy 7:menu global
3 % chargement
Donner le nom du fichier :
% on donne le nom du fichier dans lequel l'application a été sauvegardée

1:creer 2:supprimer 3:déplacer 4:renommer 5:modifier 6:utilitaires 7:arrêt
QUEL EST VOTRE CHOIX ?
6 % utilitaires
1:contrôles 2:sauvegarde 3:chargement 4:lister 5:compiler 6:hardcopy 7:menu global
5 % compilation
NOEUD est un agent qui n'est ni un module ni dans un module.
Le contrôle des modules est terminé.
L'application n'est pas compilée !

Le système a détecté la présence d'un agent qui n'a pas été déclaré en module, procédons donc à cette opération et relançons la compilation.

1:creer 2:supprimer 3:déplacer 4:renommer 5:modifier 6:utilitaires 7:arrêt
QUEL EST VOTRE CHOIX ?
6 % utilitaires
1:contrôles 2:sauvegarde 3:chargement 4:lister 5:compiler 6:hardcopy 7:menu global
5 % compilation
Le contrôle des modules est terminé.
Le port A de l'agent CAIR n'est pas utilisé à bon escient !
L'application n'est pas compilée !

Il subsiste, en effet, un demi-lien sur ce port. Les demi-liens n'étant autorisés que sur l'agent racine, il faut donc supprimer celui-ci et recompiler.

1:creer 2:supprimer 3:déplacer 4:renommer 5:modifier 6:utilitaires 7:arrêt
QUEL EST VOTRE CHOIX ?
6 % utilitaires
1:contrôles 2:sauvegarde 3:chargement 4:lister 5:compiler 6:hardcopy 7:menu global
5 % compilation
Le contrôle des modules est terminé.
Donner le nom du fichier : 10 lettres maximum
% on donne le nom du fichier qui contiendra l'application compilée, si ce fichier est chargée ultérieurement l'utilisateur sera placé en mode mise à jour globale

La compilation du réseau de modules, représenté sur les figures 3.4.9 et 3.4.10, se trouve dans le fichier TOUT.mentol.

Ce fichier sera chargé dans l'environnement MENTOR avec la commande :
?.**DEVIN**
FILENAME : **TOUT**

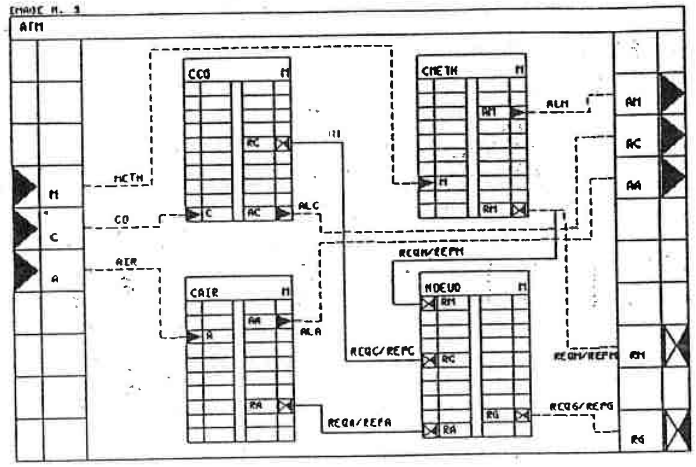


Figure 3.4.9

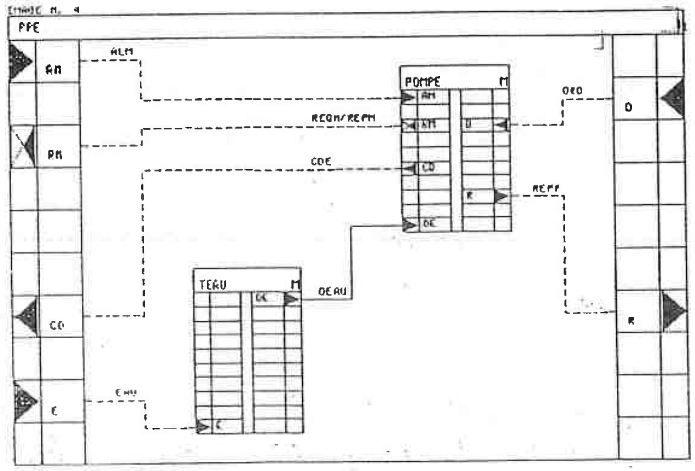


Figure 3.4.10

Pour visualiser le programme LSD construit on exécute la commande Mentol :
?**P***
On obtient le programme suivant :

```
application APP
    presentation
      $TEXTE1
    description
      modules
        module TEAU
            presentation
                $TEXTE1
            description
                interface
                    e-port E:EAU;s-port DE:DEAU;
                $CORPS1
        fin-module;
        module POMPE
            presentation
                $TEXTE1
            description
                interface
                    s-port R:REPP;e-port O:ORD;e-port DE:DEAU;s-port CD:CDE;se-port
                    RM:REQM reponse REPM;e-port AM:ALM;
                $CORPS1
        fin-module;
        module NOEUD
            presentation
                $TEXTE1
            description
                interface
                    se-port RC:REQC reponse REPC;se-port RM:REQM reponse REPM;es-port
                    RG:REQG reponse REPG;
                    se-port RA:REQA reponse REPA;
                $CORPS1
        fin-module;
        module CAIR
            presentation
                $TEXTE1
            description
                interface
                    es-port RA:REQA reponse REPA;e-port A:AIR;s-port AA:ALA;
                $CORPS1
        fin-module;
        module CMETH
            presentation
                $TEXTE1
            description
                interface
                    e-port M:METH;es-port RM:REQM reponse REPM;
                    s-port AM:ALM;
                $CORPS1
        fin-module;
        module CCO
            presentation
                $TEXTE1
```

```
        description
            interface
                e-port C:CO;es-port RC:REQC reponse REPC;s-port AC:ALC;
            $CORPS1
    fin-module;
liaisons
    EXTERIEUR vers NOEUD.RG;
    POMPE.R vers EXTERIEUR;
    EXTERIEUR vers POMPE.O;
    CAIR.AA vers EXTERIEUR;
    CCO.AC vers EXTERIEUR;
    POMPE.RM vers CMETH.RM;
    EXTERIEUR vers TEAU.E;
    POMPE.CD vers EXTERIEUR;
    EXTERIEUR vers CAIR.A;
    EXTERIEUR vers CCO.C;
    EXTERIEUR vers CMETH.M;
    CMETH.AM vers POMPE.AM et EXTERIEUR;
    TEAU.DE vers POMPE.DE;
    NOEUD.RM vers CMETH.RM;
    NOEUD.RC vers CCO.RC;
    NOEUD.RA vers CAIR.RA;
messages
    message EAU
        $DESCR-MESS1
    fin-message;
    message DEAU
        $DESCR-MESS1
    fin-message;
    message REPP
        $DESCR-MESS1
    fin-message;
    message ORD
        $DESCR-MESS1
    fin-message;
    message CDE
        $DESCR-MESS1
    fin-message;
    message REPM
        $DESCR-MESS1
    fin-message;
    message REQM
        $DESCR-MESS1
    fin-message;
    message ALM
        $DESCR-MESS1
    fin-message;
    message REPC
        $DESCR-MESS1
    fin-message;
    message REQC
    $DESCR-MESS1
    fin-message;
    message REPG
    $DESCR-MESS1
    fin-message;
    message REQG
```



```
        $DESCR-MESS1
    fin-message;
    message REPA
        $DESCR-MESS1
    fin-message;
    message REQA
        $DESCR-MESS1
    fin-message;
    message AIR
        $DESCR-MESS1
    fin-message;
    message ALA
        $DESCR-MESS1
    fin-message;
    message METH
        $DESCR-MESS1
    fin-message;
    message CO
        $DESCR-MESS1
    fin-message;
    message ALC
    $DESCR-MESS1
    fin-message;
types
    $L-TYPES1
fin-application.
```

Revenons au niveau graphique pour une modification de structure.
On charge l'application puis on créé l'agent AGENT1, le port AGENT1.P1 et la liaison TEAU.DE - AGENT1.P1 dans le réseau de la figure 3.4.10 (cf. Fig. 3.4.11).

Ilexiste une liaison ou un attachement partant du port de type S de l'agent plein. Voulez-vous changer le type S en SC ou SD ?
O
Donner le nouveau type : SC ou SD
SC

On déclare AGENT1 en module et on lance la commande de compilation afin de compléter le fichier d'interprétation, MODIF.mentol, rempli au fur et à mesure des modifications.

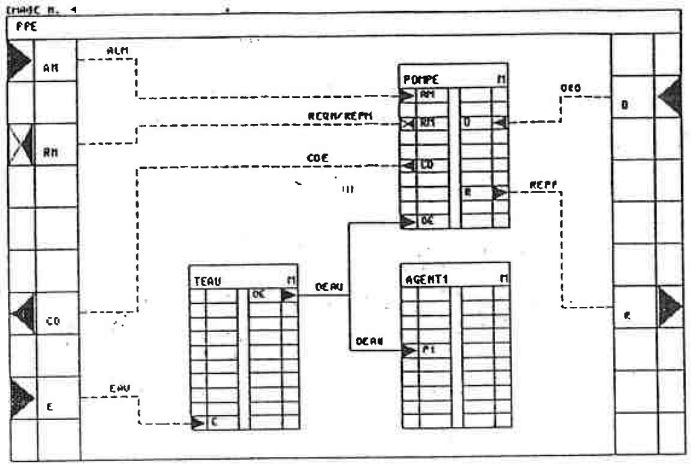


Figure 3.4.11

Ce fichier d'interprétation sera "appliqué", sous MENTOR, au programme LSD, créé précédemment, de la manière suivante :

?.**LOAD**

FILENAME : %nom du fichier

.**XIN**

FILENAME : **MODIF**

Le résultat de cette opération donne le programme LSD suivant :

```
application APP
    presentation
      $TEXTE1
    description
      modules
        module TEAU
            presentation
                $TEXTE1
            description
                interface
                    e-port E:EAU;s-port DE:DEAU;
                $CORPS1
        fin-module;
```



```
    module POMPE
        presentation
            $TEXTE1
        description
            interface
                s-port R:REPP;e-port O:ORD;e-port DE:DEAU; s-port CD:CDE;se-port
                RM:REQM reponse REPM;e-port AM:ALM;
            $CORPS1
    fin-module;
    module NOEUD
        presentation
            $TEXTE1
        description
            interface
                se-port RC:REQC reponse REPC;se-port RM:REQM reponse REPM;es-port
                RG:REQG reponse REPG;
                se-port RA:REQA reponse REPA;
            $CORPS1
    fin-module;
    module CAIR
        presentation
            $TEXTE1
        description
            interface
                es-port RA:REQA reponse REPA;e-port A:AIR;s-port AA:ALA;
            $CORPS1
    fin-module;
    module CMETH
        presentation
            $TEXTE1
        description
            interface
                e-port M:METH;es-port RM:REQM reponse REPM;
                s-port AM:ALM;
            $CORPS1
    fin-module;
    module CCO
        presentation
            $TEXTE1
        description
            interface
                e-port C:CO;es-port RC:REQC reponse REPC;s-port AC:ALC;
            $CORPS1
    fin-module;
    module AGENT1
        presentation
            $TEXTE1
        description
            interface
                e-port P1:DEAU;
            $CORPS1
    fin-module;
liaisons
    EXTERIEUR vers NOEUD.RG;
    POMPE.R vers EXTERIEUR;
    EXTERIEUR vers POMPE.O;
    CAIR.AA vers EXTERIEUR;
```

```
    CCO.AC vers EXTERIEUR;
    POMPE.RM vers CMETH.RM;
    EXTERIEUR vers TEAU.E;
    POMPE.CD vers EXTERIEUR;
    EXTERIEUR vers CAIR.A;
    EXTERIEUR vers CCO.C;
    EXTERIEUR vers CMETH.M;
    CMETH.AM vers POMPE.AM et EXTERIEUR;
    TEAU.DE vers POMPE.DE ou AGENT1.P1;
    NOEUD.RM vers CMETH.RM;
    NOEUD.RC vers CCO.RC;
    NOEUD.RA vers CAIR.RA;
messages
    message EAU
        $DESCR-MESS1
    fin-message;
    message DEAU
        $DESCR-MESS1
    fin-message;
    message REPP
        $DESCR-MESS1
    fin-message;
    message ORD
        $DESCR-MESS1
    fin-message;
    message CDE
        $DESCR-MESS1
    fin-message;
    message REPM
        $DESCR-MESS1
    fin-message;
    message REQM
        $DESCR-MESS1
    fin-message;
    message ALM
        $DESCR-MESS1
    fin-message;
    message REPC
        $DESCR-MESS1
    fin-message;
    message REQC
    $DESCR-MESS1
    fin-message;
    message REPG
    $DESCR-MESS1
    fin-message;
    message REQG
        $DESCR-MESS1
    fin-message;
    message REPA
        $DESCR-MESS1
    fin-message;
    message REQA
        $DESCR-MESS1
    fin-message;
    message AIR
        $DESCR-MESS1
```



```
    fin-message;
    message ALA
        $DESCR-MESS1
    fin-message;
    message METH
        $DESCR-MESS1
    fin-message;
    message CO
        $DESCR-MESS1
    fin-message;
    message ALC
    $DESCR-MESS1
    fin-message;
types
    $L-TYPES1
fin-application.
```

Notons la création du module **AGENT1** et du port correspondant et la modification de la liaison bipoint **TEAU.DE vers POMPE.DE** en une liaison multipoint en sélection (noeud "**ou**"). Si on recharge l'application au niveau graphique pour créer une autre hiérarchie avec la commande *4:ailleurs*, le système répond :

ATTENTION cette commande vous fait sortir du mode modification ! On continue : O/N ?

BIBLIOGRAPHIE

[DON 79] DONZEAU-GOUGE, HUET, KAHN
The MENTOR program manipulation system.
Rapport INRIA, 10/79.

[HAB 79] HABERMANN A.N.
The Gandalf Research Project.
Carnegie-Mellon University, 1979.

[KRA 80] KRAMER, MAGEE, SLOMAN, LISTER
Distributed process control systems : programming and configuration.
RR 80/12, Imp. College, LONDRES, 5/80.

[MEL 82] MELESE B.
METAL, un langage de spécification pour le système MENTOR.
TSI, Vol. 1, 4, 7/1982.

[MEL 85] MELESE, MIGOT, VEROVE
The MENTOR-V5 documentation.
INRIA, RTO43, 1/85.

[MEY 85] MEYER B.
Etapes sur le chemin du génie logiciel.
Thèse d'Etat, Nancy I, 1985.

[REI 84] REISS S.
Graphical program development with PECAN.
Sigplan Notices, Vol. 19, 5/1984.

[TEI 81] TEITELBAUM, REPS.
The Cornell Program Synthetizer.
CACM, Vol. 24, n°9, 1981.

CONCLUSION

L'outil graphique tel qu'il a été présenté précédemment correspond à la version actuellement opérationnelle. Les commandes de documentation (liste des agents, des modules, des messages, etc) n'ont pas été implantées; par contre la structure de données décrivant les hiérarchies a été conçue de façon à faciliter la mise en oeuvre de telles commandes (cf. la structure de données décrite dans **[LAL 84]**).
Si cet outil a atteint les objectifs fonctionnels fixés, on peut toutefois lui souhaiter une évolution tant au niveau de la rapidité d'exécution des commandes que de la convivialité.
L'utilisation de la notion de "segment" de la bibliothèque IGL permettrait de définir en tant qu'entité, tout ensemble d'instructions. Par exemple, une suite d'instructions permettant de dessiner un agent et ses ports peut constituer un segment. Les programmes sources y gagneraient en clarté, et l'affichage des dessins en rapidité.
La convivialité, quant à elle, serait améliorée en substituant au système de menu actuel, un système de multifenêtrage complété par une "souris".

Un inconvénient de cette maquette réside dans la gestion manuelle des fichiers de transition. L'automatisation de cette gestion constitue donc un prolongement indispensable.

Pour la spécification détaillée, le concepteur dispose d'un squelette de programme en LSD, reflètant la structure organique, et du langage de manipulation d'arbres MENTOL.
Pour contourner la lourdeur de ce dernier, il faudra construire une bibliothèque de procédures prédéfinies visibles comme cela a été fait pour le langage PASCAL (cf. procédure DECVAR [**MEL 81**]). Elles ne posent aucun problème de fond.

Un autre point qui mériterait développement concerne l'optimisation du code généré par la compilation ou l'interprétation du niveau graphique.
En effet, dans l'exemple du § III.4.2 on peut remarquer les commandes successives suivantes:
@L L et @L R qui déplacent le pointeur @L vers la gauche puis vers la droite !.
Si la génération de cette séquence ne peut être évitée, une analyse du fichier généré, supprimerait de tels problèmes et accélérerait la construction de l'arbre de syntaxe abstraite.

L'environnement mixte actuel ne constitue qu'une maquette démonstrative d'interfaçage. L'effort de réalisation a particulièrement porté sur les problèmes de contrôle de cohérence:

- cohérence des chemins de communication au niveau graphique,

- cohérence du réseau de modules au niveau graphique,
- cohérence entre description graphique et description textuelle.

La solution proposée a le mérite de la simplicité. Les 2 éditeurs sont indépendants et interfacés par passage de fichiers de procédures.
La solution qui aurait consisté à développer un éditeur syntaxique doté d'une possibilité de "vue graphique" aurait été d'un niveau de complexité incomparablement plus grand: en effet, les modifications de structure au niveau textuel auraient dû pouvoir être interprétées graphiquement, c'est-à-dire que l'éditeur aurait dû pouvoir par lui-même composer à l'écran la représentation d'une application quelconque. Dans notre cas la composition des écrans relève de l'utilisateur. On sait la complexité de ces problèmes de composition automatique d'image. En interdisant les modifications structurelles au niveau textuel, on maintient à "peu de frais" la cohérence entre la vue graphique et les arbres de syntaxe abstraite. Les inconvénients pour l'utilisateur seront insignifiants lorsque la traduction entre éditeurs sera transparente (cf. §IV.4).

Ce travail n'aborde pas du tout l'édition des spécifications formelles de comportement des entités aux phases en amont de la conception (cf.**[LON 87]**). Ceci montre combien les environnements de conception d'applications, parce qu'ils manipulent une pluralité de formes de descriptions de natures très différentes, posent de nombreux problèmes même sur le seul plan de l'édition des descriptions. Les éditeurs syntaxiques seront au centre de ces environnements mais ils apparaissent trop limités dans leurs versions actuelles.

BIBLIOGRAPHIE

[LAL 84] LALLIER M.
Un outil graphique d'aide à la conception de systèmes répartis.
DEA, NANCY I, 9/84.

[LON 87] LONCHAMP J.
Conception des applications informatiques réparties en commande de procédés industriels : une démarche, des outils.
Thèse d'Etat, NANCY I, 5/87.

[MEL 81] MELESE B.
Mentor : l'environnement PASCAL.
INRIA, Rapport technique n°5, 10/1981.

ANNEXES

ANNEXE 1: DESCRIPTION DU LANGAGE DE SPECIFICATION DETAILLE (LSD)

Cette proposition de définition est tirée de [**LON 87**].

Toutes les notions structurantes nommées (applications, modules, processus, types de messages, etc.) seront décrites selon le modèle suivant :

```
NOTION nom-notion
     [PRESENTATION                          [....] : clause
          texte de présentation]                 facultative.
     DESCRIPTION
          description formelle
FIN-NOTION
```

Le texte de présentation facultatif peut prendre diverses formes selon les notions concernées et les préférences du concepteur :

- texte en langage naturel,
- spécification en termes des relations de précédence pour les modules ou processus,
- pré et post conditions pour les fonctions, sous-programmes, opérations, etc.

a) Description d'une application :

```
APPLICATION nom-application
     [PRESENTATION
          texte de présentation]
     DESCRIPTION
          MODULES
               liste-descriptions-modules
          LIAISONS
               liste-descriptions-liaisons (externes
               et inter-modules)
          MESSAGES
               liste-descriptions-types-messages
               (utilisés sur les liaisons ci-dessus)
          TYPES
               liste-déclarations-types-données
               (utiles pour décrire les types de messages
               ci-dessus)
          [REPARTITION
               liste des sites et des assignations de
               modules aux sites ]
FIN-APPLICATION
```

b) Les types de messages :

```
MESSAGE nom-message
     [PRESENTATION
          texte de présentation]
     DESCRIPTION
          {VIDE          }                  {....}
          {liste-champs  }                  {....} : choix.
FIN-MESSAGE                                 {....}
```


Un champ est décrit classiquement par :

```
  nom-champ : type-données
```

ou par :

```
  CAS nom-champ1     (un champ de type scalaire du message)
     QUAND liste-valeurs1 =>nom-champ2 : type-données1
     QUAND liste-valeurs2 =>nom-champ2 : type-données2
        ...
     [AUTREMENT nom-champ2 : type-données n]
  FIN-CAS
```

Exemple :

```
  CAS no-produit
     QUAND 1, 2 => besoin : REEL
     QUAND 3, 4 => besoin : BOOLEEN
     AUTREMENT besoin : ENTIER
  FIN-CAS
```

Fin exemple.

c) Les liaisons :

Une liaison n'est pas nommée; sa description prend l'une des trois formes suivantes :

- liaison bipoint :

```
  désignation-port VERS désignation-port
```

- liaison multipoint en diffusion :

```
  désignation-port VERS désignation-port ET ... ET
       désignation-port
```

- liaison multipoint en sélection :

```
  désignation-port VERS [nom-chemin :] désignation-port
       OU ... OU [nom-chemin :] désignation-port
```

Désignation-port prend une des formes suivantes :

- nom-port
- nom-module.nom-port
- EXTERIEUR (les liaisons externes sont fictivement terminées par un port EXTERIEUR).

Dans une liaison multipoint en sélection tous les chemins sont nommés ou aucun ne l'est; dans le premier cas, la sélection d'un chemin précis au niveau des primitives de réception est possible, alors que dans le second cas, la sélection est nécessairement indéterministe.

d) Les modules :

```
MODULE nom-module
  [PRESENTATION
    texte de présentation]
  DESCRIPTION
    INTERFACE
      liste-descriptions-ports-module
        CORPS
          DECLARATIONS
            liste-déclarations (constantes, variables, types,
              fonctions, procédures, utilisables par tous les
              processus du module)
          PROCESSUS
            liste-descriptions-processus
          LIAISONS
            liste-descriptions-liaisons (internes aux modules)
```

ATTACHEMENTS
liste-descriptions-attachements
MESSAGES
liste-descriptions-types-messages (ceux utilisés sur les liaisons internes)
FIN-MODULE

Un attachement est décrit par :
. nom-E(ou ES)-port de module A nom-E(ou ES)-port de processus
. ou nom-S(ou SE)-port de processus A nom-S(ou SE)-port de module.

e) Les ports de modules :

Ils établissent uniquement l'interface externe des modules (les propriétés attachées aux ports seront précisées au niveau des processus accueuillant ces ports).

Il y a quatre types de ports :
- port d'entrée :
  E-PORT nom-port : type-message
- port de sortie :
  S-PORT nom-port : type-message
- port d'entrée/sortie : (correspond à un service rendu)
  ES-PORT nom-port : type-message1 REPONSE type-message2
- port de sortie/entrée : (correspond à une demande de service)
  SE-PORT nom-port : type-message1 REPONSE type-message2

f) Les processus :

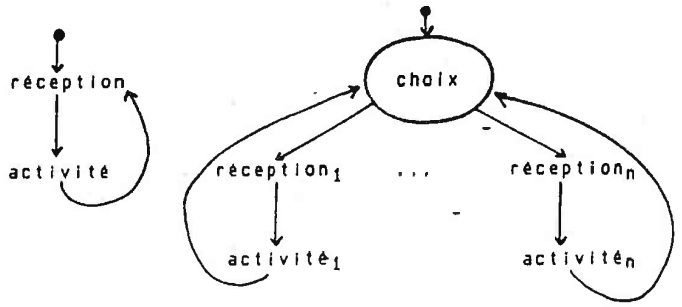
On distingue processus "normaux" et processus d'interruption. Leur durée de vie est celle de l'application.

Un processus normal est décrit par :
PROCESSUS nom-processus [PRIORITE entier>=0]
[PRESENTATION
texte de présentation]
DESCRIPTION
INTERFACE
liste-descriptions-ports-processus
CORPS
DECLARATIONS
liste-déclarations (constantes, variables, types, procédures, fonctions, exceptions)
TRAITEMENTS
traitements
FIN-PROCESSUS

Les priorités des processus servent à résoudre les éventuels conflits d'allocation des processeurs d'un site aux processus prêts à s'exécuter (c'est à dire non bloqués dans une action de communication ou dans une phrase de délai). L'ordonnancement des processus normaux peut être préemptif ou non.

La partie traitements comporte une boucle infinie qui inclut une ou plusieurs "activités", déclenchées par une réception de message. Lorsqu'il y a plusieurs "activités",

l'aiguillage entre elles se fait par une phrase de choix (cf. §11); chaque "activité" peut comporter éventuellement des choix et réceptions, c'est à dire des "sous-activités" :



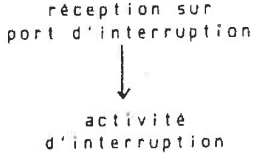
Un processus d'interruption est décrit par :
PROCESSUS INTERRUPTION nom-processus [PRIORITE entier>=0]
Même description que pour un processus normal si ce n'est que la clause INTERRUPTION est associée aux E-PORT d'interruption.
FIN-PROCESSUS

Une interruption est matérialisée par l'arrivée d'un message sur un port d'interruption :
E-PORT nom-port : type-message INTERRUPTION

Chaque port d'interruption est associé à un type d'interruption physique et un type d'interruption physique ne peut être associé qu'à un seul port d'interruption. Le détail de la liaison entre message d'interruption et interruption physique n'est pas spécifié à ce niveau (il peut cependant être décrit informellement dans la partie PRESENTATION du processus d'interruption.

Implicitement, la priorité d'un processus d'interruption est supérieure à celle de tous les processus normaux : la priorité qui peut figurer, permet de hiérarchiser les interruptions de plusieurs processus d'interruption.

La partie traitements comporte une réception sur le port d'interruption suivie de l'"activité" d'interruption :



g) Les ports de processus : (hors ports d'interruption)

Par rapport à la description d'un port de module on trouve en plus, pour les E-PORT et les ES-PORT, une éventuelle clause de priorité et pour les E-PORT une éventuelle clause de consommation (traduisant l'existence d'un tampon et fixant sa politique de gestion).

. **E-PORT** nom-port : type-message [**PRIORITE** entier>=0]
[**CONSOMMATION** { **DERNIER** { **AVEC** / **SANS** } **REMISE** / **FIFO** / **LIFO** / **HASARD** / ... }]

. **ES-PORT** nom-port : type-message1 [**PRIORITE** entier>=0]
REPONSE type-message2

La priorité peut être utile lorsqu'il y a des choix entre plusieurs réceptions dans le corps des processus (cf. primitive **SELECT** ci-après).

La clause de consommation est utile lorsqu'une émission non bloquante crée une accumulation de messages sur un port d'entrée. La consommation peut être :

- non exhaustive : le dernier message reçu seulement est consommé avec ou sans remise (dans ce denier cas le même message peut être consommé plusieurs fois); les autres messages sont détruits.
- exhaustive : tous les messages sont consommés selon diverses politiques (FIFO, LIFO, HASARD, ...); un tampon de taille suffisante est supposé exister. Dans la réalité ceci ne peut pas toujours être garanti et au niveau des langages de programmation le cas du tampon plein doit être considéré (blocage de l'émetteur, écrasement, exception,...)

Dans une liaison interne à un module, un port de processus peut être désigné par :
- nom-port
- nom-processus.nom-port
(**EXTERIEUR** ne doit pas apparaître ici).

h) Les déclarations :

Elles peuvent apparaître dans n'importe quel ordre :
- déclaration de constante symbolique :
CONST nom-constante : type-données [:= valeur]
- déclaration de variable :
VAR nom-variable : type-données [:= valeur]
L'affectation d'une valeur est possible pour les types scalaires. Les notations classiques jouent pour l'écriture des constantes.
- déclaration de type :
En plus des types prédéfinis **ENTIER**, **REEL**, **BOOLEEN**, **CHAINE**, on peut déclarer
  - des types structure
TYPE nom-type : **ENUMERATION** liste-valeurs **FIN-TYPE**

  - des type énumération
TYPE nom-type : **STRUCTURE** liste-champs **FIN-TYPE**
  - des types abstraits
TYPE nom-type
[**PRESENTATION**
texte descriptif]
DESCRIPTION
liste-opérations
FIN-TYPE
Une opération est caractérisée par son "profil" et un texte libre pour définir informellement sa sémantique :
OPERATION
[**PRESENTATION**
texte descriptif]
PROFIL
domaine -> domaine
FIN-OPERATION
où domaine s'écrit : nom-type ou
(nom-type$_1$ X...X nom-type$_n$).

- déclaration d'exception :

Elle peut apparaître au niveau d'un module ou d'une fonction ou procédure. Ceci fixe la portée de son nom, c'est à dire la partie de l'application où l'exception peut être prise en compte grâce à un récupérateur portant son nom. Ailleurs, elle peut être récupérée anonymement en même temps que d'autres exceptions (cf. § i13, ci après).
EXCEPTION nom-exception

- déclaration de fonction et procédure :
FONCTION nom-fonction (liste-paramètres-formels typés)
RETOURNE nom-type
[**PRESENTATION**
texte descriptif]
DESCRIPTION
DECLARATIONS
liste-déclarations (constantes, variables, types, fonctions, procédures, exceptions locales à la fonction)
TRAITEMENTS
traitements
FIN-FONCTION
où les paramètres formels typés s'écrivent
nom-paramètre : type-données

Dans son code, le résultat de la fonction est désigné par nom-fonction et se manipule comme une variable locale ayant pour type celui indiqué après **RETOURNE**.

PROCEDURE nom-procédure (paramètre-formel$_1$ { **ENTREE** / **SORTIE** / **MAJ** } : type-données$_1$, ..., paramètre-formel$_n$ { **ENTREE** / **SORTIE** / **MAJ** } : type-données$_n$)
[**PRESENTATION**
texte descriptif]

```
DESCRIPTION
     DECLARATIONS
          liste-déclarations    (constantes, variables,
               types, fonctions, procédures, exceptions
               locales à la procédure)
     TRAITEMENTS
          traitements
FIN-PROCEDURE
```

i) Les traitements :

Une partie traitements (ou code) est constituée par un ensemble de phrases séparées par des ";".
On distingue les types de phrases suivants :

i1) phrase de commentaire :

```
% texte de commentaire%
```

i2) phrase vide :

```
RIEN
```

i3) phrase conditionnelle :

```
SI condition ALORS code1[SINON code2]
```

i4) phrase de cas:

```
CAS variable     (de type scalaire)
     QUAND liste-valeurs1 => code1
     QUAND liste-valeurs2 => code2
     ...
     [AUTREMENT coden]
FIN-CAS
```

i5) phrases répétitives :

```
REPETER code FIN-REPETER

TANT QUE condition REPETER code FIN-REPETER

REPETER code JUSQUA condition FIN-REPETER
```

i6) phrases d'appel :

```
nom-procédure(liste-paramètres-effectifs)

nom-opération(liste-paramètres-effectifs)
```

Les paramètres effectifs sont des constantes, des variables ou des appels de fonctions.
Peuvent être appelées des fonctions, procédures et opérations déclarées dans le processus concerné ou des fonctions, procédures et opérations déclarées au niveau du module englobant le processus concerné.

i7) phrase d'affectation :

```
variable := { constante
            { variable
            { nom-fonction(liste-paramètres-effectifs)
```

Le langage n'offre pas d'arithmétique; les calculs sont décrits informellement dans des paragraphes, procédures ou

fonctions. L'affectation est cependant utile pour faire évoluer les variables d'état.

Les conditions sont elles aussi réduites en conséquence : variables ou fonctions à résultat booléen ou expressions de comparaison (=,≠,>=,<=,<,>,ET,OU,NON).

i8) phrase d'attente :

```
ATTENDRE délai
```

où délai est un réel positif ou nul représentant un nombre de secondes.

i9) phrase de retour (en fin de procédure ou fonction) :

```
RETOURNER
```

i10) phrases de communication :

- émission bloquante en attente de réponse ou d'acquittement:

```
B-ENVOYER message SUR { nom-S-PORT [VOIE nom-chemin]
                      { nom-SE-PORT [VOIE nom-chemin]
                                    ATTENDRE nom-message
 [ALORS code1 QUAND DELAI délai => code2
FIN-B-ENVOYER]
```

où message s'écrit : { nom-message ; ( ) pour un signal ; (liste-valeurs-scalaires) }

Un délai maximum de blocage peut être fixé. Le déblocage se fait sur réception du (ou des) acquittement(s) de prise en compte ou de la réponse. En cas de dépassement du délai, les messages ou acquittements qui surviennent postérieurement et avant la prochaine émission, sont détruits.

- émission non bloquante :

```
NB-ENVOYER message SUR nom-S-PORT [VOIE nom-chemin]
```

- réception bloquante :

```
B-RECEVOIR nom-message SUR { nom-E-PORT [VOIE nom-chemin]
                           {      [AVEC ACQUITTEMENT]
                           { nom-ES-PORT [VOIE nom-chemin]
                           {             ALORS code
                           {             REPONDRE message
```

Un délai maximum d'attente peut être fixé en imbriquant la réception dans une phrase de CHOIX avec délai, le déblocage se faisant sur réception d'un message.
Dans toutes les phrases de réception les messages ont implicitement le type spécifié au niveau du port.

i11) phrase de choix indéterministe :

Il s'agit d'une construction essentielle pour le type d'applications considéré :

```
CHOIX
     QUAND garde1 => code1
     QUAND garde2 => code2
          ...
     [QUAND DELAI délai => coden ]
FIN-CHOIX
```

"branches"

Une garde est constituée d'une condition booléenne (facultative) et d'une phrase de réception. La garde ne peut pas porter sur le contenu des messages en réception.

Exemple :

```
CHOIX
     QUAND PLEIN<n B-RECEVOIR PRODUIRE SUR P
          => ..............
     QUAND PLEIN>0 B-RECEVOIR CONSOMMER SUR C
          => ..............
FIN-CHOIX
```

Fin exemple.

Une branche est ouverte si l'expression booléenne est vraie (si aucune branche n'est ouverte une exception standard est déclenchée). Une branche est franchissable si elle est ouverte et si la phrase de réception est débloquée. Si plusieurs branches sont simultanément franchissables le choix se fait en fonction des priorités associées aux ports ou à défaut de manière indéterministe. La branche de DELAI est exécutée lorsqu'aucune autre branche n'est franchissable après expiration du délai. Un délai nul permet de construire une réception non bloquante.

i12) phrase de signalement d'exception :

SIGNALER nom-exception

Peut apparaître dans la partie traitements des sous-programmes et des processus ou dans les récupérateurs d'exception (pour propager explicitement une exception).

i13) le récupérateur d'exceptions :

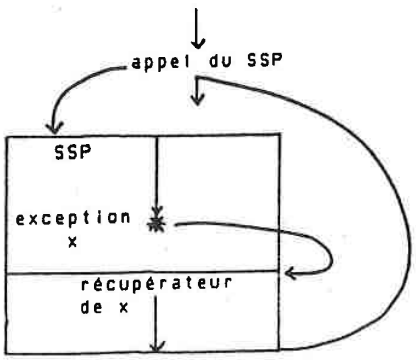
```
EXCEPTION[liste-noms-exception]
     [PRESENTATION
          texte de présentation]
     DESCRIPTION
          code
FIN-EXCEPTION
```

Le mot EXCEPTION est suivi d'un nom ou d'une liste de noms d'exceptions visibles. Si un récupérateur ne porte pas de nom, il récupère toutes les exceptions non récupérées par ailleurs de l'unité à la fin de laquelle il se trouve. Il y a un récupérateur au plus par nom d'exception et par unité.

Les exceptions peuvent se produire dans la partie traitements des sous-programmes et des processus. Le LSD ne permet pas de spécifier un traitement particulier en cas d'exception dans un récupérateur (implicitement l'exception est ignorée).

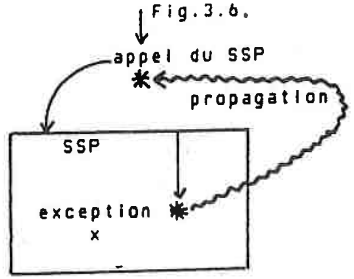
Le schéma de récupération varie suivant la nature de l'unité dans laquelle se produit l'exception :

- pour les sous-programmes, le schéma est du type "terminaison" (sur le modèle ADA):



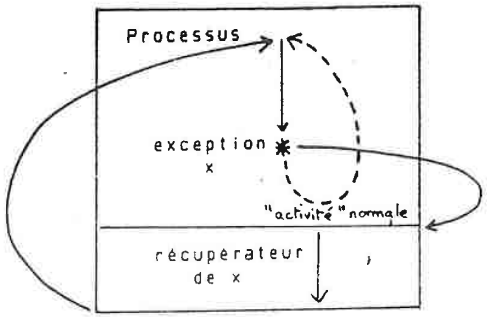
En présence d'un récupérateur de x, le SSP est terminé et le contrôle est rendu à l'appelant, après l'appel.

Fig.3.6.



En absence d'un récupérateur de x, le SSP est terminé et l'exception est propagée à l'appelant au point d'appel.

- pour les processus, le schéma est du type "reprise"; Fig.3.7.



En présence d'un récupérateur de x, celui-ci est exécuté et le processus est repris après abandon de "l'activité" en cours. En absence de récupérateur de x, "l'activité" est abandonnée (nous avons rejeté la notion de terminaison normale ou anormale des processus).

i14) phrase de réinitialisation :

REINITIALISER liste-processus (des processus normaux du même module)

Ne peut apparaître que dans un processus d'interruption.

j) **Les fonctions prédéfinies :**

Notons quelques fonctions spécifiques telles que :

TEMPS qui délivre le temps courant du site concerné

```
   ID-PROCESSUS    qui délivre le nom du processus
   ID-MODULE       qui délivre le nom du module

k) La répartition des modules sur les sites :
   REPARTITION
        SITES
             liste-noms-sites
        ASSIGNATIONS
             liste-assignations
   Chaque assignation s'écrit :
        nom-module SUR nom-site

        Dans les systèmes où la répartition peut être modifiée
   dynamiquement, il s'agit des assignations initiales.
        Un module ne peut être assigné qu'à un seul site; un
   site peut accueillir plusieurs modules.
```


ANNEXE 2 : PROGRAMME LSD. METAL

Ce programme correspond à un "LSD réduit" décrivant le réseau de modules qui est créé au niveau graphique.

Ce programme a la structure suivante :

Définition of LSD is

Rules

Abstract syntax

Chapter ... ;

Rules

Abstract syntax

End chapter ;

:
:

End définition ;

La notion de "**chapter**" rend le programme plus lisible.

La zone "**rules**" contient un ensemble de production de la **syntaxe concrète** à chacune desquelles on associe une **fonction de construction d'arbre de syntaxe abstraite**.

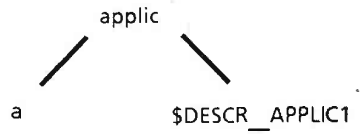
exemple:

```
<APPLIC>:: = application<EN__TETE><DESCR__APPLIC>fin__application#.;
      applic(<EN__TETE>,<DESCR__APPLIC>)
<EN__TETE>:: = <NOM>;
      <NOM>
<NOM>:: = %IDENTIFICATEUR;
      nom-atom(%IDENTIFICATEUR)
```

("application", "fin__application " et "." sont des mots clés et caractères du LSD).

A la première règle de production est associée la fonction de compilation d'arbre "applic" qui construit un arbre dont la racine est l'opérateur de même nom "applic" ayant 2 fils décrit par <EN__TETE> et <DESCR__APPLIC>.
La deuxième règle de production n'a pas de fonction de construction d'arbre propre. Elle utilise celle du non-terminal <NOM>, c'est-à-dire "nom-atom".

Cette fonction a le suffixe "-atom" pour spécifier que la valeur de l'identificateur constituera un noeud de l'arbre.
Exemple :
Arbre qui correspond à ces 3 règles avec l'identificateur NOM ayant pour valeur "a":



La deuxième branche de l'arbre n'étant pas encore définie, elle est matérialisée par la métavariable $DESCR__APPLIC1.

La **syntaxe abstraite** qui décrit la représentation arborescente des programmes LSD est définie dans la zone "**abstract syntax**". Elle est composée d'**opérateurs** et de **phyla** : les opérateurs seront les noeuds de l'arbre. Ils sont d'arité fixe ou variable. Les opérateurs d'arité nulle sont les feuilles de l'arbre et correspondent aux actions du langage. Les opérateurs d'arité fixe (non nulle) peuvent avoir des fils de types différents, tandis que les fils d'un noeud de liste doivent tous être de même type. Le type d'un noeud est le phylum auquel ce noeud appartient. Les phyla sont des ensembles, non vides d'opérateurs. A chaque emplacement d'un arbre de syntaxe abstraite est associé un phylum qui indique les opérateurs qui ont le droit de se trouver à cet emplacement. Le phylum associé à un emplacement donné ne dépend que du père de cet emplacement.

exemple:

```
APPLIC :: = applic;
applic -> EN__TETE DESCR__APPLIC;
EN__TETE:: = en__tête1 nom;
nom -> implemented as STRING;
```

Le phylum APPLIC contient l'opérateur "applic". Cet opérateur, de même nom que la fonction de construction d'arbre correspondant à la règle de production <APPLIC>:: = **application**<EN__TETE><DESCR__APPLIC>**fin__application**#.;, possède 2 fils appartenant respectivement aux phyla EN__TETE et DESCR__APPLIC.
Le phylum EN__TETE possède 2 opérateurs, c'est-à-dire qu'à l'emplacement EN__TETE d'un programme on peut trouver soit le sous-arbre repéré par en__tête1 soit un nom correspondant à une feuille, car l'opérateur "nom" n'a pas de fils.

Le programme **LSD.métal** contient donc la syntaxe concrète et abstraite ainsi que les fonctions de construction des arbres de syntaxe abstraite. Les **fonctions de décompilation** n'étant pas encore implémentées dans le langage METAL, il est nécessaire d'écrire le décompilateur en PASCAL.
Pour cela on dispose d'un squelette de programme que l'on complète selon le type de fonctions de décompilation que l'on désire obtenir pour LSD. C'est-à-dire que l'on spécifie la portion de programme associée à chaque opérateur.

Exemple:

Le sous-programme qui sera exécuté lors d'une décompilation de l'opérateur "applic" est:

```
OPAPPLIC:
    begin
    KEYPRINT(TKAPPLICATION);
    USERDECTABLE(CHILD(1,P,LOCALANG),HOLO-1;
    WLINE;
    TAB;
    USERDECTABLE(CHILD(2,P,LOCALANG),HOLO-1;
    BACKTAB;
    WLINE;
    KEYPRINT(TKFINAPPLICATION);
    WASLETTER: = FALSE;
    KEYPRINT(TKSP28);
    end;
```

Il fournit le programme suivant, avec toutes les indentations, mots clés et caractères de LSD nécessaires.

```
application $NOM
    $DESCR__APPLIC1
fin__application.
```

Les programmes **DECLSD1** (cf. annexe 3), et **LSD.métal** ci dessous, correspondent à une mise en oeuvre d'un langage MENTOR-LSD pour une grammaire restreinte de LSD.
Cette sous-grammaire permet de traduire toutes les notions introduites au cours de la spécification globale.

```
*definition de LSD.metal

definition of LSD is

  rules
    <LSD>::=<APPLIC>;
      <APPLIC>
    <APPLIC>::=application <EN_TETE><DESCR_APPLIC>fin_application #. ;
      applic(<EN_TETE>,<DESCR_APPLIC>)
    <EN_TETE>::=<NOM>;
      <NOM>
    <EN_TETE>::=<NOM> presentation <TEXTE>;
      en_tetel(<NOM>,<TEXTE>)
    <DESCR_APPLIC>::=description <MOD_LIAIS><MESS_TYPES>;
      descr_applic2(<MOD_LIAIS>,<MESS_TYPES>)
    <DESCR_APPLIC>::=description <MOD_LIAIS><MESS_TYPES><REPART>;
      descr_applic1(<MOD_LIAIS>,<MESS_TYPES>,<REPART>)

  abstract syntax
    APPLIC::=applic;
    EN_TETE::=en_tetel nom;
    DESCR_APPLIC::=descr_applic1 descr_applic2;
    applic->EN_TETE DESCR_APPLIC;
    en_tetel->NOM TEXTE;
    descr_applic1->MOD_LIAIS MESS_TYPES REPART;
    descr_applic2->MOD_LIAIS MESS_TYPES;

chapter CONSTANTS
  rules
    <NOM>::=%IDENTIFICATEUR;
      nom-atom(%IDENTIFICATEUR)
    <TEXTE>::=%CHAINECARACTERES;
      texte-atom(%CHAINECARACTERES)
    <EXTERIEUR>::=exterieur;
      exterieur-atom('exterieur')

  abstract syntax
    NOM::=nom;
    TEXTE::=texte;
    exterieur->;
    nom->implemented as STRING;
    texte->implemented as STRING;
    meta->implemented as IDENTIFIER;
end chapter;

chapter NOMP
  rules
    <NOM_POINTE>::=<NOM> #. <NOM>;
      nompointe(<NOM>.1,<NOM>.2)

  abstract syntax
    nompointe->NOM NOM;
end chapter;

chapter COMMENTS
  abstract syntax
    COMMENT::=comment;
    comment->implemented as STRING;
    comment_s->COMMENT*...;
end chapter;

chapter MOD_LIAIS
  rules
    <MOD_LIAIS>::=modules <L_MOD> liaisons <L_LIAI>;
      mod_liais(<L_MOD>,<L_LIAI>)
    <L_MOD>::=;
      l_mod-list(())
    <L_MOD>::=<L_MOD>  <MODULE>;
      l_mod-post(<L_MOD>,<MODULE>)
    <L_LIAI>::=;
      l_liai-list(())
    <L_LIAI>::=<L_LIAI>  <LIAISON>;
```



```
      l_liai-post(<L_LIAI>,<LIAISON>)

  abstract syntax
    MOD_LIAIS::=mod_liais;
    L_MOD::=l_mod;
    L_LIAI::=l_liai;
    mod_liais -> L_MOD L_LIAI;
    l_mod -> MODULE*...;
    l_liai -> LIAISON*...;
end chapter;

chapter MESS_TYPES
  rules
    <MESS_TYPES>::=messages <L_MESS> types <L_TYPES>;
      mess_types(<L_MESS>,<L_TYPES>)
    <L_MESS>::=;
      l_mess-list(())
    <L_MESS>::=<L_MESS>  <MESSAGE>;
      l_mess-post(<L_MESS>,<MESSAGE>)
    <L_TYPES>::=;
      l_types-list(())
    <L_TYPES>::=<L_TYPES>  <TYPE>;
      l_types-post(<L_TYPES>,<TYPE>)
    <MESSAGE>::=message <EN_TETE> description <DESCR_MESS> fin_message #; ;
      mess(<EN_TETE>,<DESCR_MESS>)

  abstract syntax
    MESS::=mess;
    MESS_TYPES::=mess_types;
    L_MESS::=l_mess;
    L_TYPES::=l_types;
    mess_types->L_MESS L_TYPES;
    l_mess->MESS*...;
    l_types->TYPE*...;
    mess->EN_TETE DESCR_MESS;
end chapter;

chapter LIAISON
  rules
    <LIAISON>::=<BIP>;
      <BIP>
    <LIAISON>::=<MULTIP_SEL>;
      <MULTIP_SEL>
    <LIAISON>::=<MULTIP_DIFF>;
      <MULTIP_DIFF>
    <BIP>::=<DES_PORT> vers <DES_PORT> #; ;
      bip(<DES_PORT>.1,<DES_PORT>.2)
    <MULTIP_SEL>::=<DES_PORT> vers <POR1> ou <LPOR1> #; ;
      multip_sel(<DES_PORT>,<POR1>,<LPOR1>)
    <MULTIP_DIFF>::=<DES_PORT> vers <DES_PORT> et <LPOR> #; ;
      multip_diff(<DES_PORT>.1,<DES_PORT>.2,<LPOR>)

  abstract syntax
    LIAISON::=bip multip_sel multip_diff;
    bip->DES_PORT DES_PORT;
    multip_sel->DES_PORT POR1 LPOR1;
    multip_diff->DES_PORT DES_PORT LPOR;
end chapter;

chapter DES_PORT
  rules
    <DES_PORT>::=<NOM>;
      <NOM>
    <DES_PORT>::=<NOM_POINTE>;
      <NOM_POINTE>
    <DES_PORT>::=<EXTERIEUR>;
      <EXTERIEUR>
    <DES_POR1>::=<NOM> #: <DES_PORT>;
      despor1(<NOM>,<DES_PORT>)

  abstract syntax
    DES_PORT::=nom nompointe exterieur;
    despor1->NOM DES_PORT;
```

```
end chapter;

chapter PORT
  rules
    <LPOR1>::=<POR1>;
      lpor1-list((<POR1>))
    <LPOR1>::=<LPOR1> ou <POR1>;
      lpor1-post(<LPOR1>,<POR1>)
    <LPOR>::=<DES_PORT>;
      lpor-list((<DES_PORT>))
    <LPOR>::=<LPOR> et <DES_PORT>;
      lpor-post(<LPOR>,<DES_PORT>)
    <POR1>::=<NOM>;
      <NOM>
    <POR1>::=<NOM_POINTE>;
      <NOM_POINTE>
    <POR1>::=<EXTERIEUR>;
      <EXTERIEUR>
    <POR1>::=<DES_POR1>;
      <DES_POR1>

  abstract syntax
    LPOR1::=lpor1;
    LPOR::=lpor;
    POR1::=DES_PORT despor1;
    lpor1->POR1+...;
    lpor->DES_PORT+...;
end chapter;

chapter MODULE
  rules
    <MODULE>::=module <EN_TETE> <DESCR_MOD> fin_module #; ;
      module(<EN_TETE>,<DESCR_MOD>)
    <DESCR_MOD>::=description <INTERFACE> <CORPS>;
      descr_mod(<INTERFACE>,<CORPS>)
    <INTERFACE>::=interface <L_PORTS_M>;
      <L_PORTS_M>
    <L_PORTS_M>::=;
      l_ports_m-list(())
    <L_PORTS_M>::=<L_PORTS_M>  <PORT_M>;
      l_ports_m-post(<L_PORTS_M>,<PORT_M>)
    <PORT_M>::=<E>;
      <E>
    <PORT_M>::=<S>;
      <S>
    <PORT_M>::=<ES>;
      <ES>
    <PORT_M>::=<SE>;
      <SE>
    <E>::=e_port <NOM> #: <NOM> #; ;
      e(<NOM>.1,<NOM>.2)
    <S>::=s_port <NOM> #: <NOM> #; ;
      s(<NOM>.1,<NOM>.2)
    <ES>::=es_port <NOM> #: <NOM> reponse <NOM> #; ;
      es(<NOM>.1,<NOM>.2,<NOM>.3)
    <SE>::=se_port <NOM> #: <NOM> reponse <NOM> #; ;
      se(<NOM>.1,<NOM>.2,<NOM>.3)

  abstract syntax
    MODULE::=module;
    DESCR_MOD::=descr_mod;
    INTERFACE::=l_ports_m;
    PORT_M::=e s es se;
    module->EN_TETE DESCR_MOD;
    descr_mod->INTERFACE CORPS;
    l_ports_m->PORT_M*...;
    e->NOM NOM;
    s->NOM NOM;
    es->NOM NOM NOM;
end chapter;

chapter ESSAI
  rules
```



```
    <CORPS>::=  corps  ;
      corps-atom('corps')
    <DESCR_MESS>::=  descr_mess  ;
      descr_mess-atom('descr_mess')
    <TYPE>::=  type  ;
      type-atom('type')
    <REPART>::=  repart  ;
      repart-atom('repart')

  abstract syntax
    CORPS::=corps;
    DESCR_MESS::=descr_mess;
    TYPE::=type;
    REPART::=repart;
    corps->;
    descr_mess->;
    type->;
    repart->;
end chapter;

chapter POINTS_D_ENTREE
  rules
    <LSD> ::= #[APPLIC] <APPLIC>;
      <APPLIC>
    <LSD> ::= #[APPLIC] <METAVAR>;
      <METAVAR>
    <LSD> ::= #[EN_TETE] <EN_TETE>;
      <EN_TETE>
    <LSD> ::= #[EN_TETE] <METAVAR>;
      <METAVAR>
    <LSD> ::= #[DESCR_APPLIC] <DESCR_APPLIC>;
      <DESCR_APPLIC>
    <LSD>::=#[DESCR_APPLIC] <METAVAR>;
      <METAVAR>
    <LSD>::=#[NOM] <NOM>;
      <NOM>
    <LSD>::=#[NOM] <METAVAR>;
      <METAVAR>
    <LSD>::=#[TEXTE] <TEXTE>;
      <TEXTE>
    <LSD>::=#[TEXTE] <METAVAR>;
      <METAVAR>
    <LSD>::=#[MOD_LIAIS] <MOD_LIAIS>;
      <MOD_LIAIS>
    <LSD>::=#[MOD_LIAIS] <METAVAR>;
      <METAVAR>
    <LSD>::=#[L_MOD] <L_MOD>;
      <L_MOD>
    <LSD>::=#[L_MOD] <METAVAR>;
      <METAVAR>
    <LSD>::=#[L_LIAI] <L_LIAI>;
      <L_LIAI>
    <LSD>::=#[L_LIAI] <METAVAR>;
      <METAVAR>
    <LSD>::=#[MESS_TYPES] <MESS_TYPES>;
      <MESS_TYPES>
    <LSD>::=#[MESS_TYPES] <METAVAR>;
      <METAVAR>
    <LSD>::=#[MESS] <MESSAGE>;
      <MESSAGE>
    <LSD>::=#[MESS] <METAVAR>;
      <METAVAR>
    <LSD>::=#[L_MESS] <L_MESS>;
      <L_MESS>
    <LSD>::=#[L_MESS] <METAVAR>;
      <METAVAR>
    <LSD>::=#[L_TYPES] <L_TYPES>;
      <L_TYPES>
    <LSD>::=#[L_TYPES] <METAVAR>;
      <METAVAR>
    <LSD>::=#[LIAISON] <LIAISON>;
      <LIAISON>
    <LSD>::=#[LIAISON] <METAVAR>;
```

```
      <METAVAR>
    <LSD>::=#[DES_PORT] <DES_PORT>;
      <DES_PORT>
    <LSD>::=#[DES_PORT] <METAVAR>;
      <METAVAR>
    <LSD>::=#[LPOR1] <LPOR1>;
      <LPOR1>
    <LSD>::=#[LPOR1] <METAVAR>;
      <METAVAR>
    <LSD>::=#[LPOR] <LPOR>;
      <LPOR>
    <LSD>::=#[LPOR] <METAVAR>;
      <METAVAR>
    <LSD>::=#[POR1] <POR1>;
      <POR1>
    <LSD>::=#[POR1] <METAVAR>;
      <METAVAR>
    <LSD>::=#[MODULE] <MODULE>;
      <MODULE>
    <LSD>::=#[MODULE] <METAVAR>;
      <METAVAR>
    <LSD>::=#[DESCR_MOD] <DESCR_MOD>;
      <DESCR_MOD>
    <LSD>::=#[DESCR_MOD] <METAVAR>;
      <METAVAR>
    <LSD>::=#[INTERFACE] <L_PORTS_M>;
      <L_PORTS_M>
    <LSD>::=#[INTERFACE] <METAVAR>;
      <METAVAR>
    <LSD>::=#[PORT_M] <E>;
      <E>
    <LSD>::=#[PORT_M] <METAVAR>;
      <METAVAR>
    <LSD>::=#[PORT_M] <S>;
      <S>
    <LSD>::=#[PORT_M] <ES>;
      <ES>
    <LSD>::=#[PORT_M] <SE>;
      <SE>
    <METAVAR>::=%METAVAR;
      meta-atom(%METAVAR)
end chapter;

end definition
```


ANNEXE 3 : PROGRAMME DE DECOMPILATION

Le squelette de programme fourni avec l'environnement MENTOR - METAL **[MEL 85]**, a été complété pour le LSD réduit.

```
program DECLSD1;

     $import
          'SWITCH (pascal)':DECOMPARAMS;
          'TABLAN (pascal)':TABNAMES;
          'INTER (pascal)':CURRENTP,FLAGON;
          'TREEFACE (pascal)':ISALIST,OPER,HEAD,TAIL,ISEMPTY,CHILD;
          'SCHEMA (pascal)':OPKINDOF,LGPARAM;
          'DIS (pascal)':STOPDIS,BEGEMPH,ENDEMPH,GLUE,LINE,MARK,RELEASE,
                         TAB,BACKTAB,CHARWRITE,WRITEON,BIGWRITE;
          'BECANE (pascal)':STRINGOF;
          'SYSFCT (pascal)':UPPERC,LOWERC,LOWERB,UPPERB;
          'IDENT (pascal)':PRNAME;
          'ANNOT (pascal)':DECCOMMS,HASVISCOMMS$
     $export
          DECLSD$
     const
          PRE      =0;
          POST     =1;
          FUNNY    =-87654321;
          IDENT16  =16;
          MAXSTRING=160;
          IDENTFONT=0;
          KWFONT   =1;
          COMMON   =4;
          COMMOFF  =5;
          FLCOMMENT=2;
          MAXINAMES=2500;
          OPEXTERIEUR=1 (* exterieur *);
          OPNOM    =2 (* nom *);
          OPTEXTE  =3 (* texte *);
          OPMETA   =4 (* meta *);
          OPCOMMENT=5 (* comment *);
          OPCORPS  =6 (* corps *);
          OPDESCRMESS=7 (* descr_mess *);
          OPTYPE   =8 (* type *);
          OPREPART=9 (* repart *);
          OPAPPLIC=18 (* applic *);
          OPENTETE1=19 (* en_tete1 *);
          OPDESCRAPPLIC2=20 (* descr_applic2 *);
          OPNOMPOINTE=21 (* nompointe *);
          OPMODLIAIS=22 (* mod_liais *);
          OPMESSTYPES=23 (* mess_types *);
          OPMESS   =24 (* mess *);
          OPBIP    =25 (* bip *);
          OPDESPOR1=26 (* despor1 *);
          OPMODULE=27 (* module *);
          OPDESCRMOD=28 (* descr_mod *);
          OPE      =29 (* e *);
          OPS      =30 (* s *);
          OPCOMMENTS=35 (* comment_s *);
          OPLMOD   =36 (* l_mod *);
          OPLLIAI  =37 (* l_liai *);
          OPLMESS  =38 (* l_mess *);
          OPLTYPES=39 (* l_types *);
          OPLPOR1  =40 (* lpor1 *);
          OPLPOR   =41 (* lpor *);
          OPLPORTSM=42 (* l_ports_m *);
          OPDESCRAPPLIC1=47 (* descr_applic1 *);
          OPMULTIPSEL=48 (* multip_sel *);
          OPMULTIPDIFF=49 (* multip_diff *);
          OPES     =50 (* es *);
          OPSE     =51 (* se *);
          TKAPPLICATION=1 (* application *);
          TKFINAPPLICATION=2 (* fin_application *);
          TKPRESENTATION=3 (* presentation *);
          TKDESCRIPTION=4 (* description *);
          TKEXTERIEUR=5 (* exterieur *);
          TKMODULES=6 (* modules *);
          TKLIAISONS=7 (* liaisons *);
          TKMESSAGES=8 (* messages *);
          TKTYPES =9 (* types *);
          TKMESSAGE=10 (* message *);
```



```
          TKFINMESSAGE=11 (* fin_message *);
          TKVERS   =12 (* vers *);
          TKOU     =13 (* ou *);
          TKET     =14 (* et *);
          TKMODULE=15 (* module *);
          TKFINMODULE=16 (* fin_module *);
          TKINTERFACE=17 (* interface *);
          TKEPORT =18 (* e_port *);
          TKSPORT =19 (* s_port *);
          TKESPORT=20 (* es_port *);
          TKREPONSE=21 (* reponse *);
          TKSEPORT=22 (* se_port *);
          TKCORPS =23 (* corps *);
          TKDESCRMESS=24 (* descr_mess *);
          TKTYPE   =25 (* type *);
          TKREPART=26 (* repart *);
          TKSP28   =28 (* . *);
          TKSP29   =29 (* ; *);
          TKSP30   =30 (* : *);
     type ALFA     =packed array[1..8]of CHAR;
          OPERATORS=0..255;
          KINDS    =INTEGER;
          SMALLBUFS=array[1..IDENT16]of CHAR;
          BIGBUFS =array[1..MAXSTRING]of CHAR;
          OPERKIND=(KOPIDENT,KOPSTRING,KOPNUMBER,KOPVOID,KOPCHAR,KOPINT,KOPGATE,
                 KOPTRUC,KOPUNARY,KOPBINARY,KOPTERNARY,KOPSTARLIST,KOPPLUSLIST);
          TREES    =@INTEGER (* private type *);
          LANGUAGE=@INTEGER (* private type *);
          SYMBOL   =INTEGER;
          TTABNAMES=array[0..MAXINAMES]of SYMBOL;
          TDECOMPARAMS=
               record
                    TREE:TREES;
                    HOLO:INTEGER;
                    LANGUE:LANGUAGE
               end;
     var  CURRENTP:TREES (* tree denoted by the current pointer @k *);
          TABNAMES:TTABNAMES;
          WASLETTER:BOOLEAN (* true if the last written character is a letter *)
          VISCOMM:BOOLEAN (* true if comments has to be unparsed *);
          WASCOM:BOOLEAN (*true if comment has been unparsed *);
          PADIDENT:INTEGER (* minimum size of identifiers output *);
          LOCALANG:LANGUAGE;
          DECOMPARAMS:TDECOMPARAMS;
          BASEKEYWORD:INTEGER;

     function ISALIST(OP:OPERATORS;L:LANGUAGE):BOOLEAN;external (*
             Return the structure of the operator OP *);

     function OPKINDOF(OP:OPERATORS;L:LANGUAGE):OPERKIND;external (*
             Returns the value of the language KIND parameter *);

     function LGPARAM(LANGUE:LANGUAGE;KIND:INTEGER):INTEGER;external (*
             Returns true is the FLAG is set *);

     function FLAGON(FLAG:INTEGER):BOOLEAN;external (* In video mode, returns true
             when the screen is full; used in USERDECTABLE to stop unparsing *);

     function STOPDIS:BOOLEAN;external (*
             Returns the code of the root operator of T *);

     function OPER(T:TREES):OPERATORS;external (* The TREE L must be a list.
             Returns the first element of this list *);

     function HEAD(L:TREES):TREES;external (* The TREE L must be a list.
             Returns a list which is L without its first element *);

     function TAIL(L:TREES):TREES;external (* The TREE L must be a list.
             Returns true is this list is empty. *);

     function ISEMPTY(L:TREES):BOOLEAN;external (*
             Returns the N th son of the tree *);
```

```
function CHILD(N:INTEGER;T:TREES;L:LANGUAGE):TREES;external (*
        The first argument of STRINGOF must be an atomic tree,that is a
        tree with a nullary operator as root operator.
        It gives back the corresponding string and its length in B and LNG .
        Very usefull to get the atomic objects like identifiers, strings...*);

procedure STRINGOF(T:TREES;var B:BIGBUFS;var LNG:INTEGER;L:LANGUAGE);
        external (* Takes a character an gives it back in upper cases *);

function UPPERC(CH:CHAR):CHAR;external (*
        Takes a character and gives it back in lower cases *);

function LOWERC(CH:CHAR):CHAR;external (*
        Takes a buffer and gives it back in lower cases *);

procedure LOWERB(var B:BIGBUFS;L:INTEGER);external (*
        Takes a buffer and gives it back in upper cases *);

procedure UPPERB(var B:BIGBUFS;L:INTEGER);external (*
        Returns in BUF the text of the symbol SYMB. Used in KEYPRINT *);

procedure PRNAME(SYMB:INTEGER;var BUF:SMALLBUFS;var LNG:INTEGER);external
        (*Unparsing of comments *);

function DECCOMMS(WHERE:TREES;PREFIX:BOOLEAN;HOLO:INTEGER):BOOLEAN;
        external;

function HASVISCOMMS(WHERE:TREES;PREFIX:BOOLEAN):BOOLEAN;external (*
        In video mode, to enter in bold fount *);

procedure BEGEMPH;external (* In video mode, to come back in normal fount *);

procedure ENDEMPH;external (* Writes a space separator *);

procedure GLUE;external (* Begins a new line *);

procedure LINE;external (*
        Marks the current column; the next line will start at that column *);

procedure MARK;external (* To forget the previous MARK *);

procedure RELEASE;external (* Tabulation from the current column.
        Will be effective on the next line*);

procedure TAB;external (*
        Back tabulation: comes back to the previous position.
                Will be effective only on the next line *);

procedure BACKTAB;external (*
        Writes a single character on the current output device *);

procedure CHARWRITE(CH:CHAR);external (*
        Writes the buffer CH of length LNG on the current output device
        without splitting in font FONT.
        Use this procedure to write strings no longer than 16 *);

procedure WRITEON(CH:SMALLBUFS;N:INTEGER;FONT:INTEGER);external (*
        Write the buffer BUF of length LNG on the current output
        device without splitting.
        PAD is the minimum size of the output. If  PAD < LNG  the
        size will be LNG characters. If  PAD > LNG  the size will
        be PAD characters long ending with PAD-LNG blank characters.
        PAD may be used to make alignments.
        Use this procedure to output buffer no longer than 160 *);

procedure BIGWRITE(var BUF:BIGBUFS;LNG,PAD:INTEGER);external (*
        Procedure that writes a comment line
        on the current output device.*);

procedure PRINTCOMM(P:TREES);
     var  BB:BIGBUFS;
          L,I:INTEGER;
     begin
```



```
        STRINGOF(P,BB,L,LOCALANG);
        BIGWRITE(BB,L,PADIDENT)
        end;

    procedure WLINE;
        begin
        WASLETTER:=FALSE;
        LINE
        end (*WLINE P may be any atomic tree. Its corresponding
              string will be written without splitting *);

    procedure PRINTATOM(P:TREES);
        var  BB:BIGBUFS;
             L,I:INTEGER;
        begin
        STRINGOF(P,BB,L,LOCALANG);
        if WASLETTER then GLUE;
        BIGWRITE(BB,L,PADIDENT);
        WASLETTER:=TRUE
        end (* P may be any atomic tree. Its corresponding
              string will be written without splitting but preceded by a given
              character ( here a $ ). Used here to write meta-variables *);

    procedure PRINTMETA(P:TREES);
        var  BB:BIGBUFS;
             L,II:INTEGER;
        begin

STRINGOF(P,BB,L,LOCALANG);
        if WASLETTER then GLUE;
        for II:=L+1 downto 2 do BB[II]:=BB[II-1];
        BB[1]:='$';
        BIGWRITE(BB,L+1,PADIDENT);
        WASLETTER:=TRUE
        end (* To write a keyword. TOK is the constant declared for the
               keyword. *);

    procedure KEYPRINT(TOK:INTEGER);
        var  BUFFER:SMALLBUFS;
             LNG:INTEGER;
        begin
        if WASLETTER then GLUE;
        PRNAME(TABNAMES[BASEKEYWORD+TOK],BUFFER,LNG);
        WRITEON(BUFFER,LNG,KWFONT);
        WASLETTER:=TOK in[1..26]
        end (* To write holophrasting characters *);

    procedure PRINTHOLO(OP:OPERATORS);
        var  BUFFER:SMALLBUFS;
             LNG:INTEGER;
        begin
        if WASLETTER then GLUE;
        if ISALIST(OP,LOCALANG) then
             begin
             BUFFER[1]:='.';
             BUFFER[2]:='.';
             BUFFER[3]:='.';
             LNG:=3
             end
        else begin
             BUFFER[1]:='#';
             LNG:=1
             end;
        WRITEON(BUFFER,LNG,KWFONT);
        WASLETTER:=TRUE
        end (*USERDECTABLE is the unparsing procedure. *);

    procedure USERDECTABLE(P:TREES;HOLO:INTEGER);
         (* Unparsing of list nodes *)

         procedure DECLIST(P:TREES);
              (* Vertical unparsing of a list node (exple: statements list).
                 If the list needs a separator, the corresponding keyword
```

```
        must be put at the place of the meta-variable $SEPARATOR *);

    procedure HORIZDECLST(PTEMP:TREES);
        label
            1;
        begin
        MARK;
        1: USERDECTABLE(HEAD(PTEMP),HOLO-1);
        PTEMP:=TAIL(PTEMP);
        if PTEMP#nil then
            begin
            WASLETTER:=FALSE;
            KEYPRINT(TKSP29);
            goto 1
            end;
        RELEASE
        end (* noms de ports separes par OU *);

    procedure HORIZDECLST2(PTEMP:TREES);
        label
            1;
        begin
        MARK;
        1: USERDECTABLE(HEAD(PTEMP),HOLO-1);
        PTEMP:=TAIL(PTEMP);
        if PTEMP#nil then
            begin
            KEYPRINT(TKOU);
            goto 1
            end;
        RELEASE
        end (* noms de ports separes par ET *);

    procedure HORIZDECLST3(PTEMP:TREES);
        label
            1;
        begin
        MARK;
        1: USERDECTABLE(HEAD(PTEMP),HOLO-1);
        PTEMP:=TAIL(PTEMP);
        if PTEMP#nil then
            begin
            KEYPRINT(TKET);
            goto 1
            end;
        RELEASE
        end (* Body of DECLIST *);

    begin
    case OPER(P) of
        OPCOMMENTS: ;
        OPLMOD: VERTICDECLST(P);
        OPLLIAI: VERTICDECLST(P);
        OPLMESS: VERTICDECLST(P);
        OPLTYPES: VERTICDECLST(P);
        OPLPOR1: HORIZDECLST2(P);
        OPLPOR: HORIZDECLST3(P);
        OPLPORTSM:
            begin
            KEYPRINT(TKINTERFACE);
            WLINE;
            TAB;
            HORIZDECLST(P);
            WASLETTER:=FALSE;
            KEYPRINT(TKSP29);
            BACKTAB;
            end
    end
    end (*DECLIST Unparsing of nullary nodes *);

procedure DECTA0(var P:TREES);
    begin
    case OPER(P) of
```



```
        OPEXTERIEUR: PRINTATOM(P);
        OPNOM: PRINTATOM(P);
        OPTEXTE:
            begin
            CHARWRITE('''');
            WASLETTER:=FALSE;
            PRINTATOM(P);
            WASLETTER:=FALSE;
            CHARWRITE('''');
            end;
        OPMETA: PRINTMETA(P);
        OPCOMMENT: PRINTCOMM(P);
        OPCORPS:
            begin
            PRYNTATOM(P);
            end;
        OPDESCRMESS:
            begin
            KEYPRINT(TKDESCRIPTION);
            WLINE;
            TAB;
            PRINTATOM(P);
            BACKTAB;
            end;
        OPTYPE:
            begin
            PRINTATOM(P);
            end;
        OPREPART:
            begin
            PRINTATOM(P);
            end
    end
    end (*DECTA0 Unparsing of unary nodes *);

procedure DECTA1(var P:TREES);
    begin

    end;

procedure DECTA2(var P:TREES);
    begin
    case OPER(P) of
        (* Put here a case for each binary operator*)
        OPAPPLIC:
            begin
            KEYPRINT(TKAPPLICATION);
            USERDECTABLE(CHILD(1,P,LOCALANG),HOLO-1);
            WLINE;
            TAB;
            USERDECTABLE(CHILD(2,P,LOCALANG),HOLO-1);
            BACKTAB;
            WLINE;
            KEYPRINT(TKFINAPPLICATION);
            WASLETTER:=FALSE;
            KEYPRINT(TKSP28);
            end;
        OPENTETE1:
            begin
            USERDECTABLE(CHILD(1,P,LOCALANG),HOLO-1);
            WLINE;
            TAB;
            KEYPRINT(TKPRESENTATION);
            WLINE;
            TAB;
            USERDECTABLE(CHILD(2,P,LOCALANG),HOLO-1);
            BACKTAB;
            BACKTAB;
            end;
        OPDESCRAPPLIC2:
            begin
            KEYPRINT(TKDESCRIPTION);
            WLINE;
```

```
     TAB;
     USERDECTABLE(CHILD(1,P,LOCALANG),HOLO-1);
     WLINE;
     USERDECTABLE(CHILD(2,P,LOCALANG),HOLO-1);
     BACKTAB;
     end;
OPNOMPOINTE:
     begin
     USERDECTABLE(CHILD(1,P,LOCALANG),HOLO-1);
     WASLETTER:=FALSE;
     KEYPRINT(TKSP28);
     USERDECTABLE(CHILD(2,P,LOCALANG),HOLO-1);
     end;
OPMODLIAIS:
     begin
     KEYPRINT(TKMODULES);
     WLINE;
     TAB;
     USERDECTABLE(CHILD(1,P,LOCALANG),HOLO-1);
     WASLETTER:=FALSE;
     KEYPRINT(TKSP29);
     BACKTAB;
     WLINE;
     KEYPRINT(TKLIAISONS);
     WLINE;
     TAB;
     USERDECTABLE(CHILD(2,P,LOCALANG),HOLO-1);
     WASLETTER:=FALSE;
     KEYPRINT(TKSP29);
     BACKTAB;
     end;
OPMESSTYPES:
     begin
     KEYPRINT(TKMESSAGES);
     WLINE;
     TAB;
     USERDECTABLE(CHILD(1,P,LOCALANG),HOLO-1);
     WASLETTER:=FALSE;
     KEYPRINT(TKSP29);
     BACKTAB;
     WLINE;
     KEYPRINT(TKTYPES);
     WLINE;
     TAB;
     USERDECTABLE(CHILD(2,P,LOCALANG),HOLO-1);
     BACKTAB;
     end;
OPMESS:
     begin
     KEYPRINT(TKMESSAGE);
     USERDECTABLE(CHILD(1,P,LOCALANG),HOLO-1);
     WLINE;
     TAB;
     USERDECTABLE(CHILD(2,P,LOCALANG),HOLO-1);
     BACKTAB;
     WLINE;
     KEYPRINT(TKFINMESSAGE);
     end;
OPBIP:
     begin
     USERDECTABLE(CHILD(1,P,LOCALANG),HOLO-1);
     KEYPRINT(TKVER[);
     USERDECTABLE(CHILD(2,P,LOCALANG),HOLO-1);
     end;
OPDESPOR1:
     begin
     USERDECTABLE(CHILD(1,P,LOCALANG),HOLO-1);
     WASLETTER:=FALSE;
     KEYPRINT(TKSP30);
     USERDECTABLE(CHILD(2,P,LOCALANG),HOLO-1);
     end;
OPMODULE:
     begin
```

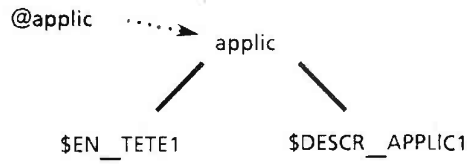


```
     KEYPRINT(TKMODULE);
     USERDECTABLE(CHILD(1,P,LOCALANG),HOLO-1);
     WLINE;
     TAB;
     USERDECTABLE(CHILD(2,P,LOCALANG),HOLO-1);
     BACKTAB;
     WLINE;
     KEYPRINT(TKFINMODULE);
     end;
OPDESCRMOD:
     begin
     KEYPRINT(TKDESCRIPTION);
     WLINE;
     TAB;
     USERDECTABLE(CHILD(1,P,LOCALANG),HOLO-1);
     WLINE;
     USERDECTABLE(CHILD(2,P,LOCALANG),HOLO-1);
     BACKTAB;
     end;
OPE:
     begin
     KEYPRINT(TKEPORT);
     USERDECTABLE(CHILD(1,P,LOCALANG),HOLO-1);
     WASLETTER:=FALSE;
     KEYPRINT(TKSP30);
     USERDECTABLE(CHILD(2,P,LOCALANG),HOLO-1);
     end;
OPS:
     begin
     KEYPRINT(TKSPORT);
     USERDECTABLE(CHILD(1,P,LOCALANG),HOLO-1);
     WASLETTER:=FALSE;
     KEYPRINT(TKSP30);
     USERDECTABLE(CHILD(2,P,LOCALANG),HOLO-1);
     end
end
end (*DECTA2 Unparsing of ternary nodes *);

procedure DECTA3(var P:TREES);
     begin
     case OPER(P) of
          OPDESCRAPPLIC1:
               begin
               KEYPRINT(TKDESCRIPTION);
               WLINE;
               TAB;
               USERDECTABLE(CHILD(1,P,LOCALANG),HOLO-1);
               WLINE;
               USERDECTABLE(CHILD(2,P,LOCALANG),HOLO-1);
               WLINE;
               USERDECTABLE(CHILD(3,P,LOCALANG),HOLO-1);
               BACKTAB;
               end;
          OPMULTIPSEL:
               begin
               USERDECTABLE(CHILD(1,P,LOCALANG),HOLO-1);
               KEYPRINT(TKVERS);
               USERDECTABLE(CHILD(2,P,LOCALANG),HOLO-1);
               KEYPRINT(TKOU);
               USERDECTABLE(CHILD(3,P,LOCALANG),HOLO-1);
               end;
          OPMULTIPDIFF:
               begin
               USERDECTABLE(CHILD(1,P,LOCALANG),HOLO-1);
               KEYPRINT(TKVERS);
               USERDECTABLE(CHILD(2,P,LOCALANG),HOLO-1);
               KEYPRINT(TKET);
               USERDECTABLE(CHILD(3,P,LOCALANG),HOLO-1);
               end;
          OPES:
               begin
               KEYPRINT(TKESPORT);
               USERDECTABLE(CHILD(1,P,LOCALANG),HOLO-1);
```

```
                    WASLETTER:=FALSE;
                    KEYPRINT(TKSP30);
                    USERDECTABLE(CHILD(2,P,LOCALANG),HOLO-1);
                    KEYPRINT(TKREPONSE);
                    USERDECTABLE(CHILD(3,P,LOCALANG),HOLO-1);
                    end;
               OPSE:
                    begin
                    KEYPRINT(TKSEPORT);
                    USERDECTABLE(CHILD(1,P,LOCALANG),HOLO-1);
                    WASLETTER:=FALSE;
                    KEYPRINT(TKSP30);
                    USERDECTABLE(CHILD(2,P,LOCALANG),HOLO-1);
                    KEYPRINT(TKREPONSE);
                    USERDECTABLE(CHILD(3,P,LOCALANG),HOLO-1);
                    end
          end
          end (*DECTA3Body of USERDECTABLE*);
     begin
     if(P#nil)and not STOPDIS then
          begin
          if P=CURRENTP then BEGEMPH;
          if HOLO>0 then
               begin
               if VISCOMM then
                    begin
                    WASCOM:=DECCOMMS(P,TRUE,HOLO);
                    if WASCOM then WLINE
                    end
               else WASCOM:=FALSE;
               case OPKINDOF(OPER(P),LOCALANG) of
                    KOPIDENT,KOPSTRING,KOPNUMBER,KOPVOID,KOPCHAR,KOPINT,
                     KOPGATE,KOPTRUC: DECTA0(P);
                    KOPUNARY: DECTA1(P);
                    KOPBINARY: DECTA2(P);
                    KOPTERNARY: DECTA3(P);
                    KOPSTARLIST,KOPPLUSLIST: DECLIST(P)
               end;
               if VISCOMM then
                    begin
                    if HASVISCOMMS(P,FALSE) then WLINE;
                    WASCOM:=DECCOMMS(P,FALSE,HOLO);
                    if WASCOM then WLINE
                    end
               else WASCOM:=FALSE
               end
          else  (* Holophrast *)
               PRINTHOLO(OPER(P));
          if P=CURRENTP then ENDEMPH
          end
     end (*USERDECTABLE*);

procedure LSDDECOMP(X:TREES;HOLOPH:INTEGER;L:LANGUAGE);
     begin
     WASLETTER:=FALSE;
     PADIDENT:=0;
     VISCOMM:=FLAGON(FLCOMMENT);
     LOCALANG:=L;
     BASEKEYWORD:=LGPARAM(LOCALANG,1);
     if HOLOPH=-1 then HOLOPH:=10000 (* P* COMMAND*);
     USERDECTABLE(X,HOLOPH)
     end

procedure DECLSD;
     begin
     with DECOMPARAMS do LSDDECOMP(TREE,HOLO,LANGUE)
     end;

begin

end.
```


ANNEXE 4 : UN EXEMPLE DE CONSTRUCTION DE PROGRAMME LSD A L'AIDE DE MENTOL

Lors de la mise en oeuvre de l'environnement MENTOR-LSD, MENTOR associe à chaque opérateur un schéma prédéfini.
Par exemple, à l'opérateur "**applic**" est associé un pointeur **@applic** qui repère l'arbre abstrait:

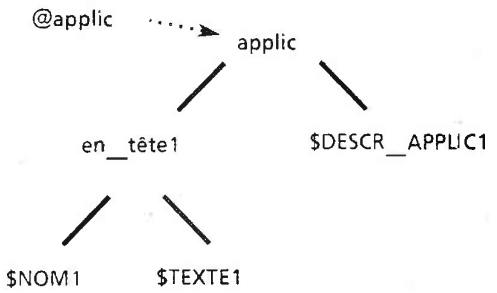


On utilisera ces schémas et quelques fonctions MENTOL pour créer un programme LSD.
Au fur et à mesure de cette construction on présentera l'arbre abstrait correspondant. Une vue arborescente étant absolument nécessaire pour une bonne compréhension des fonctions MENTOL présentées au paragraphe IV.3 .

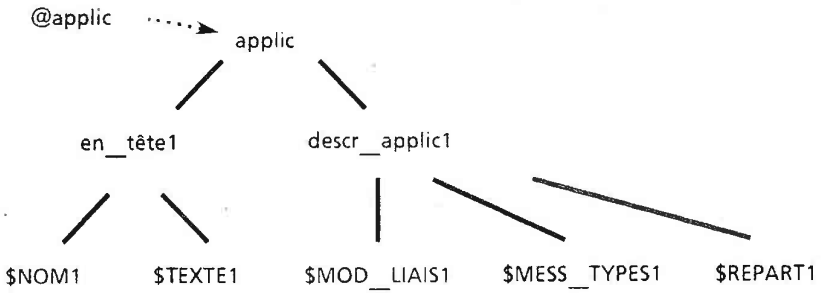
Le caractère "?" signale que MENTOL attend une commande. Les commentaires seront précédés du caractère "§".

```
?@A:@applic                    § le programme sera repéré par le
                                 pointeur @A
?@A P                          § imprimer pour voir
application $EN__TETE1
     $DESCR__APPLIC1           § cf. l'arbre abstrait ci-dessus
fin__application.
?@A S                          § accès au premier fils à gauche
?@A P
$EN__TETE1
?@A C @en__tête1               § changer une partie de l'arbre par le
                                 schéma prédéfini de en__tête1
?@A P
$NOM1
     présentation
          $TEXTE1
?@A U                          § remonter d'un niveau dans l'arbre
?@A P
```

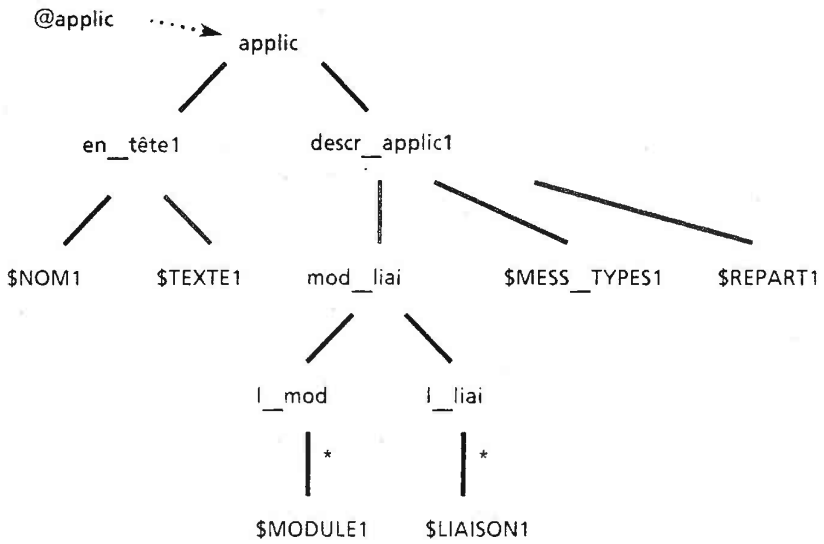
```
application $NOM1
      présentation
            $TEXTE1
      $DESCR_APPLIC1
fin_application.
```




```
?@A S2                                  § accès au deuxième fils
?@A P
$DESCR_APPLIC1
?@A C @descr_applic1
?@A P
description
      $MOD_LIAIS1
      $MESS_TYPES1
      $REPART1
?@A U
?@A P
application $NOM1
      présentation
            $TEXTE1
      description
            $MOD_LIAIS1
            $MESS_TYPES1
            $REPART1
fin_application.
```

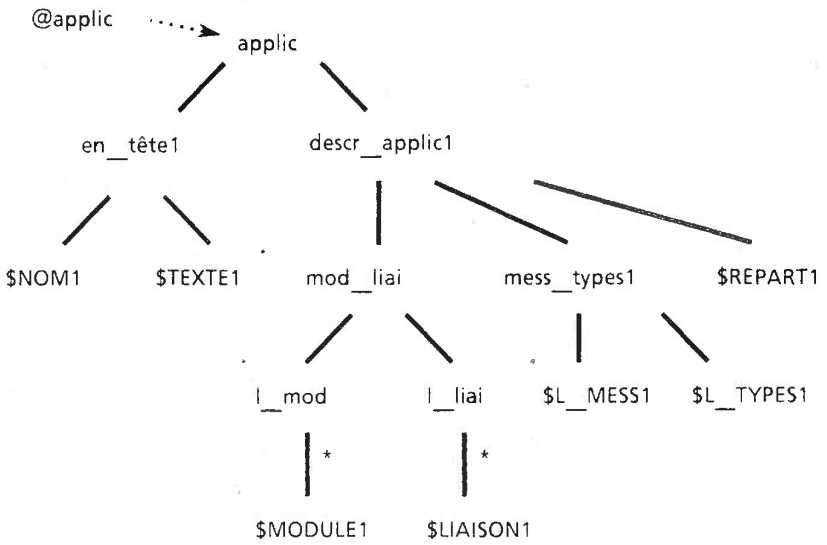

```
?@A S2S
?@A P
$MOD_LIAIS1
?@A C @mod_liais
?@A P
modules
      $L_MOD1;
liaisons
      $L_LIAI1;
?@A S
?@A C @l_mod                            § cet opérateur est un noeud de type
                                        liste; une * est portée sur l'arbre abstrait
                                        pour repérer ce type de noeud.
?@A U
?@A S2 C @l_liai
?@A U*                                  § on remonte à la racine
?@A P
application $NOM1
      présentation
            $TEXTE1
      description
            modules
                  $MODULE1;
            liaisons
                  $LIAISON1;
            $MESS_TYPES1
            $REPART1
fin_application.
```




```
?@A S2S2
?@A P
$MESS__TYPES1
?@A C @mess__types
?@A P
messages
      $L__MESS1;
types
      $L__TYPES1
?@A U*
?@A P
```



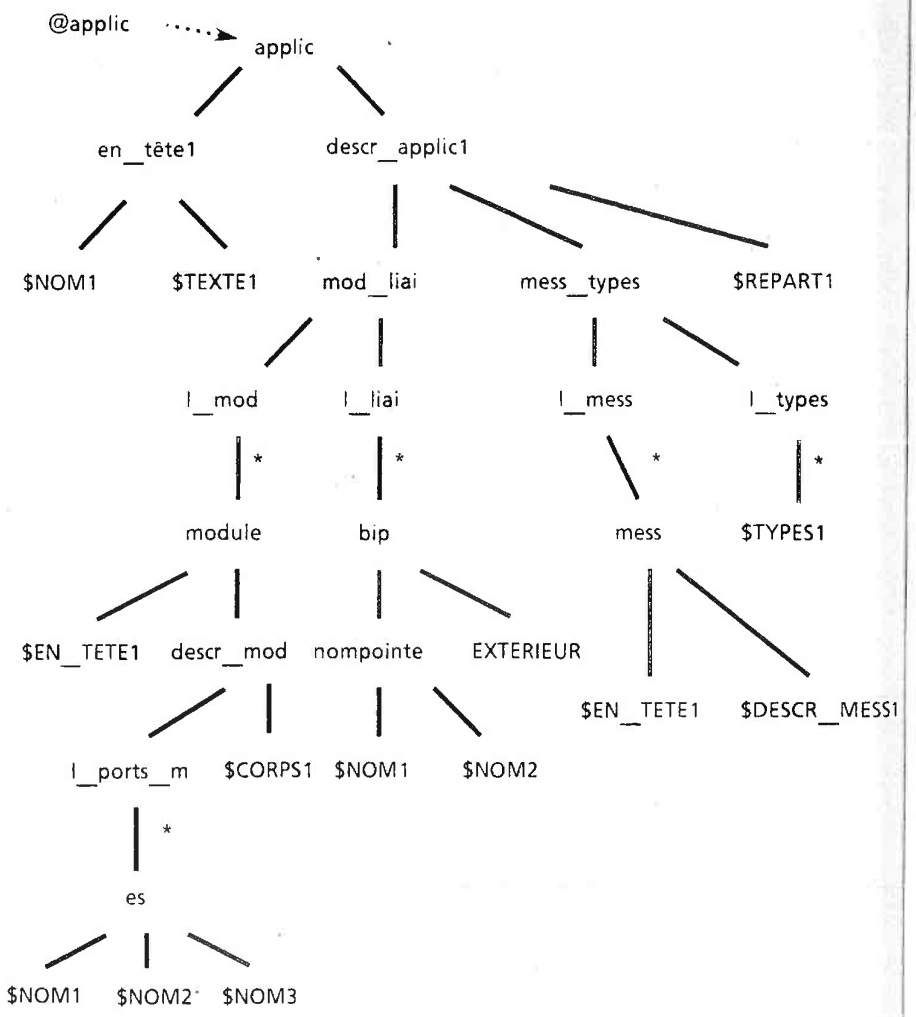
```
application $NOM1
      présentation
            $TEXTE1
      description
            modules
                  $MODULE1;
            liaisons
                  $LIAISON1;
            messages
                  $L__MESS1;
            types
                  $L__TYPES1
            $REPART1
fin__application.
```



```
?@A S2SSS
?@A P
$MODULE1
?@A C @module
?@A P
module $EN_TETE1
      $DESCR_MOD1
fin_module
?@A S2
?@A C @descr_mod
?@A P
description
      $INTERFACE1
      $CORPS1
?@A S
?@A C @I_ports_m
?@A P
interface
      $PORT_M1;
?@A S
?@A C @es
?@A P
es_port $NOM1 : $NOM2 réponse $NOM3;
?@A U*
?@A P*
application $NOM1
      présentation
            $TEXTE1
      description
            modules
                  module $EN_TETE1
                        description
                              interface
                                    es_port $NOM1 : $NOM2 réponse $NOM3;
                              $CORPS1
                  fin_module;
            liaisons
                  $LIAISON1;
            messages
                  $L_MESS1;
            types
                  $L_TYPES1
            $REPART1
fin_application.
?@A S2SS2S
?@A P
$LIAISON1
?@A C @bip
?@A P
```



```
$DES_PORT1 vers $DES_PORT2
?@A S
?@A C @nompointe
?@A P
$NOM1 . $NOM2
?@A U
?@A S2
?@A C:&
[NOM]:EXTERIEUR                    § saisie d'un nom au clavier
?@A U*
?@A S2S2S
?@A P
$L_MESS1
?@A C @I_mess
?@A S
?@A C @mess
?@A P
message $EN_TETE1
      $DESCR_MESS1
fin_message
?@A U*
?@A S2S2S2
?@A P
$L_TYPES1
?@A C @I_types
?@A P
$TYPE1
?@A U*
?@A P*
application $NOM1
      présentation
            $TEXTE1
      description
            modules
                  module $EN_TETE1
                        description
                              interface
                                    es_port $NOM1 : $NOM2 réponse $NOM3;
                              $CORPS1
                  fin_module;
            liaisons
                  $NOM1.$NOM2 vers EXTERIEUR;
            messages
                  message $EN_TETE1
                        $DESCR_MESS1
                  fin_message;
            types
                  $TYPES1
            $REPART1
fin_application.
```



ANNEXE 5 : LES PROCEDURES MENTOL CACHEES

Les fonctions MENTOL suivantes ont été créées pour permettre la compilation et l'interprétation des commandes graphiques. L'utilisateur n'y a pas accès directement.
Avant de les décrire, précisons les effets des instructions MENTOL de base utilisées.

: affectation simple
exemple: @K : @e => @K et @e repèrent le même arbre;

= copie
exemple: @K = @P => @K et @P repèrent 2 arbres identiques mais distincts;

C change un arbre en un autre
exemple: @K C @e => le sous-arbre repéré par @K est remplacé par l'arbre repéré par @e;

E évaluation de l'arbre, c'est-à-dire remplacer toute métavariable de l'arbre par un arbre de même nom;

I insertion à droite sous un opérateur de type liste;

F recherche d'un schéma dans un arbre;

D détruit un sous-arbre;

? (inst1,inst2) si l'instruction précédant le point d'interrogation "réussit" alors l'instruction "inst1" est exécutée sinon "inst2" est exécutée;

(inst1 / inst2) l'instruction "inst1" est exécutée, si elle "réussit", "inst2" est exécutée et la liste entre parenthèses n'est plus évaluée sinon "inst2" n'est pas exécutée mais l'évaluation de la liste se poursuit (l'instruction de cas n'échoue pas);

(liste insts)n la liste d'instructions est effectuée n fois. Si n est remplacée par "*" la boucle est infinie;

$n
$-n permet de sortir de n boucles;
idem que $n mais en signalant que la commande à échouée;

.equal<@K,@P> retourne vrai si les pointeurs @K et @P repèrent un même endroit dans l'arbre abstrait;

.eqval<@K,@P> retourne vrai si les 2 pointeurs désignent des arbres identiques;

.is<@K,@P> retourne vrai si les 2 pointeurs désignent des opérateurs identiques;

La traduction d'une structuration en programme LSD débute par l'appel d'une fonction d'initialisation qui créé le programme suivant avec comme nom d'application le nom de l'agent racine.

```
application nomapplication
    présentation
        $TEXTE1
    description
        modules
            $L_MOD1;
        liaisons
            $L_LIAI1;
        messages
            $L_MESS1;
        types
            $L_TYPES
fin_application.
```

% fonction d'initialisation

```
.def <.init <@NOMAP> , ( @k:@applic ; @EN_TETE1:@EN_TETE1 ; :E ;
@NOM1:@NOMAP; :E ; @DESCR_APPLIC1:@DESCR_APPLIC2 ; :E ;
@MOD_LIAIS1:@MOD_LIAIS ; :E ; @MESS_TYPES1:@MESS_TYPES ; :E) >
```

Les autres procédures sont regroupées selon les entités structurantes qu'elles concernent:

a) Les modules.

% création du premier module

```
.def <.lmod.<@NOMMOD , @LPM> , ( @L_MOD1:@L_MOD ; :E ;
@MODULE1:@MODULE ; :E;@EN_TETE1:@EN_TETE1; :E ; @NOM1:@NOMMOD;
:E ; @DESCR_MOD1:@DESCR_MOD ; :E ; @INTERFACE1:@LPM ; :E) >
```

Elle remplace $L_MOD1 par:

```
module nommodule
    présentation
        $TEXTE1
    description
        $INTERFACE1
        $CORPS
    fin_module
```

La métavariable $INTERFACE1 étant remplacée par la liste des ports passée en paramètre via le pointeur @LPM.

% création d'un autre module

```
.def <.mod <@NOMMOD,@LPM> , ( U* ; S2SSSR* ; I@MODULE ; R ; S ;
C@EN_TETE1 ; S ; C@NOMMOD ; U2 ; S2 ; C@DESCR_MOD ; SC@LPM ; U*) >
```

Rajoute la description d'un nouveau module dans le texte.

% création d'un module sans interface

```
.def<.admod<@M>,(@A C @module;@A SC @en_tete1;@A SSC @M;@A S2 C
@descr_mod;U*;S2SSS;S;?(UR*;I @A),C @A;U*)>
```

Ajoute la description d'un module sans son interface dans la liste des modules. Fonction utilisée en mode mise à jour.

% renommer un module

```
.def<.renoma<@M,@N>,(U*;S2SSS;@A:@K;(SS;.eqval<@M,@K>;
?(C@N;$2),(@AR;@K:@A))*;U*;S2SS2S;@A:@K;(F@M;?(@B:@K;@BU;@BS;.equal
<@K,@B>;?C@N,;@AR; ?(@K:@A),$2) ,$1)*;U*)>
```

Renomme un module ainsi que toutes les occurences de ce nom apparaissant dans la description des liaisons. En effet, le nom du module peut apparaitre dans une notation pointée de désignation de port (nommod.nomport).

% supprimer un module

```
.def<.supmod<@M>,(U*;S2SS;F @M;U2;D;S;?,C @I_mod;U*)>
```

Suppression d'un module. Si ce module est le dernier on le remplace par la métavariable $MODULE1.

```
.def<.supmod2<@M>,(U*;S2SS2;@A:@K;(F@M;?(U2;@B:@K;@B
U;.equal<@A,@B>;?D,(U;D);S;?U,C @I_liai),$1)*;U*)>
```

Suppression de toutes les liaisons mettant en jeu le module supprimé.

b) Les liaisons.

% création de la première liaison

```
.def <.lliaison<@L> , ( @L_LIAI1:@L_LIAI ; :E ; @LIAISON1:@L ; :E) >
```

Remplace $L_LIAI1 par la liaison passée en paramètre.

% création d'une autre liaison

```
.def <.liaison<@L> , ( U* ; S2SS2SR*I@L) >
```

Rajoute une liaison dans la liste des liaisons.

% création de liaison en mode mise à jour

```
.def <.tradporsc<@M,@A,@B> , (U* ; S2SS2S ; S ; ?((F @M
;?(U;S3;?(S;?(R*;II@B),C@B), (@C C @multip_sel ; @C SC @M ; @C S2C @K S2 ; @C
S3 C @B ; C @C); $2), (U ; R ; ?S,(I@A ; $3)))*),C @A ; U*)>
```

Création d'une liaison multipoint en sélection ou modification de la liaison existante (ajout de ports de réception).

```
.def <.tradporsd<@M,@A,@B>,(U*;S2SS2S; S ; ?((F @M ; ?(U ; S3 ;
?(S;?(R*;II@B),C@B), (@C C @multip_diff ; @C SC @M ; @C S2C @K S2 ; @C S3 C @B
; C @C); $2), (U ; R ; ?S,(I@A ; $3)))*),C @A ; U*)>
```

Création d'une liaison multipoint en diffusion ou modification de la liaison existante (ajout de ports de réception).

```
.def <.tradporse<@M,@A,@B> , (U* ; S2SS2S ; S ; ?((F @M ; ?(U ; S3 ; ?(S R* ; II@B) ,
(@CC @multip_sel ; @C SC @M ; @C S2C @K S2 ; @C S3 C @B ; C @C); $2), (U ; R ;
?S,(I@A ; $3)))*),C @A ; U*)>
```

Création d'une liaison bidirectionnelle.

```
.def<.tradpors<@M>,(U* ; S2SS2S ; S ; ?(U ; R* ; I @M) , C @M ; U*)>
```

Ajoute une liaison de type quelconque dans la liste.

```
.def<.tradporscex<@M,@A,@L>,(U*;S2SS2S;S;?((F@M;?(U;S3;?(S;?(R*;I@L),
(C@lpor1;S;C@L)), (@C C @multip_sel;@C S C @M;@C S2C @KS2;@C S3 C @lpor1;
@CS3SC @L;C @C);$2),(U;R;?S,(I @A;$3)))*),C @A;U*)>
```

Création d'une liaison multipoint en sélection avec un port de type EXTERIEUR côté réception ou modification d'une liaison bipoint.

```
.def<.tradporsdex<@M,@A,@L>,(U*;S2SS2S;S;?((F@M;?(U;S3;?(S;?(R*;I@L),
(C@lpor;S;C @L)),(@C C @multip_diff;@C SC @M;@C S2C @KS2;@C S3 C @lpor;
@CS3SC @L;C @C);$2),(U;R;?S,(I @A;$3)))*),C @A;U*)>
```

Création d'une liaison multipoint en diffusion avec un port de type EXTERIEUR côté réception ou modification d'une liaison bipoint.

```
.def<.tradporseex<@M,@A,@L>,(U*;S2SS2S;S;?((F@M;?(U;S3;?(SR*;I@L),
(@CC@multip_sel;@C S C @M;@C S2 C @K S2;@C S3 C @lpor1;@C S3S C@L;C
@C);$2), (U;R;?S,(I @A;$3)))*),C @A;U*)>
```

Création d'une liaison bidirectionnelle en sélection avec un port de type EXTERIEUR côté réception ou modification d'une liaison bipoint.

```
.def<.tradporexec2<@M,@A>,(U*;S2SS2;(F@M;?(US;S;?U2R,$2),)*;U;S3;
?,(@CC@multip_sel;@C SC @K S;@C S2 C @K S2;@K C @C;S3);C @A;U*)>
```

Modification d'une liaison en une liaison en sélection avec un port EXTERIEUR en émission.

```
.def<.tradporexed2<@M,@A>,(U*;S2SS2;(F@M;?(US;S;?U2R,$2),)*;U;S3;
?,(@CC@multip_diff;@C SC @K S;@C S2 C @K S2;@K C @C;S3);C @A;U*)>
```

Modification d'une liaison en une liaison en diffusion avec un port EXTERIEUR en émission.

```
.def<.tradporexes2<@M,@A>,(U*;S2SS2;(F@M;?(US;S;?U2R,$2),)*;U;
@CC@multip_sel; @C S C @K S;@C S2 C @K S2;@K C @C;S3;C @A;U*)>
```

Modification d'une liaison bidirectionnelle bipoint en une liaison bidirectionnelle multipoint.

```
.def<.tradporex2<@A,@B>,(U*;S2SS2S; (S; S; ?(U2;R), (U;S3; ?(S; ?(S; ?(U2; @C C
@K; U;@CSR* I @K S2; eqval2<@C,@B>;?$5,(R;$4)),U2R),UR),R))*;S3SSR*II@A)>
```

Rajoute la liste de ports repérée par @A à la liaison partant d'un port EXTERIEUR vers la liste de ports repérée par @B.

%supprimer une liaison

```
.def<.otelien<@M>,(U*;S2SS2;F @M;?(U;D;S;?,C @I_liai;U*),)>
```

Suppression du lien complet issu du port repéré par @M. Si ce lien est de type multipoint, il est recréé ultérieurement avec les ports de réception restants. Si ce port est le dernier on le remplace par $LIAISON1.

```
.def<.otedemilien<@A>,(U*;S2SS2S;(S;S;?(U2;R),(U;S3;?(S;?(U;@B C @K;U;@B
SR*I @K S2; eqval2<@B,@A>;?$4,R),(U;(.is<@A,@lpor>/@B C @lpor;@B C
@lpor1);@B SC @K S2; eqval2<@B,@A>;?$4,R)),R))*;D;S;?,C @I_liai;U*)>
```

Suppression d'un demi-lien de type sélection ou diffusion avec un port EXTERIEUR en émission.

```
.def<.otelienexbip<@A>,(U*;S2SS2;F @A;D;S;?,C @I_liai;U*)>
```

Suppression d'un demi-lien de type bipoint avec un port EXTERIEUR en émission.

```
.def<.eqval2<@A,@B>,(.nbfilsegal<@A,@B>;?((.is<@A,@lpor>/(.is<@B,@lpor
>; ?,$-4);.is<@A,@lpor1>/(.is<@B,@lpor1>;?,$-4));@D:@A S;(@C = @D;@B F
@C;?(@B U;@D R;?,$4),$-3)*),$-1)>
```


Vérifie si les 2 arbres ont les mêmes fils sans pour autant qu'ils soient à la même place.

```
.def<.nbfilsegal<@A,@B>,(@A S;@B S;(@A R;?(@B R;?,$-2),(@B R;?$-2,$2))*)>
```

c) Les ports.

%création du premier port

```
.def <.lport <@M , @L> , (U* ; S2SS ; F @M ; U2S2S ; C @L ; U* )>
```

Créé le premier port sur le module repéré par @M. Fonction utilisée en mode mise à jour.

%création d'un autre port

```
.def <.port <@M , @L> , (U* ; S2SS ; F @M ; U2S2SSR* ; I @L ; U* )>
```

Rajoute un port au module repéré par @M. Fonction utilisée en mode mise à jour.

%ajouter le message au port simple (sans réponse)

```
.def <.admespors <@M , @P , @N> , ( U* ; S2SS ; F @M ; U2S2S ; F @P ; US2C@N ; U*
)>
```

%ajouter le message au port avec réponse

```
.def <.admesporse < @M , @P , @N1 , @N2> , ( U* ; S2SS ; F @M ; U2S2S ; F @P ; US2
C @N1 ; US3 C @N2 ; U* )>
```

%renommer un port

```
.def<.renomp<@M,@P1,@P2>,(U*;S2SSS;@A:@K;(SS;.eqval<@M,@K>;
?(U2;S2SS;@A:@K;(S;.eqval<@P1,@K>;?(C @P2;$4),(@A R;@K:@A))*),(@A
R;@K:@A))*;@A = @nompointe;@A SC @M;@A S2 C @P1;(U*;S2SS2;F @A;? S2 C
@P2,$1)*;U*)>
```

%supprimer un port

```
.def<.otepor3<@M,@P>,(U*;S2SS;F@M;U2S2S;F@P;U;D;S;?,C
@I_ports_m;U*)>
```

d) Les messages.

%création du premier message

```
.def <.lmess<@L> , ( @I_mess1:@I_mess ; :E ; @mess1:@L ; :E )>
```

Remplace $L_MESS1 par le message passé en paramètre.

%création d'un autre message

```
.def <.mess<@L> , ( U* ; S2S2SSR*I@L )>
```

Rajoute un message à la liste des messages.

%ajouter un message s'il y a lieu

```
.def <.ajoutmess<@M> , ( U* ; S2S2S ; F @M ; ? ; (@N C @mess ; @N S C@M ; S ; S ;
?(U;R* ; I @N ), C @N ; U* ))>
```

Ajoute un message si celui-ci n'existe pas déjà. Fonction utilisée lors d'une mise à jour.

%supprimer des messages inutiles

```
.def<.supmess,(U*;S2S2S;@L:@K;S;(@C:@K;R;?(@S:@K;L),@S:@L;S;S;?(@MC@K;U
2                                                    )                  ,
(@MC@K;U);.searchmess<@M>;?,(@K:@C;D;S;?,C@I__mess);@K:@S;.equal<@L
>; ?$,)*;U*)>
```

Suppression de toutes les définitions de messages n'apparaissant plus dans la liste des ports de modules. Cette fonction est exécutée avant de sortir du mode mise à jour.

```
.def<.searchmess<@M>,(U*;S2SSS;(@L:@K;S2;SS;(@P:@K;S2;.eqval<@M>;
?$2,@K:@P;R;?,$1)*;@K:@L;R;?,$-1)*;U*)>
```

Toutes ces procédures sont appelées lors de la traduction d'une structure graphique en programme LSD mais ne sont pas accessibles directement à l'utilisateur.

BIBLIOGRAPHIE

[LON 87] LONCHAMP J.
Conception des applications informatiques réparties en commande de procédés industriels : une démarche, des outils.
Thèse d'Etat, NANCY I, 5/87.

[MEL 85] MELESE, MIGOT, VEROVE
The MENTOR-V5 documentation.
INRIA, RTO43, 1/85.

NOM DE L'ETUDIANT : LALLIER Martial

NATURE DE LA THESE : Doctorat 3ème cycle en Informatique

VU, APPROUVE ET PERMIS D'IMPRIMER

NANCY, le 14 OCT. 1988 n° 1734

LE PRESIDENT DE L'UNIVERSITE DE NANCY I

UNIVERSITE NANCY I LE PRESIDENT

R. MAINARD

Résumé

La conception des applications informatiques réparties soulève, entre autres problèmes, celui de l'édition et du contrôle des différents niveaux de descriptions. Proposer un outil qui assiste l'utilisateur dans ces tâches fastidieuses de manipulation des spécifications constitue l'objectif principal de ce travail.

Après un bref survol du contexte général de la conception des applications réparties en commande de procédés et du projet global auquel se rattache cette proposition, la définition et la mise en oeuvre d'un environnement d'édition et de contrôle évolué, graphique et syntaxique sont abordés. Un exemple complet illustre ses modalités d'utilisation.

Mots clés

Conception, environnement, outils, structuration, spécification.