THESE

présentée à

Université Scientifique et Médicale de Grenoble

Institut National Polytechnique de Grenoble

pour obtenir le grade de

DOCTEUR de 3ème CYCLE

par

Martine LUCAS

CONCEPTION MODULAIRE DES SYSTEMES D'EXPLOITATION.

OUTILS POUR LA PROGRAMMATION MODULAIRE

Thèse soutenue le 27 juin 1977 devant la Commission d'Examen :

Monsieur	G. VEILLON	: Président
Monsieur	S. KRAKOWIAK	: Rapporteur
Messieurs	J. BELLINO C. BETOURNE J.P. VERJUS	: Examinateurs

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Monsieur Philippe TRAYNARD : Président

Monsieur Pierre-Jean LAURENT : Vice Président

PROFESSEURS TITULAIRES

MM.	BENOIT Jean	Radioélectricité
	BESSON Jean	Electrochimie
	BLOCH Daniel	Physique du solide
	BONNETAIN Lucien	Chimie minérale
	BONNIER Etienne	Electrochimie et électrométallurgie
	BOUDOURIS Georges	Radioélectricité
	BRISSONNEAU Pierre	Physique du solide
	BUYLE-BODIN Maurice	Electronique
	COUMES André	Radioélectricité
	DURAND Francis	Métallurgie
	FELICI Noël	Electrostatique
	FOULARD Claude	Automatique
	LESPINARD Georges	Mécanique
	MOREAU René	Mécanique
	PARIAUD Jean-Charles	Chimie-Physique
	PAUTHENET René	Physique du solide
	PERRET René	Servomécanismes
	POLOUJADOFF Michel	Electrotechnique
	SILBER Robert	Mécanique des fluides

PROFESSEUR ASSOCIE

M.	ROUXEL Roland	Automatique

PROFESSEURS SANS CHAIRE

MM.	BLIMAN Samuel	Electronique
	BOUVARD Maurice	Génie mécanique
	COHEN Joseph	Electrotechnique
	LACOUME Jean-Louis	Géophysique
	LANCIA Roland	Electronique
	ROBERT François	Analyse Numérique
	VEILLON Gérard	Informatique fondamentale et appliquée
	ZADWORNY François	Electronique

UNIVERSITE SCIENTIFIQUE
ET MEDICALE DE GRENOBLE

Monsieur Gabriel CAU : Président
Monsieur Pierre JULLIEN : Vice Président

MEMBRES DU CORPS ENSEIGNANT DE L'U.S.M.G.

PROFESSEURS TITULAIRES

MM.	AMBLARD Pierre	Clinique de dermatologie
	ARNAUD Paul	Chimie
	ARVIEU Robert	I.S.N.
	AUBERT Guy	Physique
	AYANT Yves	Physique approfondie
Mme.	BARBIER Marie-Jeanne	Electrochimie
MM.	BARBIER Jean-Claude	Physique expérimentale
	BARBIER Reynold	Géologie appliquée
	BARJON Robert	Physique nucléaire
	BARNOU Fernand	Biosynthèse de la cellulose
	BARRA Jean-René	Statistiques
	BARRIE Joseph	Clinique chirurgicale
	BEAUDOING André	Clinique de pédiatrie et puériculture
	BELORIZKY Elie	Physique
	BERNARD Alain	Mathématiques pures
Mme.	BERTRANDIAS Françoise	Mathématiques pures
MM.	BERTRANDIAS Jean-Paul	Mathématiques pures
	BEZEZ Henri	Pathologie chirurgicale
	BLAMBERT Maurice	Mathématiques pures
	BOLLIET Louis	Informatique (IUT B)
	BONNET Jean-Louis	Clinique ophtalmologique
	BONNET-EYMARD Joseph	Clinique gastro-entérologique
Mme.	BONNIER Marie-Jeanne	Chimie générale
MM.	BOUCHERLE André	Chimie et toxicologie
	BOUCHEZ Robert	Physique nucléaire
	BOUSSARD Jean-Claude	Mathématiques appliquées
	BOUTET DE MONVEL Louis	Mathématiques pures
	BRAVARD Yves	Géographie
	CABANEL Guy	Clinique rhumatologique et hydrologique
	CALAS François	Anatomie
	CARLIER Georges	Biologie végétale
	CARRAZ Gilbert	Biologie animale et pharmocodynamie
	CAU Gabriel	Médecine légale et toxicologie
	CAUQUIS Georges	Chimie organique
	CHABAUTY Claude	Mathématiques pures
	CHARACHON Robert	Clinique oto-rhino-laryngologique
	CHATEAU Robert	Clinique de neurologie
	CHIBON Pierre	Biologie animale
	COEUR André	Pharmacie chimique et chimie analytique
	CONTAMTIN Robert	Clinique gynécologique
	COUDERC Pierre	Anatomie pathologique

Mme.	DEBELMAS Anne-Marie	Matière médicale
MM.	DEBELMAS Jacques	Géologie générale
	DEGRANGE Charles	Zoologie
	DELORMAS Pierre	Pneumophtisiologie
	DEPORTES Charles	Chimie minérale
	DESRE Pierre	Métallurgie
	DESSAUX Georges	Physiologie animale
	DODU Jacques	Mécanique appliquée (IUT I)
	DOLIQUE Jean-Michel	Physique des plasmas
	DREYFUS Bernard	Thermodynamique
	DUCROS Pierre	Cristallographie
	GAGNAIRE Didier	Chimie physique
	GALVANI Octave	Mathématiques pures
	GASTINEL Noël	Analyse numérique
	GAVEND Michel	Pharmacologie
	GEINDRE Michel	Electroradiologie
	GERBER Robert	Mathématiques pures
	GERMAIN Jean-Pierre	Mécanique
	GIRAUD Pierre	Géologie
	JANIN Bernard	Géographie
	KAHANE André	Physique générale
	KOSZUL Jean-Louis	Mathématiques pures
	KLEIN Joseph	Mathématiques pures
	KRAVTCHENKO Julien	Mécanique
	KUNTZMANN Jean	Mathématiques appliquées
	LACAZE Albert	Thermodynamique
	LACHARME Jean	Biologie végétale
Mme.	LAJZEROWICZ Janine	Physique
MM.	LAJZEROWICZ Joseph	Physique
	LATREILLE René	Chirurgie générale
	LATURAZE Jean	Biochimie pharmaceutique
	LAURENT Pierre-Jean	Mathématiques Appliquées
	LEDRU Jean	Clinique médicale B
	LE ROY Philippe	Mécanique (IUT I)
	LLIBOUTRY Louis	Géophysique
	LOISEAUX Pierre	Sciences nucléaires
	LONGEQUEUE Jean-Pierre	Physique nucléaire
	LOUP Jean	Géographie
Melle	LUTZ Elisabeth	Mathématiques pures
MM.	MALINAS Yves	Clinique obstétricale
	MARTIN-NOEL Pierre	Clinique cardiologique
	MAZARE Yves	Clinique médicale A
	MICHEL Robert	Minéralogie et pétrographie
	MICOUD Max	Clinique maladies infectieuses
	MOURIQUAND Claude	Histologie
	MOUSSA André	Chimie nucléaire
	NOZIERES Philippe	Spectrométrie physique
	OZENDA Paul	Botanique
	PAYAN Jean-Jacques	Mathématiques pures
	PEBAY-PEYROULA Jean-Claude	Physique
	PERRET Jean	Semeiologie médicale (Neurologie)
	RASSAT André	Chimie systématique
	RENARD Michel	Thermodynamique
	REVOL Michel	Urologie
	RINALDI Renaud	Physique
	DE ROUGEMONT Jacques	Neuro-chirurgie
	SEIGNEURIN Raymond	Microbiologie et Hygiène
	SENGEL Philippe	Zoologie
	SIBILLE Robert	Construction mécanique (IUT I)

MM.	SOUTIF Michel	Physique générale
	TANCHE Maurice	Physiologie
	TRAYNARD Philippe	Chimie générale
	VAILLANT François	Zoologie
	VALENTIN Jacques	Physique nucléaire
	VAUQUOIS Bernard	Calcul électronique
Mme.	VERAIN Alice	Pharmacie galénique
MM.	VERAIN André	Physique
	VEYRET Paul	Géographie
	VIGNAIS Pierre	Biochimie médicale

PROFESSEURS ASSOCIES

MM.	CRABBE Pierre	CERMO
	DEMBICKI Eugéniuz	Mécanique
	JOHNSON Thomas	Mathématiques appliquées
	PENNEY Thomas	Physique

PROFESSEURS SANS CHAIRE

Melle	AGNIUS-DELORD Claudine	Physique pharmaceutique
	ALARY Josette	Chimie analytique
MM.	AMBROISE-THOMAS Pierre	Parasitologie
	ARMAND Gilbert	Géographie
	BENZAKEN Claude	Mathématiques appliquées
	BIAREZ Jean-Pierre	Mécanique
	BILLET Jean	Géographie
	BOUCHET Yves	Anatomie
	BRUGEL Lucien	Energétique (IUT I)
	BUISSON René	Physique (IUT I)
	BUTEL Jean	Orthopédie
	COHEN ADDAD Pierre	Spectrométrie physique
	COLOMB Maurice	Biochimie
	CONTE René	Physique (IUT I)
	DELOBEL Claude	M.I.A.G.
	DEPASSEL Roger	Mécanique des fluides
	FONTAINE Jean-Marc	Mathématiques pures
	GAUTRON René	Chimie
	GIDON Paul	Géologie et minéralogie
	GLENAT René	Chimie organique
	GROULADE Joseph	Biologie médicale
	HACQUES Gérard	Calcul numérique
	HOLLARD Daniel	Hématologie
	HUGONOT Robert	Hygiène et médecine préventive
	IDELMAN Simon	Physiologie animale
	JOLY Jean-René	Mathématiques pures
	JULLIEN Pierre	Mathématiques appliquées
Mme.	KAHANE Josette	Physique
MM.	KRAKOWIACK Sacha	Mathématiques appliquées
	KUHN Gérard	Physique (IUT I)
	LUU DUC Cuong	Chimie organique
	MAYNARD Roger	Physique du solide
Mme.	MINIER Colette	Physique (IUT I)
MM.	PELMONT Jean	Biochimie
	PERRIAUX Jean-Jacques	Géologie et minéralogie
	PFISTER Jean-Claude	Physique du solide
Melle	PIERY Yvette	Physiologie animale

MM.	RAYNAUD Hervé	M.I.A.G.
	REBECQ Jacques	Biologie (CUS)
	REYMOND Jean-Charles	Chirurgie générale
	RICHARD Lucien	Biologie végétale
Mme.	RINAUDO Marguerite	Chimie macromoléculaire
MM.	ROBERT André	Chimie papetière
	SARRAZIN Roger	Anatomie et chirurgie
	SARROT-REYNAULD Jean	Géologie
	SIROT Louis	Chirurgie générale
Mme.	SOUTIF Jeanne	Physique générale
MM.	STIEGLITZ Paul	Anesthésiologie
	VIALON Pierre	Géologie
	VAN CUTSEM Bernard	Mathématiques appliquées

MAITRES DE CONFERENCES ET MAITRES DE CONFERENCES AGREGES

MM.	ARMAND Yves	Chimie (IUT I)
	BACHELOT Yvan	Endocrinologie
	BARGE Michel	Neuro-chirurgie
	BEGUIN Claude	Chimie organique
Mme	BERIEL Hélène	Pharmacodynamie
MM.	BOST Michel	Pédiatrie
	BOUCHARLAT Jacques	Psychiatrie adultes
Mme.	BOUCHE Liane	Mathématiques (CUS)
MM.	BRODEAU François	Mathématiques (IUT B) (Personne étrangère habilitée à être directeur de thèse)
	CHAMBAZ Edmond	Biochimie médicale
	CHAMPETIER Jean	Anatomie et organogénèse
	CHARDON Michel	Géographie
	CHERADAME Hervé	Chimie papetière
	CHIAVERINA Jean	Biologie appliquée (EFP)
	CONTAMIN Charles	Chirurgie thoracique et cardio-vasculaire
	CORDONNIER Daniel	Néphrologie
	COULOMB Max	Radiologie
	CROUZET Guy	Radiologie
	CYROT Michel	Physique du solide
	DENIS Bernard	Cardiologie
	DOUCE Roland	Physiologie végétale
	DUSSAUD René	Mathématiques (CUS)
Mme.	ETERRADOSSI Jacqueline	Physiologie
MM.	FAURE Jacques	Médecine légale
	FAURE Gilbert	Urologie
	GAUTIER Robert	Chirurgie générale
	GIDON Maurice	Géologie
	GROS Yves	Physique (IUT I)
	GUIGNIER Michel	Thérapeutique
	GUITTON Jacques	Chimie
	HICTER Pierre	Chimie
	JALBERT Pierre	Histologie
	JULIEN-LAVILLAVROY Claude	O.R.L.
	KOLODIE Lucien	Hématologie
	LE NOC Pierre	Bactériologie-virologie
	MACHE Régis	Physiologie végétale
	MAGNIN Robert	Hygiène et médecine préventive
	MALLION Jean-Michel	Médecine du travail
	MARECHAL Jean	Mécanique (IUT I)
	MARTIN-BOUYER Michel	Chimie (CUS)
	MICHOULIER Jean	Physique (IUT I)

MM.	NEGRE Robert	Mécanique (IUT I)
	NEMOZ Alain	Thermodynamique
	NOUGARET Marcel	Automatique (IUT I)
	PARAMELLE Bernard	Pneumologie
	PECCOUD François	Analyse (IUT B) (Personnalité étrangère habilité à être directeur de thèse)
	PEFFEN René	Métallurgie (IUT I)
	PERRIER Guy	Géophysique-Glaciologie
	PHELIP Xavier	Rhumatologie
	RACHAIL Michel	Médecine interne
	RACINET Claude	Gynécologie et obstétrique
	RAMBAUD André	Hygiène et hydrologie (Pharmacie)
	RAMBAUD Pierre	Pédiatrie
	RAPHAEL Bernard	Stomatologie
Mme.	RENAUDET Jacqueline	Bactériologie (Pharmacie)
MM.	ROBERT Jean-Bernard	Chimie physique
	Romier Guy	Mathématiques (IUT B) (Personnalité étrangère habilité à être directeur de thèse)
	SCHAERER René	Cancérologie
	SHOM Jean-Claude	Chimie générale
	STOEBNER Pierre	Anatomie pathologie
	VROUSOS Constantin	Radiologie

MAITRES DE CONFERENCES ASSOCIES

MM.	DEVINE Roderick	Spectro physique
	HODGES Christopher	Transition de phases

Fait à SAINT MARTIN D'HERES, NOVEMBRE 1976.

MAITRES DE CONFERENCES

MM.	ANCEAU François	Mathématiques appliquées
	CHARTIER Germain	Electronique
	GUYOT Pierre	Chimie minérale
	IVANES Marcel	Electrotechnique
	JOUBERT Jean-Claude	Physique du solide
	MORET Roger	Electrotechnique nucléaire
	PIERRARD Jean-Marie	Mécanique
	SABONNADIERE Jean-Claude	Informatique fondamentale et appliquée
Mme.	SAUCIER Gabrièle	Informatique fondamentale et appliquée

MAITRE DE CONFERENCES ASSOCIE

M.	LANDAU Ioan	Automatique

CHERCHEURS DU C.N.R.S. (Directeur et Maîtres de Recherche)

MM.	FRUCHART Robert	Directeur de Recherche
	ANSARA Ibrahim	Maître de Recherche
	CARRE René	Maître de Recherche
	DRIOLE Jean	Maître de Recherche
	MATHIEU Jean-Claude	Maître de Recherche
	MUNIER Jacques	Maître de Recherche

Que cette page exprime mes remerciements.

A Monsieur G. VEILLON, pour l'honneur qu'il me fait en acceptant de présider ce jury,

A Messieurs J. BELLINO, C. BETOURNE et J.P. VERJUS, pour avoir bien voulu faire partie du jury et pour l'intérêt qu'ils ont porté au projet pendant toute sa réalisation,

A Monsieur S. KRAKOWIAK, directeur de recherche, qui a su faire de ce projet un véritable travail d'équipe.

Je tiens à remercier chacun des membres de cette équipe et particulièrement Messieurs J. MOSSIERE et J. MONTUELLE, sans le soutien et la compréhension desquels cette thèse ne serait pas écrite.

J'apprécie aussi la solidarité des secrétaires du laboratoire, qui ont accepté de se répartir la frappe de cette thèse en plus de leur travail habituel.

Que tous ceux que j'ai rencontrés pendant ce temps de recherche trouvent ici le témoignage de ma reconnaissance pour ce qu'ils m'ont apporté.

TABLE DES MATIERES

I - INTRODUCTION

I.1. Motivation de la programmation modulaire

La décomposition en parties est une méthode couramment utilisée pour faciliter la compréhension, la conception et la construction d'ensembles complexes. Il est reconnu qu'une grande part des difficultés de la programmation provient de notre incapacité à maîtriser la complexité des objets construits ; aussi, la décomposition des programmes en parties est-elle couramment pratiquée. On parle alors souvent de modules et de programmation modulaire. Avant de donner un contenu plus précis à ces termes, rappelons les principaux avantages attendus de la programmation modulaire :

- Meilleure compréhension du fonctionnement du programme

Chaque module remplit une fonction déterminée, dont la définition doit être indépendante de son mode de réalisation; le problème de la réalisation d'une fonction particulière par un module peut alors être disjoint du problème de la connexion du module au reste du système. La structure de l'ensemble est ainsi rendue plus apparente. La compréhension du fonctionnement d'un module donné ne doit pas nécessiter une connaissance détaillée du reste du système; enfin, la simple réduction de la taille, qui résulte de la division, tend à réduire la complexité de chaque partie.

- Facilité de production

Une fois la décomposition définie, les diverses parties du système peuvent être réalisées en parallèle par des équipes distinctes ayant peu d'interactions. Les communications d'information sont restreintes au strict nécessaire; en particulier, les informations sur le fonctionnement interne d'un module peuvent rester inconnues des utilisateurs de ce module et ne doivent pas pouvoir être utilisées par eux.

- Facilité d'entretien

Tout programme est destiné à être modifié, soit parce que ses spécifications évoluent, soit parce qu'on y a détecté des erreurs, soit enfin parce qu'on souhaite améliorer ses performances. La programmation modulaire doit permettre

- de localiser aisément le module où une erreur s'est produite, et de corriger cette erreur sans avoir à connaître l'ensemble du système.
- de remplacer un module par un autre module remplissant la même fonction sans avoir à modifier le reste du système (selon le principe de l' "échange standard" utilisé dans d'autres domaines); plus généralement toute modification peut être apportée à un module à condition de respecter ses connexions avec le reste du système.
- de réutiliser un module existant pour assurer une fonction donnée, sans avoir à écrire un nouveau programme (nécessité d'avoir un stock de produits suffisamment généraux).

De nombreux travaux récents essaient de rendre systématique la décomposition. Sans vouloir en donner une liste exhaustive, signalons quelques unes des étapes, ou grandes lignes, de cette recherche en les illustrant par quelques exemples. Après des travaux de "formalisation" des notions de module (Parnas 72) et de types (Morris 73), on tente actuellement d'inclure ces notions dans des langages de programmation ou dans des systèmes d'aide à l'écriture des programmes. Schématiquement, on peut classer ces travaux en deux groupes, selon que l'accent est mis sur la notion de type ou sur l'assemblage des modules. Dans la première catégorie, citon CLU (Liskov 72, 74a, 74b) et ALPHARD (Wulf 76) : à partir des classes de SIMULA (Dahl 72), on restreint les opérations sur des données à celles définies dans le type et on interdit l'accès à la représentation. En ce qui concerne l'assemblage des modules, l'idée de distinguer écriture des modules et connexion a été introduite dans (De Remer 75). Comme extension de la notion de bloc et de procédure, apparaît dans plusieurs langages celle de module. C'est le cas en particulier de MODULA (Wirth 77) et EUCLID (Lampson 77). Le système CIVA (Derniame 75) est un système d'aide à la programmation orienté vers des applications de gestion. Quelques langages sont destinés plus spécifiquement à l'écriture de systèmes d'exploitation, par exemple Concurrent Pascal (Brinch Hansen 74), LIS (Ichbiah 74) et LEST (Bétourné 76). Ces deux derniers permettent en particulier de définir finement l'implantation des structures de données.

I.2. Présentation du projet

C'est autour de cette idée de découpage, d'éléments de programmes pouvant s'assembler comme un jeu de construction qu'a débuté le projet. Le but de

l'équipe n'était pas de rechercher de nouvelles structures, mais plutôt d'essayer, en faisant un choix parmi les structures proposées, de construire un outil simple et cohérent d'aide à la production de systèmes. Le terme de module étant employé dans des sens différents, selon les auteurs, nous avons dans un premier temps essayé de dégager les propriétés principales de "nos" modules. En particulier, la différence entre bloc de construction et modèle d'objet structuré est apparue peu à peu. L'aspect "modèle d'objets structurés" faisant déjà l'objet d'études poussées (Algol 68, Van Winjgaarden 69 par exemple), nous avons plutôt poursuivi l'idée de bloc de construction, essayant de développer les propriétés des modules technologiques. A savoir :

a) Un module forme un tout logique, dont on attend certaines fonctions précises. Chacune de ces fonctions est une "patte" qui sort du module et peut être connectée à une autre patte.

b) La séparation entre la spécification et la réalisation. En effet, lors de la réalisation de la boite, on ne connaît initialement que les différentes fonctions que l'on doit fournir à l'extérieur. Si l'on considère deux modules devant réaliser les mêmes fonctions, mais fabriqués différemment, la différence ne doit pas apparaître à l'utilisateur du module.

c) L'aspect "standard" des modules, qui découle du fait précédent. On doit pouvoir remplacer facilement un module par un autre de même spécification.

d) On doit pouvoir réutiliser ces objets "standards" pour la construction de plusieurs systèmes. D'où la nécessité de les conserver sous une forme la plus générale possible, c'est-à-dire sous forme symbolique et possédant un certain nombre de paramètres dont la valeur n'est pas fixée.

Définir les propriétés d'un module n'était intéressant que dans la mesure où un ensemble d'outils adaptés à l'écriture et à l'utilisation des modules était construit. C'est ainsi que nous avons été conduits à spécifier SESAME, ensemble d'outils incluant le module comme notion de base. Les choix faits dans cette spécification visaient à séparer au maximum les problèmes tout en définissant des outils minimaux car l'équipe n'était pas suffisamment nombreuse pour envisager un gros projet. Ont donc été éliminés des sujets comme tests de cohérence entre spécification et réalisation, problèmes de protection autre que la

structure modulaire dont l'étude n'a pas un rapport fondamental avec les modules. Nous avons conservé dans ce projet la nécessité :

- d'un langage d'écriture de modules,
- d'un langage de connexion entre modules,
- d'un ensemble de programmes permettant de conserver et de réutiliser des modules : le bibliothécaire.

La spécification de chacune de ces parties a ensuite été faite, en même temps qu'apparaissait la nécessité de spécifier parallèlement un petit système modèle, PROSPER, dont le rôle était :

- d'étudier les propriétés de la décomposition modulaire,
- de rendre plus clairs les besoins au niveau langage et connexion,
- de servir de test constant des outils réalisés.

La conception de PROSPER s'est donc faite parallèlement à une étude de la méthodologie de conception résultant de l'existence de l'outil que constituent les modules.

Le choix du découpage du programme en modules est un problème important. Il faut en effet trouver des modules qui soient le plus possible logiquement indépendants, afin de minimiser les connexions entre modules. Ce problème, ainsi que celui de la réalisation des modules est abordé dans (Montuelle 77).

Enfin, la réalisation de PROSPER permettait de souligner les caractéristiques propres au système producteur (SESAME) et au système produit (PROSPER).

Le choix fait de réaliser SESAME sur une grosse machine, mais de destiner les systèmes produits à être exécutés sur une petite machine accentue les particularités de chacun des systèmes producteur et produit. SESAME et les systèmes produits s'exécutent sur des machines différentes, nous avons été conduits à construire un compilateur plutôt qu'un interpréteur. Pour la même raison, chercher à faire le plus possible de tests de cohérence au niveau de SESAME, c'était chercher à faire le plus possible de tests statiquement.

Par la suite, une des utilisations possibles de SESAME et PROSPER serait pour l'enseignement des systèmes d'exploitation. Alors qu'actuellement, il

n'y a pas de milieu entre la machine nue et un gros système d'exploitation, on pourrait faire écrire aux étudiants de petits morceaux de système, étudier par exemple différents algorithmes pour la réalisation d'un allocateur, et ce en ne prenant connaissance que de la partie qui les intéresse.

On peut résumer l'évolution de la conception et de la réalisation du projet en suivant les publications de l'équipe. Tout d'abord les problèmes de la programmation modulaire sont évoqués (Krakowiak 74). Après la mise au point d'une définition commune sur les notions de types, modules, ensemble d'objets ou de valeurs (Mossière 74), une première spécification des outils, c'est-à-dire des langages d'écriture et de connexion, du bibliothécaire apparaît (Krakowiak 75, Mossière 75, Krakowiak 76). Faisant suite à une étude plus générale sur la synchronisation dans les langages de haut niveau (Lucas 74), une étude est faite pour caractériser des primitives de synchronisation adaptées aux modules (Lucas 75). La spécification des outils devient de plus en plus précise (Cheval 76a, Cheval 76b), et une première version du bibliothécaire est décrite (Cristian 76). Ensuite, apparaissent des réflexions sur la méthode de conception propre à l'utilisation des modules. Ces réflexions sont faites à partir de réalisations concrètes : le bibliothécaire (Cristian 77) et PROSPER (Cheval 77, Montuelle 77).

Les principaux choix et difficultés de réalisation du projet sont décrits dans une partie complémentaire (Lucas 77).

Le schéma situé à la fin du chapitre donne une vue d'ensemble du projet tel qu'il a été conduit (l'ordre chronologique apparaît de gauche à droite). Les trois parties en cours de réalisation sont donc celles de droite : compilateur générant du code pour T1600, connecteur, et système modèle.

I.3. Présentation de la thèse dans le cadre du projet

Deux études ont donc été faites en parallèle : celle du système producteur SESAME, et celle du système produit : PROSPER. L'étude en vue de spécifier un petit système produit, faisant apparaître une méthode de décomposition à l'aide des modules, peut être trouvée dans (Montuelle 77). Nous présentons dans cette partie de la thèse les principaux choix de conception de SESAME, tandis qu'une partie complémentaire à chacune de ces deux thèses (Lucas 77) décrit les principaux problèmes soulevés lors de la réalisation du projet, ainsi que les choix de réalisation.

Après le chapitre I d'introduction, nous étudions dans le chapitre II les deux notions principales que l'on trouve sous le nom de module : les modèles d'objets structurés, que nous appelons générateurs de types, et les blocs de constructions auxquels nous conservons le nom de module.

Nous étudions plus particulièrement ces derniers en insistant sur la différence et l'indépendance entre la spécification et la réalisation d'un module, différence illustrée par des exemples. Enfin, avant d'indiquer quel est le schéma de module que nous avons adopté, et les objets nécessaires à sa traduction, nous présentons un exemple de langage qui possède à la fois des modules et des générateurs de type.

Dans le chapitre III nous présentons différentes possibilités pour inclure notre schéma de module dans un ensemble d'outils. Ces solutions tiennent compte du fait que les modules doivent pouvoir être écrits indépendamment, connectés entre eux, et conservés en vue d'être réutilisés. Elles comportent donc toutes un langage dans lequel le module est une notion de base. Afin de pouvoir réutiliser les modules, différents modes de paramétrage sont présentés :

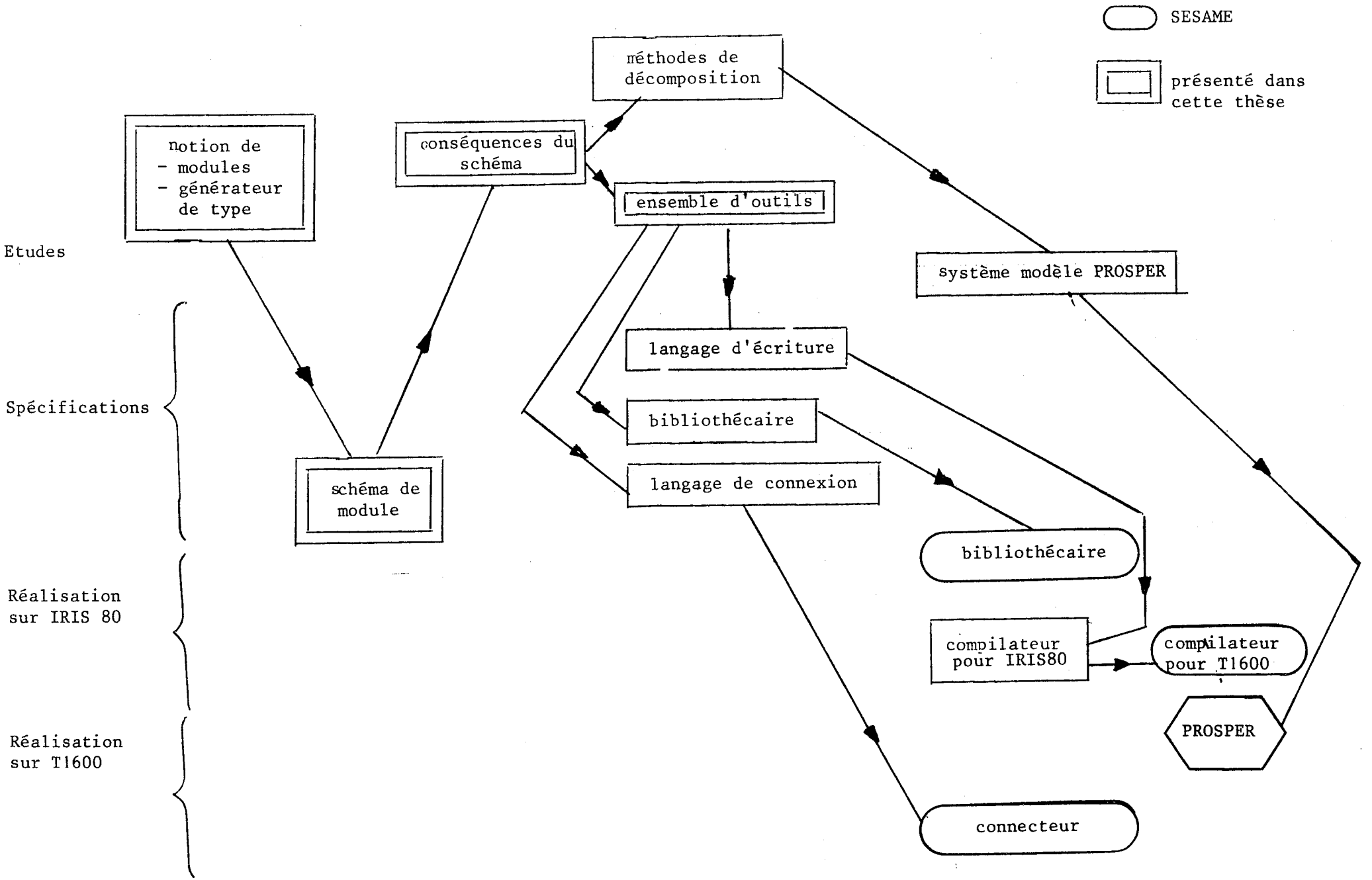
a) paramétrage permettant d'ajuster la structure de l'objet représenté par le module : ce sont les métavariables, dont nous discutons la nature et les moments de liaison,

b) paramétrage permettant de retarder le moment de liaison de certaines procédures. Il s'agit en fait d'un mécanisme d'indirection inspiré de celui de la désignation des fichiers.

Nous présentons dans ce même chapitre quelques caractéristiques du langage d'écriture, et faisons apparaître la commodité, sinon la nécessité de partager des types entre plusieurs modules. Plusieurs solutions sont proposées pour réaliser ce partage. Ensuite, nous présentons différents choix possibles en ce qui concerne la connexion, allant du choix d'un langage d'écriture intégrant le système de connexion à un choix permettant de séparer phase de traduction et phase de connexion. Nous tirons les conséquences de ce dernier choix et développons les caractéristiques d'un langage de connexion. Le bibliothécaire joue un rôle important pour la réutilisation des modules. Aussi en donnons-nous des caractéristiques. Ce chapitre se termine sur un rappel des choix faits et une vue d'ensemble des outils que nous avons réalisé, c'est-à-dire de SESAME.

Dans le chapitre IV, nous étudions quelques caractéristiques de primitives de synchronisation adaptées aux modules. La relation entre processus et module est tout d'abord définie. On étudie ensuite le rapport entre synchronisation et spécification, synchronisation et réalisation; puis apparaît le fait que la synchronisation des processus ne peut se faire sans introduire de nouvelles variables dans le module. Ce chapitre se termine avec la présentation et la critique de deux primitives de synchronisation adaptées aux modules : les moniteurs et les expressions de chemin.

Dans le chapitre V, qui sert de conclusion, nous récapitulons les principales idées présentées et indiquons quelques voies de développement possibles du projet.



VUE D'ENSEMBLE DU PROJET

II - NOTION DE MODULE

II.1 - Modules et générateurs de types

Les avantages énumérés dans l'introduction ne peuvent être obtenus que si l'on peut isoler dans le programme des fonctions dont l'effet puisse être considéré indépendamment de leur mode de réalisation. C'est en particulier un des aspects de la notion de procédure dans de nombreux langages de programmation. Mais l'idée de décrire un objet indépendamment de son mode de réalisation peut être développée dans deux directions:

- représentation d'un objet par un certain nombre de ses propriétés caractérisant ses relations avec le monde extérieur ;

- modèle pour la construction d'un ensemble d'objets appartenant à une même classe, définie par des propriétés communes.

La notion de module peut donc trouver deux applications: en tant que bloc de construction, connectable à d'autres blocs pour construire des programmes, et en tant que modèle d'une classe d'objets dont on peut construire des représentants. Nous parlerons désormais respectivement de modules et de générateurs de type.

La représentation des deux notions, module et générateur de type, consiste en un ensemble de variables, représentant un objet structuré, et des opérations que l'on peut effectuer sur cet objet, la seule façon de modifier la valeur des variables est de passer par les opérations définies.

La différence essentielle entre ces deux notions réside dans leur moment de mise en oeuvre. Alors que les modules sont constitués de variables et d'opérations dont la structure se retrouve au moment de l'exécution, les générateurs de types ne sont que des indications données au compilateur.

II.2 - Générateurs de types

La définition d'un générateur de type peut être comparée à celle d'un type dans les langages évolués (PASCAL, ALGOL 68...). Elle décrit donc la

structure d'un objet complexe. Mais alors que dans les définitions des types habituels chaque élément de l'objet structuré est accessible de façon compatible avec son type, ici les seuls accès permis sont définis de façon explicite par les opérations. Cette propriété constitue un net avantage par rapport aux classes de SIMULA (Dahl 66) où, malgré le regroupement de l'objet et des procédures d'accès, tous les accès élémentaires à la structure sont permis.

Etant donné un type, ou modèle d'objet structuré, deux options sont possibles pour la création d'un objet de ce type:

- la création se fait de façon statique, c'est-à-dire lors de la déclaration d'un tel objet ;

- ou bien on possède un opérateur de création (le *new* de SIMULA), la création est alors dynamique et permet de définir une procédure d'initialisation différente pour chaque modèle.

Le contrôle des appels des procédures définies sur l'objet peut être effectué à la compilation.

Nous pouvons illustrer ces propriétés par un exemple tiré de (Liskov 74a). On veut définir le générateur de type(*cluster*) "pile d'éléments". La définition se décompose en trois parties:

1/ description de l'interface, c'est-à-dire de ce qui est accessible de l'extérieur du cluster. Il s'agit du nom du type (*stack*)et des opérations possibles sur ce type (*create, push, pop, top, erase top, empty*).

2/ description de la représentation de l'objet. Ici un entier, un nom de type (celui passé en paramètre au cluster: *type*, et un tableau d'éléments de ce type).

3/ description de la réalisation des opérations. La représentation de l'objet n'est accessible qu'à l'intérieur de ces opérations.

```
stack: cluster (element_type : type)
     is  create, push, pop, top, erasetop, empty ;

  rep(type_param: type) = (tp: integer;
                           e_type: type ;
                           stk: array[1..]
                                 of type_param ;
  create: operation ;
       s:rep(element type) ;
       s.tp := 0;
       s.tp_type := element_type ;
      return s ;
      end

  push: operation(s: rep, v:s.e_type);

      s.tp := s.tp+1 ;
      s.stk [s.tp] := v;
      return ;
      end

  pop: operation(s: rep) returns s.e_type ;

      if s.tp = 0 then error ;
      s.tp := s.tp-1 ;
      return s.stk[s.tp+1] ;
      end

  top: operation(s:rep) returns s.e_type;

      if s.tp = 0 then error ;
      return s.stk[s.tp] ;
      end

  erasetop: operation (s:rep);

      if s.tp = 0 then error ;
      s.tp := s.tp-1 ;
      return ;
      end

  empty: operation(s:rep) returns boolean ;

      return s.tp = 0 ;
      end

  end stack
```

On obtient ainsi un modèle de pile, objet structuré dont la création se fait dynamiquement à l'aide de l'opération *create*.

Lors de l'exécution de: *S : stack (integer)$create*
un objet ayant la structure décrite dans le cluster est créé et initialisé.

Le compilateur contrôle que les opérations effectuées sur les objets ainsi créés sont licites. Les opérations sont attachées au type et non à chaque objet.

II.3 - Définition d'un module

Les modules se présentent comme des blocs de construction de programmes. Ils demeurent au moment de l'exécution des blocs connectés entre eux.

Il est donc possible d'envisager des compilations séparées. La solution la plus simple consiste alors à ne faire que des créations statiques de modules (compilation = création), car elles permettent de faire des connexions statiques. Nous allons décrire de façon plus précise ce qu'est un module.

Un module est un programme dont seuls certains objets (dits externes) sont accessibles de l'extérieur. Ces objets externes peuvent à priori etre exécutables (procédures) ou non (données). Partant de l'idée qu'une donnée ne doit être définie (pour des raisons de sécurité) qu'en même temps que les opérations autorisées sur elle, nous restreindrons, par convention, les objets externes d'un module à être des procédures (pouvant être des fonctions, c'est-à-dire fournir une valeur appartenant à un type spécifié). Cette convention revient à interdire la modification d'une donnée d'un module autrement que par l'appel d'une procédure externe, la lecture étant également assurée par un appel de fonction. Une convention équivalente (Parnas 72) consiste à admettre des données externes accessibles en lecture seule depuis l'extérieur du module. Les paramètres d'une procédure externe ne peuvent être passés que par valeur.

Un module comprend donc:

- un ensemble de procédures externes, accessibles depuis l'extérieur du module,
- un ensemble de procédures } internes, inacessibles directement
- un ensemble de données } depuis l'extérieur du module.

La définition des procédures externes (nom et éventuellement type, nombre et type de paramètres, description de l'effet) constitue l'interface du module. Le terme de spécification est également employé. Lorsque nous parlerons de "spécification(s)", sans préciser, ce terme sera synonyme d'"interface" ; le terme de "spécification de réalisation" s'applique à la description d'une réalisation particulière d'un module.

Avant la première utilisation d'un module, ses données internes doivent être dans un état initial bien défini. La définition de cet état doit faire partie des spécifications du module, soit directement soit comme résultat de l'exécution d'une procédure d'initialisation spécifiée.

A tout instant (en dehors de l'exécution d'une procédure externe), les variables internes du module ont des valeurs dont l'ensemble définit l'état du module à cet instant. Une autre façon de définir l'état, qui ne fait pas intervenir les variables internes, consiste à donner la suite d'appels de procédures (avec les valeurs des paramètres) entre l'instant initial (ou tout instant où l'état est déjà défini) et l'instant actuel. L'état d'un module résume donc, en ce qui concerne son comportement futur, toute l'histoire antérieure du module.

II.4 - Spécifications d'un module

L'effet d'une procédure externe peut s'exprimer en fonction de sès paramètres et de l'état du module au moment de l'appel. Toutefois, on ne peut utiliser les variables internes pour la description de l'état si l'on souhaite que la spécification soit indépendante du mode de réalisation. Nous introduirons donc des variables d'état, distinctes des variables internes, en vue de caractériser l'état du module. La spécification d'un module est donc une description de l'effet des procédures externes au moyen des variables d'état.

L'effet d'une procédure peut être défini de deux façons:

- de façon implicite, par des assertions portant sur les valeurs des variables d'état et des paramètres de la procédure avant et après son appel. Plus généralement, ces assertions peuvent s'appliquer aux états avant et après exécution d'une séquence d'appels de procédures externes du même module ;

- de façon explicite, par un algorithme exprimé au moyen des mêmes variables et procédures.

Pour une réalisation donnée du module, on devra exprimer ces mêmes procédures externes au moyen de variables internes. Etablir la validité d'une réalisation (la validité de la spécification étant postulée), c'est prouver l'équivalence des deux représentations (spécification au moyen de variables d'état, réalisation au moyen de variables internes) des procédures externes. C'est en général un problème difficile car les deux représentations peuvent être très différentes. On demande à une spécification d'être claire et à une réalisation d'être efficace.

Une réalisation particulière d'un module spécifié par un algorithme à l'aide de variables d'état peut être obtenue en représentant chaque variable d'état par une variable interne de même type. Réciproquement, il est possible de spécifier un module par une réalisation particulière. Dans tous les cas, les variables d'état n'ont pour l'utilisateur d'un module qu'une existence abstraite.

II.5 - Exemples de spécification et réalisation de modules

Les problèmes de spécification et de concordance entre spécification et réalisation n'ont pas été abordés dans notre projet.

Mais pour clarifier les notions qui viennent d'être introduites elles vont être illustrées par des exemples.

Exemple 1 : racine carrée

Ce module comprend une procédure externe, de type fonction à valeur réelle

à un paramètre réel non négatif. L'état du module est indépendant des appels antérieurs ; il n'y a donc pas lieu d'introduire de variables d'état.

ext function racine (x:real) : real ;

: if x<0 then erreur else
{abs(racine(x)-√x)<10⁻⁶}

La spécification ne fait intervenir que le paramètre et le résultat de la procédure externe.

Exemple 2 : table de symboles

Nous définissons une table de symboles comme un ensemble de doublets *(clé, inf)* où *clé* est une variable de type *T1* et *inf* une variable de type *T2*. Quatre procédures permettent de manipuler cette structure de données, supposée initialement vide.

entrer(c : T1, p : T2) introduit le doublet (c,p) dans la table à condition qu'elle ne soit pas pleine et qu'un doublet de clé c n'y figure pas déjà.

resultat(c : T1) : T2 si un doublet de clé c figure dans la table, délivre la valeur de l'élément *inf* correspondant.

supprimer(c : T1) si un doublet de clé c figure dans la table, supprime ce doublet de la table.

modifier(c : T1, q:T2) si un doublet de clé c figure dans la table, remplace la partie *inf* de ce doublet par q .

Cette description peut être rendue plus formelle en utilisant le vocabulaire des ensembles.

```
        type doublet : (clé: T1, inf : T2) ;
        var x : doublet ;
        set S of doublet ;
initialisation:S := ∅ ;
ext procedure entrer (c: T1; I : T2) ; clé of x := c ; inf of x := I ;
                                        if card(S) = nmax then erreur1
                                        else if ∃z : (z∈S)∧(clé of z =clé of x)
                                        then erreur2 else
                                                S := S∪{x}

ext function resultat(c:T1) : T2 ; if ∃z : (z∈S)∧(clé of z = c) then
                                    resultat := inf of z else erreur3 ;

ext procedure supprimer (c:T1) ; if ∃z : (z∈S)∧(clé of z = c) then
                                 S := S-{x} else erreur4 ;

ext procedure modifier (c: T1, p : T2) if ∃z : (z∈S)∧(clé of z = c) then
                                        inf of z := p else erreur5 ;
```

Enfin, on peut expliciter, en donnant une réalisation particulière, le test d'appartenance.

```
        type doublet : (clé: T1, inf : T2) ;
var n : integer ;
var a : array [1..nmax] of doublet ;
var x : doublet ;
initialisation : n := 0 ;

function index (c: T1; m : integer) : integer ;
        begin
        var i : integer ;
        i := 1 ;
        while i≤m and clé of  a[i] ≠ c do
                i := i+1 ;
        index := i
        end ;
        {cette procédure interne réalise le test d'appartenance d'un élément
         de clé c à a[1..m] (oui si index ≤ m)}
```

```
ext procedure entrer (c: T1; I :T2); clé of x := c ; inf of x := I ;
                                     if n = nmax then erreur1
                                     else begin i := index (clé of x, n);
                                          if i<n then erreur2 else
                                          begin
                                          n := n+1 ;
                                          a[n] := x
                                          end
                                     end ;
ext function resultat (c: T1) : T2 ; i := index (c, n) ;
                                     if i<n then resultat := inf of  a[i]
                                     else erreur3 ;

ext procedure supprimer (c: T1) ; i := index (c, n) ;
                                  if i<n then erreur4 else
                                     begin {compactage}
                                      a[i] := a[n];
                                     n := n-1
                                     end ;

ext procedure modifier (c: T1; p : T2) ; i := index (c,n) ;
                                         if i>n then erreur5 else
                                         inf of a[i] := p ;
```

Remarque:
le type doublet est ici interne au module. Il n'est donc pas possible de l'utiliser comme paramètre de la procédure *entrer*. Le paragraphe suivant donne une solution à ce problème en "exportant" les types. Nous verrons (§ III.3.) d'autres solutions possibles .

II.6 - Utilisation simultanée des modules et générateurs de types

On peut remarquer qu'un langage comme Modula (Wirth 77) renferme à la fois la notion de module et celle de générateur de types.

En effet, un module (au sens de Modula) comprend un ensemble de déclarations de constantes, types, variables et opérations sur ces variables. La liste des

définitions externes (objets exportés) et celle des références externes (objets importés) apparaissent en début de module. La valeur de la représentation interne du module ne peut être modifiée de l'extérieur du module que par l'intermédiaire de procédures définies comme étant externes.

Un module (au sens de Modula) ne comportant pas d'autres objets importés ou exportés que des variables ou procédures, constitue un module tel que nous l'avons défini plus haut.

L'aspect "générateur de type" apparaît dans la possibilité d'exporter aussi des noms de types. Cela permet d'utiliser dans un module *M2* des objets du type *T* défini dans *M1* et exporté. En dehors de *M1*, l'accés à la représentation d'un objet de type *T* n'est possible que par l'intermédiaire des opérations définies et exportées dans *M1*, et importées dans le module où est déclaré l'objet.

L'intérêt de cette notion peut être illustré par l'exemple suivant:

```
module M ;
    define T, op1, op2, op3 ;
      :  {declaration du type T et des operations op1, op2, op3}
    end M

module M1 ;
    use T, op1, op2 ;
      :  {declaration d'un objet de type T, sur lequel seules les operations
          op1 et op2 sont permises}
    end M1

module M2
    use T, op1, op3 ;
      :  {declaration d'un objet de type T sur lequel seules les
          opérations oP1 et op3 sont permises}
    end M2
```

dans *M* les procédures *op1*, *op2* et *op3* sont créées,dans *M1* comme dans *M2* , c'est un objet d'un tel type qui est créé, c'est-à-dire sa représentation physique.

Ainsi, quel que soit le nombre de modules qui utilisent le type *T* , les opérations *op1*, *op2* et *op3* n'existent qu'en un seul exemplaire.

D'autre part, il est possible de donner des droits différents à des modules différents, par le biais de la liste d'utilisation.

II.7 - Notre choix et ses conséquences

Les générateurs de types constituent donc un mode de structuration des objets. Par rapport aux types des langages évolués, ils procurent une plus grande protection puisque seules sont accessibles les parties que l'on définit comme telles, et que de plus, la manière dont elles sont accessibles est précisée. Cependant, ces modèles n'ont plus d'existence propre après la compilation. Cela veut dire que s'ils permettent de structurer le texte source d'un gros programme, cette structuration ne se retrouve pas dans le texte binaire ; c'est pourquoi nous avons conservé comme notion de base de notre projet ce que nous avons appelé module, c'est-à-dire le bloc de construction.

Un module comprend donc pour nous, tant dans le texte source que dans le texte binaire, une représentation de l'objet qu'il implémente. Ce sont les variables globales du module, ou par abus de langage, variables d'état. Ces variables ont une valeur initiale lors de la création du module, valeur qui peut être modifiée à chaque appel de procédure du module. Les variables d'état conservent donc leur valeur entre deux utilisations du module. C'est ce qu'on appelle des variables rémanentes.

A chaque représentation, contrairement aux générateurs de types, correspond un ensemble de procédures et fonctions qui ont accès aux valeurs des variables d'état et peuvent les modifier. Seules les procédures définies comme externes sont accessibles de l'extérieur du module. A l'intérieur de chaque procédure, interne ou externe, peuvent être définies variables et procédures locales. A toute exécution d'une procédure P correspond, comme c'est le cas habituellement, un exemplaire de ces données locales.

Celles-ci ont donc une durée de vie correspondant au temps d'exécution de la procédure qui les englobe.

Notre choix était aussi dès le départ, de développer le plus possible les propriétés de bloc de construction, en vue de réutiliser les modules déjà écrits, c'est pourquoi les modules binaires devront comporter des renseignements sur l'interface: nom des procédures externes, nombre et type des paramètres. Nous décrivons dans (Lucas 77) les conséquences de l'existence de ces différents objets sur le schéma d'adressage.

Enfin, bien que nous ayions écarté au départ le développement des générateurs de types mélangés aux modules, comme c'est le cas dans Modula, nous avons été obligés par la suite d'envisager une construction, les environnements, qui fournisse des propriétés analogues. Nous verrons (§ III.3) que c'est une construction moins puissante que les types exportés, mais dont l'intérêt réside dans son indépendance par rapport aux modules. Elle a donc pu être étudiée à part.

III. INCLUSION D'UN SCHEMA DE PROGRAMMATION MODULAIRE DANS UN ENSEMBLE D'OUTILS

Nous avons présenté un schéma de modules et fait apparaître deux niveaux d'écriture des modules : celui de la spécification, et celui de la réalisation.

En s'imposant une discipline stricte de traduction, une sorte de " schéma de réalisation", on peut traduire de tels modules à l'aide des langages existants, qu'ils soient de haut ou de bas niveau. Mais si la traduction n'est pas faite automatiquement, rien ne vient garantir que ce schéma sera effectivement utilisé, et qu'il n'y aura pas des "exceptions" faites pour des raisons d'efficacité ou d'habitude. La meilleure garantie contre l'indiscipline du programmeur est donc de fournir un langage d'écriture de modules. Ce chapitre étudie quelques caractéristiques d'un ensemble d'outils, dont le langage d'écriture, qui pourrait permettre l'utilisation, et la réutilisation de modules pour la construction de gros programmes.

III.1. Modèles de modules

a) Introduction des métavariables

On cherche à concevoir des modules utilisables dans plusieurs programmes distincts. Il semble raisonnable de prévoir que les valeurs d'un certain nombre de paramètres (ou métavariables) ne seront pas fixées une fois pour toutes. En effet, si l'on a besoin dans une application du module suivant :

```
module  Pile_entiers;
const  nmax = 20;
type  Elem = integer;
var  P:array  [0 ..nmax] of Elem;
ext procedure empiler;
                    :{déclaration des opérations possibles sur P}
end Pile_entiers
```

Peut-être aura-t-on besoin dans une autre application d'une pile de 26 éléments de type caractère. Les opérations restent essentiellement les mêmes quant à leur description. Seuls changent le nombre maximal d'éléments et le type de ces éléments.

Si on définit le modèle de module :

```
modèle  Pile (&nmax:integer;&Elem:type);
   var  P:array [0  ..&nmax] of &Elem ;
        :
end Pile
```

Le module *Pile_entiers* correspond à : *Pile (20, integer)*
alors que le second serait défini par *Pile (26,char)*.
Le modèle de module est donc un texte en langage source comportant des métavariables et permettant d'engendrer un module par affectation de valeurs aux métavariables.

b) Nature des métavariables

Les métavariables ont été introduites pour paramétrer les modules, dont la spécification des opérations est déterminée. Il s'agit seulement de modifier la structure de donnée sur laquelle s'appliquent ces opérations. Les métavariables sont des constantes pour un module donné, constantes dont la valeur peut être affectée par simple remplacement lexicographique et au plus tard lors de la création du module. Les constantes les plus simples sont celles d'un type déterminé (*&nmax* dans l'exemple précédent). Mais nous avons vu dans le même exemple que passer un type par son nom, ou sa structure peut être intéressant. Les autres valeurs : noms de variables et morceaux de programmes ne sont pas compatibles avec la notion de module. En effet, la notion de variable globale à plusieurs modules est inexistante. Quant au remplacement lexicographique d'une métavariable par un morceau de programme, cela pourrait conduire à dénaturer complètement la fonction spécifiée.

Dans la mesure où nous fournissons un moyen de lier des noms de procédure après la création des modules (III.2), nous interdisons de mettre des noms de procédure comme métavariables. Les contrôles du nombre et du type des paramètres sont ainsi laissés au connecteur.

c) Différents moments de liaison

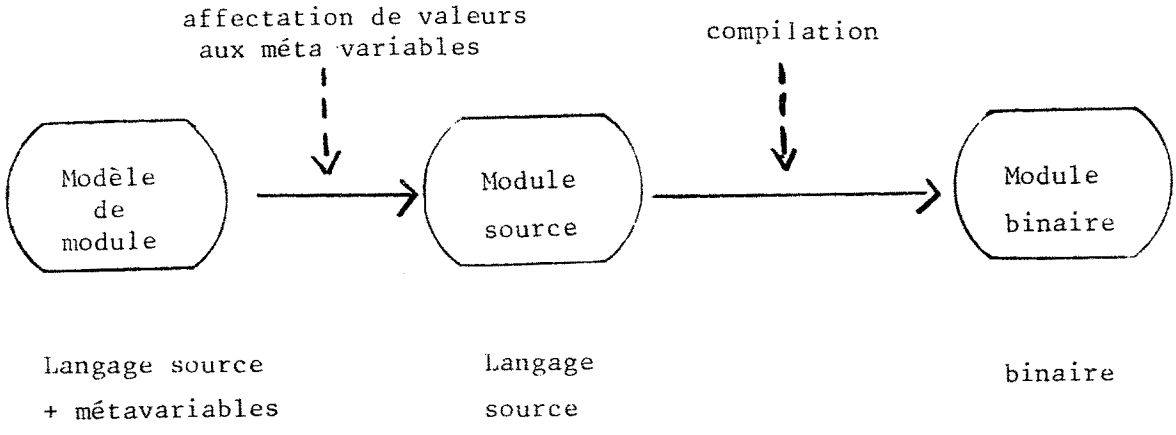
Les métavariables doivent être affectées au plus tard au moment de la création, car la représentation de l'objet et le code généré dépendent de ces valeurs.

Selon le moment de liaison et de création, des tests de cohérence différents peuvent être faits.

Une solution consiste à effectuer un remplacement lexicographique sans autre test au préalable. On confie alors le soin au compilateur d'effectuer les tests de cohérence si la création est statique, et de générer les instructions effectuant ces mêmes tests si la création est dynamique.

Dans le cas où les métavariables sont des constantes d'un type donné, cette solution n'est vraiment pas avantageuse. En effet, il suffit d'indiquer dans l'en-tête du modèle du module quel type doit avoir la métavariable (*integer* pour *&nmax*) pour pouvoir contrôler dès la compilation dans tous les cas (et avant remplacement de la métavariable par sa valeur) si la valeur est bien du type voulu. Pour les types, il ne suffit pas de donner leur nom. En effet, les fonctions utilisent ces types en supposant définies sur eux certaines opérations. Celles-ci doivent donc être également indiquées dans l'en-tête du module. Dans l'exemple de la pile, il suffit d'indiquer que l'affectation doit être définie pour le type passé en paramètre. Ce souci de cohérence se retrouve dans Alphard (Wulf 76). Lorsque la création est statique, le paramétrage précédent, c'est-à-dire défini de façon précise dans l'en-tête du module permet de séparer complètement l'étape d'affectation des métavariables de celle de la compilation.

Dans ce cas, les modèles de modules subissent les transformations suivantes :



Le module écrit en langage source a peu d'intérêt en lui-même; de plus, le passage d'un modèle de module en module est très simple à effectuer. En conséquence, il suffit de conserver des modèles de modules et des modules binaires (bien entendu, certains modèles de modules peuvent ne pas comporter des méta variables et ne diffèrent donc pas de modules sources).

III.2. Procédures fictives

Un autre paramétrage intéressant est inspiré par la méthode d'accès habituelle utilisée pour les fichiers. Les modules binaires (et donc les modules sources et les modèles de module) pourront comporter des références externes non définies, une procédure externe pouvant être désignée par un nom fictif; le nom réel correspondant sera défini à la connexion.

Exemple :

On se propose d'écrire un programme chargé de trier des nombres lus sur un support externe. Pour mettre au point ce programme, on utilise un générateur de nombres aléatoires.

On écrit alors trois modules :

- un module d'entrée contenant une procédure externe qui délivre la valeur du nombre suivant :

```
ext function  nbsuiv : real ;
```

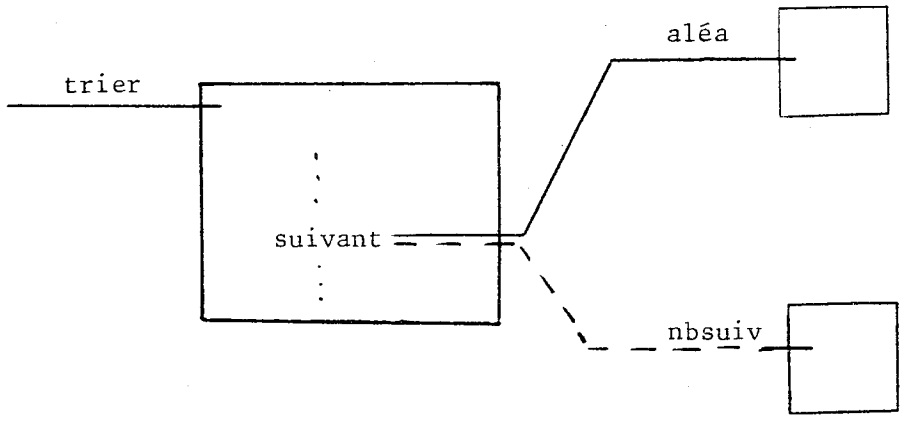
- un module générateur de nombres aléatoires, utilisables au moyen d'une procédure externe qui délivrer un nouveau nombre à chaque appel,

```
ext function alea : real :
```

- un module de tri qui fait appel à une procédure externe de nom conventionnel *suivant*. A la compilation du programme, ce nom ne désigne aucun objet.

```
undef function suivant : real ;
ext function tri (n : integer) : file ;
   begin
   {cette procédure lit n nombres de type real et délivre un fichier
    tri contenant les nombres triés par ordre croissant}
      :
   z := suivant ;
      :
   end;
```

Au cours de la phase de mise au point, on fera correspondre à *suivant* la procédure *alea* ; lors de la phase d'exploitation, on remplacera *alea* par *nbsuiv*. On peut ainsi réaliser deux programmes distincts sans avoir à modifier le texte d'aucun module. La puissance de cette construction est analogue à celle que permettrait le passage de paramètres du type *procédure*; la liaison, tout en restant postérieure à la phase de compilation, est réalisée avant la phase d'exécution.



III.3. Caractéristiques du langage d'écriture des modules

a) Généralités

La notion de base du langage d'écriture est le module. Cette notion met en jeu des éléments de structuration (procédures), qui ne peuvent être commodément exprimés que dans un langage de haut niveau.

Tel qu'il a été défini au chapitre II, un module est constitué d'un ensemble de variables et de procédures internes et externes; les variables ne sont modifiables que par l'intermédiaire des procédures externes. Cette dernière propriété peut se traduire de deux façons :

- Soit les variables définies comme externes ne sont accessibles qu'en lecture à l'extérieur du module. C'est le choix de Parnas (Parnas 72) et Brinch Hansen dans ses classes (Brinch Hansen 74)
- Soit on ne permet pas la définition de variable comme externe. L'accès à la valeur se fait alors par l'intermédiaire d'une fonction.

Les paramètres des procédures et fonctions externes sont des objets ou valeurs venues d'autres modules. Si l'on interdit d'accéder à une variable à l'extérieur de son module, il ne faut autoriser le passage de paramètres que par valeur. Dans ce cas, le langage devra nécessairement permettre de rendre comme résultats des valeurs d'objets structurés.

b) Environnement

Il peut arriver que certains types, ou certaines opérations sur des valeurs d'un type particulier doivent être utilisées dans plusieurs modules. C'est le cas dans l'exemple du chapitre précédent (II.5) où le type *T1* d'un élément de table doit être connu par le module gérant la table et par ceux utilisant cette table. C'est le cas aussi du type *doublet* qui gagnerait à être défini dans chacun des modules utilisateurs. Plusieurs solutions sont envisageables.

Si le module est un élément du langage qui apparaît à l'intérieur d'un programme, il est possible de définir des types et opérations globaux. Mais cette solution ne permet pas de différencier les droits des différents modules. Tous ont accès à tous les types globaux et à toutes leurs opérations.

Nous avons rencontré (II.6) une autre solution à ce problème: il s'agit des types exportés de Modula (Wirth 77). Rappelons qu'il est possible dans Modula de définir un type comme externe à un module. Les opérations définies sur les valeurs ou objets de ce type sont définies comme externes. Ainsi dans les autres modules peut-on déclarer des objets de ce type sans qu'il y ait recopie des opérations. Avec cette solution, les critiques précédentes ne sont plus possibles. On peut se demander cependant pourquoi définir ce type dans tel module plutôt que tel autre.

La solution que nous avons adoptée consiste à regrouper un tel ensemble de déclarations de constantes, de types et d'opérations sur les valeurs de ce type dans un "environnement", entité différente du module et définie en dehors.

Une solution est bien sûr de recopier l'environnement dans chaque module où on en a besoin. Mais cela n'est ni commode, ni sûr lorsque cette recopie est manuelle. Il est donc intéressant de conserver ces environnements dans une bibliothèque, où l'on pourra les consulter ou les recopier.

On peut alors signaler lors de l'écriture d'un module que l'on veut utiliser tel ou tel environnement.

Exemple :

```
environnement CØMPL;
     type complex  = (Im,Ré : integer);
     function  Add (A,B:complex) : complex;
     begin
       Im of Add := Im of A + Im of B;
       Ré of Add := Ré of A + Ré of B;
     end;
end CØMPL
Module M;
   env  CØMPL;
   var  S,I : complex;
   begin
        S:=0;
        Add(S,3);
         :
         {utilisation du type complex}
   end M
```

Lors de la compilation du module *M* on fera comme si les déclarations de l'environnement étaient insérées à cet endroit, c'est-à-dire que le compilateur génèrera le code qu'il aurait généré si le texte de l'environnement se trouvait recopié à cet endroit.

III.4. Connexion entre modules

a) Généralités

Les modules étant des blocs de construction, il est nécessaire de pouvoir les relier entre eux. Plusieurs solutions sont possibles, selon que l'on veut faire une connexion statique ou dynamique que les phases de connexion, compilation, exécution sont séparées ou non.

Il s'agit donc d'avoir un moyen de nommer, de créer, et de relier entre eux différents modules. Dans tous les cas, des tests de cohérence devront

être effectués quant à l'existence des procédures, au nombre et au type de leurs paramètres. Si c'est une création dynamique, ces tests devront être insérés dans le texte du programme. On gagne en efficacité si ces tests peuvent être effectués avant l'exécution. C'est le cas si la connexion est statique :
Si elle se fait à la phase de compilation, certaines vérifications pourront être faites à ce moment-là, comme des vérifications de nom, de concordance de types de paramètres, etc... Un inconvénient de cette dernière solution en est le manque de souplesse. En effet, si une connexion est à changer, il faut tout recompiler alors que dans le premier cas il suffit de changer la connexion par instruction.

Une solution intermédiaire est de fixer une étape de connexion entre la compilation et l'exécution. Cela s'accorde bien avec le choix de faire du module une unité de compilation. Alors que dans les autres solutions les directives de connexion font partie du langage d'écriture de modules, cette dernière solution impose l'existence d'un langage de connexion. Quelles en sont les caractéristiques ?

b) Langage de connexion

Le langage de connexion doit permettre d'énumérer un ensemble de modules, et de décrire les relations qui doivent exister entre ces modules pour qu'ils forment un système complet. L'éditeur de lien classique reçoit en entrée des modules "éditables", c'est-à-dire qui ont déjà été compilés. Il se contente d'effectuer les connexions demandées. On peut donner à l'interpréteur du langage de connexion un rôle plus important, et lui permettre d'avoir, en plus du rôle d'éditeur de lien, celui du langage de commande, c'est-à-dire qu'il pourra appeler le bibliothécaire (cf.III.5) pour obtenir les modèles et modules nécessaires, puis le compilateur pour transformer les modules sources en modules binaires. Si le module binaire contient suffisamment de renseignements, il pourra en outre vérifier lors de l'édition de lien des cohérences d'interface.
Le langage comprend donc deux parties distinctes :

- Les déclarations, d'une part, qui permettent d'obtenir tous les modules binaires nécessaires à la construction d'un système. Ils sont obtenus, comme on l'a vu, par affectation des métavariables , puis appel au compilateur, sauf si un tel module binaire existe déjà en bibliothèque.

Si par exemple on dispose en bibliothèque d'un modèle de module dont l'en-tête est le suivant :

modèle table (&N : integer ; &T : type);

l'exécution de :

Tab : modèle = table (50,real)

aura pour effet :

1°/ D'obtenir du bibliothécaire le modèle *table*

2°/ De créer le texte du module correspondant à *table (50,real)* en remplaçant dans tout le texte du modèle *&N* par *50* après avoir vérifié que *50* est une valeur du type *integer*, et *&T* par *real* après avoir vérifié que *real* est un identificateur de type.

3°/ D'appeler le compilateur et d'obtenir ainsi le module binaire correspondant au module.

4°/ De lui donner le nom de *Tab*

Eventuellement, une directive doit permettre de demander au bibliothécaire de conserver ce module binaire. Si une autre utilisation en est nécessaire, il suffira alors d'écrire

Tab : module binaire = table (50,real)

La seule opération faite sera alors la première.

- Les affectations d'autre part, qui permettent d'identifier des procédures fictives à des procédures réelles. Ainsi si une procédure fictive *error* est déclarée dans le module *Tab* et si on a créé un module binaire *Err* de traitement d'erreurs contenant une procédure de nom *faute*, on identifie la procédure *error* du module *Tab* et la procédure *faute* du module *Err* en écrivant :

Tab·error := Err·faute

Cet ensemble de déclarations et affectations constitue un programme de connexion. Il faut y ajouter une façon d'indiquer quel est le point de départ de l'exécution de système.

En complétant l'exemple développé jusqu'ici, on obtient :

On dispose en bibliothèque de deux modèles de modules :

```
modèle
      table (&N : integer ; &T (type);
  undef procedure error ;
  ext procedure entier ;
  ext procedure sortir ;
end table
modèle exemple (&T : integer) ;
   undef procedure lire ;
   undef procedure écrire ;
   ext procédure vasy ;
end exemple
```

et d'un module binaire *erreur* contenant une définition de procédure externe du nom *faute*. On peut utiliser ces composants dans le programme de connexion ci-dessous :

```
tab : modèle = table (50,real) ;
ex  : modèle = exemple (100) ;
err : module binaire = erreur ;
tab.error :=err.faute ;
ex.lire := tab.entrer ;
ex.ecrire := tab.sortir ;
start ex.vasy ;
```

Par analogie avec les modèles de modules, et toujours dans le souci de pouvoir réutiliser des morceaux de programmes, on est conduit à définir des procédures de connexions, qui sont des programmes de connexion paramétrés. La même discussion peut avoir lieu au sujet du moment et de la façon d'effectuer l'affectation de ces nouvelles métavariables.

L'exemple précédent pourrait s'écrire alors :

```
proc  tableau (&x : integer ; &bidon : proc) ;
tab : modèle = table (50,real);
ex : modèle = exemple (&x);
tab.error := &bidon ;
ex.lire := tab.entrer ;
ex.écrire := tab.sortir ;
end tableau ;
err : module binaire = erreur ;
tableau (100, err.faute);
start ex.vasy ;
```

Nous avons vu qu'il suffisait de conserver des modèles de module. De la même façon, seule la conservation des procédures de connexion est intéressante.

III.5. Le bibliothécaire

Nous regroupons sous le nom de bibliothécaire l'ensemble des programmes chargés de la gestion (catalogage, conservation, recherche, édition) d'un certain nombre d'objets du système.

Contrairement aux systèmes de gestion de fichier, qui sont chargés de gérer un petit nombre de gros volumes d'information, il s'agit ici de gérer un très grand nombre de petits objets.

Différents objets sont à conserver.

En langage source, nous avons rencontré

- les modèles de modules
- les environnements

En langage de connexion : les procédures de connexion.

Enfin nous avons des modules binaires.

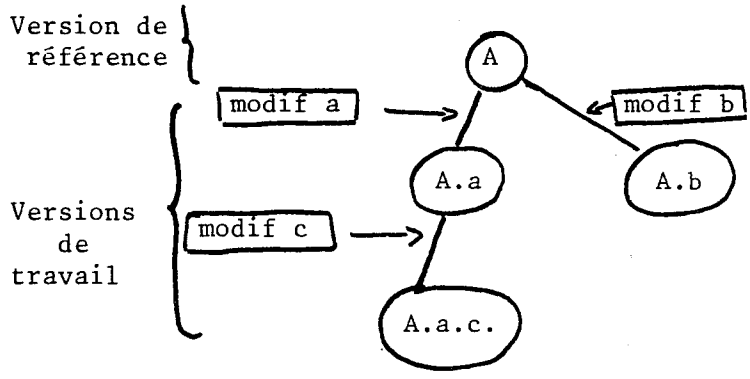
La construction de systèmes à partir de modules donnant lieu aux opérations suivantes :

- création et suppression de modules sources nommés
- modification de modules sources existants, avec ou sans conservation de la version initiale
- production de modules objets par compilation de modules sources.
- production de macro-modules par connexion des modules objets
- recherche de modules source suivant certains critères
- extraction d'information diverses (impression de texte source, de commentaires, de catalogues, de la liste des composants d'un macro-module, etc...)

On voit apparaître des relations entre :

- un module source et les divers modules objets compilés à partir de lui
- les diverses versions résultant des modifications d'un même module source
- les modules composants d'un macro-module.

Un module source subit des modifications au cours de sa mise au point. Etant donnée une version, le bibliothécaire doit permettre de retrouver à partir de quelle version et quelle modification elle a été obtenue. La méthode de travail habituelle consiste à définir des versions de travail d'un module par modification d'une version de référence. Chacune de ces versions peut elle-même être modifiée à son tour. On définit ainsi une structure arborescente entre les versions, avec la relation "est une version modifiée de ". La racine de l'arborescence est une version de référence.



Les opérations courantes sont :

- création ou destruction d'une version
- remplacement d'une version par l'une de ses descendantes.

D'autre part, chaque module binaire est issu d'une compilation avec un ensemble de valeurs des métavariables, valeurs qui doivent pouvoir être retrouvées. Il est du rôle du bibliothécaire de garantir la cohérence entre ces divers objets :

- cohérence entre module source et module objet : tout module objet doit pouvoir être reconstruit à partir du module source dont il est issu. Donc la destruction ou le remplacement d'un module source entraîne la destruction ou le remplacement de tous les modules objets qui en sont issus. Tout module objet doit contenir ou répérer les paramètres éventuels de la compilation. Enfin, le bibliothécaire ne doit pas permettre la modification directe des modules objets.

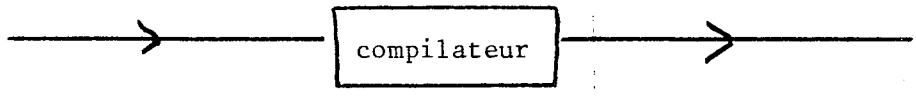
- cohérence entre les versions d'un module : toute version de travail doit pouvoir être reconstruite à partir de la version de référence dont elle est issue et des modifications successives. Par suite, la destruction d'une version entraîne la destruction de tous ses descendants.

- cohérence entre macro-modules et modules constituants : un macro-module résulte de la connexion de plusieurs modules objets. Il est entièrement décrit par un programme de connexion. Ce programme n'a de sens que si les modules sources constituants existent.

III.6. Vue d'ensemble de SESAME

Nous avons présenté plusieurs solutions pour résoudre le problème de l'inclusion de la notion de module dans un ensemble d'outils. Nous avons fait apparaître la nécessité d'un langage d'écriture incluant la notion de module. Des choix faits au niveau de ce langage découle l'existence d'autres outils. D'une façon générale nous avons essayé de séparer au maximum les moments de transformation et d'utilisation des modules, afin de les rendre plus apparents, et de pouvoir éventuellement y travailler de façon indépendante.

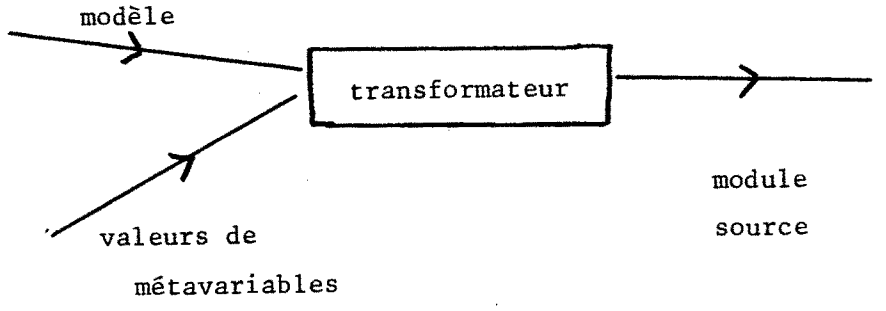
Une première décision est donc de séparer création des modules et exécution. L'utilisation d'un compilateur s'impose alors. Nous avons voulu faire du module un bloc de construction élémentaire. C'est pourquoi nous avons pris comme unité de compilation le module. Un appel au compilateur se traduit par l'obtention d'un module binaire pouvant contenir des références non résolues.



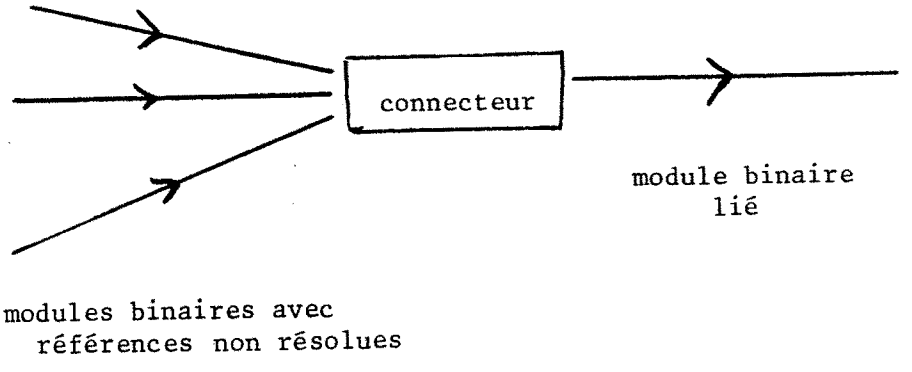
Module source

Module binaire
avec références
non résolues

- Un module source est obtenu après affectation de valeurs aux méta-variables
  Il existe donc un transformateur de modèle de module en module.



- Le module binaire obtenu comporte des références non résolues.Entre l'obtention de ce module binaire et l'exécution se situe donc une phase de connexion, permettant de résoudre les références de procédures fictives.

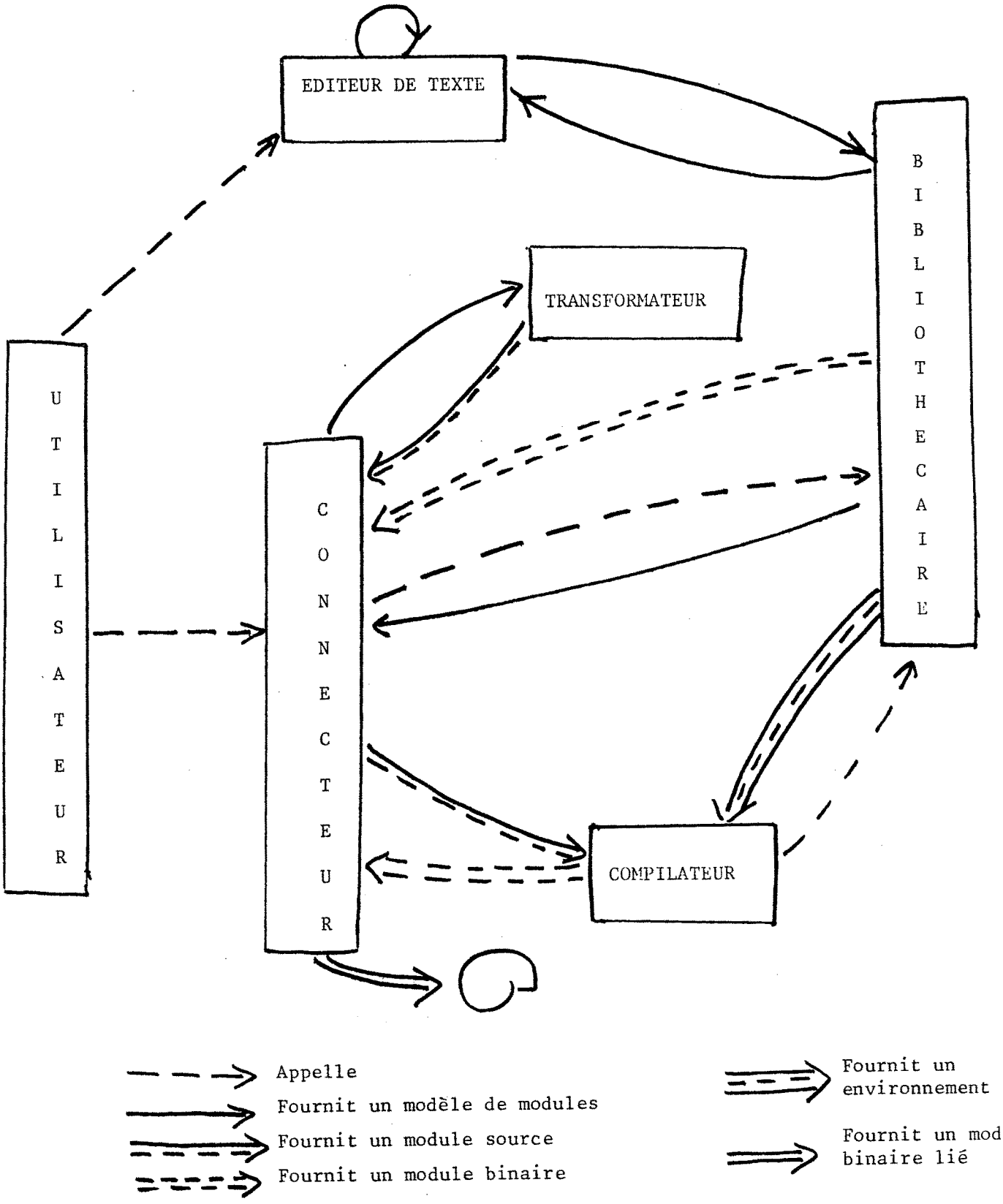


Nous avons donc développé parallèlement au langage d'écriture un langage de connexion. En fait, ce langage joue aussi le rôle de langage de commande. Cela veut dire que lors de l'analyse d'un programme de connexion, le connecteur appelle les processeurs nécessaires (bibliothécaire, compilateur) à l'obtention des modules binaires dont il a besoin. Il permet aussi d'appeler le processeur qui permet de passer de modèle de module à module source. On peut suivre la séquence des appels sur l'exemple donné dans III. 4.

Il faut rappeler cependant que le bibliothécaire gère quatre sortes d'objets : des modèles de modules, des modules binaires , des procédures de connexion et des environnements.

Texte analysé	Différents appels
tab : modèle = table (50,real)	(1) Appel au bibliothécaire qui fournit le modèle *table*.
	(2) Appel au transformateur qui contrôle *50* et *real*, puis fournit le module source voulu.
	(3) Appel au compilateur qui transforme le module source en module binaire.
ex : modèle = exemple (100)	Opérations analogues à (1) (2) (3)
err : module binaire = erreur	(4) Appel au bibliothécaire qui fournit le module binaire.
tab.error := err.faute *ex.lire := tab.entrer* *ex.écrire := tab.sortir*	connecteur

Le module binaire résultant est envoyé sur bande.
Enfin, pour obtenir des modèles de module, l'utilisateur passe par l'intermédiaire d'un éditeur de texte. Le schéma suivant donne une vue d'ensemble des outils.


EDITEUR DE TEXTE
TRANSFORMATEUR
UTILISATEUR
CONNECTEUR
BIBLIOTHECAIRE
COMPILATEUR
Appelle
Fournit un modèle de modules
Fournit un module source
Fournit un module binaire
Fournit un environnement
Fournit un module binaire lié

IV. PROCESSUS, MODULES ET SYNCHRONISATION

Le projet est conçu dans le but de permettre la production de systèmes d'exploitation. Or, dans tout système d'exploitation se déroulent des activités en parallèle. En ce sens, le processus est habituellement un élément de structuration important des systèmes d'exploitation. C'est pourquoi nous essayons ici, de faire apparaître la relation entre processus et module, avant de pouvoir donner des caractéristiques de synchronisation propres aux modules.

IV.1 Processus et modules

a) Processus

On appelle processus l'exécution d'un programme sur un ensemble de données. Il s'agit d'une activité formant logiquement un tout, dans laquelle la succession des opérations est déterminée par le texte du programme et les données initiales et externes. Cette définition conduit à avoir autant de processus que d'activités possibles en parallèles. Toutefois, la notion de processus est utilisée aussi lorsque le programme est destiné à être exécuté par un monoprocesseur ; elle sert alors comme unité de décomposition logique, permettant ainsi de découper les gros programmes. Chaque partie logiquement indépendante est considérée comme constituant un processus. On peut donner comme exemple le système GMS [Bellino 73] où un processus est associé à chaque périphérique, reflétant le parallélisme technologique, mais où l'on trouve aussi un processus par périphérique virtuel, un processus par interruption déclenchée, etc...

b) Modules

L'introduction de modules fournit une nouvelle façon de décomposer de gros programmes. Il ne s'agit plus d'isoler les parties logiquement indépendantes, mais de regrouper un certain "objet", ou structure de données, et les opérations que l'on peut effectuer dessus. On a ainsi une sorte de 'boîte noire' dont on ne voit dépasser que quelques pattes (opération externes), et que l'on peut utiliser dans la réalisation d'un programme. L'objet est repré-

senté par des variables dont la valeur à un moment donné constitue l'état du module.

L'exécution des opérations externes est possible par un ou plusieurs processus. Elle est a priori possible (sauf synchronisation explicite) dans un ordre quelconque.

c) Utilisation simultanée de processus et modules

L'exécution d'un programme se poursuit à travers différents modules, et il se peut qu'à un moment donné plusieurs processus s'exécutent dans un même module.

Les deux notions sont donc complémentaires. Cependant, si on veut garder à chacune sa propriété d'unité de décomposition, des conflits risquent d'arriver, une unité logiquement indépendante pouvant regrouper plusieurs objets structurés,ou inversement plusieurs processus pouvant correspondre à un seul objet : citons le cas du producteur-consommateur, habituellement découpé en deux processus, et dont la décomposition ne nécessite qu'un seul module, celui de la zone de communication. Aussi, pour rester cohérents avec le but initial, nous prenons comme unité de décomposition le module et gardons le processus comme unité d'exécution.

IV.2 Synchronisation des procédures d'utilisation de l'objet

L'objet réalisé par le module peut être considéré à deux niveaux : celui de la spécification, où seules interviennent les procédures externes et leurs fonctions, et celui de la réalisation, où l'on tient compte de la façon dont est implémenté l'objet. La synchronisation s'exprime à chacun de ces niveaux.

a) Au niveau de la spécification

Si l'on considère l'exemple suivant : on spécifie un module *Tampon* comme devant gérer une zone de telle sorte que l'on puisse *remplir* et *vider* cette zone (fonctions externes). On peut, à ce niveau, donner des

indications de synchronisation en disant que la zone devra être alternativement remplie, puis vidée, en commençant par un remplissage. Ces conditions ne portent que sur les procédures externes, et ne font aucune référence à la façon dont sera réalisé l'objet. Elles sont donc bien du domaine de la spécification.

b) Synchronisation au niveau de la réalisation

Supposons maintenant que ce tampon soit réalisé par une succession de cases dont, à chaque instant, une seule est le tampon courant, pointée par un indicateur.

Le module s'écrirait :

```
Module Tampon1;
  {déclaration de la zone, et de l'indicateur de tampon courant}
  ext procedure remplir (T:élément);
  ext function vider : élément;
  end Tampon1
```

Dans la réalisation des procédures *remplir* et *vider*, on utilise l'indicateur. Celui-ci n'étant pas connu au niveau de la spécification, les conditions de cohérence de son utilisation ne peuvent être exprimées qu'au niveau interne de réalisation du module.

Il existe donc aussi une synchronisation au niveau de la réalisation.

c) Conséquences pour les primitives de synchronisation

Nous venons de voir que la synchronisation doit être décrite à deux niveaux, spécification et réalisation. Notre choix étant de ne pas aborder les problèmes de cohérence entre ces deux niveaux, nous n'avons pas étudié le degré d'indépendance que l'on a dans leur traduction. Mais une fois remarqués ces deux niveaux, on peut dire que les primitives de synchronisation, au niveau réalisation donc, pourront intervenir à la fois pour la synchronisation des procédures et fonctions externes (spécification) et pour celle des procédures et fonctions internes (réalisation).

IV.3 Synchronisation des processus

Nous avons vu que la notion de processus est indépendante de celle de module. Tout module est considéré a priori comme partageable, et synchronisé de sorte que plusieurs processus puissent demander à l'exécuter en même temps. Si l'on veut différencier la réponse du module selon le processus appelant, deux choses peuvent être nécessaires. D'une part, il est indispensable que le processus se nomme, c'est-à-dire passe en paramètre une identification qui lui est propre. D'autre part, il peut être utile de de garder trace dans le module de la demande et du demandeur, d'une exécution à l'autre de la procédure. Il est alors nécessaire d'introduire comme variable du module une structure de données pour mémoriser ces renseignements.

Exemple - Un ensemble de tampons est utilisé par plusieurs processus, tampon par tampon. Seul un des producteurs produit des messages tenant sur deux tampons consécutifs. Il faut bien pouvoir reconnaître son identité pour lui affecter d'un coup les deux tampons qu'il demande. Si de plus, il faut que ces deux tampons soient vidés par le même processus, il faut conserver quelque part l'identité des tampons, puis celle du processus qui aura vidé le premier des deux, afin de vérifier que c'est le même qui vide l'autre.

Le module s'écrit alors :

```
Module Tampon2;
 {déclaration de la zone et d'indicateur de tampon courant}
 {déclaration d'un indicateur d'identité de remplisseur et
  de videur}
ext procédure remplir (E : élément ; D : boolean) ;
        {D est vrai si le processus veut 2 cases}
ext function vider (D : procname) : élément ;  {procname identifie le processus}
end Tampon2
```

IV.4 Les moniteurs

D'après ce que nous venons de voir, la procédure est la seule unité d'exécution visible de l'extérieur. On veut essayer de conserver cette propriété d'unité d'exécution au niveau de la synchronisation. Une construction proposée semble répondre à ce désir : il s'agit du moniteur de Hoare [Hoare 74], Brinch Hansen [Brinch Hansen 73] .

a) Présentation

Comme le module, un moniteur regroupe un certain nombre de données et de procédures manipulant ces données. Les données du moniteur ne sont accessibles que par les procédures du moniteur. D'autre part, ces procédures n'utilisent pas d'autres variables externes au moniteur que celles passées en paramètre.

Pour assurer la cohérence d'utilisation de ces données partagées, les procédures du moniteur sont exécutées en exclusion mutuelle les unes par rapport aux autres ; le moniteur constitue donc une ressource critique. Cela n'est pas suffisant pour exprimer la synchronisation. On a aussi besoin de pouvoir suspendre et reprendre l'exécution de processus dans certains cas.

Pour pouvoir différencier ces cas, on introduit des variables de type condition dont l'emploi n'est autorisé que dans les moniteurs,et sur lesquelles on peut appliquer deux opérations :*wait* et *signal*. Si *C* est une variable condition :

C.wait a pour effet de suspendre le processus dans la file d'attente correspondant à *C* est de libérer la section critique du moniteur.

C.signal a pour effet de réveiller un processus de la file *C* , s'il y en a, sans libérer la section critique du moniteur, et aucun sinon (pas de mémorisation).

Exemple - On reprend ici l'exemple où un producteur et un consommateur communiquent par l'intermédiaire de *N* zones tampon. On introduit deux conditions, *NON_VIDE* et *NON_PLEIN*, qui servent à suspendre respectivement le consommateur et le producteur. Les procédures accessibles de l'extérieur son préfixées par *ext*.

```
monitor ANNEAU ;
      .
      .          {déclarations}
      .
ext procedure producteur (M : message);
    begin
      if PLEIN = N then NON_PLEIN.wait;
      TAMPON [(i+PLEIN) mod N] := M;
      PLEIN := PLEIN + 1;
      NON_VIDE.signal;
   end;
ext procedure consommateur (var M : message);
    begin
      if PLEIN = 0 then NON_VIDE.wait;
      M := TAMPON [i];
      PLEIN := PLEIN - 1;
      i := (i+1) mod N;
      NON_PLEIN.signal;
    end;
end ANNEAU
```

Remarques -

1) L'utilisation d'un seul pointeur est rendue possible du fait que tout est exécuté en exclusion mutuelle.

2) Une autre conséquence de cette exclusion mutuelle est qu'il n'y a aucun changement à faire au moniteur si l'on veut maintenant l'utiliser pour plusieurs producteurs et consommateurs.

b) Critiques et extensions

On peut regretter que l'emploi des conditions soit réparti dans tout le texte, rendant peu aisée la localisation de la synchronisation.

Malgré cela, on peut considérer un moniteur comme étant un module dont une partie de la synchronisation est figée (à savoir l'exclusion mutuelle entre toutes les procédures).

Cette rigidité est pratique, car elle évite d'écrire certaines synchronisations. Cependant, la remarque 2) du paragraphe précédent fait apparaître un inconvénient de cette rigidité : il se peut que trop d'activités soient en exclusion mutuelle, et que les processus faisant appel au moniteur soient freinés inutilement dans leur exécution.

Une première idée pour généraliser ce mécanisme est donc de permettre d'indiquer quelles procédures sont à exécuter en exclusion mutuelle. D'autre part, pour mieux faire apparaître les unités qui sont synchronisées, on va remplacer les événements par des appels à des procédures qui, elles, seront synchronisées. La procédure devient donc une unité élémentaire de synchronisation et on est conduit à définir un ensemble de chemins d'exécution possibles pour elles. C'est ce qu'on fait Campbell et Habermann [Campbell 74].

IV.5 Les expressions de chemin

a) Présentation

Une expression de chemin a pour opérandes des noms de procédure du module. On peut exprimer que la procédure Q est à exécuter après la procédure P grâce à l'opérateur de séquencement. On le note de la façon suivante :

$$P \ ; \ Q$$

Remarque - La synchronisation s'applique à l'ordre d'exécution des procédures. Rien n'empêche que P et Q soient exécutées par des processus différents. Un processus voulant exécuter Q sera bloqué tant que P n'aura pas été exécutée.

On peut aussi exprimer qu'une procédure au choix peut être exécutée parmi plusieurs. La notation :

$$P + Q$$

indique que l'on peut exécuter à ce moment, soit P soit Q.

Les parenthèses jouant le rôle habituel dans les expressions, on peut composer séquence et choix exclusif pour obtenir des expressions plus complexes.

Exemple - $P + (Q \ ; \ R) + S$ indique que l'on a le choix entre l'exécution de P, l'exécution de Q suivie de celle de R et l'exécution de S.

Les délimiteurs *path* et *end* marquent le début et la fin d'une expression de chemin. De plus, ils indiquent que le chemin peut être exécuté zéro, une ou

plusieurs fois consécutives. Si l'on veut exprimer cette dernière propriété à l'intérieur d'un chemin pour une partie de celui-ci, on utilise l'opérateur*.

Exemple - L'expression *path P ; (Q ; R)*; S end* permet les séquences :

P ; S
P ; Q ; R ; S
P ; Q ; R ; Q ; R ; S
etc...

ainsi que toute séquence formée d'une séquence quelconque de ces chemins permis.

Autre exemple :

Nous traitons de nouveau le cas d'un producteur et d'un consommateur ; mais, pour simplifier, ils communiquent cette fois par l'intermédiaire d'une seule zone tampon. La seule contrainte de synchronisation à imposer dans ce cas est qu'il faut remplir la zone avant de la vider, et ce un nombre variable de fois consécutives.

```
module ZONE;
    .
    .      {déclarations}
    .

ext procedure remplir (m : message);
                {remplit la zone avec m}

ext procédure vider (var m : message);
                {met le contenu de la zone dans m}

synchro
     path remplir ; vider end
end ZONE
```

b) Particularités et critiques

On peut remarquer que les constructions présentées sont telles que toutes les procédures d'un même chemin ne sont exécutables qu'une par une.

Toutes les procédures d'un même chemin sont donc exécutées en exclusion mutuelle les unes par rapport aux autres. Par contre, il n'y a pas de contrainte entre procédures apparaissant dans deux chemins différents ; ces procédures sont donc exécutables en parallèle. Pour cette raison,l'utilisation des expressions de chemin semble donc plus souple que cette des moniteurs.

Dans la recherche de moyens de structurer des programmes, et de séparer les parties indépendantes, un des aspects attirants des expressions de chemin est l'écriture de la synchronisation en dehors des procédures. Cette séparation n'est cependant pas toujours réalisable.

En effet, une des particularités des constructions présentées est qu'elles ne permettent de décrire que des chemins inconditionnels ; c'est-à-dire que ne doivent apparaître ici que les procédures dont l'exécution, une fois commencée, doit se dérouler jusqu'au bout.

Or il y a des cas où l'exécution d'une procédure dépend de l'état du module. Si on a le programme :

```
procedure Q ;
      :
      :
    if CØND then P ;
      :
      :
end Q ;
```

aucun chemin ne faisant intervenir que *P* ou *Q* ne convient, puisque l'exécution de *Q* doit s'interrompre si *CØND* n'est pas réalisé et que l'exécution de *P* doit être suspendue tant que *CØND* n'est pas réalisée.

Pour tourner cette difficulté, on peut avoir recours à l'artifice suivant : lorsque *CØND* est réalisée, et seulement dans ce cas, on fait appel à une procédure artificielle *(Accord)* dont la seule fonction est de bloquer l'exécution de *P* tant que c'est nécessaire. Son corps est donc vide d'instruction. L'appel à *P* est fait inconditionnellement. On obtient :

```
procedure Q ;
        :
        :
      if  CØND  then  Accord ;
       P ;
end Q ;
```

Le chemin de synchronisation s'écrit alors

path Accord ; P end

Le processus exécutant la procédure *P* sera débloqué lorsqu'un autre processus exécutera la procédure *Accord.*

La simulation de sémaphores est un exemple simple de ces procédures vides.

```
module SEMA ;
ext procedure P ; begin end ;
ext procedure V ; begin end ;
synchro
   path V ; P end
end SEMA
```

Cette simulation fait, de plus, apparaître le fait que cette construction correspond exactement à la programmation habituelle d'un événement à l'aide de sémaphore. L'avantage vient ici du fait que la synchronisation apparaît à l'extérieur des procédures.

Dans certains cas particuliers, on peut utiliser la notion conditionnelle des chemins [Habermann 75]. Mais d'après sa définition, ceci n'est possible que si les tests portent sur des valeurs constantes des variables d'état.

Dès que la synchronisation dépend de paramètres, il faut revenir à la construction précédente (procédures vides) ou à un autre artifice de programmation. Les autres extensions présentées par Habermann, en même temps que cette notation conditionnelle, laissent subsister les critiques formulées.

On peut remarquer, enfin, qu'il existe des procédures propres à la synchronisation (procédures vides par exemple) et, puisque les morceaux de programme à synchroniser n'apparaissent dans les chemins que sous forme de noms de procédures, le découpage en procédures dépend de la synchronisation. Cela veut dire qu'en général il ne suffira pas de changer la partie synchronisation (chemins) pour modifier la synchronisation du module.

Une étude approfondie de ces mécanismes, en vue de leur implantation dans le langage d'écriture, n'a pu être réalisée. La première version de SESAME ne comportera donc que des primitives élémentaires [LUCAS 77].

Les principaux résultats énoncés ici : relation entre processus et modules, synchronisation à deux niveaux, ont cependant permis une meilleure compréhension du problème.

V - CONCLUSION

L'étude des différents sens donnés au terme "module" a fait apparaître des propriétés communes : découpage de gros programmes en unités plus petites, séparation de la spécification et de la réalisation, recherche de l'indépendance des unités par minimisation des connexions. Elle a aussi fait apparaître deux catégories de modules : les blocs de construction et les modèles d'objets structurés. La suite de l'étude a porté sur l'aspect bloc de construction.

Pour nous, un module est constitué par des variables globales représentant l'état du module, et des procédures et fonctions qui ont accès à ces variables et peuvent modifier leur valeur. Seules sont visibles de l'extérieur certaines procédures et fonctions définies comme externes. Pour le langage d'écriture que nous avons défini, un module constitue également une unité de compilation. On a ainsi, après compilation, des blocs de construction connectables entre eux.

Afin de permettre la réutilisation des modules dans la construction de plusieurs programmes, deux modes de paramétrage semblent intéressants :

a) Le paramétrage de la structure de données sur laquelle sont définies les opérations du module. Ces paramètres, ou métavariables ont une valeur constante pour un module donné, et l'affectation de cette valeur doit être faite avant la création du module. Selon le choix fait, cette affectation peut avoir lieu à l'exécution ou à la compilation. La solution que nous avons adoptée est de donner dans l'en-tête du module suffisamment d'indications pour pouvoir détecter certaines erreurs de cohérence avant l'affectation ; l'affectation a lieu avant la compilation.

b) Le paramétrage des procédures, inspiré de ce qui est utilisé dans les systèmes de gestion de fichiers, et qui permet de laisser libres à la compilation certaines références externes. Ces références externes seront résolues après les différentes compilations, dans une phase de connexion.

Il est aussi nécessaire de pouvoir utiliser dans des modules différents un même ensemble de constantes, types et opérations sur les valeurs de ces types. Cela nous a conduits à définir des "environnements", structures permettant de regrouper un tel ensemble de définitions indépendemment des modules. Chaque module référence explicitement le ou les environnements dont il a besoin; le compilateur agit lors de la rencontre d'une telle référence comme si l'ensemble des définitions de l'environnement était inséré à cet endroit.

La connexion des modules entre eux peut être faite de différentes façons. Elle peut être faite à la compilation si le langage d'écriture permet de regrouper plusieurs modules dans un même programme. Si le langage contient des directives de connexion, celle-ci peut être dynamique. Nous avons choisi,puisque l'unité de compilation est pour nous le module de faire apparaître une phase de connexion distincte. Cela nous a conduits à développer un langage de connexion, avec ses déclaration, affectations etprocédures. Le connecteur utilise des objets conservés par un bibliothécaire. Il s'agit :

- des modules paramétrés, ou modèles de modules,
- des modules binaires,
- des procédures de connexion.

On trouve aussi dans le bibliothécaire des environnements, qui sont utilisés par le compilateur. Le bibliothécaire est un programme de gestion et de conservation des objets que l'on vient d'énumérer.

Une de ses caractéristiques est de préserver la cohérence entre les différents objets qu'il conserve (différentes versions d'un même modèle de module, modules binaires ne correspondant qu'à des modules sources existants en bibliothèque..)

Enfin, nous avons présenté une brève étude des relations entre processus et module. Nous avons ainsi conservé au processus son rôle d'unité d'exécution tandis que le module devient lui l'unité de construction de systèmes. Quelques particularités d'unesynchronisation adaptée aux modules ont été présentées, et illustrées par la présentation de deux primitives, les moniteurs [Brinch Hansen 73, Hoare 74] et les expressions de chemin [Campbell 74, Habermann 75].

Ces différentes études ont conduit à la spécification, puis à la réalisation de SESAME, ensemble d'outils permettant d'utiliser le module comme unité de construction. SESAME est constitué principalement :

- d'un langage d'écriture des modules, issu du langage Pascal [Jensen 74], qui est analysé et transformé par un compilateur,

- d'un langage de connexion,

- d'un programme de conservation et de gestion des modules, le bibliothécaire.

Il fournit un ensemble d'outils permettent d'utiliser de façon cohérente les modules que nous avons définis.

Les principaux problèmes soulevés par leur réalisation, ainsi que les choix qui ont été faits sont décrits dans [Lucas 77].

SESAME constitue une aide à la réalisation d'un système conçu et découpé à l'aide de nos modules. On trouvera dans [Montuelle 77] une étude de la manière de décomposer ainsi de gros programmes, et de réaliser des modifications dues à un changement de spécification du système. Cette étude a conduit à la conception, puis à la réalisation de PROSPER, petit système modèle permettant de contrôler la validité des choix faits pour SESAME.

Ont été volontairement écartés du projet dans un premier temps les problèmes liés à la protection, aux traitements d'erreurs, et aux tests de concordance ce la réalisation à la spécification. Une version prototype de SESAME est actuellement en fin de réalisation. Certains modules de PROSPER sont réalisés. Cette réalisation parallèle du système producteur et d'un système produit a permis un constant réajustement de l'outil pour en faire un outil réellement utilisable. Une utilisation dans l'enseignement permettra de confirmer ou d'infirmer sa maniabilité.

B I B L I O G R A P H I E

Bekkers 74 — BEKKERS Y.
A comparison of two high level synchronizing concepts, The Queen's University of Belfast (1974)

Bellino 73 — BELLINO J.
Mécanismes de base dans les systèmes superviseurs : conception et réalisation d'un système à accès multiples, Thèse, Grenoble (1973)

Bétourné 76 — BETOURNE C., FERAUD L., JOULIA J., RIGAUD J.M.
Le langage d'écriture de systèmes "LEST", *Congrès AFCET,* Paris (1976)

BrinchHansen 73 — BRINCH HANSEN P.
Concurrent programming concepts, *Computing surveys,* 5,4 (1973)

Brinch Hansen 74 — BRINCH HANSEN P.
Concurrent PASCAL, a programming language for operating system design, California Institute of Technology, TR n° 10 (1974)

Campbell 74 — CAMPBELL R.H., HABERMANN A.N.
The specification of process synchronization by path expressions *Coll. sur les aspects théoriques et pratiques des systèmes d'exploitation,* IRIA, Paris (1974)

Cheatham 72 — CHEATHAM T.E., Jr. WEGBREIT B.
A laboratory for the study of automatic programming, *Proc. SJCC, vol. 40* (1972)

Cheval 76a — CHEVAL J.L., CRISTIAN F., KRAKOWIAK S., LUCAS Ma., MONTUELLE J., MOSSIERE J.
Conception modulaire des systèmes d'exploitation, Rapport de Recherche n° 32, IMAG, Grenoble (1976)

Cheval 76b — CHEVAL J.L., CRISTIAN F., KRAKOWIAK S., LUCAS Ma., MONTUELLE J., MOSSIERE J.
Un système d'aide à l'écriture des systèmes d'exploitation, *Congrès AFCET*, Paris (1976)

Cheval 77 — CHEVAL J.L., CRITIAN F., KRAKOWIAK S., MONTUELLE J., MOSSIERE J.
An experiment in modular program design *Proc. IFIP Congress*, Toronto (1977)

Corwin 72 — CORWIN P., WULF W.A.
A software laboratory, Carnegie-Mellon University (1972)

Cristian 76 — CRISTIAN F., GAUDUEL F.
Réalisation d'un bibliothécaire pour le projet SESAME, Projet de 3ème année ENSIMAG, Grenoble (1976)

Cristian 77 — CRISTIAN F.
A case study in modular design, *Proc. Int. Computing symposium*, Liège (1977)

Crocus 75 — CROCUS
Systèmes d'exploitation des ordinateurs. Principes de conception Dunod (1975)

Dahl 66 — DALH O.J., NIGAARD K.
Simula, an Algol based simulation language, *CACM*, 9,9 (1966)

Dahl 72 — DAHL O.J., DIJKSTRA E.W., HOARE C.A.R.
Structured programming, Academic Press (1972)

De Remer 75 — DE REMER F., KRON H.
Programming in the large vs. programming in the small, Proc. Int. Conference on Reliable Software, *SIGPLAN Notices* 10,6 (1975)

Derniame 74 — DERNIAME J.C.
Le projet CIVA, un système de programmation modulaire, Thèse, Nancy (1974)

Dijkstra 67 — DIJKSTRA E.W.
Cooperating sequential processes, in *Programming languages*, F. Genuys Ed., Academic Press (1967)

Dijkstra 72 — DIJKSTRA E.W.
Notes on structured programming, Academic Press (1972) (in Dahl 72)

Flon 75 — FLON L.
Program design with abstract data types, Carnegie Mellon University (1975)

Flon 76 — FLON L., HABERMANN A.N.
Towards the construction of verifiable software systems, *Proc. ACM symposium on data (abstraction, structure)*, (1976)

Habermann 75a — HABERMANN A.N.
Synchronization of communicating process, *CACM* 15,3 (1972)

Habermann 75b — HABERMANN A.N.
Path expressions, Carnegie Mellon University (1975)

Habermann 76 — HABERMANN A.N., FLON L., COOPRIDER L.
Modularization and hierarchy in a family of operating systems. *CACM*, vol. 19 n° 5 (1976)

Hoare 74 — HOARE C.A.R.
Monitors : an operating system structuring concept, *CACM* 17,10 (1974)

Ichbiah 74 — ICHBIAH J.D., RISSEN J.P., HELIARD J.C., COUSOT P.
The system implementation lauguage LIS, Technical report 4549E/En, CII (1974)

Jensen 74 — JENSEN K., WIRTH N.
PASCAL : user manual and report, Lecture notes in computer science n° 18, Springer (1974)

Krakowiak 74 — KRAKOWIAK S., MOSSIERE J.
Quelques problèmes de la programmation modulaire, note interne, Grenoble (1974)

Krakowiak 75 — KRAKOWIAK S., MOSSIERE J.
Systèmes d'aide à la programmation de systèmes, *Journée sur les systèmes d'aide à la programmation*, IRIA, Paris (1975)

Krakowiak 76 — KRAKOWIAK S., LUCAS Ma., MONTUELLE J., MOSSIERE J.
A modular approach to the structured design of operating systems, *Proc. MRI symposium on computer software engineering*, Polytechnic Institute, New York (1976)

Lampson 77 — LAMPSON B.W., HORNING J.J., LONDON R.L., MITCHELL J.G., POPEK G.J.
Euclid report, *ACM SIGPLAN notices*, (1977)

Liskov 72 — LISKOV B.H.
A design methodology for reliable software systems, *Proc. AFIPS FJCC* (1972)

Liskov 74a — LISKOV B., ZILLES S.
Programming with abstract data types. Proc. of a symposium on very high level languages, *ACM SIGPLAN notices* (1974)

Liskov 74b — LISKOV B.
A note on CLU, Computation structures groupe memo 112, MIT project MAC (1974)

Lucas 74 — LUCAS Ma.
Expression du parallélisme dans les langages de haut niveau, Rapport de DEA, Grenoble (1974)

Lucas 75 — LUCAS Ma.
Primitives de synchronisation pour langages de haut niveau, Séminaire de programmation, Grenoble (1975)

Lucas 77 — LUCAS Ma., MONTUELLE J.
Conception modulaire des systèmes d'exploitation. Description du projet SESAME, Complément de thèse, Grenoble (1977)

Maynard 70 — MAYNARD B.
Modular programming, Butterworths (1970)

Montuelle 77 — MONTUELLE J.
Conception modulaire des systèmes d'exploitation. Méthodes et exemple d'application, Thèse Docteur-Ingénieur, Grenoble (1977)

Morris 73 — MORRIS J.
Types are not sets, *ACM*, Symposium on principles of programming languages (1973)

Mossière 74 MOSSIERE J.
Sur les notions de types, de modules, ... Note interne, Grenoble (1974)

Mossière 75 MOSSIERE J.
Quelques outils d'aide à la programmation des systèmes d'exploitation, Séminaire de programmation, Grenoble (1975)

Naur 69 NAUR P.
Programming by action clusters, BIT, 9,3 (1969)

Parnas 72 PARNAS D.L.
A technique for software module specification with examples, *CACM* 15,5 (1972)

Parnas 75a PARNAS D.L.
The influence of software structure on reliability, Proc. Int. Conf. on Reliable Software, *SIGPLAN Notices* 10,6 (1975)

Parnas 75b PARNAS D.L., HANDZEL G.
More on specification techniques for software modules, BS 1 75/1 *Technische Hochschule Darmstadt* (1975)

Van Winjgaarden 69 VAN WINJGAARDEN, MAILLOUX, PECK, KOSTER
Report on the algorithmic Language Algol 68, MR 101, *Mathematisch Centrum* (1969)

Wirth 77 WIRTH N.
Modula, a language for modular multiprogramming, *Software - Practice and Experience*, vol. 7 (1977)

Wulf 76 WULF W.A., LONDON R.L., SHAW M.
Abstraction and verification in ALPHARD : introduction to language and methodology, Carnegie-Mellon University (1976)