THÈSE

de

DOCTORAT D'ÉTAT ES-SCIENCES MATHÉMATIQUES

présentée

A L'UNIVERSITÉ DE LILLE

par

Didier LANCIAUX

pour obtenir

le grade de Docteur Es-Sciences

Spécialité : Mathématiques
Mention : Informatique

MÉCANISMES DE STRUCTURATION DES SYSTÈMES À CAPACITÉS

Soutenue le 28 octobre 1978 devant la Commission d'examen composée de :

MM.	**P. BACCHUS**	Président
	C. CARREZ **V. CORDONNIER** **C. KAISER** **W.A. WULF**	Examinateurs

NIVERSITE DES SCIENCES
T TECHNIQUES DE LILLE

01.02.1976

DOYENS HONORAIRES de l'Ancienne Faculté des Sciences

MM. R. DEFRETIN, H. LEFEBVRE, M. PARREAU.

PROFESSEURS HONORAIRES des Anciennes Facultés de Droit et Sciences Economiques, des Sciences et des Lettres

M. ARNOULT, Mme BEAUJEU, MM. BROCHARD, CHAPPELON, CHAUDRON, CORDONNIER, CORSIN, DEHEUVELS, DEHORS, DION, FAUVEL, FLEURY, P. GERMAIN, HEIM DE BALSAC, HOCQUETTE, KAMPE DE FERIET, KOUGANOFF, LAMOTTE, LASSERRE, LELONG, Mme LELONG, MM. LHOMME, LIEBAERT, MARTINOT-LAGARDE, MAZET, MICHEL, NORMANT, PEREZ, ROIG, ROSEAU, ROUBINE, ROUELLE, SAVART, WATERLOT, WIEMAN, ZAMANSKI.

PRESIDENTS HONORAIRES DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

MM. R. DEFRETIN, M. PARREAU.

PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

M. J. LOMBARD.

PROFESSEURS TITULAIRES

M.	BACCHUS Pierre	Astronomie
M.	BEAUFILS Jean-Pierre	Chimie Physique
M.	BECART Maurice	Physique Atomique et Moléculaire
M.	BILLARD Jean	Physique du Solide
M.	BIAYS Pierre	Géographie
M.	BONNEMAN Pierre	Chimie Appliquée
M.	BONNOT Ernest	Biologie Végétale
M.	BONTE Antoine	Géologie Appliquée
M.	BOUGHON Pierre	Algèbre
M.	BOURIQUET Robert	Biologie Végétale
M.	CELET Paul	Géologie Générale
M.	CONSTANT Eugène	Electronique
M.	DECUYPER Marcel	Géométrie
M.	DELATTRE Charles	Géologie Générale
M.	DELHAYE Michel	Chimie Physique
M.	DERCOURT Michel	Géologie Générale
M.	DURCHON Maurice	Biologie Expérimentale
M.	FAURE Robert	Mécanique
M.	FOURET René	Physique du Solide
M.	GABILLARD Robert	Electronique
M.	GLACET Charles	Chimie Organique
M.	GONTIER Gérard	Mécanique
M.	GRUSON Laurent	Algèbre
M.	GUILLAUME Jean	Microbiologie
M.	HEUBEL Joseph	Chimie Minérale
M.	LABLACHE-COMBIER Alain	Chimie Organique
M.	LANSRAUX Guy	Physique Atomique et Moléculaire
M.	LAVEINE Jean-Pierre	Paléontologie
M.	LEBRUN André	Electronique
M.	LEHMANN Daniel	Géométrie

Mme	LENOBLE Jacqueline	Physique Atomique et Moléculaire
M.	LINDER Robert	Biologie et Physiologie Végétales
M.	LOMBARD Jacques	Sociologie
M.	LOUCHEUX Claude	Chimie Physique
M.	LUCQUIN Michel	Chimie Physique
M.	MAILLET Pierre	Sciences Economiques
M.	MONTARIOL Frédéric	Chimie Appliquée
M.	MONTREUIL Jean	Biochimie
M.	PARREAU Michel	Analyse
M.	POUZET Pierre	Analyse Numérique
M.	PROUVOST Jean	Minéralogie
M.	SALMER Georges	Electronique
M.	SCHILTZ René	Physique Atomique et Moléculaire
Mme	SCHWARTZ Marie-Hélène	Géométrie
M.	SEGUIER Guy	Electrotechnique
M.	TILLIEU Jacques	Physique Théorique
M.	TRIDOT Gabriel	Chimie Appliquée
M.	VIDAL Pierre	Automatique
M.	VIVIER Emile	Biologie Cellulaire
M.	WERTHEIMER Raymond	Physique Atomique et Moléculaire
M.	ZEYTOUNIAN Radyadour	Mécanique

PROFESSEURS SANS CHAIRE

M.	BELLET Jean	Physique Atomique et Moléculaire
M.	BODARD Marcel	Biologie Végétale
M.	BOILLET Pierre	Physique Atomique et Moléculaire
M.	BOILLY Bénoni	Biologie Animale
M.	BRIDOUX Michel	Chimie Physique
M.	CAPURON Alfred	Biologie Animale
M.	CORTOIS Jean	Physique Nucléaire et Corpusculaire
M.	DEBOURSE Jean-Pierre	Gestion des entreprises
M.	DEPREZ Gilbert	Physique Théorique
M.	DEVRAINNE Pierre	Chimie Minérale
M.	GOUDMAND Pierre	Chimie Physique
M.	GUILBAULT Pierre	Physiologie Animale
M.	LACOSTE Louis	Biologie Végétale
Mme	LEHMANN Josiane	Analyse
M.	LENTACKER Firmin	Géographie
M.	LOUAGE Francis	Electronique
Mle	MARQUET Simone	Probabilités
M.	MIGEON Michel	Chimie Physique
M.	MONTEL Marc	Physique du Solide
M.	PANET Marius	Electrotechnique
M.	RACZY Ladislas	Electronique
M.	ROUSSEAU Jean-Paul	Physiologie Animale
M.	SLIWA Henri	Chimie Organique

MAITRES DE CONFERENCES (et chargés d'Enseignement)

M.	ADAM Michel	Sciences Economiques
M.	ANTOINE Philippe	Analyse
M.	BART André	Biologie Animale
M.	BEGUIN Paul	Mécanique
M.	BKOUCHE Rudolphe	Algèbre
M.	BONNELLE Jean-Pierre	Chimie
M.	BONNEMAIN Jean-Louis	Biologie Végétale
M.	BOSCQ Denis	Probabilités
M.	BREZINSKI Claude	Analyse Numérique
M.	BRUYELLE Pierre	Géographie

.../...

M. CARREZ Christian	Informatique
M. CORDONNIER Vincent	Informatique
M. COQUERY Jean-Marie	Psycho-Physiologie
Mle DACHARRY Monique	Géographie
M. DEBENEST Jean	Sciences Economiques
M. DEBRABANT Pierre	Géologie Appliquée
M. DE PARIS Jean-Claude	Mathématiques
M. DHAINAUT André	Biologie Animale
M. DELAUNAY Jean-Claude	Sciences Economiques
M. DERIEUX Jean-Claude	Microbiologie
M. DOUKHAN Jean-Claude	Physique du Solide
M. DUBOIS Henri	Physique
M. DYMENT Arthur	Mécanique
M. ESCAIG Bertrand	Physique du Solide
Me EVRARD Micheline	Chimie Appliquée
M. FONTAINE Jacques-Marie	Electronique
M. FOURNET Bernard	Biochimie
M. FORELICH Daniel	Chimie Physique
M. GAMBLIN André	Géographie
M. GOBLOT Rémi	Algèbre
M. GOSSELIN Gabriel	Sociologie
M. GRANELLE Jean-Jacques	Sciences Economiques
M. GUILLAUME Henri	Sciences Economiques
M. HECTOR Joseph	Géométrie
M. JACOB Gérard	Informatique
M. JOURNEL Gérard	Physique Atomique et Moléculaire
Mle KOSMAN Yvette	Géométrie
M. KREMBEL Jean	Biochimie
M. LAURENT François	Automatique
Mle LEGRAND Denise	Algèbre
Mle LEGRAND Solange	Algèbre
M. LEROY Jean-Marie	Chimie Appliquée
M. LEROY Yves	Electronique
M. LHENAFF René	Géographie
M. LOCQUENEUX Robert	Physique Théorique
M. LOUCHET Pierre	Sciences de l'Education
M. MACKE Bruno	Physique
M. MAHIEU Jean-Marie	Physique Atomique et Moléculaire
Me N'GUYEN VAN CHI Régine	Géographie
M. MAIZIERES Christian	Automatique
M. MALAUSSENA Jean-Louis	Sciences Economiques
M. MESSELYN Jean	Physique Atomique et Moléculaire
M. MONTUELLE Bernard	Biologique Appliquée
M. NICOLE Jacques	Chimie Appliquée
M. PAQUET Jacques	Géologie Générale
M. PARSY Fernand	Mécanique
M. PECQUE Marcel	Chimie Physique
M. PERROT Pierre	Chimie Appliquée
M. PERTUZON Emile	Physiologie Animale
M. PONSOLLE Louis	Chimie Physique
M. POVY Lucien	Automatique
M. RICHARD Alain	Biologie
M. ROGALSKI Marc	Analyse
M. ROY Jean-Claude	Psycho-Physiologie
M. SIMON Michel	Sociologie
M. SOMME Jean	Géographie
Mle SPIK Geneviève	Biochimie
M. STANKIEWICZ François	Sciences Economiques
M. STEEN Jean-Pierre	Informatique

.../...

M.	THERY Pierre	Electronique
M.	TOULOTTE Jean-Marc	Automatique
M.	TREANTON Jean-René	Sociologie
M.	VANDORPE Bernard	Chimie Minérale
M.	VILLETTE Michel	Mécanique
M.	WALLART Francis	Chimie
M.	WERNIER Georges	Informatique
M.	WATERLOT Michel	Géologie Générale
Mme	ZINN-JUSTIN Nicole	Algèbre

Je tiens à remercier vivement :

Monsieur le Professeur Bacchus de l'honneur qu'il me fait d'avoir bien voulu présider mon jury de thèse,

Messieurs les Professeurs Carrez et Cordonnier de s'être intéressés à mes travaux et de participer à mon jury,

Monsieur le Professeur Kaiser de l'aide et des encouragements qu'il m'a prodigués tout au long de notre collaboration à l'IRIA,

Monsieur le Professeur Wulf de m'avoir accueilli dans son équipe à Carnegie-Mellon University me permettant ainsi de mener à bien ce travail.

Je remercie également mes amis et collègues de l'IRIA, du groupe Cornafion et de CMU, en particulier Madame le Professeur Jones, et, d'une manière générale tous ceux avec qui j'ai eu l'occasion de travailler dans le cadre des divers projets et études auxquels j'ai participé.

Merci à Madame Suard et Mademoiselle Loubressac qui ont assuré la réalisation matérielle de cette thèse.

TABLE DES MATIERES

INTRODUCTION

Le travail présenté dans cette thèse est centré sur la structuration des systèmes d'exploitation. La complexité et la taille des systèmes s'accroissant, il est devenu difficile de les appréhender dans leur ensemble et des méthodes de structuration sont maintenant nécessaires. De plus, la complexité des systèmes a des incidences sur leur fiabilité. La difficulté de conserver une compréhension globale d'un ensemble où les imbrications sont très complexes conduit à des erreurs et fait même souvent que leur correction tend à en générer des nouvelles. Suivre des principes de découpage qui restreignent la vue du programmeur à ce qui le concerne et même la réalisation de preuves de programmes ne sont pas des méthodes suffisantes pour réaliser un système fiable (encore qu'elles soient nécessaires). Le matériel lui-même peut être en défaut et des garde-fous doivent être mis en place pour limiter les incidences des erreurs qui en résultent. Enfin, avec le développement du partage d'informations et celui des applications où les informations manipulées sont sensibles (bases de données bancaires, contrôle de procédés complexes,...), les systèmes actuels se révèlent inadaptés. La gestion des noms y devient complexe et la protection des informations y est illusoire.

Ainsi il s'avère nécessaire de mettre en place de nouveaux mécanismes qui fournissent au concepteur et à l'utilisateur un univers plus hospitalier supportant les méthodes de structuration souhaitées.

1. Motivations

C'est un principe maintenant largement adopté qu'un système se décrit en termes d'objets. On y distingue des agents exécutant les algorithmes (et éventuellement matérialisés par des processus) sur des entités passives qui sont justement les objets. Un objet est d'abord un modèle intuitif appréhendé par le concepteur. La matérialisation de ce modèle se réalise en définissant ses opérations de manipulation. C'est le comportement de ces opérations qui décrit complètement l'abstraction ainsi mise en place. Par exemple, une pile est complètement décrite par les opérations ajouter, enlever, lire. Une telle méthode permet d'ignorer complètement les contraintes de réalisation au niveau de l'utilisation de l'objet.

Les objets peuvent être regroupés en classes d'équivalence : elles sont formées des objets sur lesquels sont définies les mêmes opérations. On parle alors de types d'objets. Un type est défini par l'ensemble des opérations caractéristiques d'une classe d'objets et l'on dit d'un objet de cette classe qu'il appartient au type. Eventuellement les opérations caractéristiques d'un type sont alors regroupées dans un module. Ces notions, intégrées dans des langages tels que CLU [LZ74] ou Alphard [Wu 74b] peuvent être retrouvées dans des structures plus anciennes telles que celles de SIMULA.

Une autre notion toute aussi importante est nécessaire pour que l'on dispose d'un outil réel de construction de système. Il s'agit de l'extensibilité. Il doit être possible de créer de nouveaux objets en termes d'abstractions déjà définies : les opérations qui caractérisent un type doivent pouvoir être décrites en termes d'opérations sur des types déjà existants et pouvoir elles-mêmes servir de support à la définition de nouveaux types. Sinon, le concepteur doit sans cesse réimplanter les algorithmes chaque fois qu'il introduit une nouvelle abstraction.

Les principes énoncés précédemment permettent une meilleure appréhension d'un problème complexe et l'utilisation d'abstractions rigoureusement définies permet la mise en place de preuves dont la portée était jusqu'alors limitée à des programmes trop simples pour être efficaces.

Les notions d'objet et de module évoquées sont le plus souvent supportées seulement par le compilateur et non par le système d'exploitation et encore moins par la machine. Ainsi, après la mise en oeuvre du système il ne subsiste que des objets très primitifs (des segments par exemple) et la structuration permise au niveau du langage n'est plus apparente. Cependant, si le système complet participait à la conservation de ces notions, un environnement plus hospitalier pourrait être mis à la disposition de l'utilisateur aussi bien que

du concepteur de systèmes lui-même. Ceci contribuerait à rendre plus aisées les modifications ou la maintenance du système.

Plus les structures mises en place sont externes, plus elles sont spécifiques. Eventuellement la réalisation d'une base de données n'utilisera pas les structures logiques que le système de gestion de fichiers fournit à l'utilisateur mais des structures plus internes utilisées justement pour réaliser les fichiers. Ainsi sa réalisation n'est pas contrainte et une organisation plus adaptée des données peut être mise en place. Dans les systèmes classiques il est rarement possible d'accéder à des structures internes et le concepteur est conduit à réinventer des mécanismes propres à son application. Une telle tâche serait évitée si après avoir été mises en oeuvre à la conception les notions relatives au découpage du système étaient maintenues à l'exécution. Ceci requiert une structure d'adressage qui matérialise la modularité des programmes et en permette le partage.

A part quelques exceptions, les systèmes actuels se présentent encore comme un tout monolithique où l'on ne distingue que, d'une part, le système d'exploitation proprement dit et d'autre part, l'utilisateur. Ceux-ci sont séparés par une barrière supposée infranchissable et matérialisée par les modes maître et esclave. Toutes les structures mises en place à la conception sont alors amalgamées d'un côté ou de l'autre de cette barrière. Le but d'un tel mécanisme est de protéger le système vis à vis de l'utilisateur. Quant à la protection des objets de l'utilisateur, elle repose sur les mécanismes que fournit le système auquel il lui est demandé de faire confiance. L'absence de structuration réelle du système et le manque de fiabilité qui en résulte conduisent à douter de l'efficacité des mécanismes de protection fournis par le système. Ceci est encore renforcé par le fait qu'éventuellement la mise en place d'une application particulière nécessite de franchir la barrière de protection . Ceci donne à son réalisateur le pouvoir de contourner les mécanismes réalisés dans le système. Linden [Li76] rapporte que pour l'année 74 il y a eu aux Etats-Unis plus de 300 cas de fraudes dont la plupart impliquaient l'accès à des informations financières. La perte moyenne dans chacun de ces cas étaient de l'ordre de 500 000 dollars. Ceci illustre le besoin d'une protection plus appropriée. La protection des informations de l'utilisateur nécessite que celles-ci soient matérialisées dans les termes où elles doivent être protégées ; elle suppose donc la conservation de la notion d'objet. Ainsi qu'il a été noté, le plus souvent la notion d'objet n'est présente dans les systèmes qu'à un niveau très rudimentaire. Il n'est possible de protéger que des fichiers, voire des segments, structures trop grossières pour répondre aux besoins réels de la protection.

On notera enfin que la fiabilité des systèmes a longtemps été abordée comme un problème de méthodes de construction et de preuves de programme. Il est certain qu'une meilleure compréhension du système appréhendé sous forme de modules de taille limitée et dont le comportement a été prouvé contribue à limiter la production d'erreurs. Il ne semble pourtant pas que les preuves de programmes permettent de les éviter toutes. Quand bien même les spécifications des modules et les preuves seraient correctes, la traduction des algorithmes peut être erronée et le matériel peut générer des erreurs. Ici encore, la conservation des modules et objets à l'exécution met en place les garde-fous nécessaires. L'existence de modules dont les connexions sont clairement définies et limitées au strict nécessaire permet d'éviter la propagation des erreurs entre modules indépendants. Ceci favorise la localisation des erreurs condition nécessaire à leur traitement. Par ailleurs, le problème de la fiabilité rejoint celui de la protection. Fournir un système fiable passe par le respect de l'intégrité de ses composants. L'intégrité d'un objet est respectée si seules les opérations définies sur son type peuvent y accéder. Alors seules sont en cause ces opérations. Inversement, il est des cas où l'effondrement du système pose un problème de protection. Ainsi un système construit pour protéger l'information doit éliminer les sources d'écroulement. Il en résulte que la protection de l'information doit tenir compte des erreurs et assurer l'intégrité des objets contre elles.

2. But de l'étude

Les remarques précédentes suggèrent une solution commune à l'ensemble des problèmes évoqués. Celle-ci repose sur le maintien à l'exécution des structures mises en place à la conception. Ainsi les objets et modules utilisés pour structurer les programmes devraient rester apparents au cours de leur exécution. On se propose ici de définir les mécanismes à mettre en place pour cela. Rappelons que ces mécanismes doivent répondre aux contraintes suivantes :

- assurer la protection des objets,
- agir uniformément sur tous les objets,
- permettre l'extensibilité du système, c'est-à-dire la création de nouvelles abstractions dans les termes des abstractions déjà mises en place,
- permettre d'assurer la fiabilité du système.

3. Plan de la thèse

Cette étude est divisée en trois parties.

La première partie est consacrée aux mécanismes de structuration. Plus précisemment, on y présente les architectures à capacités. Le chapitre I rappelle les concepts fondamentaux de la protection. Les contraintes de protection consuisent au découpage du système en termes de modules dont la matérialisation à l'exécution réalise des "domaines de protection". Celle-ci repose sur l'utilisation de la notion de capacité. Une fois ce concept présenté, nous étudions la mise en place des objets et l'extensibilité dans un tel système. Le chapitre II définit la structure d'adressage nécessaire au partage des objets. Alors les mécanismes introduits au chapitre I peuvent être développés et nous étudions leur réalisation sur la structure d'adressage proposée. Afin de montrer ses avantages nous décrivons également comment cette structure pourrait être étendue au cas des réseaux. A l'issue de la première partie, l'architecture d'un système à capacités a été complètement définie.

La deuxième partie est relative aux problèmes que posent les architectures à capacités. L'expérience des réalisations actuelles montre que ces architectures conduisent à la prolifération dans le système d'objets de petite taille. Il en résulte une mauvaise utilisation des transferts entre mémoires. De plus le partage potentiel de tous les objets rend leur gestion complexe. Le chapitre III adresse ces problèmes non encore résolus dans les réalisations actuelles et propose des solutions. Enfin, le maintien à l'exécution des modules de programmes et la méthodologie qu'impose la protection impliquent de nombreux changements de contextes d'exécution. Le chapitre IV étudie la nature de ces contextes, appelés ici environnements, et leur gestion.

La troisième partie est consacrée à l'étude de la fiabilité. Le chapitre V développe la méthodologie adoptée pour compléter l'apport des architectures à capacités. Celle-ci est basée sur la tolérance aux fautes. On y étudie également le problème du report des erreurs vers les modules appropriés. Les méthodes de découpage en modules limitent les actions possibles en cas d'erreur car la représentation des abstractions n'est pas accessible en dehors des modules qui les mettent en place. Le chapitre VI étudie ces limites et propose une méthode qui permet à l'utilisateur d'une abstraction d'en contrôler totalement la cohérence.

En conclusion nous analysons les avantages et les limites actuelles des architectures à capacités.

PREMIERE PARTIE

ARCHITECTURE A CAPACITES

CHAPITRE I

LA PROTECTION DES OBJETS

Le terme "protection" désigne l'action des mécanismes (programmés ou câblés) qui contrôlent l'accès aux objets du système. Ces mécanismes ont pour but :

- de maintenir l'intégrité des objets : ceux-ci ne doivent pouvoir être manipulés qu'en accord avec leurs spécifications, c'est-à-dire par les opérations que l'on a définies sur eux et par elles seulement.
- d'assurer la sélectivité des accès : l'ensemble des opérations permises à un utilisateur sur un objet définit les droits de cet utilisateur sur l'objet ; il doit être possible de faire varier ces droits en fonction de l'utilisateur de l'objet.

A priori, on peut assurer la protection à différentes étapes de la chaîne de traitement des programmes : à la compilation, à l'édition des liens, à l'exécution. Cependant, la protection prend tout son sens avec le partage des objets. Il en résulte des problèmes que le compilateur ne peut traiter seul. Ainsi, l'isolation des utilisateurs et le contrôle des ressources partagées ne peuvent être assurées que par le système. Eventuellement, les différentes étapes de la chaîne de traitement peuvent coopérer à la mise en place de la protection [Dar 77].

Pour être effective, la protection des objets doit être assurée de manière fiable. En conséquence, elle doit être assurée de manière uniforme sur tous les composants du système. La protection repose sur la sureté des mécanismes qu'elle met en jeu. Lorsqu'elle est assurée à l'aide de contrôles intermédiaires dont la mise en place est laissée à l'initiative du réalisateur d'une entité, la cohérence et la sureté de fonctionnement de ces divers mécanismes ne peuvent pas être contrôlées. Au contraire la protection doit être réalisée à l'aide d'un mécanisme unique. Afin d'agir sur tous les composants du système, ce mécanisme doit être implanté à son niveau le plus interne. Le mécanisme que nous allons étudier ici est un mécanisme de contrôle à l'exécution destiné à être implanté au niveau de la machine. Ceci devra être complété par la définition d'une stratégie de protection exprimée dans un langage approprié [JL 78]. Enfin, on devra prouver que cette stratégie est correcte.

1. - Problèmes de protection

1.1. - Méfiance mutuelle

Le problème de méfiance mutuelle relève du partage de procédure. Il est donc relatif à la construction modulaire et à l'extensibilité du système.

Supposons que l'on rende public un nouveau sous-système et que celui-ci soit formé d'un ensemble de procédures et d'objets. Le concepteur de ce sous-système souhaite qu'il conserve son intégrité. Pour cela, il ne doit être utilisé qu'à l'aide des procédures mises en place. Il en est de même des objets propres au sous-système ; ils ne doivent être accessibles qu'aux procédures du sous-système. Ces exigences conduisent à réaliser l'isolation du système et des utilisateurs à l'aide du mode maître. Mais, supposons que l'on veuille construire ce système de manière extensible et permettre la construction de nouveaux sous-systèmes en termes des opérations fournies par les sous-systèmes déjà existants. Alors chaque sous-système de niveau plus interne doit être protégé de ceux qui l'utilisent. Multics [Gra 68, Sal 74, Cro 75 Chap. 5] fournit un exemple d'une telle réalisation dans laquelle chaque sous-système nouveau a un pouvoir inférieur (i.e. numéro d'anneau supérieur) à ceux des sous-systèmes qu'il utilise. Cependant, cette structure conduit à regrouper plusieurs sous-systèmes avec le même degré de protection (sous le même pouvoir ou numéro d'anneau).

A l'inverse, l'utilisateur d'une procédure (ou d'un sous-système) exécute celle-ci dans son propre espace. En général, cette procédure s'exécute avec le même pouvoir que le programme utilisateur ; éventuellement si elle appartient au système, elle s'exécute avec un pouvoir supérieur. Une telle situation est fréquente quand un programme utilise des procédures écrites par d'autres utilisateurs. Il est ainsi possible de faire réaliser à ces procédures des actions qui ne correspondent pas à leur spécification ni aux intentions de leur utilisateur. On peut imaginer que ces procédures, ayant inutilement accès aux objets de l'utilisateur y réalisent des actions malveillantes. Plus simplement, ces procédures peuvent être erronées et porter atteinte à l'intégrité de ses objets. Ce problème, connu sous le nom de "problème du Cheval de Troie" [Bra 73] ne semble pas pouvoir être totalement résolu. Sa solution suppose :

- que seuls les objets transmis en paramètres soient accessibles au sous-système,

- que le sous-système réalise sur ces objets les opérations spécifiées et elles seules,
- que le sous-système ne puisse divulguer ou mémoriser l'information qui lui est transmise (cf. 1.3) en paramètre.

La malveillance potentielle d'un sous-système devrait être limitée aux seuls objets transmis en paramètres par l'utilisateur (ou un autre sous-système). Une telle exigence est rarement réalisée dans les systèmes actuels. Ainsi, la relation d'ordre qui existe entre les anneaux de Multics permet à une procédure exécutée dans l'anneau n d'accéder à tous les objets protégés dans des anneaux de numéro supérieur à n.

La combinaison des deux sortes de contraintes présentées ici forme le "problème de la méfiance mutuelle" [Sch 72]. Il exprime d'une part la méfiance du sous-système vis-à-vis des malveillances potentielles de ce sous-système. La coopération entre sous-systèmes mutuellement méfiants exige :

- l'isolation réciproque des sous-systèmes,
- le contrôle de l'accès au sous-système en des points d'entrée spécifiés qui définissent les opérations mises en place par ce sous-système,
- le contrôle des paramètres reçus par le sous-système,
- la possibilité pour un sous-système de restreindre les droits d'accès aux objets qu'il transmet en paramètres à d'autres sous-systèmes.

1.2. - Révocation des droits

Initialement le créateur d'un objet possède toutes les opérations définies sur cet objet. Le partage sélectif des objets entre utilisateurs ou sous-systèmes nécessite que ces droits puissent être transmis et révoqués :

- D'une part, le créateur d'un objet doit pouvoir transférer aux utilisateurs de son choix les droits qu'il possède sur cet objet en restreignant éventuellement les possibilités de manipulation (en diminuant les droits) à un sous-ensemble des opérations définies sur lui. Cette possibilité devrait être étendue aux sous-systèmes méfiants ; un tel sous-système devrait ainsi pouvoir restreindre les droits d'utilisation d'un objet qu'il transmet en paramètre à un autre sous-système.
- D'autre part, les droits concédés sur un objet doivent éventuellement pouvoir être retirés sélectivement à tout instant. Ceci est appelé le problème de la révocation des droits.

La révocation des droits peut être immédiate ou différée. Dans les systèmes actuels, celle-ci est réalisée à travers les catalogues. Il en résulte qu'elle n'a pas d'effet sur les objets déjà liés à l'espace d'exécution d'un programme. On dit qu'elle est différée. Dans la mesure où l'on veut permettre la coopération de sous-systèmes mutuellement méfiants et la variation des droits, la révocation devrait avoir un effet immédiat. Dans un système à partage sélectif, la distribution des droits est nécessaire, la révocation ne l'est pas. De plus, la révocation nécessite des mécanismes d'adressage et de protection particuliers [Sch 72, Coh 75, Red 74] car elle impose d'être capable de retrouver toutes les chaînes d'accès à l'objet. Le problème de la révocation ne sera pas abordé ici.

1.3. - Etanchéité

Le problème d'étanchéité a été introduit par Lampson [Lam 73].

Schématiquement, un sous-système est dit totalement étanche lorsqu'il lui est impossible de conserver trace de l'information qui lui a été communiquée à travers les objets reçus en paramètres. On peut toujours définir des mécanismes qui interdiront la copie d'information par le sous-système. Cependant, Lampson exhibe des procédés de divulgation de cette information qui sont plus subtils et qui sont plus difficiles à contrôler. Par exemple, l'ouverture et la fermeture d'un fichier à un rythme convenu est un moyen de divulguer à l'extérieur du sous-système l'information acquise. D'une manière générale, le problème de l'étanchéité requiert des méthodes de preuve de programme. Une autre forme d'étanchéité, l'étanchéité sélective a été étudiée [Den 74]. Elle consiste à ne contrôler que les paramètres définis comme confidentiels par l'utilisateur de sous-système. Le problème d'étanchéité reste un problème ouvert qui ne sera pas abordé dans cette étude.

2. - Notion de domaine

Dans [Lam 71], Lampson met en évidence l'existence de domaines de protection. Au cours de son exécution, un programme ne conserve pas le même pouvoir d'accès sur les objets qu'il utilise ; on exprime cela en disant qu'il s'exécute dans des domaines de protection différents. Un domaine est un ensemble d'objets et de droits d'accès à ces objets à un instant donné. Si par exemple, un programme utilise un fichier, les droits qu'il possède sur ce fichier lui permettent sans

doute d'y lire ou d'y écrire ; par contre, au cours de l'exécution de ces opérations, la représentation du fichier devient accessible. L'exécution de ce programme se fait dans deux domaines : celui où le fichier n'est visible qu'à travers des opérations de lecture ou d'écriture sur fichier, et celui où ces opérations se réalisent. Dans un système classique de construction monolithique, tel que Siris 8, un programme ne s'exécute que dans deux domaines : celui de l'utilisateur, et celui du système où tout objet devient accessible. Ainsi une relation d'ordre est définie entre ces deux domaines et leur méfiance mutuelle ne peut être assurée.

Les contraintes d'extensibilité évoquées en introduction de même que le principe de méfiance mutuelle requièrent que cette notion de domaine soit généralisée de la manière suivante :

- le système doit pouvoir être découpé en autant de domaines qu'il réalise de fonctions distinctes,
- le même mécanisme de protection doit être mis à la disposition de l'utilisateur,
- les domaines ainsi définis doivent être indépendants, en particulier, il ne doit pas y avoir de relation d'ordre entre domaines, c'est une condition nécessaire pour être capable de réaliser la méfiance mutuelle.

Selon le principe de méfiance mutuelle, tout sous-système s'exécutant dans un domaine ne doit avoir accès qu'aux objets décrits dans ce domaine et ceci avec les droits qui sont relatifs à ce domaine. La définition d'un domaine a donc une incidence sur les mécanismes de désignation des objets. D'une part, seuls les objets du domaine doivent pouvoir être désignés, d'autre part, les droits d'utilisation d'un objet par le sous-système doivent être contrôlés. Les domaines doivent donc être indépendants entre eux et les seules communications entre domaines ont lieu par transmission de paramètres.

Ainsi, les opérations définies sur des objets différents sont réalisées dans des domaines distincts, c'est ce qui permet de contrôler leur utilisation et d'assurer l'intégrité des objets qu'elles manipulent. Le partage d'objets avec des droits différents entraine leur utilisation dans des domaines différents. De même, l'exécution d'une procédure génère des objets locaux propres à cette exécution ; par définition, il en résulte que chaque exécution d'une même procédure a lieu dans un domaine distinct.

Enfin, les changements de domaines doivent être contrôlés. Dans l'exemple cité en début de ce paragraphe, c'est réalisé en des points d'entrée du système qui sont mis en place à l'aide d'opérations du genre "appel superviseur". La notion d'objet permet d'assurer l'ensemble des contrôles de manière uniforme, en traitant le domaine comme un objet. Le changement de domaine requiert alors la présentation d'un objet décrivant le domaine appelé et les droits appropriés.

3. - Méthodologie

On se propose d'étudier ici comment se définissent les domaines de protection. Le paragraphe précédent suggère que cela se fasse en termes d'opérations sur des objets, chaque invocation d'une opération engendrant un domaine. La structuration d'un système protégé semble donc équivalente à la décomposition d'un système en termes d'objets et d'opérations pour laquelle différentes méthodes sont déjà connues [Pa 72a, Ha 76, 78] et éventuellement introduites dans des langages [LZ 74, Wu 76a, 76b, Mos 77]. Selon ces méthodes, le système s'organise en termes de modules. En première approximation, un module regroupe des procédures et les objets qu'utilisent ces procédures. Avec les paramètres que reçoivent les procédures, ces objets sont les seuls accessibles dans le module. La notion de module permet ainsi la programmation de sous-systèmes mutuellement méfiants :

- en regroupant dans un même module les opérations du sous-système,
- en faisant de ces opérations les seules procédures du module que l'on puisse invoquer à l'extérieur du module.

On peut alors se demander si la structure qui résulte de l'application de ces méthodes s'identifie aux besoins de la protection.

Reprenons à nouveau l'exemple précédent. La complexité du système de gestion de fichiers conduit sans doute à le découper en plusieurs modules. Deux solutions extrêmes peuvent être envisagées : l'une consiste à considérer l'ensemble des modules participant à la gestion des fichiers comme s'exécutant dans un seul domaine, l'autre consiste à ce que chaque module intervenant dans la décomposition du système de gestions de fichiers engendre un domaine de protection distinct. Mis à part les contraintes d'extensibilité qui requièrent de permettre le partage de chacun des modules entrant dans cette décomposition, d'autres remarques conduisent à adopter cette seconde solution.

Protection et fiabilité sont deux thèmes difficilement dissociables. La protection exige la fiabilité des mécanismes qui l'assurent mais la fiabilité nécessite de son côté l'existence d'un mécanisme de protection. Afin d'améliorer la fiabilité d'un système complexe, il convient de le décomposer en modules de

petite taille dont les interactions sont clairement définies. Cependant, il n'est pas toujours facile de prévoir les interactions possibles entre ces modules ; en particulier ces interactions changent justement lorsque l'un des modules se comporte anormalement. Ainsi, si l'un des modules est erroné, c'est l'ensemble du système auquel il participe qui peut se comporter anormalement. Or il est impossible d'éviter toutes les causes de mauvais fonctionnement d'un programme. Une méthode de programmation "défensive" peut éventuellement limiter les effets d'un mauvais fonctionnement potentiel. Une telle méthode consiste à insérer dans les programmes des tests de détection d'erreur et à vérifier que les modules interagissent comme prévu. Elle peut conduire à une complexité plus grande encore du programme. Une solution plus générale consiste à introduire un mécanisme de protection défini indépendamment du découpage fonctionnel en termes de modules. La réalisation de modules fiables se définit exactement comme un problème de méfiance mutuelle et le rôle du mécanisme de protection est d'éviter la propagation des erreurs au-delà des modules où elles se produisent. Pour que l'effet de protection recherché soit atteint, chaque procédure devra s'exécuter dans un domaine propre. On appelle ceci le principe du moindre privilège ; il conduit à un découpage en modules aussi simples que possible et à l'exécution des programmes dans des "petits domaines". L'invocation d'une opération d'un module génère un nouveau domaine.

Les avantages de l'observation du principe du moindre privilège portent sur :

- la correction des fautes de programmation : celles-ci sont plus faciles à trouver car les erreurs dans un module se traduiront moins probablement par un comportement anormal d'un autre module. Par ailleurs il est également moins probable que la correction d'une erreur dans un module introduise des erreurs dans d'autres modules,
- le test du bon fonctionnement d'un programme : il est plus facile de tester un module à la fois. De plus, puisque l'espace d'exécution du module est clairement défini à l'exécution, les interactions non prévues dans les tests sont évitées,
- la modification des programmes ; les interactions entre modules étant clairement définies, les modules qui sont affectés par une modification sont également clairement définis,

- la correction des erreurs : pour qu'une erreur puisse être corrigée efficacement, il importe que les dommages causés puissent être limités et clairement établis. L'utilisation de petits domaines contribue à permettre la détection des erreurs le plus tôt possible et limiter ainsi leur action,
- la preuve des programmes : la complexité des preuves croît plus vite que la longueur des programmes. La décomposition en modules simples dont les interactions sont clairement définies permet de simplifier les preuves à condition que ces modules soient indépendants.

4. - Notion de capacité

Si l'on résume les paragraphes précédents, les caractéristiques que doit présenter un mécanisme adapté à la réalisation de systèmes protégés sont les suivantes :

- assurer l'intégrité des objets,
- assurer la sélectivité des accès,
- mettre en place la notion de domaine.

L'intégrité des objets et la sélectivité peuvent être réalisés en plaçant un intermédiaire entre l'objet et son utilisateur. On peut imaginer utiliser des procédures protégées qui, agissant en intermédiaire, permettent de contrôler que l'objet est utilisé correctement et que l'utilisateur possède les droits requis pour y accéder. D'une part, on doit faire en sorte que les contrôles réalisés par les intermédiaires ne puissent être évités par l'utilisateur ; d'autre part, les intermédiaires eux-mêmes doivent être protégés. Enfin, l'intervention des contrôleurs intermédiaires résulte en un ralentissement important des performances.

Différentes implantations de la notion de domaine ont été réalisées ou proposées. On peut citer la structure en anneaux de Multics [Sal 74], les structures d'adressage hiérarchiques de Gemau [GR 75] ou de [Pr 73] et la réalisation de Spier [Sp 73]. Cependant le concept qui seul permet de répondre à l'ensemble des exigences d'un mécanisme de protection approprié est la notion de capacité [DVh 66, Fab 68].

4.1. - Définition

Rappelons qu'un objet est caractérisé par les opérations qui permettent de le manipuler et sa représentation. Une capacité est une référence à un objet (à sa représentation) qui décrit également les opérations autorisées sur l'objet sous contrôle de cette capacité.

L'accès à un objet ne peut se faire que par l'intermédiaire d'une capacité. Ainsi toute tentative de manipulation d'un objet par une opération qui n'est pas spécifiée dans la capacité de l'objet est détectée. La sélection d'un objet par un programme dans son espace d'exécution se fait en désignant la capacité qui repère cet objet. Celle-ci est le premier maillon de la chaîne d'accès à l'objet. Ceci implique que la modification des capacités soit strictement contrôlée. En particulier il ne doit pas être possible de modifier le repère qu'elle contient ni les droits. Les capacités doivent donc elles-mêmes être protégées.

La nécessité d'utiliser une capacité pour accéder à un objet, présente les avantages suivants :

- elle facilite la construction et la protection d'ensembles d'objets car seuls les objets dont il existe une capacité dans l'espace d'exécution peuvent être désignés,
- elle permet de traduire la variation des droits sur un objet en fonction du domaine où s'exécute l'utilisateur,
- elle introduit une séparation explicite entre l'objet et les opérations sur l'objet,
- enfin, elle permet, comme on le verra, de protéger de manière uniforme toutes les entités du système.

4.2. - Manipulation des capacités

Nous avons noté la nécessité de protéger les capacités. Ce sont les structures élémentaires à l'aide desquelles sont mises en place les structures plus complexes fournies par les modules du système ou des utilisateurs. La partie du système qui gère les capacités (et entre autres les crée) est donc nécessairement la plus protégée. Elle constitue le domaine de protection initial dans le système. On l'appelle le noyau de protection. La réalisation d'un

mécanisme unique permettant d'assurer de manière uniforme la protection de toutes les entités du système contribue à diminuer la taille du noyau de protection. Ainsi la contrainte citée en introduction selon laquelle la sûreté du noyau de protection doit pouvoir être prouvée se trouve facilitée.

La protection des capacités peut être réalisée de deux façons :

- soit chaque emplacement de mémoire est muni d'un préfixe qui indique si son contenu est ou non une capacité ; c'est la méthode appliquée dans la machine Burroughs 6700 [Cro 75 chap. 3] et dans celle de l'université Rice [Feu 73] ;
- soit les capacités sont regroupées dans des segments particuliers appelés capacitifs ou C-listes et qui sont eux-mêmes protégés : alors l'objet élémentaire de protection est en fait la C-liste.

L'avantage de la deuxième solution est d'imposer un regroupement des capacités, son intérêt apparaîtra dans le chapitre III. Alors, on distingue deux classes d'objets primitifs réalisés par la machine, les C-listes destinées à assurer la protection et les segments de données destinés à recevoir l'information. Un segment est un objet sur lequel sont définies les opérations lire, écrire, exécuter. La notion de capacité a été introduite initialement dans des machines à mémoire segmentée ; la table des segments qui dans Multics est associée à chaque processus est un exemple de C-liste. Dans une telle optique, la capacité intervient à l'insu du processus au cours du calcul d'adresse et n'est pas un objet manipulable explicitement sauf par le noyau de protection. Au contraire, la réalisation de sous-systèmes méfiants requiert que l'on puisse réduire au strict nécessaire les droits sur un objet transmis en paramètre. Alors les capacités deviennent des objets manipulables par tous les domaines. Afin d'être capable de contrôler sélectivement l'utilisation des capacités, il convient de définir des droits de manipulation des capacités de la même manière que pour les autres objets. Aussi, outre des droits d'accès à l'objet qu'elle repère, une capacité doit contenir des droits relatifs à son utilisation. Ces mêmes droits permettent éventuellement de mettre en place le mécanisme assurant l'étanchéité. Hydra en fournit un exemple [Coh 75]. Plus simplement le contrôle d'utilisation des capacités peut être réalisé en faisant de la C-liste un objet sur lequel on définit des opérations de lecture et d'écriture. Ainsi la copie d'une capacité exige le droit de lire la C-liste source et le droit d'écrire dans la C-liste destinataire. Une copie avec restriction des droits permet de transmettre librement les droits sur un objet entre sous-systèmes.

5. - Notion d'environnement

Rappelons les principes de réalisation de la protection que nous avons présentés précédemment. Ceux-ci supposent que le système soit découpé en modules aussi petits que possible. Un module est, pour l'utilisateur, essentiellement un ensemble d'opérations (des procédures). L'invocation d'une opération d'un module provoque la mise en place d'un nouveau domaine indépendant. Ainsi, le contrôle des changements de domaines se ramène au contrôle de l'invocation des opérations ; il est réalisé en utilisant l'objet procédure dont l'invocation requiert la présentation d'une capacité. Un objet procédure est formé d'un ensemble d'objets : les segments de données qui en contiennent le code et les objets qui lui sont associés en permanence. Ainsi une procédure peut se représenter initialement à l'aide d'une C-liste. L'opération définie sur l'objet procédure est l'opération d'appel qui reçoit les capacités des paramètres de l'appelant et réalise l'activation de la procédure appelée. Nous reviendrons ultérieurement (chapitre IV) sur la définition de l'"objet" procédure. L'activation de la procédure se traduit par la mise en place de son espace d'exécution ; il se matérialise par une C-liste destinée à recevoir les capacités des paramètres et des objets locaux créés au cours de son exécution. On appelle cette C-liste l'environnement de la procédure. De plus, la capacité de la C-liste représentant la procédure y est chargée au moment de l'appel de sorte que ses objets permanents restent accessibles. On remarque que l'environnement définit l'espace d'adressage de la procédure dans lequel celle-ci désigne par l'indice de leur capacité (on dit aussi le "nom"), les objets qu'elle manipule. La figure 1 schématise l'environnement d'une procédure. Par définition, il matérialise le domaine associé à l'exécution de la procédure. L'indépendance des domaines,nécessaire à la réalisation de la méfiance mutuelle,est ainsi assurée par l'indépendance des espaces d'adressage. Parmi les objets permanents de la procédure, certains peuvent être des C-listes ou des procédures qui décrivent les opérations possibles. L'environnement définit donc en fait la racine d'une arborescence d'objets accessibles dont l'ensemble forme un domaine de protection.

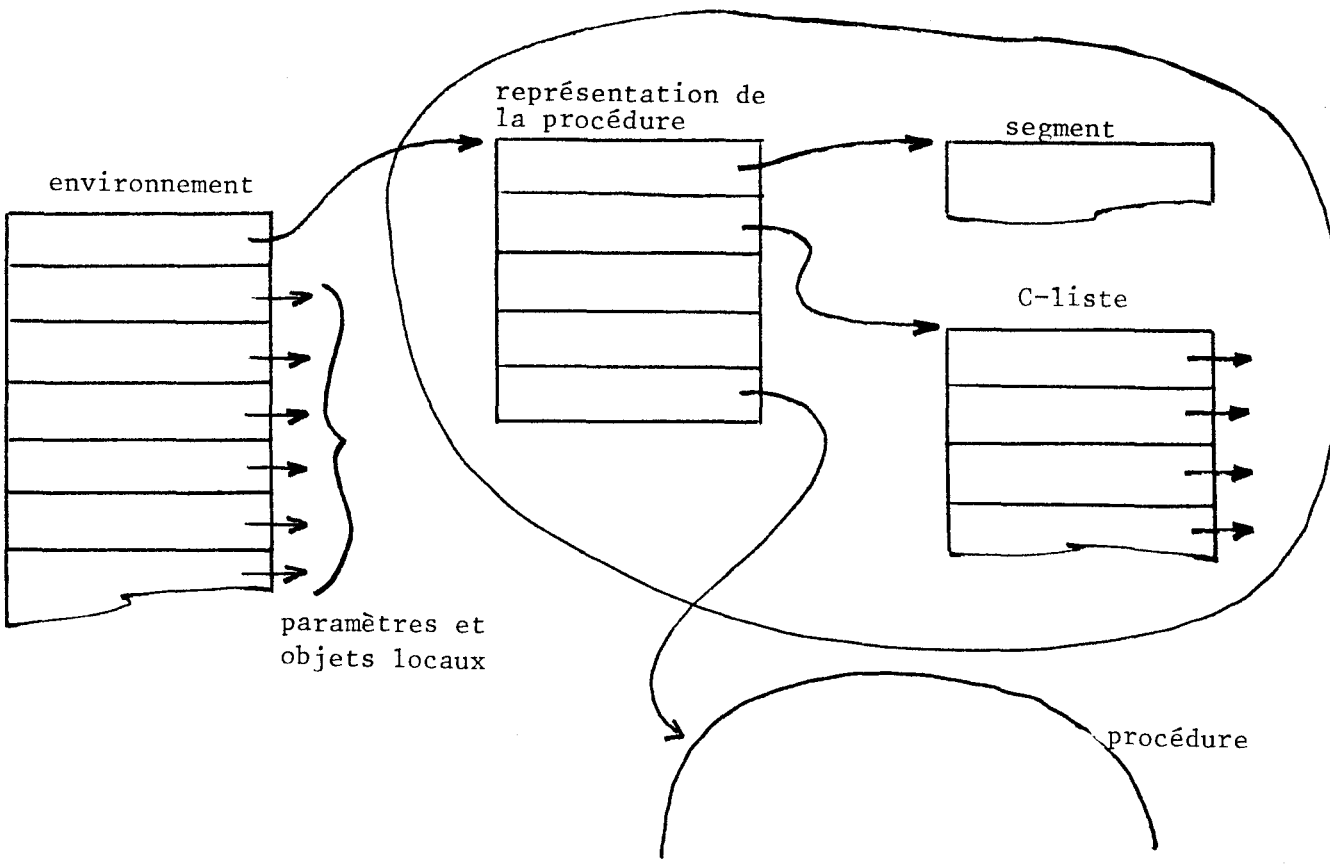


Figure 1 : Représentation d'un domaine

Un module est une structure de regroupement d'opérations. Il peut donc être représenté par une C-liste contenant les capacités des procédures qui réalisent ces opérations. Selon le schéma précédent, il n'est pas nécessaire de protéger le module ni d'en faire un objet puisqu'il ne participe pas à la réalisation de la protection.

Une autre solution consiste au contraire à protéger le module et non les opérations, qui ne sont alors accessibles qu'à travers l'objet module. Selon ce schéma, l'invocation d'une opération nécessite la possession d'une capacité du module. Elle se réalise en désignant l'opération par l'indice de sa capacité dans la C-liste qui représente le module.

A l'extrême, il n'est plus nécessaire de distinguer les représentations des différentes opérations du module. Dans Cm* [KL 78, Jo 77, Jo 78] un module est représenté par une C-liste regroupant tous les objets du module. L'un d'eux est un segment qui contient les indices, dans la C-liste, des capacités des opérations. Le nom d'opération connu extérieurement et fourni par l'appelant, est utilisé pour retrouver dans ce segment l'indice, dans la C-liste, de la capacité de l'opération. Dans la Plessey 250 [Eng 74, FL 76, Ban 78] une convention de programmation réserve éventuellement les premiers emplacements de la C-liste aux capacités des segments de code des procédures. Les opérations

d'un module sont désignés explicitement par l'utilisateur du module à l'aide de l'indice de leur capacité. L'environnement créé à l'appel contient alors la capacité de la C-liste représentant le module.

6. - Construction d'objets - Exemple : le système Plessey 250

Les notions introduites jusqu'à présent nous permettent de définir des opérations et de protéger des objets primitifs. La protection d'objets plus complexes construits par programmes requiert que la représentation de ces objets ne soit accessible que par les opérations que l'on a définies sur eux. Nous appellerons ces objets, objet construits, par opposition aux objets primitifs interprétés et protégés par la machine. Un objet construit est bâti en termes d'objets déjà existants, éventuellement eux aussi fournis par logiciel. Ceci permet d'assurer l'extensibilité du système qui peut être bâti en termes de sous-systèmes décrits les uns au-dessus des autres et protégés. Dans cette optique, un objet construit a pour représentation une C-liste contenant les capacités de ses composants. La protection d'un tel objet consiste donc à n'autoriser l'accès de cette C-liste qu'aux opérations définies sur l'objet. La Plessey 250 fournit un exemple d'une réalisation de ce type où les objets sont construits comme des modules. Tous les objets de même structure sont des instances d'un même module. On dira que ces objets qui ont les mêmes opérations sont du même type. Ainsi chaque objet construit a pour représentation une C-liste qui outre ses composants contient les procédures qui réalisent les opérations définies sur lui et les objets qu'elles utilisent. L'utilisation d'un tel objet se fait en invoquant directement l'une des opérations du module qui le réalise. Alors un environnement est créé et l'opération a accès aux composants de l'objet en même temps qu'à ses propres objets. La figure 2 schématise la structure d'un objet. Le schéma peut se répéter, les composants de l'objet peuvent eux-mêmes être des objets construits réalisés par une structure de module. La figure 3 donne l'exemple d'un sémaphore où la C-liste contient la représentation du sémaphore mais aussi la liste des processus actifs sur le processeur. Le contrôle des accès est réalisé en protégeant le module qui représente l'objet. La seule action possible sur ce module est l'invocation d'une opération. Tandis que la sélectivité des accès est réalisée par la présence ou l'absence dans la représentation de l'objet de la capacité des segments de code des procédures. Ainsi la diminution des droits sur un objet peut être obtenue à l'aide d'une opération qui créerait

une nouvelle copie du module représentant l'objet en n'y conservant que les capacités d'un sous-ensemble d'opérations. Dans une certaine mesure la capacité réelle d'un objet construit est matérialisée par la C-liste du module qui le représente.

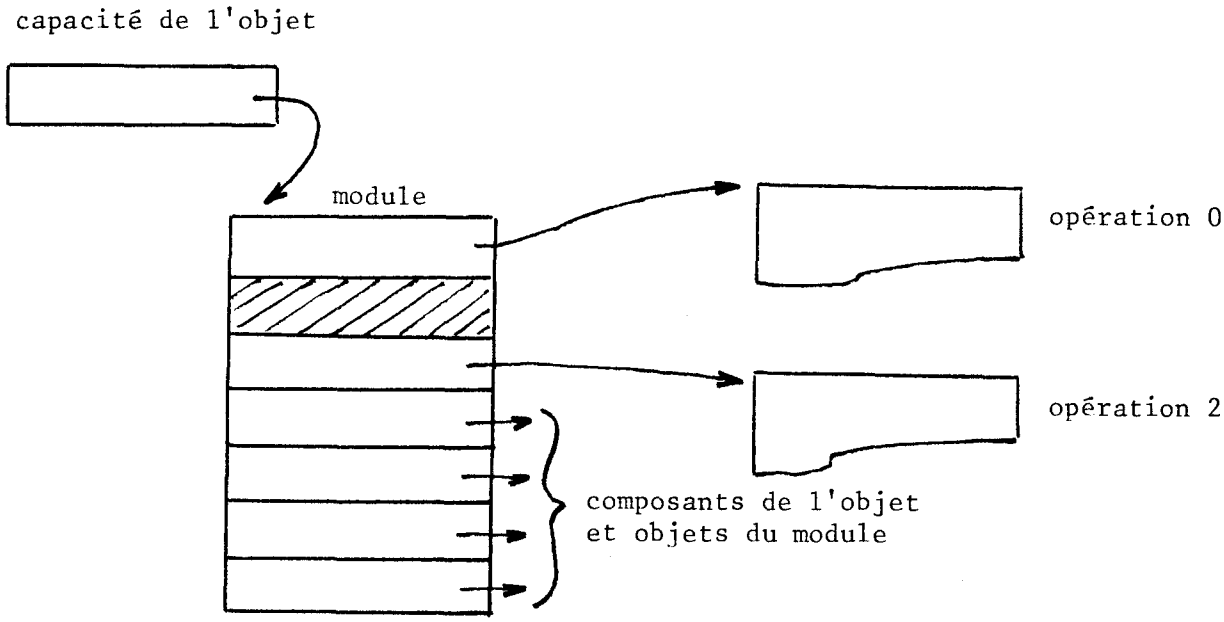


Figure 2 : Structure d'un objet construit dans le système Plessey 250

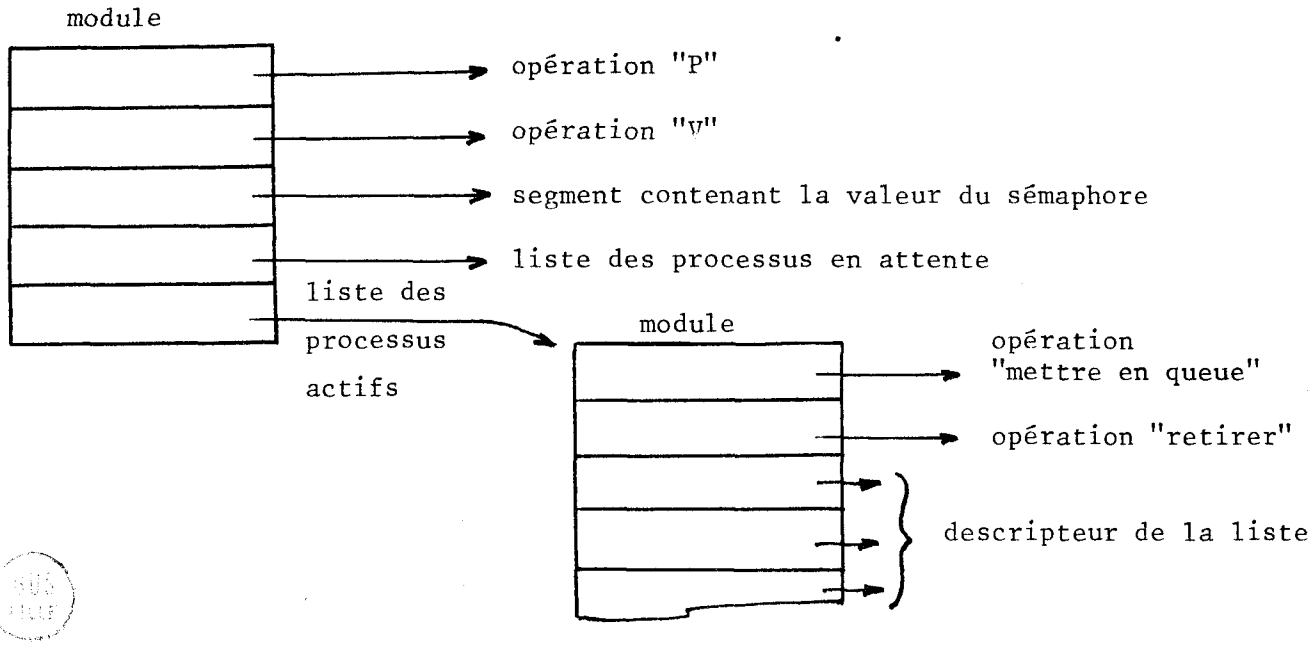


Figure 3 : Représentation possible d'un sémaphore dans le système P250

La critique d'un tel mécanisme porte sur les possibilités de structuration en termes d'opérations [FL 76]. On remarque qu'il est impossible de construire des opérations multiadiques recevant en paramètres plusieurs objets du même type. On songera par exemple à la réalisation d'une opération de multiplication de deux matrices qui nécessiterait d'une part la réalisation d'opérations entre scalaire et matrice et d'autre part la définition d'un module indépendant recevant en paramètre des matrices et utilisant ces opérations sur scalaires. Le manque de souplesse d'une telle méthode peut s'avérer contraignant tout en n'étant pas profitable sur le plan de la protection.

7. Types d'objets

7.1. Définition

Le paragraphe précédent introduisait la notion de type. On appelle type la classe d'équivalence à laquelle appartiennent tous les objets sur lesquels sont définies les mêmes opérations.

On distingue les types primitifs qui préexistent dans le système et sont reconnus par le câblé. Ceux-ci sont utilisés pour en construire d'autres de proche en proche que l'on appelle types construits et dont les opérations sont définies par programme.

Un objet de type construit a toujours pour représentation une C-liste qui contient les capacités des objets qui entrent dans sa composition et que l'on appelle ses composants.On rappelle que la protection des objets de type construit est obtenue en masquant cette représentation de sorte qu'elle ne soit accessible qu'aux opérations du type. L'opération qui fournit l'accès à la représentation d'un objet de type construit est appelée décomposition. On remarquera que la méthode de construction présentée précédemment à propos de la Plessey 250 confond décomposition et appel de procédure. La décomposition d'un objet est obtenue en rangeant dans l'environnement créé à l'appel d'une opération la capacité de la C-liste du module qui dans ce système est également la représentation de l'objet. Les limitations observées proviennent de cette méthode. Pour éviter ces limitations, il convient de dissocier la création de l'environnement et la décomposition des objets. Cependant, il est nécessaire d'associer à l'objet les opérations qui ont été définies sur lui ; c'est la condition pour être capable d'en maintenir l'intégrité et d'assurer la protection. Afin de le permettre, on introduit explicitement la notion de type [Gra 72, Wu 74a, Mo 73]. Dès lors,

la capacité d'un objet contient une marque de type. La marque de type que porte un objet est un nom unique dans le système destiné à identifier de manière non ambigue le type auquel il appartient. Seules les opérations définies sur son type (par la suite on les désignera par "opérations du type") sont autorisées à décomposer un objet de type donné.

7.2 Décomposition des objets

La décomposition d'un objet a pour but de donner à l'opération qui la réalise accès à sa représentation. De ce fait, elle fournit à cette opération l'objet C-liste qui sert de support à cette représentation et doit donc lui retourner une capacité pour cette C-liste. La décomposition d'un objet doit ne pouvoir être réalisée que par les opérations de son type. Afin de contrôler ceci, on introduit un nouvel objet, que momentanément nous appellerons maquette de transformation et qui est fourni aux seules opérations du type.

D'une manière générale, l'ensemble des contrôles qui doivent être effectués sur un objet transmis en paramètre à une opération d'un type porte sur :

- le type de l'objet, qui doit être identique à celui dont l'opération participe à la réalisation,
- les droits d'utilisation de l'objet qui doivent permettre l'invocation de l'opération.

Alors seulement l'objet peut être décomposé et l'opération exécutée. Ainsi une maquette de transformation devrait définir :

- le type auquel elle est associée,
- les droits requis sur les objets,
- la décomposition à effectuer.

Dans [Fe 74], un mécanisme câblé de ce type est présenté. Cependant, il convient de faire quelques remarques. La maquette de transformation n'est utilisée que par les opérations du type et le codage des droits associés à chaque opération est le résultat d'une convention de programmation lors de la création du type (ils ne définissent pas nécessairement par exemple chacun une opération). Par conséquent, il n'est pas besoin de les faire apparaître dans la maquette de transformation elle-même, ils peuvent être présentés comme une donnée lors de la décomposition. On notera également que la décomposition conduit toujours à une C-liste. De plus, celle-ci étant utilisée dans le sous-système qui définit le type de l'objet aucune limitation n'est nécessaire sur les droits d'accès à cette C-liste (il est d'ailleurs vraisemblable que la C-liste a été créée dans ce sous-système).

Aussi la décomposition proprement dite d'un objet est implicitement définie dès lors que le contrôle des droits a été réalisé. Il en résulte que la maquette de transformation d'un type ne fait qu'identifier ce type. Afin de mettre en place

ces maquettes, on introduit un nouveau type primitif, le type "type".

Les objets de type "type" (ou objets types) ont pour but d'identifier chacun un type. L'opération définie pour l'instant sur le type "type" est l'opération décomposer *<objet type, droits, objet>* qui réalise les contrôles et la décomposition décrits précédemment. Elle requiert la présence du droit de décomposer dans la capacité de l'objet type et retourne la capacité de la C-liste support de l'objet.

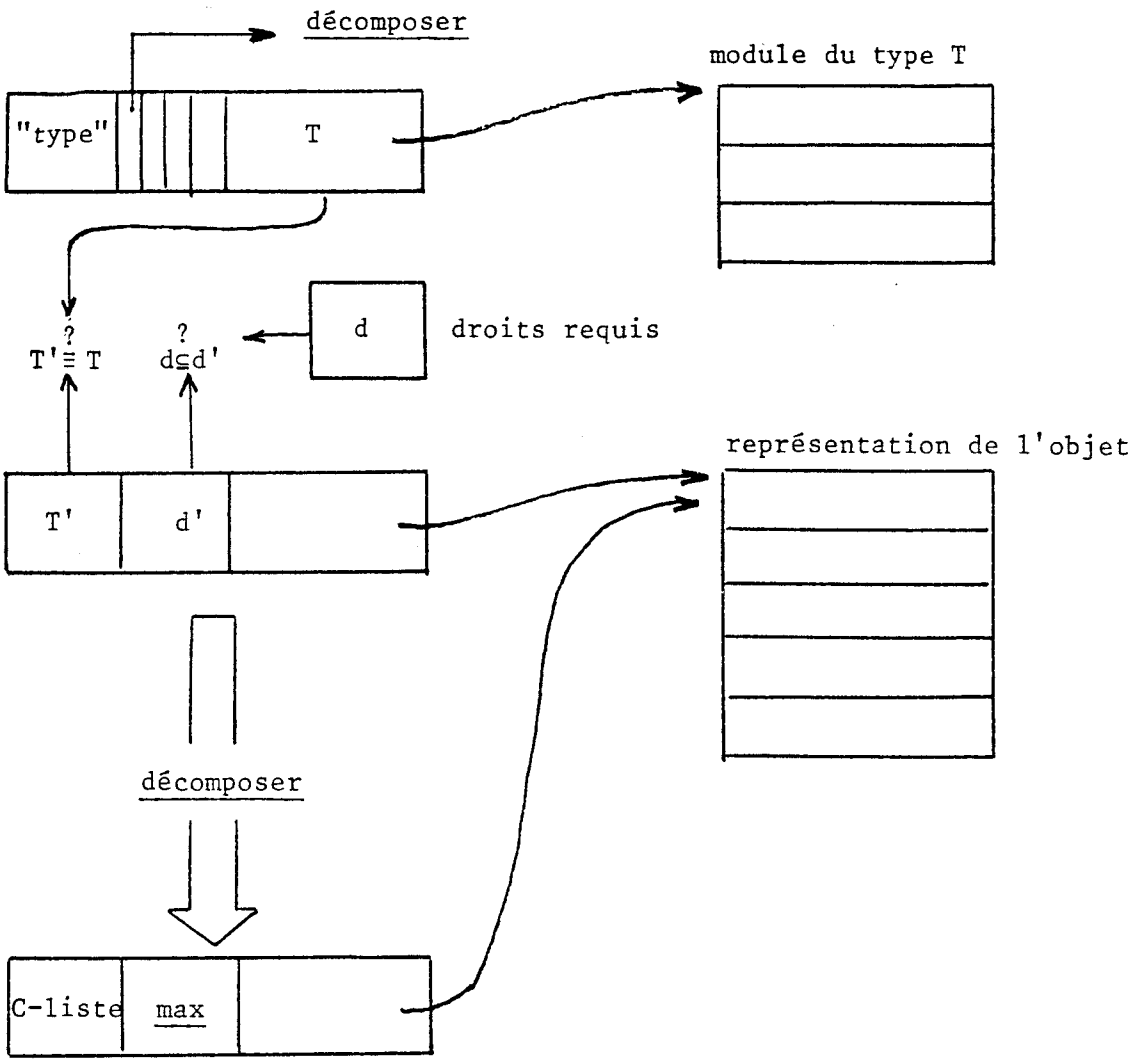


Figure 4 : Décomposition d'un objet

Un type est caractérisé par le module qui regroupe les opérations que l'on a définies sur lui. On a noté que le module ne nécessite pas d'être protégé dans

la mesure où ses opérations le sont. Il se comporte alors comme une simple structure de regroupement. Sa protection n'est nécessaire que si les opérations sont invoquées à travers l'objet module, soit pour protéger ces opérations elles-mêmes si elles ne le sont pas en tant que telles, soit pour protéger la structure du module (le respect de l'ordre de rangement des opérations par exemple si celles-ci sont nommées par l'emplacement de leur capacité dans le module). On conviendra ultérieurement de désigner les opérations à travers le module qui les regroupe (cf. chap. II et III), alors le module devient un objet dont l'utilisation doit être contrôlée ; on en fait l'objet type. L'objet type associé à un type est donc un objet construit formé à l'aide d'une C-liste, il regroupe les capacités des procédures qui décrivent les opérations du type. Les opérations définies sur ce nouveau type sont l'opération primitive décomposer déjà introduite et des opérations relatives à l'activation des opérations du type. Ces dernières, présentées ultérieurement ne sont pas contrôlées à l'aide de droits dans la capacité de l'objet type ; elles requièrent seulement d'être appliquées sur un objet type.

Hydra utilise un schéma légèrement différent de celui décrit ici [Coh 75]. L'objet type est effectivement présent. Cependant de par la structure différente des objets construits, due à des contraintes matérielles, la décomposition des objets utilise d'autres objets appelés "maquettes d'amplification" qui décrivent la structure de l'objet. De plus, la décomposition est réalisée implicitement lors de l'activation d'une procédure du type (cf. chap. IV). Ainsi l'opération décomposer sur les objets type n'a pas d'existence.

On notera qu'une opération contrôler assurant les mêmes contrôles que l'opération décomposer sans décomposer l'objet cependant permettrait à une procédure de vérifier que les paramètres effectifs sont conformes aux paramètres formels. Elle est utile dans le cas où la décomposition n'est pas réalisée par la procédure invoquée soit parce que le paramètre est de type primitif, soit parce qu'elle ne participe pas aux opérations de son type. L'équivalent est réalisé dans Hydra à l'aide des objets "maquettes de paramètres" qui, de même que les maquettes d'amplification, décrivent les paramètres formels d'une procédure.

7.3. Création des objets

La création des objets d'un type donné nécessite la connaissance de leur structure en termes de composants. Celle-ci est totalement définie par le type

auquel l'objet à créer appartient, tout au moins initialement et un générateur peut être associé à chaque type. Hydra utilise à cet effet un objet appelé maquette de création. Cependant, la structure d'un objet peut être complexe, en particulier ses composants peuvent être eux-mêmes des objets de type construit. Aussi, il n'est pas possible de la décrire complètement dans une maquette de création et celles-ci ne sont utilisées que pour définir des structures élémentaires. La création complète de l'objet est réalisée par une opération créer-objet fournie par le type au même titre que les autres opérations du type et qui utilise une maquette de création qu'elle seule possède. De même que les autres maquettes, la maquette de création est utilisée implicitement lorsque l'opération créer-objet du type est invoquée. Elle fournit une capacité pour le nouvel objet ; dans cette capacité tous les droits sur l'objet sont accordés.

Le fait de ne pas pouvoir créer directement un objet à l'aide d'une maquette a des conséquences sur l'efficacité du système. En effet, la création de chacun des composants de l'objet nécessite à nouveau l'invocation des opérations de création correspondantes. Ces composants pouvant éventuellement eux-mêmes posséder des composants de type construit il en résulte des appels successifs d'opérations jusqu'à atteindre les composants de type primitif.

Au chapitre III nous présentons une méthode qui permet d'étendre le principe des maquettes de création à des objets de structure complexe. Cependant, là encore, tous les cas ne peuvent être traités et certains types doivent être munis d'une opération de création. De même que dans Hydra, ils possèdent une maquette dont l'utilisation fournit la capacité d'un objet du type non complètement initialisé. L'application de l'opération décomposer sur cet objet permet alors d'en atteindre la représentation afin de compléter son initialisation.

La création d'un nouveau type procède du même principe. Elle consiste en la construction des procédures du type puis en la création de l'objet type associé à ce nouveau type. De même que l'objet type associé à un type donné en contrôle l'utilisation, on peut définir un objet type associé au type "type" et fourni par le noyau. Il est le générateur de tous les types dans le système.

CHAPITRE II

MISE EN PLACE DES MECANISMES

Dans le chapitre précédent, les capacités ont été considérées comme de simples repères d'objets, quoique eux-mêmes protégés. On se propose d'examiner ici le problème de la réalisation des systèmes à capacités et l'implantation de domaines de protection.

Les mécanismes présentés précédemment reposent sur l'existence d'objets primitifs, les segments, qui servent de support à la représentation des structures plus complexes bâties par extension. Ainsi, la mise en place d'un système à capacités nécessite essentiellement l'existence d'une mémoire segmentée. Cependant on notera que s'il existe de nombreux modèles de mémoires segmentées, dans un système à capacités tout objet est considéré comme étant potentiellement partageable ; de ce point de vue nombre des modèles existants se révèlent non satisfaisants. La mise en place d'un système à capacités commencera par la définition d'un modèle de mémoire segmentée mieux adaptée au partage.

1. - Le problème du partage [Fab 74]

Afin d'illustrer le problème du partage d'objets entre utilisateurs, et en particulier le partage des modules qui, dans notre système, définissent les domaines de protection, nous utiliserons le modèle simple qui suit. L'exécution d'un programme exige que celui-ci puisse désigner les objets qu'il manipule parmi l'ensemble des objets connus du système. Afin de réaliser l'indépendance des programmes on convient habituellement de munir chaque programme de son propre espace de noms. Ainsi, un programme est muni d'une forme générant des noms locaux dans l'espace des noms du programme et d'une carte d'implantation permettant de localiser les objets désignés dans l'espace global des objets du système. Habituellement, cette carte localise physiquement la représentation de chaque objet. Dans un système à mémoire segmentée où chaque processus est muni d'une table de segments, celle-ci est la carte d'implantation du programme exécuté par ce processus. Un même objet utilisé par des processus différents ne possède pas nécessairement le même nom local dans leurs espaces de noms. De même, l'exécution d'un même programme par deux processus ne se traduit pas nécessairement par la génération d'une forme ni d'une carte identiques. La figure 1 schématise le modèle utilisé ici.

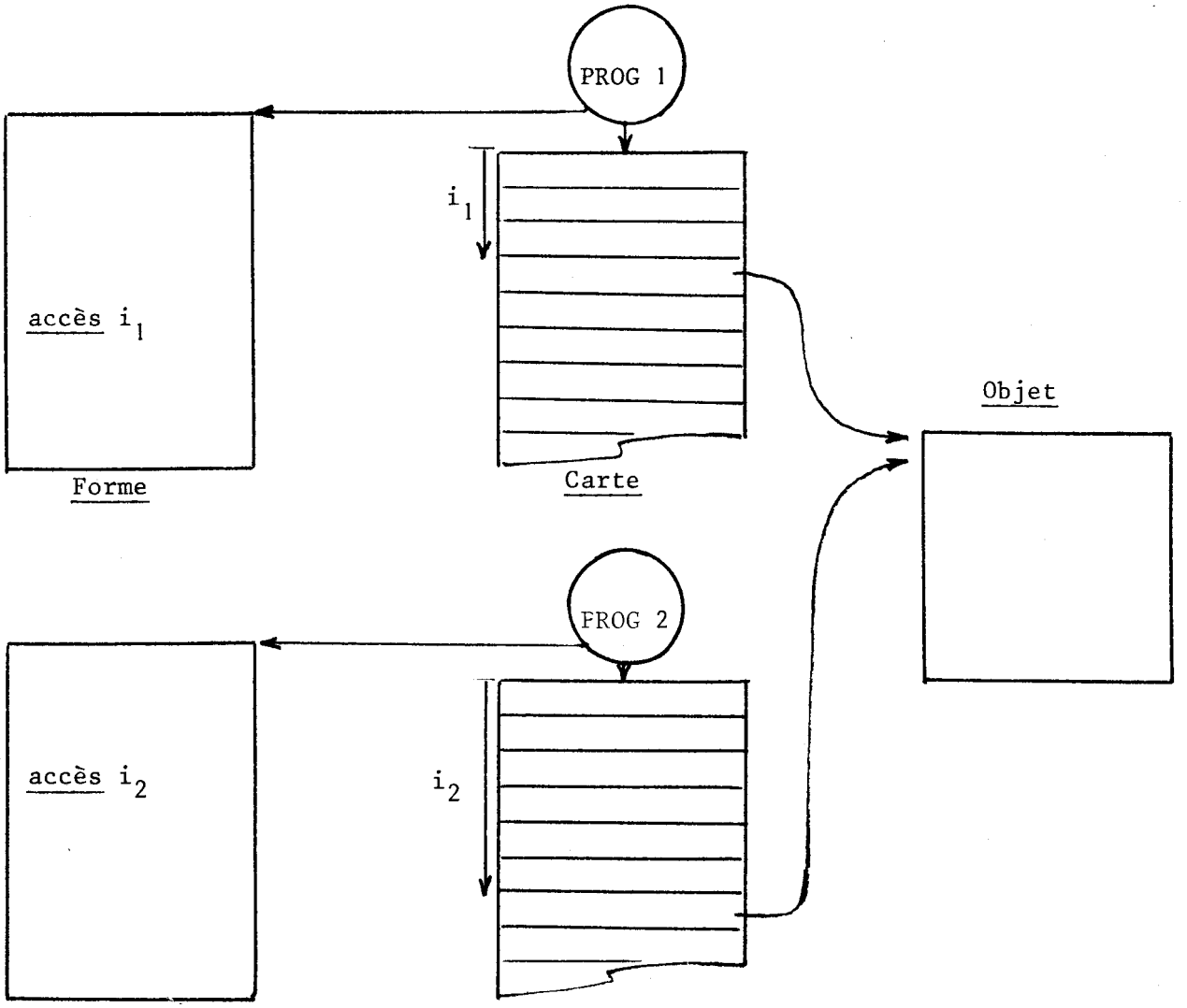


Figure 1 : Différenciation des noms locaux

Supposons maintenant que les deux programmes de la figure 1 soient susceptibles d'exécuter le même sous-programme. Les noms générés par ce sous programme doivent être interprétés ou dans la carte du programme 1 ou dans celle du programme 2 selon que l'on considère son appel par l'un ou l'autre. Ainsi les objets utilisés par ce sous-programme sont susceptibles de ne pas avoir le même nom local dans chacune de ces cartes. En conséquence deux formes du même sous-programme doivent être générées et associées à chaque programme. Si l'on souhaite réaliser un partage effectif du sous-programme et donc de sa forme, ou bien les objets utilisés doivent posséder le même nom local dans chaque carte, ou bien une carte intermédiaire doit être introduite qui permette l'interprétation des noms générés par la forme associée au sous-programme dans la carte du programme qui l'exécute. La première solution nécessite de réserver de nombreuses entrées dans chaque carte et implique la connaissance par le système des objets susceptibles d'être partagés. Par ailleurs, ces entrées étant susceptibles de ne pas être utilisées, il peut en résulter une sous-utilisation notable de l'espace mémoire. Cette solution n'est guère envisageable que pour les objets dont la

probabilité de partage est très élevée. Elle est utilisée dans Multics [Cro 75 Chap. 3] pour certaines procédures du système telles que l'éditeur de liens dynamique. Ces procédures possèdent le même nom dans toutes les tables de segments et sont prééditées à la création des processus. La deuxième solution est représentée par la figure 2. Elle est illustrée par l'utilisation d'un segment de liaison dans Multics.

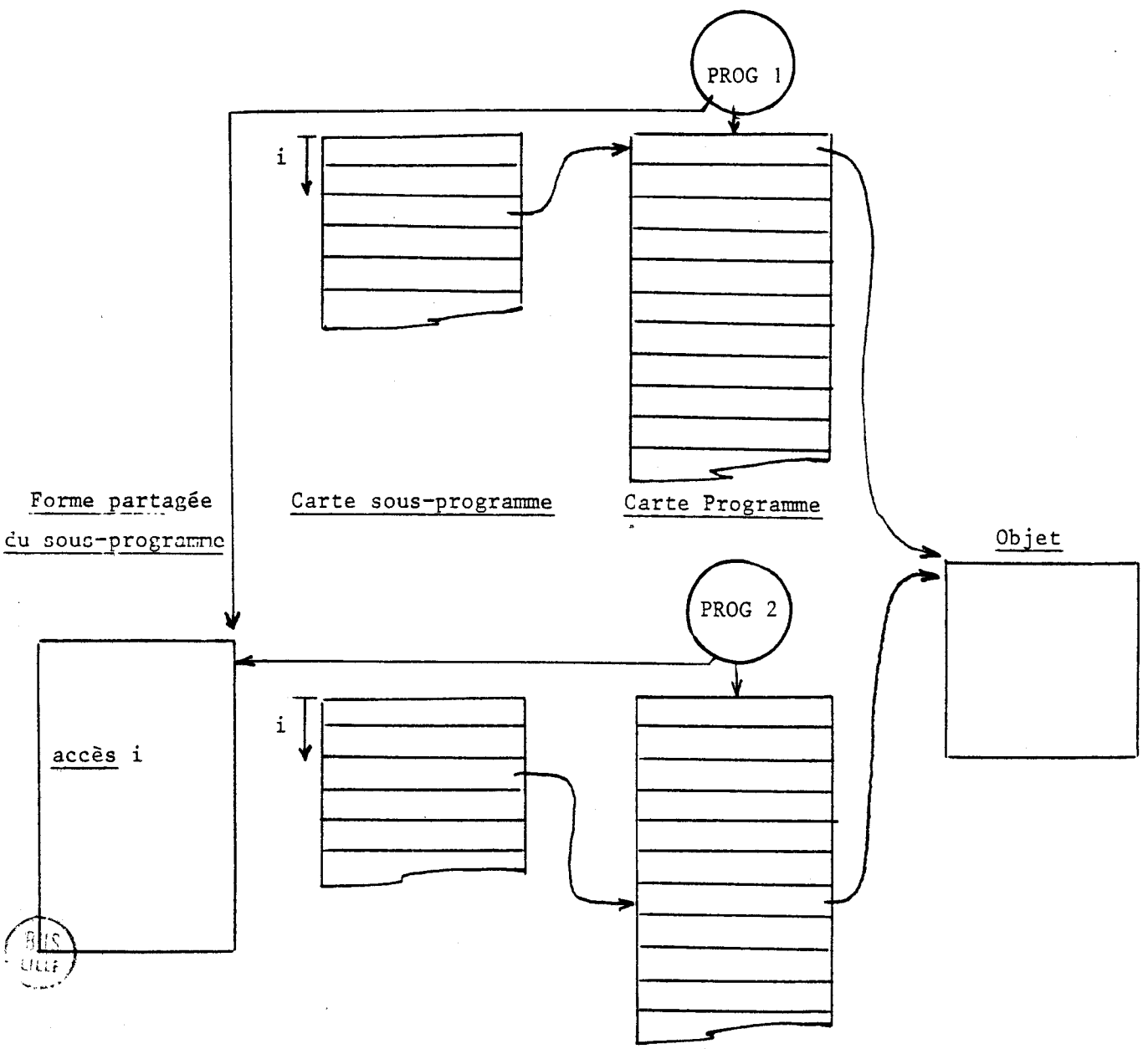


Figure 2 : Utilisation d'une carte intermédiaire

On remarque que ce schéma nécessite deux niveaux de transformation des noms :

- chaque fois qu'un sous-programme est utilisé par un nouveau processus, une nouvelle carte intermédiaire doit être créée. Ceci nécessite éventuellement de retrouver dans la carte du processus les entrées déjà allouées aux objets concernés. La gestion des noms qui en résulte est locale au processus concerné.
- lorsqu'une entrée doit être créée pour un objet dans la carte d'un processus, il est nécessaire de retrouver l'implantation de l'objet concerné pour lequel il existe éventuellement déjà une entrée dans la carte d'un autre processus. De même, lorsqu'un objet est déplacé dans la mémoire, toutes les entrées allouées pour cet objet dans différentes cartes doivent être retrouvées et mises à jour. Ainsi la gestion de la mémoire interfère avec la gestion des noms.

A priori un tel schéma va à l'encontre du principe du noyau minimum selon lequel les fonctions qui participent à la protection des objets, et c'est le cas de celles évoquées ici, devraient être minimisées autant que possible. Janson dans [Jan 76] a étudié ce problème en ce qui concerne Multics.

2. - Schéma d'un système à capacités

La difficulté du schéma présenté précédemment réside dans l'existence de plusieurs niveaux d'interprétation des noms et dans son inadéquation au partage des objets. La solution à ce problème a été esquissée précédemment : pour qu'un objet soit partageable il doit posséder un nom indépendant du contexte de son utilisation. Ainsi un système à capacités reposera sur l'attribution aux objets d'un nom unique indépendant du contexte de leur utilisation. Un tel système est muni d'une carte d'implantation unique qui permet la localisation de l'objet à partir de son nom unique. On appelle table unique des objets du système (TUOS) cette carte et descripteur de l'objet l'entrée qui lui est associée dans cette table. Tout accès à un objet a lieu indirectement à travers son descripteur.

Une capacité contient alors, outre les droits, le nom unique de l'objet. Chaque objet étant associé biunivoquement à un descripteur unique, la gestion de la mémoire ne requiert aucun accès aux différents espaces d'adressage dans lesquels les objets déplacés sont référencés. De cette manière elle peut être dissociée du reste du système qui lui, suppose l'utilisation de capacités. On notera que le rôle de la carte d'implantation intermédiaire, qui dans le schéma précédent était associée au sous-programme partagé, est rempli par une liste de capacités utilisables pour l'exécution du sous-programme (ou du module). Ainsi l'on distingue un ensemble de noms globaux : les noms uniques des objets, et un ensemble de noms locaux associé à chaque environnement : les noms d'index dans la liste de capacités de l'environnement. On a ainsi intégré les notions d'espace d'adressage et de domaine de protection. De même, la carte d'implantation du processus n'ayant plus de raison d'être, les environnements deviennent indépendants et le principe de méfiance mutuelle est respecté.

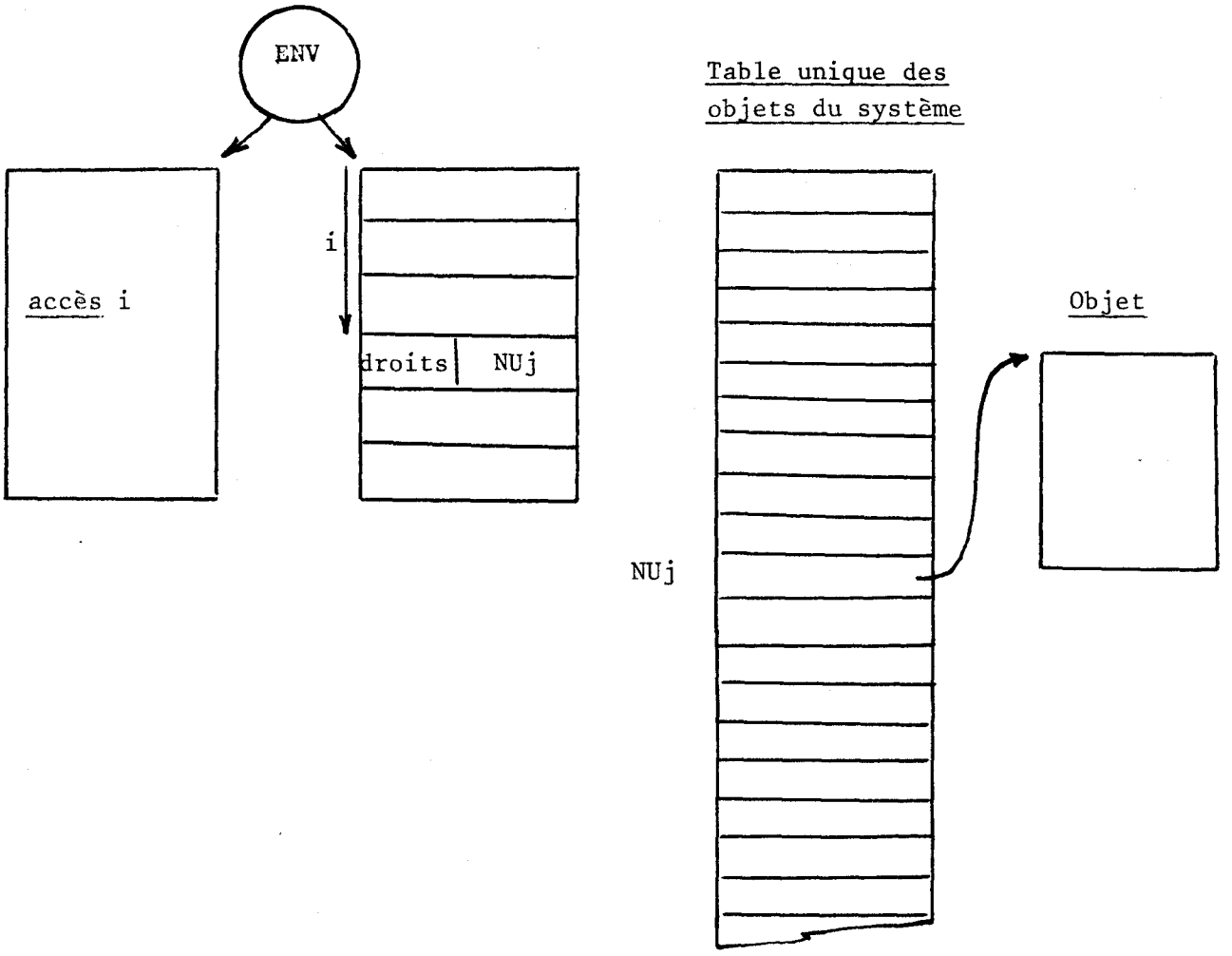


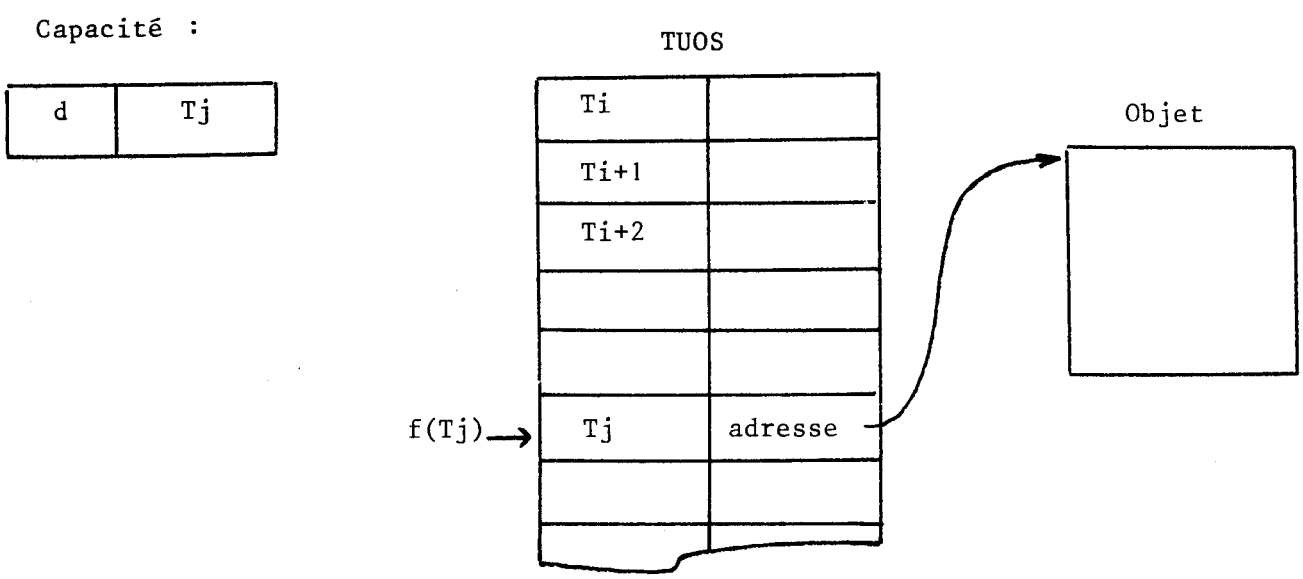
Figure 3 : Schéma d'adressage d'un système à capacités

3. - Notion de nom unique

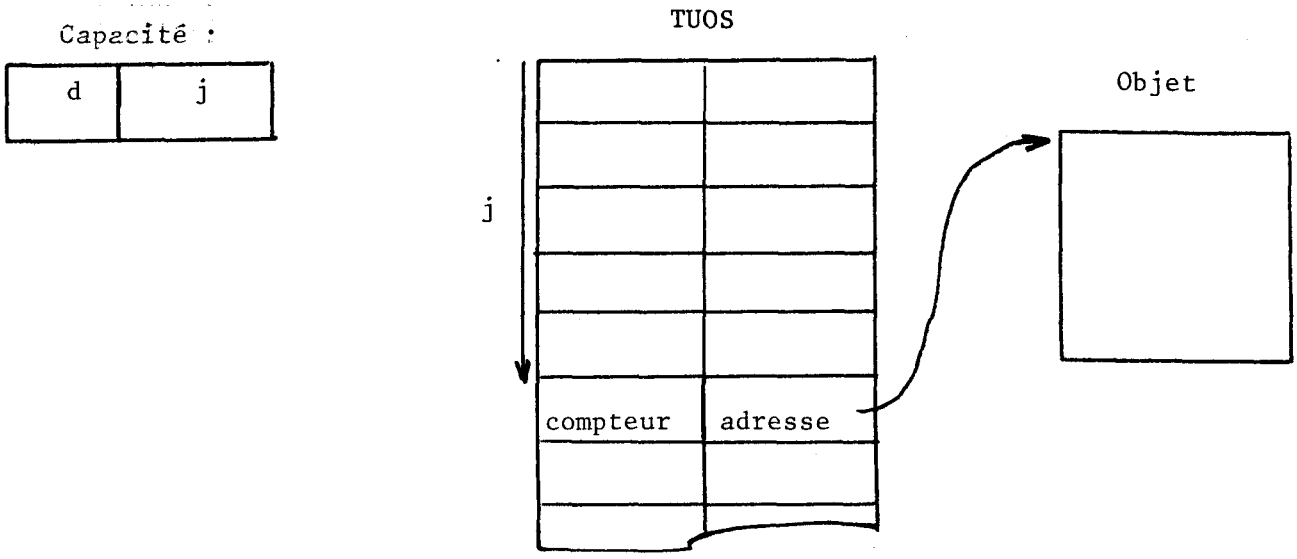
On distingue deux sortes de noms uniques selon la méthode utilisée pour les générer :

- Noms temporels. La méthode certainement la plus efficace pour générer une valeur unique est d'utiliser l'horloge du système. Il suffit que son pas de définition soit suffisament petit pour que deux lectures consécutives ne fournissent pas la même valeur. Selon cette méthode, le nom unique d'un objet est la valeur de l'horloge à l'instant de sa création, on parle alors de nom temporel. Ceci conduit généralement à choisir un pas d'horloge d'une microseconde et une taille de nom unique de huit octets afin de couvrir la durée de vie du système. Une autre méthode qui permet de diminuer la taille des noms uniques et par conséquent celle des capacités consisterait à utiliser un compteur qui ne serait incrémenté qu'à chaque création d'un nouvel objet. La figure 4a schématise l'organisation de la TUOS en pareil cas. Les descripteurs contiennent le nom temporel de l'objet qu'ils localisent et sont rangés dans l'ordre chronologique de la création des objets. On conçoit que la difficulté essentielle d'une telle organisation réside dans la réalisation d'une fonction efficace de localisation du descripteur d'un objet dans la TUOS à partir de son nom temporel.

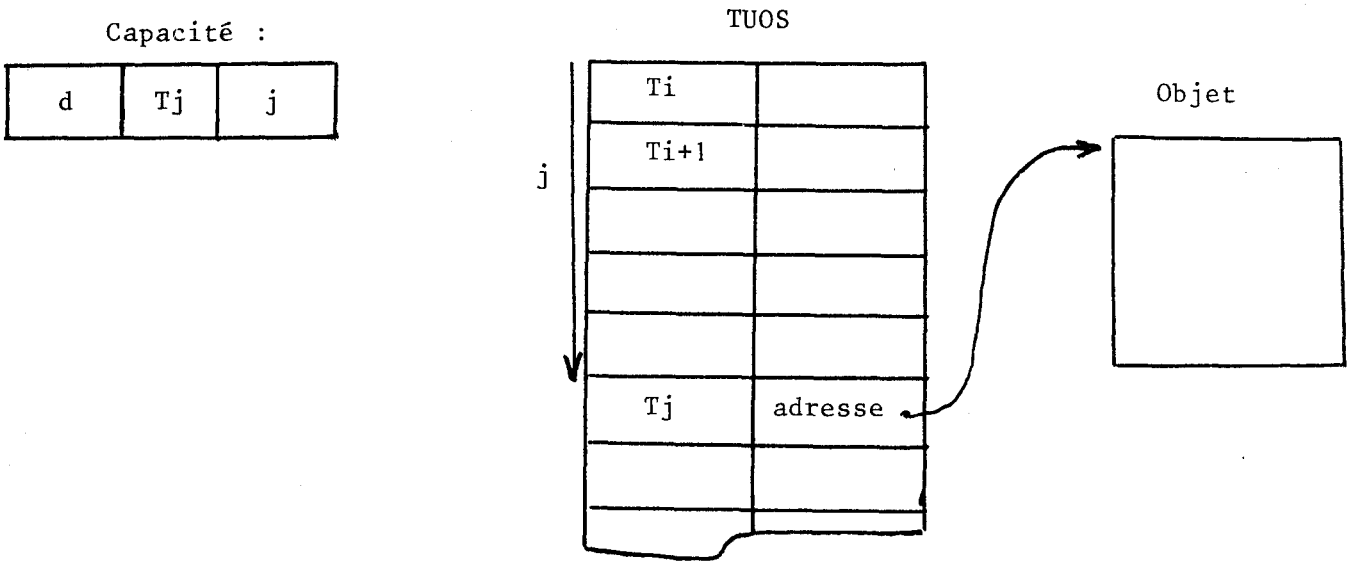
- Noms spatiaux. On appelle nom spatial d'un objet le nom de son descripteur : son adresse ou son index dans la table des descripteurs. Cette méthode permet de définir une fonction d'accès rapide et simple.



a : cas des noms uniques temporels



b : Cas des noms uniques spatiaux



c : Solution mixte

Figure 4 : Organisation de la table unique des objets du système.

La difficulté réside dans la taille de la TUOS et dans la nécessité de fournir une fonction d'accès efficace aux descripteurs des objets. La table est a priori de grande taille et les créations et destructions d'objets sont fréquentes. Il importe donc lorsqu'un objet disparait de récupérer l'espace occupé par son descripteur. Dans le cas d'une table temporelle cette gestion est simple car les descripteurs sont ajoutés dans l'ordre chronologique. De plus le nom des objets étant indépendant de l'organisation de la table, la récupération d'espace peut être réalisée simplement par une compaction périodique de la table. L'utilisation

de noms spatiaux ne permet pas cette gestion simplifiée car toute compaction entrainerait la modification de l'ensemble des noms attribués. Les entrées libérées dans la table doivent donc être gérées, par exemple à l'aide d'une structure de liste, et réattribuées. Lorsqu'un nouvel objet est créé on alloue en priorité l'une des entrées libres plutôt que d'en créer une nouvelle. A cette fin on doit assurer qu'il n'existe plus aucune capacité repérant cette entrée. afin d'éviter qu'une ancienne capacité ne repère ainsi un nouvel objet. Une méthode possible consiste à compter les capacités qui repèrent chaque objet et à maintenir à cet effet un compteur dans le descripteur associé à l'objet. Cependant, une telle méthode pénalise les opérations de manipulation des capacités car chaque copie de capacité requiert la mise à jour du compteur. Cm* utilise actuellement une table de ce type. On conçoit que lorsque la TUOS est trop grande pour tenir en mémoire centrale, cette méthode peut être la source de difficultés. Nous reviendrons sur ce sujet au chapitre suivant.

Une solution mixte est proposée dans [Lin 73]. Elle repose sur la remarque qu'un nom temporel n'est jamais réutilisé alors qu'un nom spatial autorise une fonction d'accès efficace. La table est organisée comme une table spatiale où chaque descripteur contient le nom temporel de l'objet qu'il repère (figure 4c). La capacité d'un objet contient à la fois son nom temporel et son nom spatial. Lorsqu'un accès est réalisé le descripteur est localisé à l'aide du nom spatial et le nom temporel qu'il contient est comparé à celui de la capacité. Ainsi les entrées libérées peuvent être réallouées ; s'il existe encore des capacités repérant un ancien objet l'erreur est aisément détectée. Cependant on notera qu'une telle méthode augmente considérablement la taille des capacités.

Remarque : D'autres mécanismes n'utilisant pas une table unique peuvent être imaginés. Par exemple, on peut suggérer que chaque objet reçoive un nom relatif à son type. Alors le nom du type identifie le module qui interprète le nom de l'objet pour retrouver sa représentation. Une telle solution présenterait l'avantage de limiter la taille des capacités en limitant celle des noms utilisés. Par contre elle présente l'inconvénient que les représentations des objets sont retenues par les modules et donc peu protégées en cas d'erreurs car leur gestion est laissée à ces modules. Par ailleurs, le mécanisme d'accès cessant d'être uniforme, il n'est pas possible de l'accélérer éventuellement à l'aide d'une mémoire associative.

C'est une variante de ce qui est réalisé dans un système de gestion de fichiers. Une telle solution n'est envisageable que dans les modules du noyau sur lequel de toute manière repose la protection. C'est ainsi qu'elle était utilisée dans CAL [Gr 72].

4. - Conservation des objets

Les objets doivent pouvoir être conservés d'une utilisation à l'autre. Dans un système classique, la fonction de conservation porte sur les segments (ou les fichiers) et les programmes. Elle est assurée à l'aide de catalogues. Pour conserver un objet, on le munit d'un nom symbolique que l'on répertorie dans un catalogue et le système lui alloue de l'espace en mémoire secondaire. On définit à ce moment également son mode de protection à l'aide d'une liste de contrôle d'accès. Par la suite, lors de l'édition de liens, les objets désignés par un nom symbolique sont recherchés dans les catalogues et une entrée est créée pour chacun d'eux dans la carte d'implantation du processus considéré. Les descripteurs ainsi créés contiennent également les droits concédés au processus et jouent le rôle de capacités.

On peut a priori envisager de procéder de même dans un système à capacités et considérer le mécanisme des capacités comme fournissant une structure d'éxécution et non de conservation. L'éditeur de liens créerait pour chaque objet une entrée dans la TUOS et générerait les capacités et C_listes nécessaires à la mise en place des modules de protection. Cependant, il convient de faire quelques remarques. Seuls les objets conservés dans les catalogues sont susceptibles d'être partagés et d'être référencés par des capacités. Ces objets ne peuvent avoir de composants puisque leurs capacités ne peuvent être conservées. On ne peut non plus envisager de décrire ces composants de quelque manière, à l'éditeur de liens qui est supposé générer les capacités nécessaires au programme car celui-ci ne doit pas, par définition de la protection, avoir connaissance des structures générées. Il en résulte que les objets conservés ne peuvent être construits qu'en termes de segment ou de collection d'objets déjà conservés. Il en résulte également que le système est statique puisque les seuls déplacements d'objets d'une structure à une autre ne peuvent avoir lieu que dans les catalogues. Ainsi les mécanismes à capacités ne sont utilisés de fait que pour la construction du système lui-même et la protection et l'extensibilité au niveau de l'utilisateur sont tributaires des outils proposés par ce système au niveau des catalogues. Enfin, on notera que l'éditeur de liens et le sous-système de gestion des

catalogues sont impliqués dans la mise en place d'une protection effective. L'un parce qu'il génère les capacités des objets, l'autre parce qu'il conserve des représentations d'objets non protégées. une telle structure va à l'encontre du principe du noyau de protection minimum évoqué au chapitre I.

C'est pourquoi dans un système à capacités la conservation des objets est assurée de fait par les mécanismes à capacités. Il n'y a pas de transformation de la forme des objets en d'une part la forme sous laquelle ils sont conservés et d'autres part la forme sous laquelle ils sont utilisés. Il en résulte qu'il n'y a pas à proprement parler d'édition de liens : lorsqu'un module est créé sous la forme d'une liste de capacités il est conservé sous cette forme. Il en résulte également que les catalogues sont des structures simples qui à un nom symbolique associe une capacité de l'objet. Ces structures sont elles-mêmes supportées par le mécanisme d'extension de type et utilisent des capacités. La figure 5 schématise la structure simplifiée de tels catalogues.

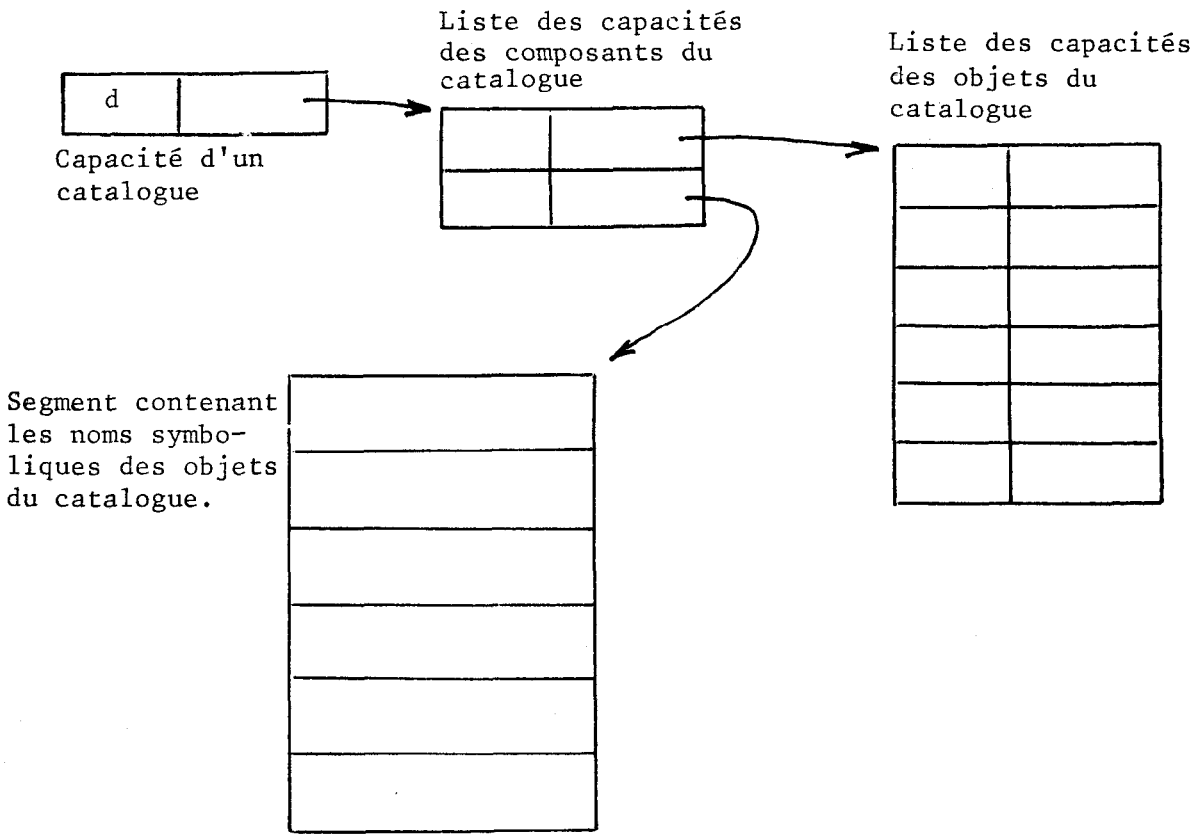


Figure 5 : Structure simplifiée d'un catalogue

5. - Un exemple : l'espace d'objets d'Hydra

Dans la mesure où les objets sont conservés à l'aide du mécanisme à capacités, la table unique des objets du système peut devenir de taille très importante. Ainsi Hydra qui a actuellement encore peu d'utilisateurs conserve environ 20 000 objets. On imagine aisément le nombre d'objets que devrait conserver un système non expérimental. L'implantation de l'espace des objets impose des contraintes contradictoires que l'on a énoncé en II 3 : la fonction d'accès dans la TUOS doit être efficace mais on doit autant que possible éviter la réutilisation des noms. On se propose d'exposer ici la solution adoptée dans Hydra.

On y distingue deux tables, l'une dite table passive, est maintenue en mémoire secondaire, l'autre, dite active, est en mémoire centrale et ne concerne que les objets utilisés. La table passive décrit l'espace de conservation. Chaque objet possède un nom unique temporel et est répertorié dans la table passive qui est organisée par ordre chronologique de création. Les descripteurs y sont regroupés en blocs de 64. Afin d'en améliorer l'accès, on maintient en mémoire centrale une table d'accès à ces blocs dont chaque entrée fournit le plus grand nom unique contenu dans le bloc correspondant. Ainsi l'accès à la table passive est réalisé d'abord en identifiant le bloc qui contient le descripteur recherché puis en réalisant une recherche linéaire dans le bloc concerné. Cet accès à la table passive n'a lieu que la première fois qu'un objet est utilisé. Par la suite on lui associe une entrée dans la table active. Celle-ci est obtenue en recopiant le descripteur de l'objet depuis la table passive. Ce descripteur est étendu pour décrire également l'implantation de l'objet en mémoire centrale.

Lorsqu'une capacité est utilisée pour la première fois, elle contient le nom unique temporel de l'objet. A l'aide de ce nom temporel, on recherche à travers une "hash table" si l'objet possède déjà une entrée dans la table active. Lorsqu'une telle entrée lui a été attribuée toute capacité utilisée pour référencer l'objet est transformée au fur et à mesure des accès pour contenir son nom spatial dans la table active (en fait l'adresse du descripteur actif). Dès lors les accès seront plus rapides. Les capacités ainsi transformée sont dite sous forme active ou capacités actives. Les capacités non transformées qui elles, sont associées à un descripteur passif sont dites sous forme passive ou capacités passives. Les noms spatiaux étant plus courts (2 octets), l'espace disponible dans les capacités actives est utilisé pour y reporter pour des raisons d'efficacité le type de l'objet (sous forme active lui aussi).

Selon ce schéma, la représentation d'un objet ne peut se trouver en mémoire centrale qu'à la condition que cet objet ait été "activé" (qu'il en existe un descripteur actif). De même, un objet n'est actif qu'à la condition qu'il soit référencé par au moins une capacité active. Aussi à chaque descripteur de la table active est associé un compteur du nombre de capacités actives de l'objet. Dès que ce compteur atteint la valeur zéro l'objet est rendu passif, c'est-à-dire que l'entrée qui lui était associée dans la table active est libérée et si nécessaire l'objet est renvoyé en mémoire secondaire. La probabilité pour que la valeur de ce compteur soit erronnée est faible car la table et les capacités actives résident toutes deux en mémoire centrale. Cependant par souci de fiabilité un code est placé dans le descripteur actif de chaque objet au moment de la création de ce descripteur et copié dans chaque capacité lorsqu'elle est activée. Il est ainsi peu probable que puisse se produire la situation décrite en 3 où par suite d'erreur une capacité d'un objet "passivé" permettrait l'accès à la représentation d'un autre objet.

Il convient de remarquer que lorsqu'un objet est rendu passif, il se peut qu'il s'agisse d'une liste de capacités. Alors ces capacités doivent elles-mêmes être remises dans leur forme passive. Il peut en résulter une réaction en chaîne qui conduit à rendre passifs d'autres objets.

On notera également que le fait que la représentation d'un objet soit renvoyée en mémoire secondaire n'a en principe pas de relation avec le fait de le rendre passif ou non. Cependant, deux conceptions sont possibles à ce sujet. On peut utiliser un support en mémoire secondaire pour mettre en place une mémoire virtuelle dans laquelle seuls les objets utilisés ont leur représentation. Lorsqu'un objet est rendu passif, sa représentation est alors déplacée de la mémoire virtuelle vers le support permanent de l'objet en mémoire secondaire. La taille d'une telle mémoire virtuelle est à équilibrer avec la taille de la table active de sorte que celle-ci reste résidente. Une autre conception est d'utiliser le support permanent de l'objet lorsque celui-ci doit être renvoyé en mémoire secondaire. Alors on ne peut conserver sur ce support des capacités actives et chaque objet renvoyé en mémoire secondaire, lorsqu'il s'agit d'une C-liste doit provoquer la transformation des capacités qu'il contient. C'est la méthode utilisée dans le système Plessey 250 [Erg 74, Cos 74]. Dans un tel cas, la transformation des capacités interfère avec la gestion de la mémoire.

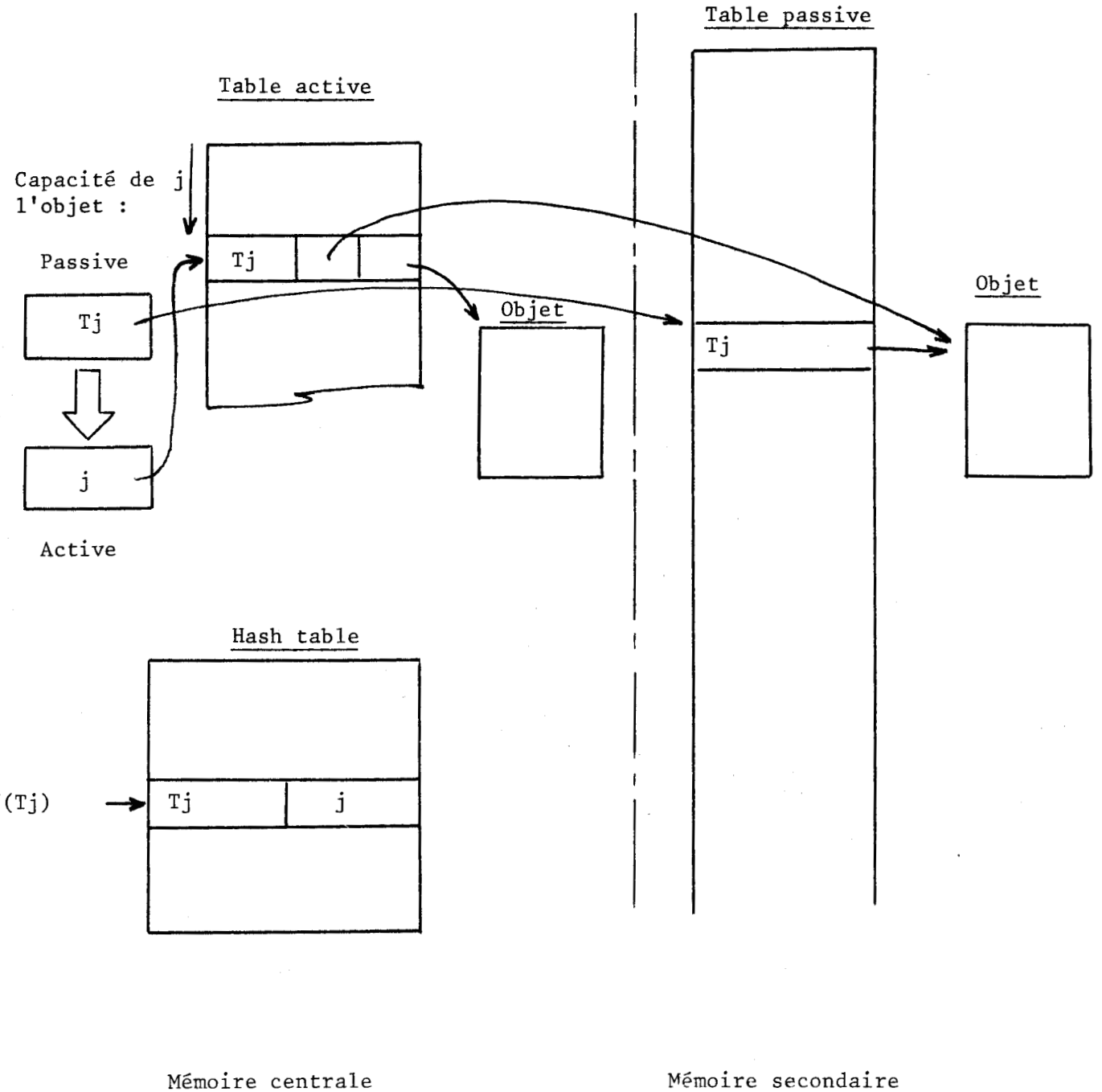


Figure 6 : Organisation de l'espace des objets d'Hydra

6. - Mise en place des mécanismes d'extension

On se propose de développer ici l'architecture d'un système utilisant les notions introduites précédemment. On remarquera que la seule table de descripteurs connue de la machine est la table active. La transformation des capacités de la forme passive à la forme active et inversement relève de la gestion de la mémoire virtuelle du système ; c'est une fonction de base du noyau de protection.

6.1. - Extension de type

La machine ne reconnaît a priori que deux types d'objets : les segments et les listes de capacités. Les listes de capacités sont utilisées soit pour décrire un espace d'adressage soit simplement comme structure de regroupement d'objets. Dans ce dernier cas, les opérations défines sur le type C-liste sont relatives à la copie d'une capacité d'une C-liste dans une autre. Les autres sont construits à partir de ces deux types de base, ils sont implantés par des sous-systèmes. Ainsi, pour permettre la construction de nouveaux types, il n'est besoin que de distinguer un type construit général. Celui-ci n'a pour but que de signifier que l'objet doit être interprété par le logiciel et d'interdire son accès aux opérations fournies par la machine.

Les opérations relatives aux C-listes ne nécessitent aucun accès aux objets dont elles manipulent les capacités. Cependant, elles doivent mettre à jour le compteur de référence. Le descripteur de l'objet est alors nécessaire et la capacité utilisée doit donc être transformée dans sa forme active. On utilise ces remarques pour limiter la taille des capacités. Puisque dès qu'une capacité est manipulée, elle nécessite l'activation de l'objet, une capacité de forme passive ne contiendra que les droits et le nom unique temporel de l'objet. La marque du type de l'objet sera contenue dans son descripteur.

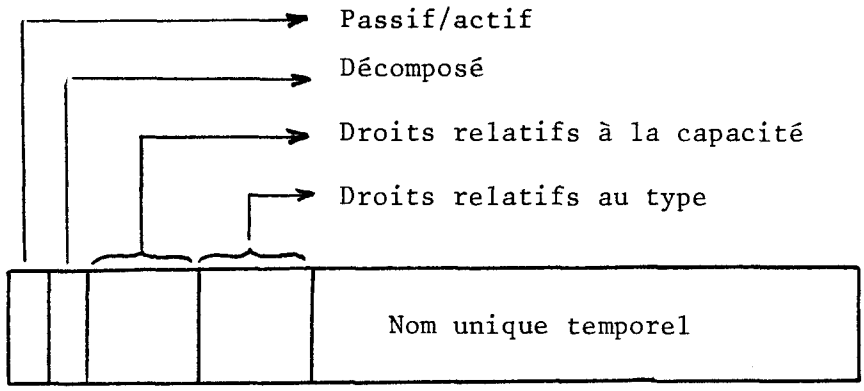
Rappelons le principe de l'extension des types défini au chapitre précédent. Un objet d'un type T_i composé d'objets de type T^1_{i-1}, T^2_{i-1}, ..., T^n_{i-1} reçoit pour représentation une C-liste qui contient les capacités de ces composants. La décomposition d'un objet construit par le gérant de son type fournit donc toujours un objet de type C-liste. Ceci permet de simplifier l'opération de décomposition. On utilise un indicateur de décomposition dans la capacité. Celui-ci permet d'interpréter la capacité comme étant celle d'une C-liste. On remarquera que la décomposition porte sur une capacité active puisqu'elle requiert l'accès aux attributs de l'objet donc à son descripteur. Il est cependant nécessaire de réaliser cette décomposition comme si elle pouvait porter sur une

capacité de forme passive. En effet, bien que ceci soit le résultat d'une faute de construction, il peut se produire qu'après décomposition une capacité active soit retournée à sa forme passive. Cette capacité repérant une C-liste et non plus l'objet composé, il importe alors de notifier cette différence d'interprétation. Ceci ne peut être fait que dans la capacité car le descripteur de l'objet est relatif à son type construit. Une telle tranformation de la capacité d'un objet après décomposition peut se produire après que la C-liste qui contient cette capacité soit retournée à sa forme passive. Alors toutes les capacités quelle contient doivent elles-même être remises à leur forme passive.

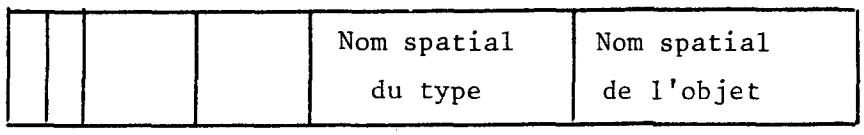
On peut maintenant donner le format des capacités. Une capacité de forme passive contient (figure 7) :

- un indicateur de forme passive/active
- un indicateur de décomposition
- la liste des droits de manipulation de l'objet introduite au chapitre précédent
- la liste des droits relatifs au type de l'objet : lorsque l'objet est décomposé, ces droits sont relatifs à la C-liste qui regroupe ses composants
- le nom unique temporel de l'objet.

Dans sa forme active, la capacité ne diffère que par le fait qu'elle contient le nom unique spatial de sa représentation et celui de son type (cf 6.2).



a) forme passive



b) forme active

Figure 7 : Format d'une capacité

On remarquera que le descripteur d'un objet est référencé par l'ensemble des capacités de l'objet. C'est lui-même un objet partagé dont la taille est assez importante. On peut mettre ceci à profit pour y ranger directement la représentation de l'objet lorsque celui-ci est petit et améliorer ainsi les temps d'accès. En particulier, lorsqu'un objet construit ne contient qu'un simple composant, la création d'une C-liste de taille unité sera évitée en rangeant directement dans le descripteur de l'objet la capacité de son composant. Le format d'un descripteur dans la table active sera donc le suivant :

- indicateur de taille : si l'indicateur est positionné, la représentation de l'objet est contenue dans le descripteur. Si l'objet est de type primitif il s'agit d'une C-liste à un élément ou d'un segment de quelques mots. S'il est de type construit on sait qu'il s'agit d'une C-liste à un seul élément.
- indicateur de type : il s'agit d'une marque de type primitif ou de la marque de type construit.
- compteur du nombre de capacités actives.
- type construit : si l'indicateur de type signifie que l'objet est de type construit alors ce champ contient la marque de ce type sous la forme d'un nom unique temporel, sinon il est vide.
- nom spatial du type (cf 6.2)
- nom unique temporel de l'objet
- taille de la représentation de l'objet
- adresse en mémoire centrale de la représentation
- localisation en mémoire secondaire de la représentation.

6.2. - Composition/décomposition des objets

L'implantation d'un type construit est réalisée à l'aide d'objets du type "type" (ou objets type) constuits sur une C-liste qui contient les capacités des différentes opérations de ce nouveau type (Chapitre I). La construction d'un nouveau type est réalisée par le noyau et est basée sur l'utilisation de maquettes de création tout comme l'est la création d'objets. Les mécanismes de création tendent à optimiser le système et seront décrits dans les chapitres ultérieurs. On se limite ici aux mécanismes cablés qui permettent leur implantation. Ceux-ci sont inspirés des mécanismes décrits au chapitre précédent. Ainsi, un type

construit est caractérisé par son nom temporel, c'est celui de l'objet type qui lui a été attribué. On définit sur cet objet l'opération appliquer qui prend pour argument l'objet type, un numéro d'opération défini sur ce type, et les paramètres à transmettre à cette opération. L'opération appliquer construite dans le noyau, a comme résultat l'activation de l'opération du type.

On remarquera également que puisque tout descripteur d'objet contient la marque de son type et que l'exécution de l'opération appliquer conduit à activer cet objet s'il ne l'est déjà, l'exécution de cette opération ne requiert pas la présentation de l'objet type. L'objet type sur lequel elle porte peut être identifié à travers le descripteur de l'objet. Ainsi l'opération appliquer s'écrira : appliquer <*nom objet, n° opération, liste de paramètres*> où nom objet est implicitement le nom du premier paramètre de l'opération à activer. La figure 8 schématise le mécanisme. On notera que contrairement au mécanisme du système Plessey 250 [FL 76], ceci n'interdit pas les opérations multiadiques. Cette limitation du système Plessey 250 provient de ce que la seule opération de décomposition est l'opération d'appel de procédure. Ce n'est pas le cas ici et les autres paramètres de l'opération activée dans le sous-système peuvent éventuellement être décomposés à l'aide de l'objet type. On peut maintenant justifier l'existence d'un nom spatial associé au type d'un objet, il s'agit du nom spatial de l'objet type correspondant. La présence de ce nom spatial dans le descripteur d'un objet ou éventuellement dans une capacité repérant cet objet a pour but d'accélérer l'opération appliquer.

Ce mécanisme cependant ne s'applique qu'aux sous-systèmes gérant des types. On est conduit à introduire une opération plus simple qui permette d'activer des procédures. C'est cette opération qui est utilisée pour réaliser l'opération appliquer. (cf chapitre IV).

Environnement courant

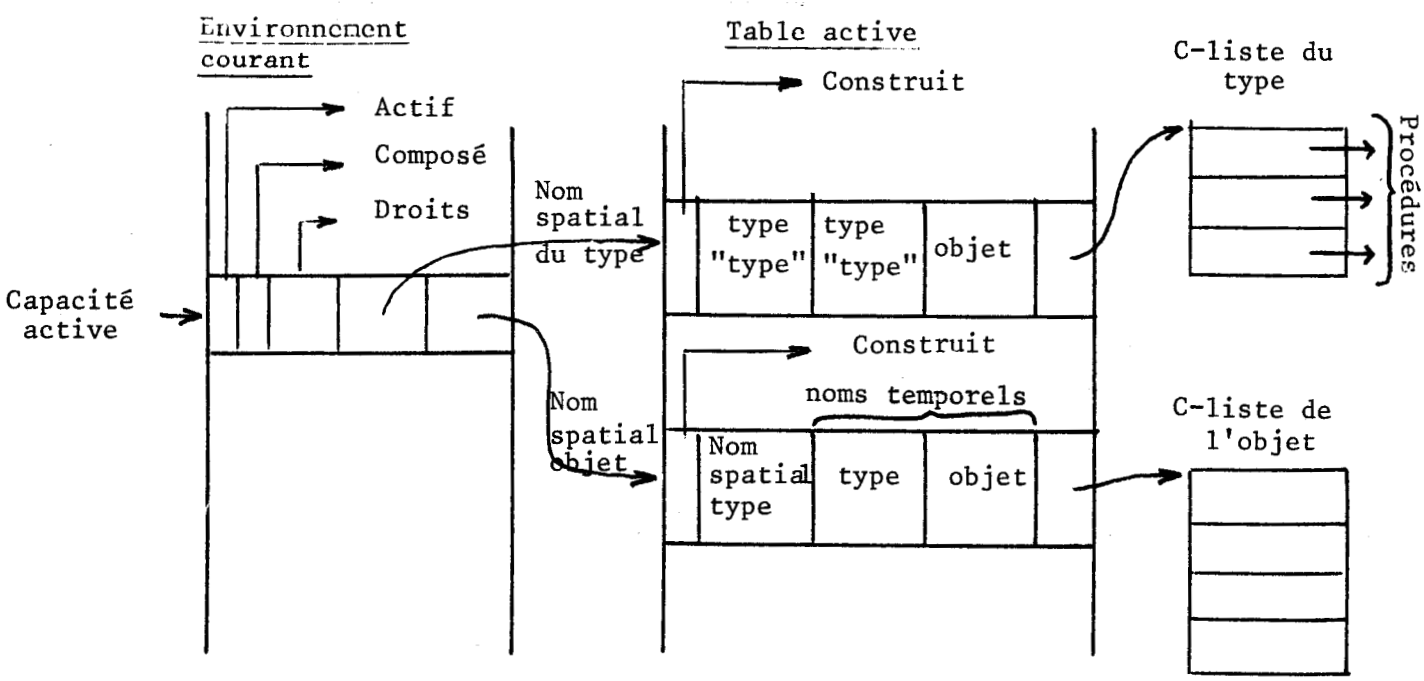


Figure 8 : Localisation des types

7. - Une tentative d'application des mécanismes précédents au cas des réseaux

La caractéristique essentielle des mécanismes présentés est de cacher et d'uniformiser la gestion des noms des objets. Il est toujours possible de mettre en place de tels mécanismes, il en est ainsi dans Multics où une procédure génère des noms locaux. Cependant l'intérêt des mécanismes à capacités réside dans la simplicité de la transformation des noms. Alors que dans Multics l'interprétation des noms jusqu'à la localisation de la représentation de l'objet correspondant requiert deux étapes complexes de transformation où interviennent la procédure, le domaine, le processus, dans un système à capacités, il ne s'agit que d'une indirection. Le problème de la gestion des noms se pose d'une manière cruciale dans les réseaux ; on se propose donc d'étendre le schéma précédent à ce cas. On ne tiendra compte dans ce qui suit que des objets actifs et on se limitera aux mécanismes d'accès.

On se place dans le cas d'un système distribué sur plusieurs machines dans lequel on souhaite depuis un site donné être capable d'activer une procédure sur un site différent sans avoir à nommer explicitement ce site. Chaque procédure doit donc être capable de nommer un quelconque objet du système comme s'il se trouvait sur le même site qu'elle même. Alors, les objets (y compris les procédures) peuvent être déplacés (voire dupliqués) sur des sites différents sans que le programmeur en ait conscience.

Afin de justifier l'utilisation du mécanisme précédent, dans un tel cas, il convient de faire quelques remarques. La représentation d'un objet d'un type donné n'est accessible que par les opérations de ce type. Aussi tant que les opérations utilisées ne sont pas celles du type, l'échange d'information au cours de l'exécution se limite à des capacités. Comme par ailleurs, il est logique de placer sur un même site les représentations des objets d'un type et les opérations qui y accèdent, les échanges réels d'informations se feront localement à un site. On peut ainsi facilement limiter les échanges entre sites à des échanges de capacités (*). Le placement d'un objet et des opérations relatives à son type sur le même site peut être renforcé par fait que la localisation du type est réalisée à travers le descripteur de l'objet (figure 9). On remarque que les opérations d'un type sont localisées à l'aide de leur nom spatial, on peut donc envisager plusieurs copies de ces opérations possédant chacune un nom spatial différent. Ainsi quand un objet doit être placé dans le réseau, il est aisé ou bien de la placer sur un site où il existe une copie des opérations de son type, ou bien de créer une copie de ces opérations.

(*) Plus précisément, on limite ainsi les transferts de représentations d'objets. Lorsque des ressources non banalisées sont en jeu, ces transferts ne peuvent être évités. Par exemple, l'impression d'un fichier nécessitera toujours sa migration vers le site où se trouve l'imprimante.

On notera cependant que ceci ne fonctionne que lorsqu'on considère des types construits. On doit aussi éviter qu'une procédure ne fasse accès à un segment où une C-liste placés sur un autre site. Ceci peut être renforcé facilement par une méthodologie appropriée selon laquelle les seuls objets primitifs utilisés sont locaux à une procédure ou participent à la structure d'un objet de type construit. Il n'y a ainsi partage que d'objets de types construits.

Nous allons supposer qu'il existe dans le réseau un générateur de noms uniques fiable. Chaque objet peut ainsi être muni d'un nom unique global dans le réseau et tout comme précédemment chaque descripteur d'objet dans la table active reste semblable, la nouveauté est que nous allons considérer l'existence d'une table active locale sur chaque site. Ainsi le même nom spatial sur deux sites différents ne correspond pas en général au même objet. Aussi nous faisons reposer les communications entre sites sur l'utilisation des noms uniques globaux. Seules des capacités sous forme passive peuvent être échangées entre sites. Tout comme dans le schéma précédent, sur chaque site, une "hash-table" permet de réaliser la correspondance entre le nom unique global et le nom spatial local afin de traduire la capacité passive reçue en une capacité active.

Lorsqu'une opération appliquer est réalisée sur un objet, le nom spatial de son type permet de localiser la procédure et de l'activer. Si le type de l'objet n'a pas d'implantation sur le site, il n'en existe pas de descripteur d'une procédure de service. La localisation du site support du type est la même que celle de la représentation de l'objet. On supposera ici que cette localisation est réalisée à l'aide du nom du site. La procédure de service du noyau reçoit les paramètres tout comme s'il s'agissait de la procédure invoquée par l'appelant. Après avoir remis les capacités dans leur forme passive et localisé le site porteur du type invoqué, le transfert des paramètres vers le site approprié est réalisé à l'aide des mécanismes de communication du réseau. Inversement lorsqu'un site reçoit un tel message, il crée un processus dont l'exécution est lancée sur une procédure de service dont le rôle est de localiser le type et d'activer la procédure invoquée depuis le site demandeur. Ainsi deux processus sont en communication pour réaliser le service demandé : le processus demandeur, en attente dans l'environnement d'une procédure de service, et le processus créé sur le site support du type.

L'activation des objets se réalise éventuellement en deux étapes. On peut imaginer qu'au cours de la création du descripteur actif sur le site demandeur, la localisation de l'objet ne soit pas encore connue ; tout comme dans le schéma précédent l'objet pouvait ne pas encore être en mémoire. Une deuxième étape est alors nécessaire pour permettre, à l'aide des noms uniques globaux, de localiser le site des objets. De même, on notera qu'un objet peut éventuellement avoir été déplacé, alors le site cible adressé peut ne pas posséder la représentation de l'objet. Cependant le même procédé peut être répété. Ainsi, le descripteur de l'objet sur ce site peut être utilisé comme un relais et renvoyer à nouveau vers un autre site et ainsi de suite. Le problème dans un tel schéma serait d'éliminer progressivement les descripteurs relais. On notera cependant que les descripteurs créés sont connus à l'aide du mécanisme d'activation. Des mécanismes de mise à jour peuvent donc être réalisés. Ces problèmes ne seront pas abordés ici.

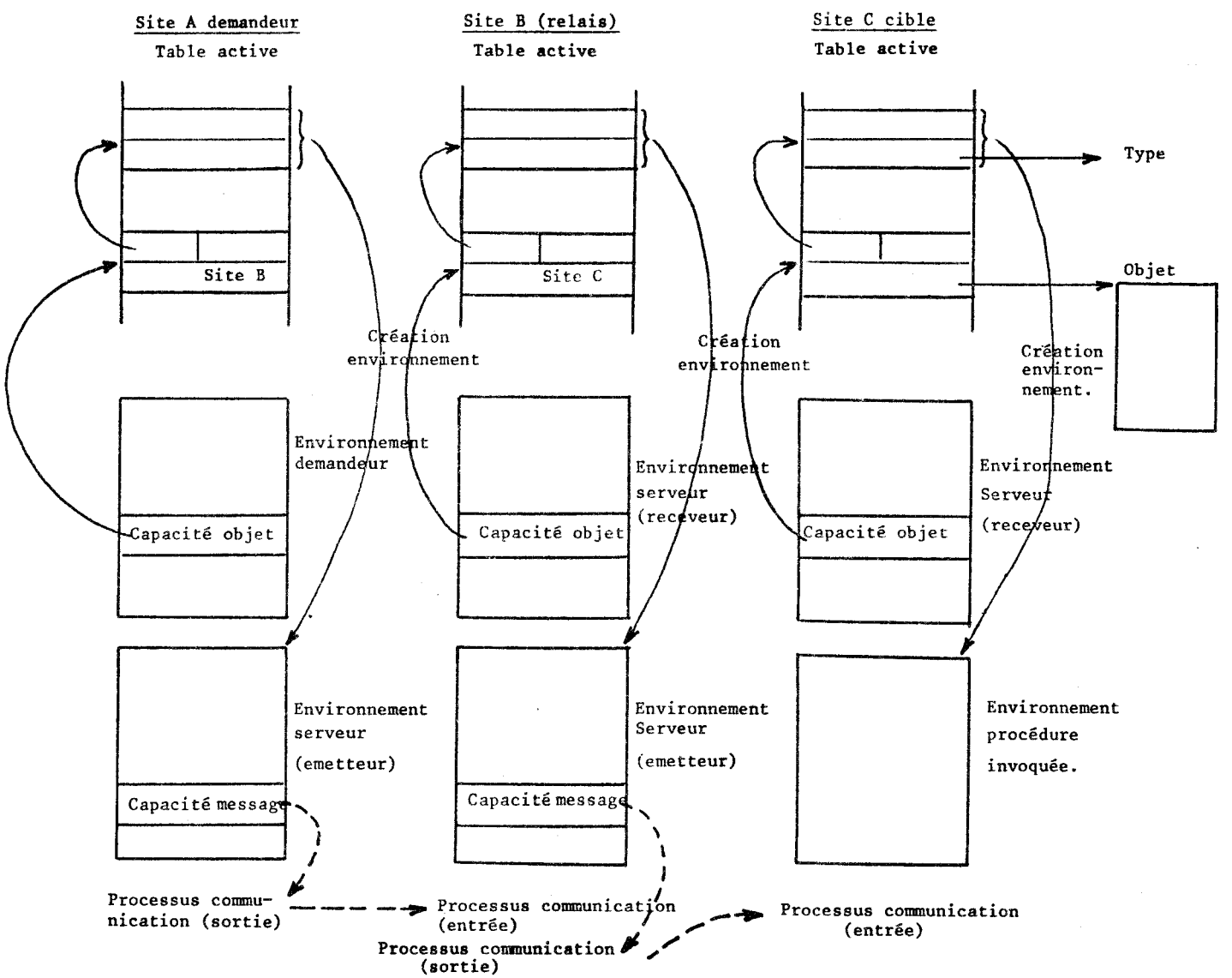


Figure 9 : Extension aux réseaux

DEUXIEME PARTIE

GESTION DES OBJETS

CHAPITRE III

GESTION DES OBJETS

1. - Evaluation des mécanismes

Nous avons noté l'intérêt de l'utilisation des mécanismes à capacités sur le plan du partage et de la gestion des noms. Cependant, il en résulte d'autres problèmes que l'on peut classer en deux catégories :

- ceux qui sont relatifs à la gestion des accès aux objets
- ceux qui sont relatifs à la gestion de la mémoire.

1.1. - Accès aux objets

On a remarqué que la TUOS du système peut être de taille importante (sans doute de l'ordre de 10^8 entrées dans un système de taille importante). Ceci résulte de la petite taille des objets due au découpage en termes de types et au principe du moindre privilège. Ainsi, alors qu'en général dans un système segmenté conventionnel la table des objets d'un processus est rendue résidente afin d'accélérer les accès, il ne peut en être de même pour la TUOS. Ceci nous a conduit à distinguer une table active et une table passive. Lorsqu'un objet est utilisé pour la première fois, il ne possède pas encore de descripteur dans la table active et la création de celui-ci nécessite une recherche dans la table passive. De même lorsqu'un objet est créé, cette création est réalisée dans la table active et doit être transmise ultérieurement dans la table passive. Plus les objets sont nombreux et plus il est probable que l'accès à un objet requiert son activation. Par ailleurs de nombreux objets sont de petite taille [Bat70] alors que la taille d'un descripteur dans la table active peut atteindre 64 octets. Plus les petits objets sont nombreux et plus l'utilisation de la mémoire est inefficace or comme nous l'avons noté plus haut, les principes utilisés conduisent à des petits objets.

Il convient à ce sujet d'interférer avec le paragraphe suivant. Il est relativement aisé, en principe, de gérer la mémoire d'un système à capacités et d'éviter la fragmentation due aux petits objets. En effet, ils sont décrits dans une table unique. Cependant ceci conduit à des transferts fréquents car ils portent sur des petits objets. Le transfert de tels objets ne réalise pas une utilisation efficace des mémoires secondaires sur lesquelles l'accès est lent et le transfert rapide.

Aussi, le cas le pire, qui consiste, pour accéder à un objet, à devoir d'abord créer un descripteur actif puis à charger l'objet en mémoire est un cas assez probable.

Or il est coûteux en temps comparativement à la taille des transferts. Il convient donc de définir des techniques nouvelles qui permettent d'améliorer l'efficacité de tels systèmes sous peine de voir les méthodes développées inutilisées. Certains concepteurs ont ainsi proposé de regrouper les petits objets dans des pages,contrairement à la technique habituelle qui consiste à décrire un segment dans plusieurs pages. Cette proposition a pour but de maintenir l'efficacité des transferts d'entrée/sortie en choisissant une taille appropriée des pages. Le problème est simplement de définir des critères de regroupement qui tiennent compte de la localité des références. Les objets étant partagés, il ne suffit pas de connaître le comportement d'un programme mais celui de l'ensemble des programmes. Or les intersections des ensembles d'objets que ces programmes manipulent peuvent être très complexes.

La partie 2 de ce chapitre propose une solution à ces problèmes. Elle consiste à regrouper statiquement les objets à l'intérieur d'un même segment.

Enfin, on notera que tout accès est réalisé par indirection à travers le descripteur de l'objet. Ceci est typique des mécanismes utilisant des tables de segments. Compte tenu du regroupement libre des objets dans les C-listes et du découpage en petits objets, il est probable qu'un objet soit accédé après plusieurs niveaux d'indirections à travers des C-listes. Mis à part les problèmes d'adressage qui en résultent il convient de remarquer que chacune de ces indirections dans une C-liste requiert elle-même une indirection à travers son descripteur. De tels mécanismes peuvent être accélérés par l'utilisation d'une mémoire associative [Fab 74]. Cependant avec de nombreux objets la probabilité est forte pour que l'objet référencé ne soit pas trouvé dans la mémoire associative. Ceci réduit son efficacité et conduirait sans doute à l'utilisation d'une mémoire associative plus importante qu'à l'habitude.

1.2. - Gestion de la mémoire

Le problème de la fragmentation de la mémoire par suite de l'existence de nombreux objets n'est qu'un problème mineur des systèmes à capacités. Le problème essentiel relatif à la gestion de la mémoire provient de ce que tous les objets ne sont accessibles que par référence. Lorsqu'il n'existe plus de capacité repérant un objet, celui-ci ne peut plus être utilisé et l'espace qu'occupe sa représentation peut être récupéré par le système. Cette disparition des références vers un objet est fréquents, car les capacités peuvent être copiées et effacées librement.

Cependant s'il n'existe plus de référence vers l'objet, celui-ci ne peut plus être identifié et l'espace qu'il occupe ne peut plus être récupéré. Il convient d'être capable de localiser l'espace mémoire inaccessible.

On remarquera que ce problème est facilement éliminé dans un système conventionnel où seuls sont conservés les objets qui ont une entrée dans les catalogues. Dès que cette entrée est détruite, et elle l'est explicitement, l'espace réservé peut être réutilisé. Les catalogues sont le seul moyen d'obtenir une référence à un objet. Une telle pratique est inutilisable dans un système à capacités car les seuls objets répertoriés dans les catalogues sont ceux dont l'utilisateur à conscience. Ces objets ont des composants qu'ils partagent éventuellement avec d'autres et qui ne sont pas visibles au niveau de l'utilisation.

Ce problème de localisation des objets non référencés s'est d'abord présenté dans les langages de programmation. Il conduit à l'utilisation d'un programme de ramasse-miettes. Dans les systèmes classiques la seule utilisation d'un ramasse-miettes a lieu à la disparition du processus. Il est alors relativement aisé de procéder à la récupération de l'espace utilisé car le processus lui-même désigne par son espace d'adressage l'ensemble des objets qui cessent d'être utilisés. A cause du partage, une telle méthode n'est pas utilisable ici. Au contraire, le ramasse-miettes consiste à détecter les objets encore accessibles ; dans l'ensemble des objets, ceux qui sont inaccessibles sont ceux qui n'appartiennent pas à l'ensemble des objets accessibles. Ce procédé, simple en apparence, est en fait complexe car il nécessite de parcourir l'ensemble de la mémoire utilisée. Ainsi, Hydra demande 3 minutes pour procéder au ramasse-miettes d'environ 20 000 objets, encore toute la mémoire centrale disponible est-elle utilisée. Le temps nécessaire au ramasse-miettes peut être considéré comme proportionnel au nombre d'objets sur lequel il doit être réalisé. Il est évident que d'autres méthodes doivent être trouvées si l'on doit procéder à ce travail sur un nombre beaucoup plus conséquent d'objets.

Décider qu'un objet est inaccessible consiste a priori à compter le nombre de références sur l'objet, c'est-à-dire le nombre de capacités qui le repère. Hydra utilise cette technique. Chaque descripteur de la table passive est muni d'un compteur qui est mis à jour lors de chaque opération portant sur une capacité. Il en est de même dans la table active où chaque descripteur est muni d'un compteur de capacités actives et d'un compteur total dont la valeur initiale est fournie par la table passive. Le premier est utilisé pour gérer la table active, le second pour aider au ramasse-miettes. On remarquera que le compteur de la table passive ne peut être mis à jour à chaque modification des compteurs de la table active. Dans le cas des objets de type "type", l'opération de mise à jour des compteurs est plus complexe car le compteur doit refléter le nombre de capacités

qui portent le nom du type, c'est-à-dire les capacités repérant l'objet type aussi bien que les capacités repérant un objet de ce type. Ceci contribue à ralentir les opérations de manipulation des capacités. Il convient de faire deux remarques concernant l'utilisation de compteurs.

Les compteurs ne reflétent pas toujours la réalité et ne permettent pas de localiser tous les objets inaccessibles. Il existe des objets repérés par des capacités qui pourtant sont inaccessibles. C'est le cas des structures bouclées telles que celle de la figure 1. Dans cette figure 1'objet A contient une capacité vers 1'objet B qui lui même contient une capacité vers A. Supposons que ces capacités soient les seules capacités repérant A et B, alors A et B sont de fait inaccessibles, pourtant ceci ne sera pas détecté. Un exemple de boucle est celui des objets "type". Un objet type contient une capacité pour chaque opération du type qui elles-même contiennent une capacité vers le type afin de réaliser la décomposition des objets. Seul un programme de ramasse-miettes peut détecter les boucles.

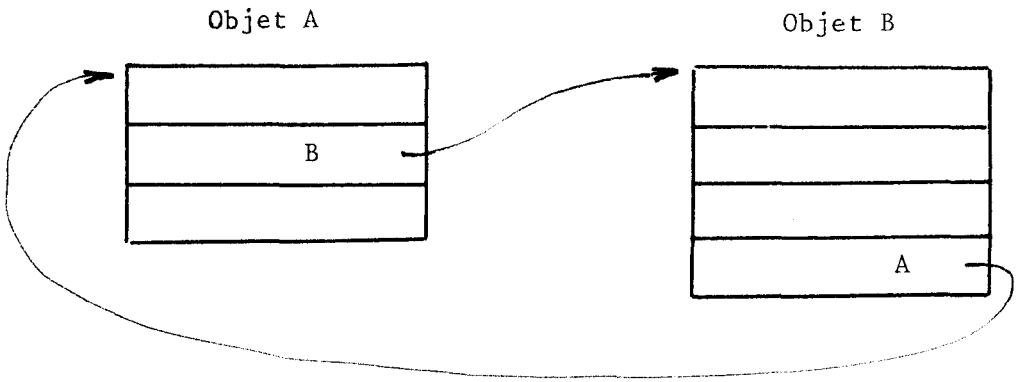


Figure 1 : Boucle de références

La seconde remarque concerne la fidélité des compteurs. On a noté que les compteurs de la table passive ne reflètent pas exactement l'état courant de l'objet en ce qui concerne le nombre de capacités qui le repèrent. Supposons la séquence suivante :

a) l'objet est activé avec un compteur de valeur n que fournit la table passive ;

b) une copie de capacité est réalisée, la valeur du compteur dans la table active est donc n + 1 alors que dans la table passive elle est n ;

c) n effacements de capacités sont réalisées.

Supposons maintenant qu'une erreur produise un écroulement du système entre les opérations b) et c). Si l'on réactive l'objet à l'aide de son descripteur dans la table passive, à la suite de l'opération c), le compteur atteindra la valeur zéro et l'objet sera détruit alors que de fait il en reste une capacité. Ceci conduit dans un tel cas à utiliser la valeur maximale du compteur et à rendre les tables aussi fiables que possible de sorte que leur contenu soit utilisable. Cependant, il se peut que la situation soit telle qu'après la restauration du système, la capacité copiée au cours de l'opération b) ait disparu (par exemple la séquence complète a été réinitialisée). Alors, au contraire le compteur a maintenant une valeur trop élevée et il ne sera jamais possible de détecter si l'objet est inaccessible. Tout au plus on peut affirmer que lorsqu'un compteur atteint la valeur zéro, l'espace occupé par l'objet est réutilisable, cependant la récupération complète de l'espace doit être réalisée par un ramassage de miettes. La partie 3 de ce chapitre présente une solution qui peut contribuer à rendre celui-ci plus performant.

2. - Une méthode de regroupement des petits objets [LSW 78]

2.1. - Composition des objets

Avant de décrire ce mécanisme, il convient de faire quelques remarques. Une première observation concerne la distinction entre les objets partagés et les objets qui sont simplement des composants d'autres objets. (Parfois les composants d'un objet sont également partagés, ce serait le cas des objets d'un catalogue). Considérons l'exemple d'un fichier partagé. Afin de permettre les opérations d'ouverture et de fermeture il est muni d'un sémaphore. Plusieurs programmes peuvent avoir une capacité pour le fichier et invoquer des opérations sur celui-ci. Le mécanisme à capacités permet à ces différents programmes d'avoir des droits différents sur le fichier. Cependant, seules les opérations du type fichier peuvent invoquer des opérations sur le sémaphore. Le sémaphore a été créé par le type fichier pour implanter le fichier ; il n'y a aucune raison pour qu'un autre sous-système ait une capacité pour cet objet. Il en résulte que les droits concédés sur un composant ont un intérêt limité. Les opérations du type fichier peuvent seulement utiliser les droits des capacités des composants comme

une protection contre elles-mêmes. Les capacités des composants d'un objet sont utilisées essentiellement comme des références et par nécessité de désigner le type de ces composants.

Notre mécanisme reconnait la différence entre objets partagés et composants. Il distingue entre objets externes et objets internes. Les objets externes correspondent aux objets d'une liste de capacités conventionnelle. Ils ont un descripteur dans la TUOS, sont désignés à l'aide de capacités et peuvent être librement partagés. L'espace qui leur est attribué en mémoire leur est propre et est différencié de celui qui est alloué à tout autre objet externe.

Au contraire, les objets internes sont utilisés pour mettre en place des composants. Leur caractéristique principale est que l'espace mémoire leur est attribué dans un objet externe qui les contient. En conséquence, il n'est pas nécessaire que les objets internes aient un descripteur dans la TUOS. L'information qui autrement se trouverait dans ce descripteur est conservée dans les objets externes qui les contiennent. Bien que les objets internes soient conçus spécifiquement pour représenter des composants, ceux-ci peuvent être implantés aussi bien comme des objets internes que comme des objets externes.

Dans un système à capacités tel que celui présenté au chapitre II, un objet de type construit est représenté par une liste de capacités qui contient les capacités de ses composants ou de segments. Avec ce nouveau mécanisme chaque objet construit est représenté par une TOL (table des objets locaux). La TOL possède une entrée pour chaque composant. Ceux-ci peuvent être de type primitif ou de type construit. Dans ce dernier cas ils peuvent alors eux-même être représentés à l'aide d'une TOL. Quand un composant est représenté à l'aide d'un objet interne, son entrée dans la TOL est un descripteur qui décrit l'espace qu'il occupe (base et limite) dans l'objet externe qui le contient. Si le composant est externe, alors son entrée dans la table est une capacité. La figure 2 montre un exemple. Comme les objets internes peuvent en contenir d'autres, un objet externe peut contenir un arborescence de TOLs. La TOL d'un objet externe composé uniquement d'autres objets externes est équivalente à une C-liste dans le système du chapitre II.

Le mécanisme proposé devrait présenter des avantages sur le plan des performances pour plusieurs raisons. Puisque les objets internes n'ont pas d'entrée dans la TUOS, celle-ci devrait être plus petite. Des mesures relevées sur Hydra à l'état passif montrent que dans la table passive, il y a 75 pour cent des objets dont le compteur de référence possède la valeur 1. Ce sont des objets qui a priori sont des composants d'autres objets.

Quand un objet est transféré de la mémoire secondaire à la mémoire principale et vice-versa, tous ses composants internes sont automatiquement transférés aussi. Le transfert est plus efficace puisque d'avantage de données sont transférées pendant une seule opération d'entrée-sortie et que les composants n'ont pas à être transférés séparément. Le premier accès à un composant est également accéléré puisqu'il est déjà en mémoire et l'indirection à travers un descripteur a lieu dans la TOL qui est déjà en mémoire et non dans la TUOS où il faudrait créer l'entrée correspondante.

Les objets internes n'ont pas besoin d'entrée dans une mémoire associative. Une entrée pour l'objet externe est suffisante pour accéder à un composant interne. Il en résulte que la gestion de la mémoire associative devrait être plus efficace.

Ce mécanisme a cependant un inconvénient. Puisque les objets internes peuvent à leur tour en contenir d'autres, la taille d'un objet interne est fixée au moment de la création de l'objet externe qui le contient. L'expérience laisse à penser que cette contrainte n'est pas sérieuse. Souvent la définition dynamique de la taille des objets n'est pas nécessaire. De plus, si l'on souhaite faire varier dynamiquement la taille d'un composant, alors celui-ci peut être implanté comme un objet externe.

Nous utilisons à titre d'exemple la structure des "superfichiers" d'Hydra [Alm 78]. Un superfichier est formé de :

- un sémaphore utilisé pour synchroniser les accès au fichier,
- un segment qui contient les informations relatives à la gestion du fichier telles que : la date de création, le nom du propriétaire ;
- le fichier proprement dit.

L'opération ouvrir (< *mode d'accès* >, < *superfichier* >) réalise une opération P sur le sémaphore du superfichier et retourne une capacité pour le fichier lui-même avec les droits d'accès demandés. Le sémaphore et les informations relatives à la gestion de fichier sont seulement accessibles par les opérations du type superfichier lors de l'ouverture et de la fermeture du superfichier. Ils ne sont donc pas partagés ; ils pourraient être implantés comme des objets internes. Le fichier composant, cependant, est potentiellement partagé (pour des lectures simultanées par exemple). En conséquence, il doit être implanté comme un objet externe. La figure 2 schématise une implantation possible.

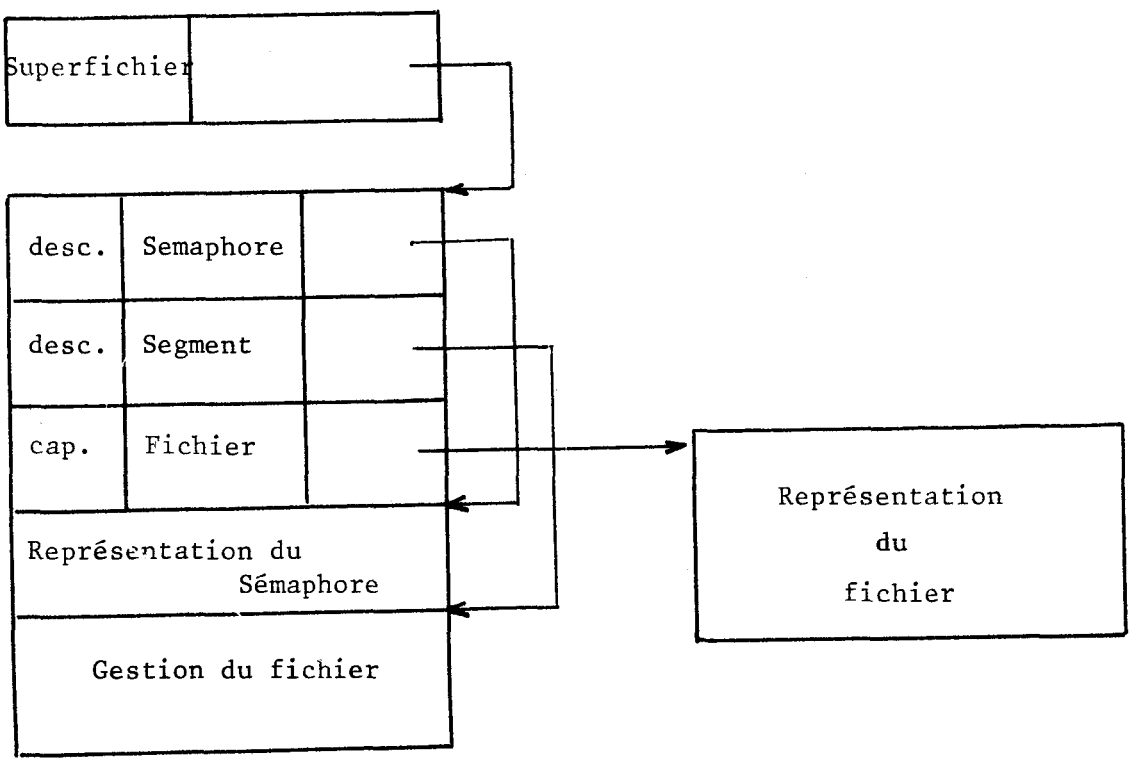


Figure 2 : Structure d'un superfichier

2.2. - Représentation des objets

On rappelle que chaque objet est référencé à travers un descripteur qui regroupe également les informations relatives à son état. Les descripteurs des objets externes sont situés dans la TUOS, ceux des objets internes sont dans la TOL de l'objet externe qui les contient. L'état d'un objet interne du point de vue de la gestion de la mémoire est celui de son objet externe. On se limite ici à l'implantation du mécanisme à capacités. Le descripteur d'un objet dans la TUOS contient grossièrement :

- son nom unique
- une adresse physique et la longueur physique de l'objet,
- le nom du type de l'objet,
- un indicateur de type construit,
- la longueur de sa TOL racine.

De la même manière, le descripteur d'un objet interne dans la TOL d'un objet externe contiendra :

- un indicateur interne/externe signifiant qu'il s'agit d'un descripteur d'objet interne ou d'une capacité d'un objet externe,
- une adresse locale et la taille de l'objet interne dans le segment support de l'objet externe,
- le nom de son type,
- un indicateur de type construit,
- un ensemble de droits.

Les composants internes d'un objet peuvent eux-même être des objets de type construit et donc éventuellement être représentés à l'aide d'une TOL et leurs opérations être implantées par les procédures d'un type. Quand les opérations sont invoquées une capacité doit leur être passée en paramètre même si l'objet est un composant interne. Il est donc nécessaire de définir des capacités pour les objets internes.

De fait, une capacité étant une référence vers un descripteur, cette définition est simple. La référence d'une capacité d'objet interne est formée de deux parties :

- le nom de l'objet externe qui le contient,
- le nom d'index du descripteur de l'objet interne dans l'objet externe.

Une capacité d'objet interne peut donc être formée directement à partir de la capacité de l'objet externe (ou à partir de celle d'un autre objet interne) et du nom d'index de l'objet interne dans la TOL qui contient son descripteur.

Les utilisateurs d'objets de types construits reçoivent donc des capacités qui leur permettent d'accéder à ces objets (par l'invocation d'opérations de leur type) et de partager ces objets avec d'autres utilisateurs. Puisque ces objets sont de type construit cependant, ces utilisateurs ne peuvent accéder à leurs composants. Quand une opération d'un type est invoquée, elle décompose l'objet et obtient ainsi l'accès à sa TOL racine qu'elle voit comme une C-liste. Lorsque l'opération lit une entrée de la TOL qui correspond à un objet externe, elle en reçoit la capacité. Quand au contraire cette entrée contient le descripteur d'un composant interne, une capacité interne est créée et fournie à l'opération. Dans les deux cas, il peut s'agir de la capacité d'un objet de type construit.

Revenons à l'exemple du superfichier. La décomposition d'un superfichier fournit au type superfichier l'accès à une TOL contenant trois éléments. L'un est le descripteur du sémaphore. La lecture de l'entrée correspondante de la TOL fournit au type une capacité interne pour un objet de type sémaphore. Après décomposition de cet objet par le type sémaphore, cette capacité donnera à celui-ci l'accès à une TOL à deux entrées décrivant le sémaphore (voir figure 3). L'une de ces entrées contient la capacité d'un objet externe : le premier processus de la liste des processus bloqués derrière la sémaphore.

Il convient de remarquer que l'implantation des objets de très petite taille dans ce schéma ne requiert pas de traitement particulier. Le but des capacités de données par exemple,ou du rangement dans le descripteur d'un objet de la capacité de son unique composant,était de limiter la fragmentation et d'améliorer les performances relatives au chargement. De tels mécanismes ne se justifient pas en ce qui concerne les composants internes d'un objet. Ainsi, la valeur du sémaphore serait sans doute rangée dans un petit segment interne à l'objet superfichier.

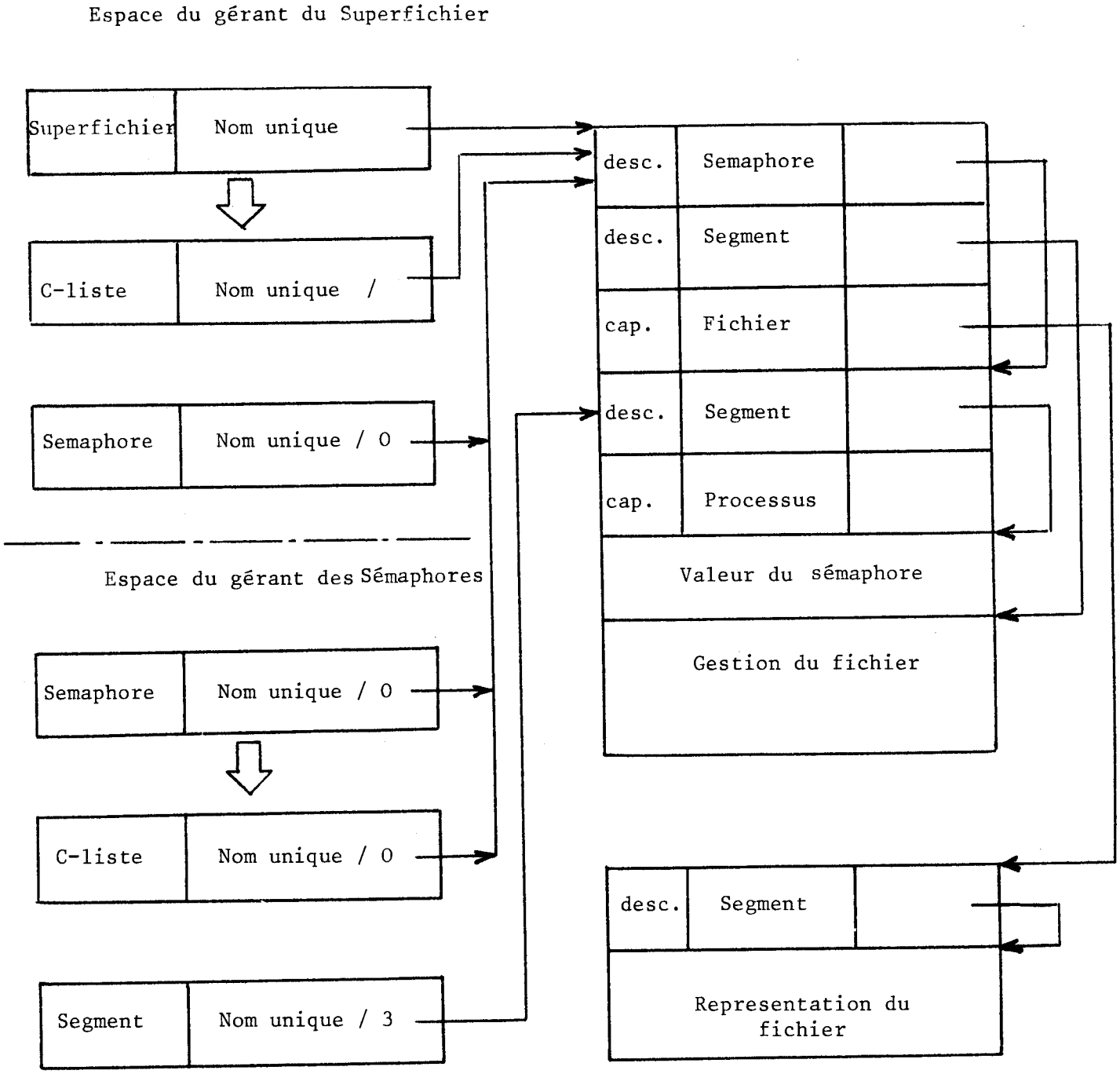


Figure 3 : Décomposition d'un superfichier

2.3. - Création d'un objet

La création d'un objet externe comprend l'allocation de la mémoire pour contenir cet objet.On appellera la mémoire allouée à un objet externe, son bloc. Si l'objet est de type construit, ce bloc comprend également l'espace alloué à sa TOL et à tous ses composants internes. Ces composants internes peuvent eux-mêmes être de type construit et être formés de composants internes ou externes. En accord avec les principes de l'"information cachée", et de l'"abstraction des données", il est nécessaire qu'un type :

- 1/ définisse s'il convient que les composants utilisés dans la création d'un objet du type soient internes ou externes,
- 2/ ne soit pas conscient de la manière dont les objets de son type sont utilisés,
- 3/ ne soit pas conscient de la manière dont les composants des objets de son type sont implantés (qu'ils soient internes à d'autres structures ou externes).

Il est clair que ces contraintes ne conviennent pas à un schéma selon lequel le bloc nécessaire à la création d'un objet externe serait dilaté au fur et à mesure des besoins pour inclure des nouveaux composants internes. Selon ce schéma, chaque type demanderait au noyau la taille de mémoire dont il a besoin et ensuite invoquerait les fonctions de création des composants de l'objet qu'il fabrique qui a leur tour demanderaient l'intervention du noyau pour augmenter la taille du bloc utilisé. Un tel schéma contredit les principes évoqués ci-dessus. La seule alternative est d'allouer l'ensemble du bloc nécessaire en une seule fois à l'aide d'une fonction du noyau. On utilise à cet effet des maquettes de création.

Lorsqu'un nouveau type est créé une maquette est associée à ce type. L'utilisation de cette maquette pour créer un objet du type requiert éventuellement des paramètres. Une maquette décrit la structure du type auquel elle est associée en termes des maquettes de ses composants. Ces maquettes de composants sont implantées comme des composants internes de la nouvelle maquette. Le noyau fournit une opération générale créer-objet qui reçoit en paramètres une maquette et les paramètres nécessaires à son interprétation. Cette opération crée un objet du type que définit la maquette,y compris ses composants, leurs composants et ainsi de suite jusqu'au niveau des objets de type primitif (que le noyau sait comment créer). Ainsi, il n'existe pas d'opération de création au niveau de chaque type et la création d'un objet ne requiert pas l'invocation en cascade des opérations de création correspondant à ses composants.

En fait, avant de détailler la création des objets, il convient de noter quelques cas particuliers. D'abord, ce type de maquettes peut ne pas être limité aux composants internes d'un objet. L'opération créer-objet du noyau peut être étendue aux composants externes. De même, il est possible de passer des objets externes en paramètres à l'opération de création. Dans un tel cas, l'entrée correspondante dans la maquette serait une entrée vide pour une capacité et une indication que cette capacité doit être reçue en paramètre au moment de la création de l'objet. Ainsi un objet peut être totalement créé et initialisé par le noyau, contrairement au mécanisme d'Hydra qui n'assure que la création du bloc et sa décomposition en C-liste et segment.

Deuxièmement, ce mécanisme suppose que les maquettes des composants sont interprétées et insérées dans la nouvelle maquette. Cependant, il est des cas où une telle inclusion devrait être évitée. Considérons le cas où une modification de l'implantation d'un type est nécessaire. Selon un tel schéma il se pourrait que des maquettes ne soient plus valables. Ce serait le cas non seulement des maquettes correspondant aux types modifiés mais aussi de toutes celles qui ont été interprétées à partir d'elles. Cette situation peut être traitée en créant des nouveaux types au lieu de modifier les types existants de sorte que tous les types construits à partir de ceux là soient eux aussi remplacés (c'est la méthode utilisée dans Hydra au cours du développement du système; lorsqu'une amélioration d'un sous-système est décidée, un nouveau type est créé qui coexiste avec l'ancien jusqu'à disparition de ses utilisateurs). Une autre possibilité est d'éviter l'inclusion des maquettes dont l'implantation des types n'est pas encore définitive. Ainsi une maquette devrait contenir aussi bien des références externes qu'internes à d'autres maquettes.

Enfin, le noyau n'est capable de créer que des objets pour lesquels il contrôle la ressource utilisée, c'est-à-dire pour lesquels il est capable d'allouer la ressource nécessaire. Il y a des cas où la création d'un objet par le noyau ne peut être complète parce qu'il n'est capable que de leur allouer un bloc et non quelque ressource nécessaire mais gérée à un plus haut niveau. Une telle création d'objet nécessite l'aide d'une opération de création fournie par le type correspondant.

Conceptuellement, du point de vue de l'utilisateur d'un type, celui-ci se caractèrise par un objet du type "type" qui regroupe les opérations de ce type. Dans le cas de l'opération créer, la capacité sera aussi bien celle d'une maquette. Quand un utilisateur appelle l'opération créer, le noyau interprète la capacité correspondante ou bien comme celle d'une procédure ou bien comme celle d'une maquette. Dans le dernier cas,il l'interprète directement

pour créer l'objet. De fait, le mécanisme reste uniforme dans les deux cas et on constatera au chapitre IV que le noyau interprétera toujours cette capacité comme celle d'une maquette.

Nous pouvons maintenant détailler les maquettes. Chaque objet de type construit est représenté à l'aide d'une TOL que l'on peut voir comme une structure (au sens d'Algol 68 par exemple). Certains éléments de cette structure peuvent être des tableaux de références. La maquette d'un objet de type construit est formée d'une maquette de descripteur pour chaque composant et d'informations relatives au nombre de descripteurs, à la taille de la maquette et aux paramètres nécessaires à son interprétation. On distingue trois types de maquettes de descripteur selon qu'elles sont relatives à :

- des objets existants,
- la maquette d'un type,
- un composant.

Le descripteur d'un objet existant est simplement une capacité pour cet objet. Quand la maquette est interprétée par le noyau, la TOL créée pour le nouvel objet contient simplement une copie de cette capacité.

Le descripteur d'un type est relatif à l'opération de création de ce type (ou à la maquette). Quand la maquette est interprétée, cette opération est invoquée et la capacité reçue en retour est insérée dans la TOL du nouvel objet.

Enfin une maquette de composant est la copie d'une maquette. Elle est un composant interne de la maquette de l'objet à créer, de sorte que les maquettes peuvent elles aussi inclure d'autres objets de la même manière que les objets en général peuvent avoir des composants internes.

Une maquette de descripteur contient donc :

- un indicateur externe/interne signifiant si le composant sera référencé par une capacité ou un descripteur interne,
- le type du descripteur,
- le type du composant et des droits,
- un pointeur vers la maquette du composant (sa copie dans le bloc ou, si le composant est de type primitif et sa valeur connue, un pointeur vers cette valeur initiale),
- la longueur du composant,
- éventuellement un facteur de répétition pour permettre la description des tableaux.

Une maquette peut être paramètrée. Dans ce cas, le noyau lie les paramètres dans l'ordre où il interprète les maquettes de descripteur. Pour chaque champ qui peut être paramètré dans un descripteur, un indicateur indique l'une des trois possibilités suivantes :

- le champ est déjà résolu,
- un paramètre est attendu,
- le champ reste indéfini.

Afin d'illustrer la représentation des maquettes, nous utiliserons l'exemple du superfichier déjà présenté. Afin de simplifier cet exemple, on considérera que le composant fichier est simplement un segment.

La création d'un superfichier nécessite que le segment qui contient les informations relatives à la gestion du fichier soient disponibles. Ceci ne peut être fourni ni par le noyau, ni par l'utilisateur. Par conséquent, on supposera qu'il existe une opération de création qui reçoit des paramètres tels que la longueur du fichier et fournit les autres informations nécessaires. La demande de création d'un superfichier active cette opération qui elle même appelle le noyau en lui présentant la maquette "superfichier" et les paramètres nécessaires à son interprétation (la longueur du segment par exemple). Le noyau retourne alors à cette procédure un objet qui n'est pas complètement initialisé. En décomposant l'objet la procédure de création est alors capable d'accéder aux composants et de compléter son initialisation. La figure 4 schématise la structure de la maquette d'un superfichier.

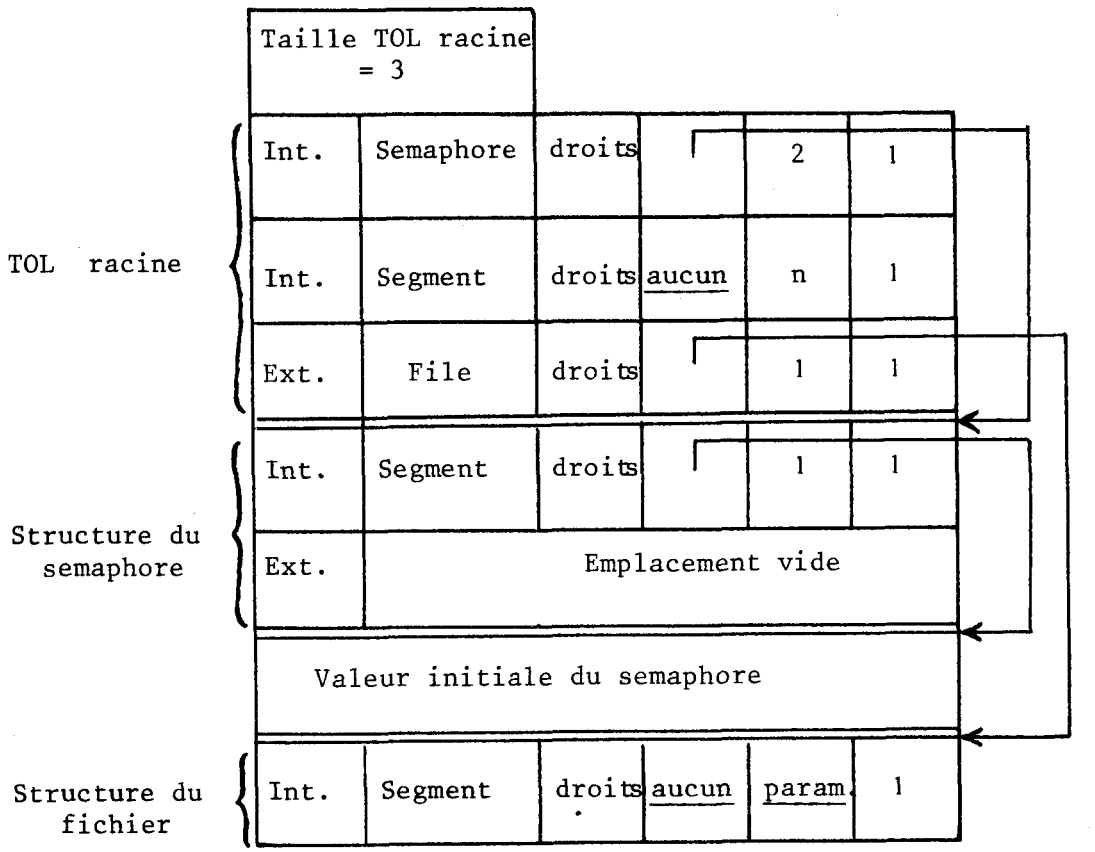


Figure 4 : Format de la maquette d'un superfichier

2.4. - Création des maquettes

Une maquette est un objet dont le type est implanté par composition d'un segment dans le noyau. La définition d'un nouveau type est essentiellement la création de sa maquette. Deux opérations sont définies sur le type maquette : l'opération créer-objet déjà décrite et l'opération copier qui consiste à copier une maquette comme un composant interne d'une autre maquette. Le but de cette opération est de permettre la création d'une maquette d'un nouveau type. Il y a des cas où cette copie ne devrait pas être réalisée (quand il est possible que l'implantation du type soit modifiée) ; elle est donc contrôlée en définissant un droit de copier. Les maquettes qui utilisent des composants dont les maquettes ne peuvent être copiées contiennent des capacités externes pour ces maquettes.

Une maquette décrit la structure d'un type en termes des types des composants. Pour créer une maquette trois sortes d'informations sont nécessaires :

- les capacités des types composants,
- la description de la structure des objets du nouveau type,
- les paramètres à utiliser pour évaluer les maquettes des types composants.

Deux commandes sont utilisées essentiellement pour décrire la structure des objets d'un nouveau type. La commande *structure <éléments>* définit une TOL, tandis que la commande *référence* < > décrit une maquette de descripteur. Cette dernière définit l'implantation du composant et quand cela est nécessaire décrit sa structure. La commande *référence* définit une référence soit vers un composant externe *<ext référence>* soit vers un composant interne.
Elle contient :

- 1/ la qualification du descripteur : le composant peut être indéfini, ou il doit être reçu en paramètre au moment de la création d'un objet du type, ou bien il est décrit par une maquette ou une structure désignée par la commande,
- 2/ une référence : soit vers une maquette soit vers une autre structure,
- 3/ une référence vers des paramètres à utiliser pour interpréter une maquette,
- 4/ un facteur de répétition pour permettre la création des tableaux.

De la même manière que la maquette elle même, la définition des champs d'une commande de référence peut nécessiter certains paramètres.

Reprenons l'exemple du superfichier. La déclaration de sa maquette s'écrirait :

Structure 3
int. réf. maquette SEMMAQUETTE, SEMINIT,1
int. réf. maquette SEGMAQUETTE, SEGLONG,1
ext. réf. maquette FICHMAQUETTE, *param* ,1

SEMMAQUETTE = maquette du type sémaphore
SEGMAQUETTE = maquette du type segment
FICHMAQUETTE = maquette du type segment
SEMINIT = 1
SEGLONG = n

Le schéma présenté devrait permettre de résoudre les problèmes qui ont été énoncés en 1.1 et qui sont relatifs à la gestion des accès. On remarque qu'il permet également la structuration des informations à l'intérieur des modules même si celles-ci ne sont pas destinées à être protégées à l'aide d'une capacité. Il fournit donc un support à la mise en place des structures du langage. On dépasserait le cadre de cette thèse en décrivant la mise en place complète du mécanisme. Celle-ci n'a été qu'esquissée et des modifications sont sans doute nécessaires pour des raisons d'efficacité. En particulier on notera que les opérations du noyau sont en principe indivisibles. La création d'un objet par interprétation d'une maquette peut être une opération longue et il n'est pas

envisageable de réserver le noyau à cet effet. La réalisation de création d'objets d'une manière concurrente par plusieurs utilisateurs peut donc être plus complexe. On notera toutefois que l'utilisation des maquettes permet d'éviter l'appel en cascade des opérations de création de chaque type composant puis de leurs composants et ainsi de suite jusqu'au dernier niveau de composition d'un type.

3. - Réalisation du ramasse-miettes

En fait l'exemple que nous allons présenter dans ce qui suit s'attaque à la fois au problème du regroupement des petits objets et au problème des références perdues. Le modèle présenté par Bishop [Bi 77] repose sur la remarque que dans un système de taille importante, le ramasse-miettes tel qu'il est effectué jusqu'à présent serait impossible à réaliser. Avec l'hypothèse que l'on ne tienne pas compte des problèmes de capacité de la mémoire, et que le temps nécessaire au ramasse-miettes soit proportionnel au nombre d'objets, Hydra demanderait 1 semaine pour réaliser le ramasse-miettes d'une mémoire de 10^{12} bits. Bischop propose donc de réaliser celui-ci en parallèle avec l'exécution des programmes utilisateur. Pour cela il est nécessaire de décrire au programme de ramasse-miettes l'ensemble des objets qu'il doit traiter.

3.1. - Notion de zone

Les objets sont regroupés en zones. Une zone est une collection d'objets qui a priori ont des relations entre eux. Un environnement est un exemple particulier de zone, une C-liste et les objets qu'elle repère peut en être un autre. Lorsqu'un objet est créé, l'utilisateur définit la zone dans laquelle il doit être créé, éventuellement l'objet sera ensuite déplacé par le ramasse-miettes. Ainsi la zone peut également être une unité de compte : l'espace mémoire alloué à un utilisateur peut être comptabilisé sur cette base.

Une zone peut se comparer à un segment dans un système classique. Elle est formée de pages et le travail du ramasse-miettes consiste entre autres à regrouper les objets dans des pages en fonction de la localité des références dans la zone. Comme la zone est l'unité utilisée par le programme de ramasse-miettes, il n'y a pas de table unique des objets mais l'équivalent d'une table locale des objets par zone. L'accès à un objet se fait en désignant la zone et l'objet dans la zone. Chaque zone peut également désigner des objets dans d'autres zones. A cet effet, on utilise des liens inter-zones (LIZ). L'utilisation

de ces liens a pour but de définir dans chaque zone ses références à l'extérieur et d'éviter ainsi la modification des noms dans une zone lorsque les objets qu'elle référence sont déplacés. Seuls les liens sont modifiés.

De même, lorsqu'un objet est déplacé dans une zone, il est nécessaire de retrouver les zones qui le référencent de sorte que leurs liens inter-zones puissent être modifiés. En conséquence, chaque zone est munie de deux listes de liens inter-zones. D'une part les liens sortants, définissent les objets qu'elle référence à l'extérieur. D'autre part, les liens entrants marquent les objets de la zone qui sont référencés depuis d'autres zones. Ces derniers identifient les zones concernées. Ainsi, lorsqu'une zone est traitée par le programme de ramasse-miettes, celui-ci peut identifier les zones qu'il va devoir traiter par conséquence. La figure 5 schématise l'utilisation des liens inter-zones. La procédure X de la zone A désigne un LIZ qui repère l'objet y de la zone B. Le lien appartient à la liste des liens sortants de A et à la liste des liens entrants de B.

Afin de maintenir les listes de liens, chaque lien est muni de deux champs : entrant et sortant. La figure 6 schématise le maintien des listes de liens. Le lien a de la zone A est un lien sortant pour la zone A, pour cette raison il est lié aux liens b et c de A. C'est aussi un lien entrant dans la zone B ; pour cette raison, il est repéré par le descripteur de B et lié au lien b de la zone A qui lui aussi est un lien entrant de la zone B. De même, il existe deux liens entrant dans la zone A, l'un dans C repère l'objet x, l'autre dans B repère l'objet v ; ainsi le lien d de B est repéré dans la liste des liens entrants de A et lié au lien e de C qui lui aussi est entrant pour A.

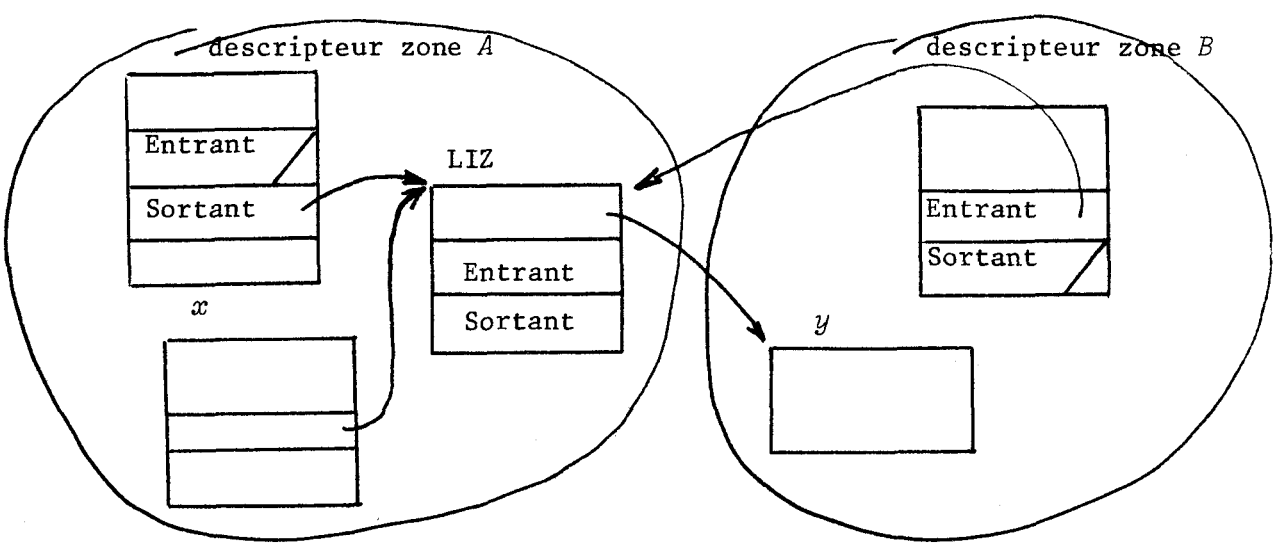


Figure 5 : Utilisation des liens inter-zones

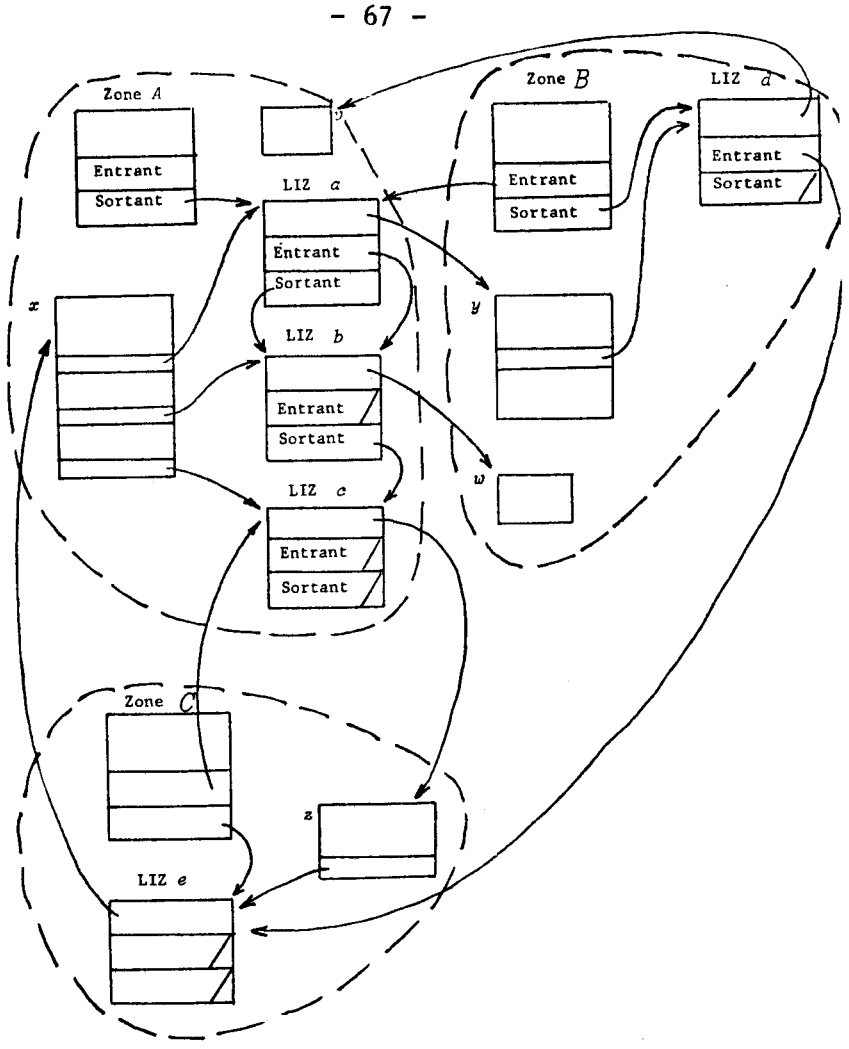


Figure 6 : Réalisation des listes de liens inter-zones.

Une zone peut également décrire un environnement. Cependant dans ce cas le mécanisme des liens inter-zone est trop lent pour être utilisé. Aussi, Bishop autorise un environnement à générer des références directes pour repérer un objet d'une autre zone. En fait, un environnement n'utilise que des références directes qui éventuellement sont générées automatiquement. Cependant si cette zone est traitée par le programme de ramasse-miettes, l'environnement doit l'être également, alors que l'utilisation des liens permettrait de ne transformer que les liens. Afin de définir les zones à traiter par le fait qu'elles utilisent des références directes, Bishop introduit des cables. Un cable est un lien qui ne repère pas un objet mais une zone ; les cables lient entre elles les zones qui utilisent des références directes. De même que pour les liens on trouvera des listes de cables entrants et sortants.

3.2. - Réalisation du ramasse-miettes

Le ramasse-miettes est réalisé par copie. Lorsqu'une zone est traitée, chaque objet pour lequel il est trouvé un lien entrant est recopié dans la nouvelle zone. Ainsi à la fin de la copie seuls subsistent les objets référencés. En fait, lorsqu'une zone doit être traitée, toutes celles qui lui sont connectées par des cables doivent l'être également. Pendant ce temps, il ne doit plus être possible de créer de nouveaux cables ou même de référencer des objets à l'aide de liens. Aussi toutes

les zones impliquées par le ramasse-miettes sont verrouillées. Lorsqu'un accès y est tenté le processus concerné est suspendu et placé en attente de la zone.

Eventuellement, il peut en résulter un "deadlock". Un environnement ne peut être traité par le ramasse-miettes si il est lié à une autre zone elle-même traitée. On a vu que toutes les zones cablées entre elles doivent être considérées en une seule fois. Ainsi le programme de ramasse-miettes commence par réserver toutes les zones connectées. Eventuellement si l'une de celles-ci est déjà impliquée dans un ramasse-miettes le nouveau traitement est momentanément abandonné. S'il était possible de toujours réserver toutes les zones impliquées, il ne pourrait y avoir de "deadlock". Cependant on a vu que le ramasse-miettes crée des cables pour les objets référencés depuis un environnement. Lorsqu'un cable concerne une zone en cours de traitement, le ramasse-miettes est arrêté, aussi des blocages peuvent apparaître. On suppose que ceci peut être évité en programmant correctement le ramasse-miettes de sorte qu'il ne génère pas de cables. Alors le ramasse-miettes peut fonctionner en parallèle car les interactions entre zones sont sans doute faibles.

L'utilisateur peut également créer explicitement des cables entre zones. Il en résulte deux problèmes. Le premier de ces problèmes est que des liens peuvent de cette façon devenir inutiles et que le ramasse-miettes ne peut en avoir connaissance car il ne traite que la zone. Aussi quand un cable est créé entre la zone A et la zone B tous les liens sortant de A sont parcourus ; dans ceux qui concernent B, un indicateur est positionné. Celui-ci permet d'alerter le ramasse-miettes qui éventuellement supprimera les liens qui ne sont plus nécessaires.

Un autre problème concerne la suppression des cables lorsqu'ils ne sont plus utilisés. A cet effet, lorsqu'un cable est créé un indicateur est positionné. Celui-ci sera changé à la première utilisation.

Finalement, il reste le problème de savoir comment commencer à traiter une zone. Il existe dans une zone des objets qui sont des composants d'autres objets et qui par conséquent ne sont pas visibles extérieurement en principe. Une liste d'objets visibles repère ceux des objets qui a priori sont accessibles extérieurement. C'est à partir de cette liste que commence le ramasse-miettes. Si un objet n'est pas accessible depuis cette liste ou s'il n'est pas composant d'un objet de cette liste il doit probablement être effacé. Si c'est le cas mais qu'il est accessible extérieurement par un lien, il est probable qu'il doit être déplacé vers l'une des zones qui le référencent. C'est, de fait, le critère qui est utilisé pour déplacer les objets.

On remarquera également que puisque le ramasse-miettes est réalisé par zones, il est possible de détecter les boucles de référence à l'intérieur de chaque zone mais pas entre zones. On peut ainsi avoir la situation de la figure 7. L'objet x de la zone A est considéré comme accessible puisqu'il existe un lien entrant m. Ainsi quand le ramasse-miettes traite la zone A l'objet x est conservé et il en serait de même de l'objet y dans la zone B.

On notera cependant que lorsque ceci se produit, l'objet de la zone traitée qui appartient à une telle boucle de références n'est accessible qu'à travers un lien inter-zone, il n'est pas référencé à l'intérieur de la zone elle-même. De tels objets qui ne sont référencés que par des zones extérieures devraient en fait être déplacés vers l'une des zones qui les référencent. Ainsi, ou bien x serait déplacé vers la zone B, ou bien y serait déplacé vers la zone A. Alors le ramasse-miettes sera capable de détecter la boucle que ce soit dans A ou dans B. Le ramasse-miettes est donc complété par un "déménageur" dont le rôle est de déplacer les objets d'une zone à une autre .

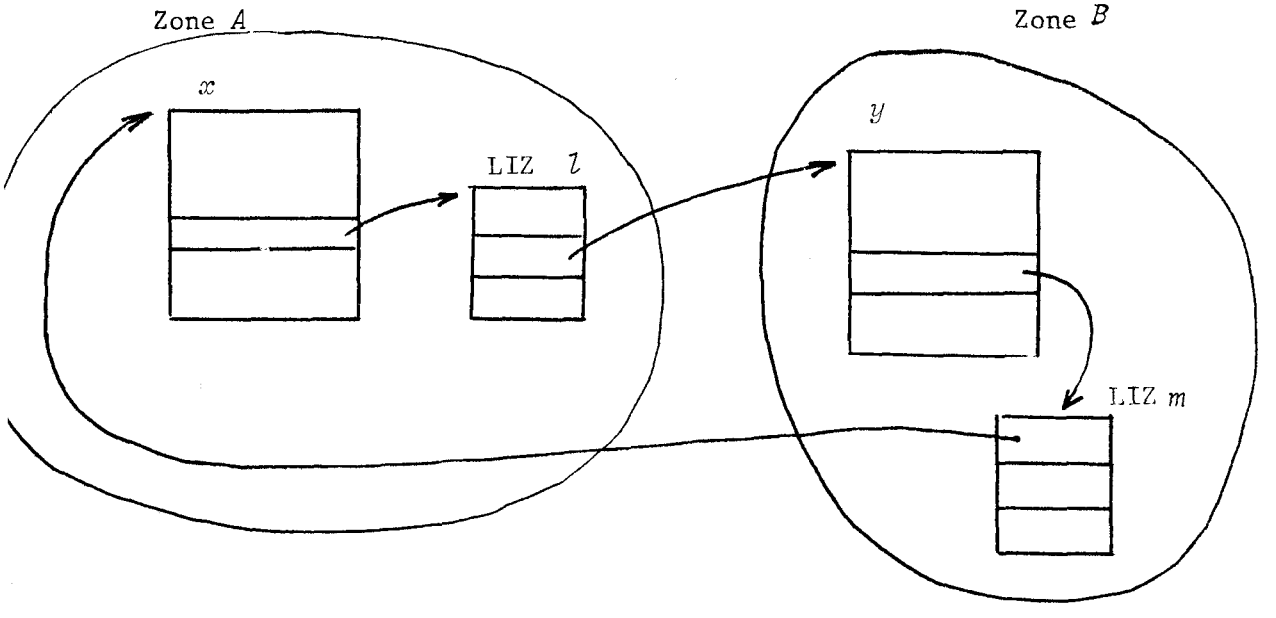


Figure 7 : Boucle de références entre zones.

4. - Discussion

L'intérêt de la méthode de Bishop repose sur le fait que le ramasse-miettes peut être effectué en parallèle. Les liens inter-zone jouent le rôle des descripteurs des mécanismes classiques à capacités. Le fait d'associer ceux-ci aux zones qui accèdent à l'objet et non à l'objet lui-même permet à peu de chose près de rendre le ramasse-miettes de la zone qui contient l'objet indépendant des zones qui utilisent l'objet. Le seul traitement complémentaire à assurer consiste à modifier les liens inter-zones. Il nécessite de repérer tous les liens concernés. En fait, ce qui fait la différence comparativement à une méthode classique de ramasse-miettes c'est que ces liens sont constamment repérés. Cependant, il en résulte un mécanisme lourd ; la copie d'une capacité d'une C-liste dans une autre trouve son équivalent dans la création d'un lien dans une nouvelle zone. Ceci requiert la mise à jour d'une liste dont éventuellement les éléments ne sont pas en mémoire. Il convient donc d'en faire une utilisation statique dans laquelle il n'y a pas ou peu de création de nouvelles références dans une zone.

La lourdeur du mécanisme des liens conduit Bishop à introduire la notion de cable. Cependant, il en résulte que toutes les zones connectées par des cables doivent être traitées en même temps par le ramasse-miettes. Celui-ci ne restera efficace que s'il y a peu de cablage entre zones, sinon on perdrait le bénéfice du parallèlisme. Ceci repose en partie sur l'utilisateur.

Enfin il convient de faire une remarque sur le placement des objets. Les objets nouvellement créés sont placés soit dans une zone désignée par le créateur, soit chez le créateur lui-même. Considérons le cas d'un objet typé formé de composants. Le gérant du type n'a pas de connaissance a priori de l'endroit où placer les composants dont il demande la création. Ceux-ci seront donc placés chez lui. Il en est de même à tous les niveaux de composition si ces composants sont eux-mêmes formés d'autres composants. Ainsi les composants d'un objet sont dispersés entre diverses zones, alors qu'il conviendrait qu'ils se trouvent chez le créateur de l'objet. Le ramasse-miettes les rassemblera peu à peu au cours du traitement des diverses zones concernées. En fait, le placement optimal est connu dès le début. Il semble donc que les deux méthodes présentées soient complémentaires. De plus, l'utilisation des mécanismes présentés en partie 2 de ce chapitre permettrait d'éviter la création de liens entre composants.

On remarquera également que le ramasse-miettes est responsable de la protection car il créé des capacités et déplace des objets. La complexité de celui-ci et le fait qu'il soit utilisé en permanence en font un point faible du système et vont à l'encontre du principe du noyau minimum. On peut également douter de l'efficacité d'un tel système sur le plan des performances ; c'est cependant le seul modèle acceptable à l'heure actuelle sur le plan du ramasse-miettes.

CHAPITRE IV

LE CONTROLE D'EXECUTION

1. - Nature des environnements

1.1. - Généralités

On rappelle que l'unité de découpage du système est le module. Un module regroupe les opérations définies sur un type. Afin de conserver ce découpage jusqu'à, y compris, l'exécution, les modules sont matérialisés à l'aide d'objets type. D'autre part, l'unité d'exécution dans le système est la procédure. Ainsi, l'opération appliquer réalisée sur un type est une extension de l'appel de procédure. Cette extension consiste à identifier le module (le type) associé au type d'un objet puis la procédure du module qui réalise l'opération invoquée. La notion d'environnement a été introduite pour permettre le partage des procédures. L'environnement définit d'abord l'espace accessible au cours d'une exécution de procédure, c'est ce qu'on a appelé plus haut un domaine. Puisque lui-même n'est pas partagé, l'environnement sert également de support aux références des objets locaux créés pour cette exécution de la procédure. Ainsi, chaque appel de procédure doit provoquer implicitement la création d'un nouvel environnement. La figure 1 schématise les différentes notions rappelées ici. On se propose d'examiner ce qu'implique cette création et de définir les opérations à mettre en place.

L'environnement a été défini comme une C-liste dont l'une des capacités repère la C-liste des objets permanents de la procédure. Cet ensemble d'objets est celui que partage tout environnement créé sur la même procédure. Entre autres, il contient les segments de code de la procédure. Une fois qu'il a été créé, pour que l'environnement devienne opérationnel, il faut encore que les paramètres effectifs soient contrôlés, voire décomposés et qu'éventuellement des objets locaux soient créés. C'est ainsi qu'est créée la pile (de données) destinée primo au stockage des variables internes créées au cours de l'exécution de la procédure, et secundo à la sauvegarde de l'état des procédures internes au module. Dès lors que l'environnement a été créé et les paramètres transmis, toutes ces opérations de contrôle ou de création peuvent être réalisées au cours de l'exécution de la procédure elle-même. Cependant, l'ensemble des objets cités étant nécessaire à l'exécution de la procédure, il serait intéressant d'en munir la procédure dès son activation. Ces objets étant locaux, ils peuvent être vus comme des composants de l'environnement. Leur création par le noyau à l'aide de maquettes peut être mise à profit pour simplifier l'allocation de mémoire et éviter sa fragmentation.

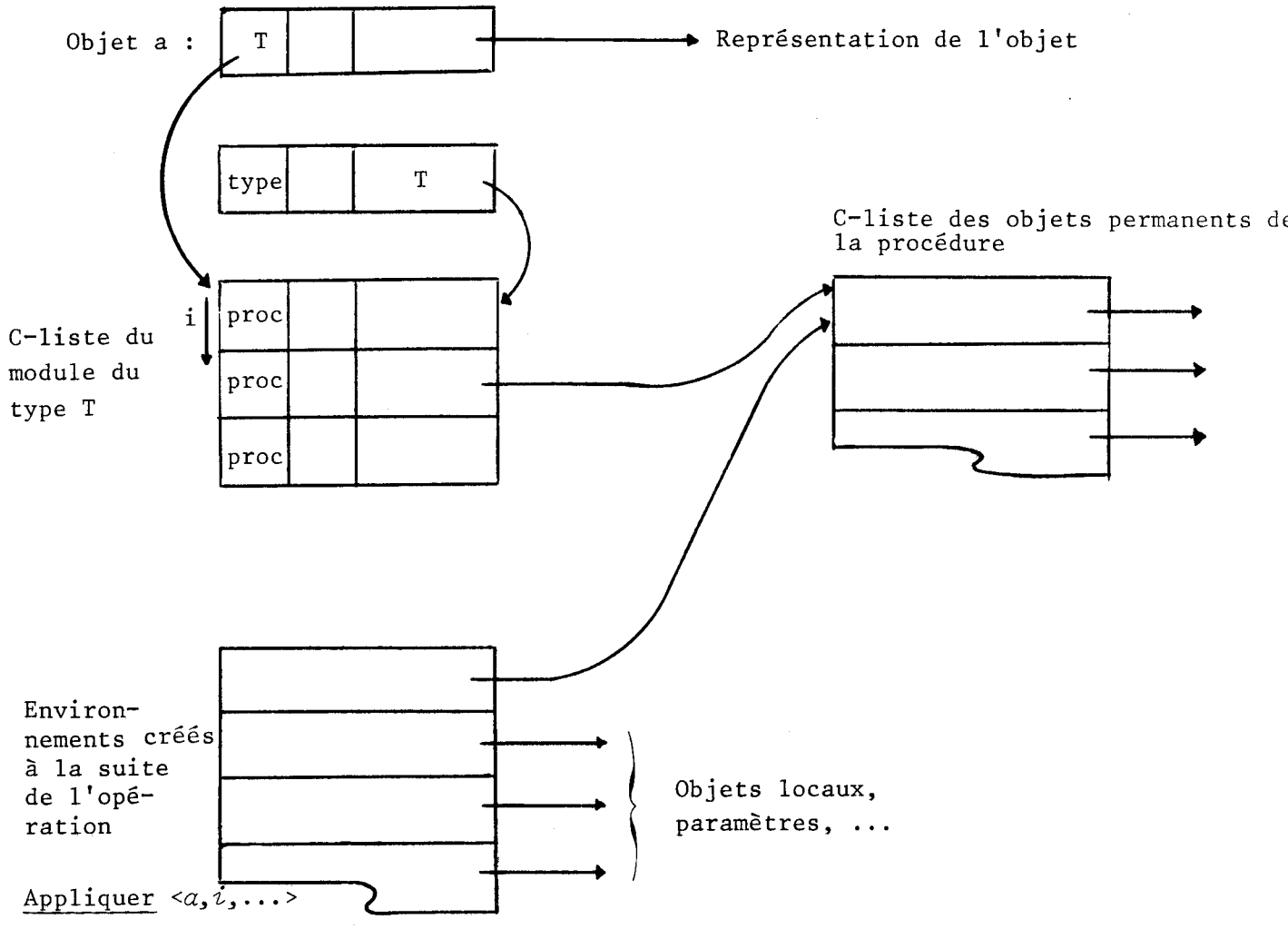


Figure 1 : Schématisation des notions utilisées.

De par le principe du moindre privilège de nombreux changements d'environnements ont lieu. Il importe donc que cette opération soit efficace. Ceci est présenté en général comme un inconvénient des systèmes à capacités. De fait cel est aussi inhérent au type de système réalisé. Dans CAP [Wal 73,74] où les environnements sont créés statiquement, l'appel de procédure est comparable à celui d'une procédure en PL1. Par contre dans Hydra où un environnement est créé à chaque appel et où des objets doivent être chargés avant l'activation de la procédure, l'appel est de l'ordre de la dizaine de millisecondes. On définira ici les mécanismes de base du contrôle d'exécution sans préjuger de leur utilisation qui elle dépend du type de système mis en place.

1.2. - Exemple : l'appel de procédure d'Hydra

On rappelle que la mémoire adressable du PDP 11 n'est que de 64 K octets. Pour pallier cet inconvénient, l'espace mémoire est découpé en pages et Hydra utilise une technique de couplage [Bet 70, Coh 75]. Une page est un objet qui est repéré à l'aide d'une capacité et qui décrit l'implantation de la page de mémoire correspondante. Ainsi l'ensemble des pages de la procédure peut être décrit dans sa C-liste. Pour qu'une page devienne adressable, elle doit avoir été couplée dans la carte d'implantation qui décrit l'espace de l'environnement.

Hydra décrit pour chaque procédure :

- un ensemble de pages utiles (EPU), ce sont celles pour lesquelles il existe une capacité dans l'objet procédure,
- un ensemble de pages chargées (EPC), c'est le sous-ensemble de EPU qui existe en mémoire principale,
- un ensemble de pages accessibles (EPA), c'est le sous-ensemble de EPC qui est couplé dans la carte d'implantation.

Chaque objet dans Hydra est formé d'une C-liste et d'un bloc de données. Ainsi une page a pour représentation un bloc de données décrivant son implantation en mémoire centrale et secondaire et une C-liste vide. Dans un objet procédure, le bloc de données contient les informations relatives à l'environnement à créer telles que : la valeur initiale du compteur ordinal, la taille de pile de données nécessaire, l'ensemble des pages à charger initialement (EPCI), l'ensemble des pages à coupler initialement (EPAI) etc ... . La C-liste contient elle les capacités des objets de la procédure (par exemple celles des pages) et des maquettes de paramètres ou de création d'objets (figure 2).

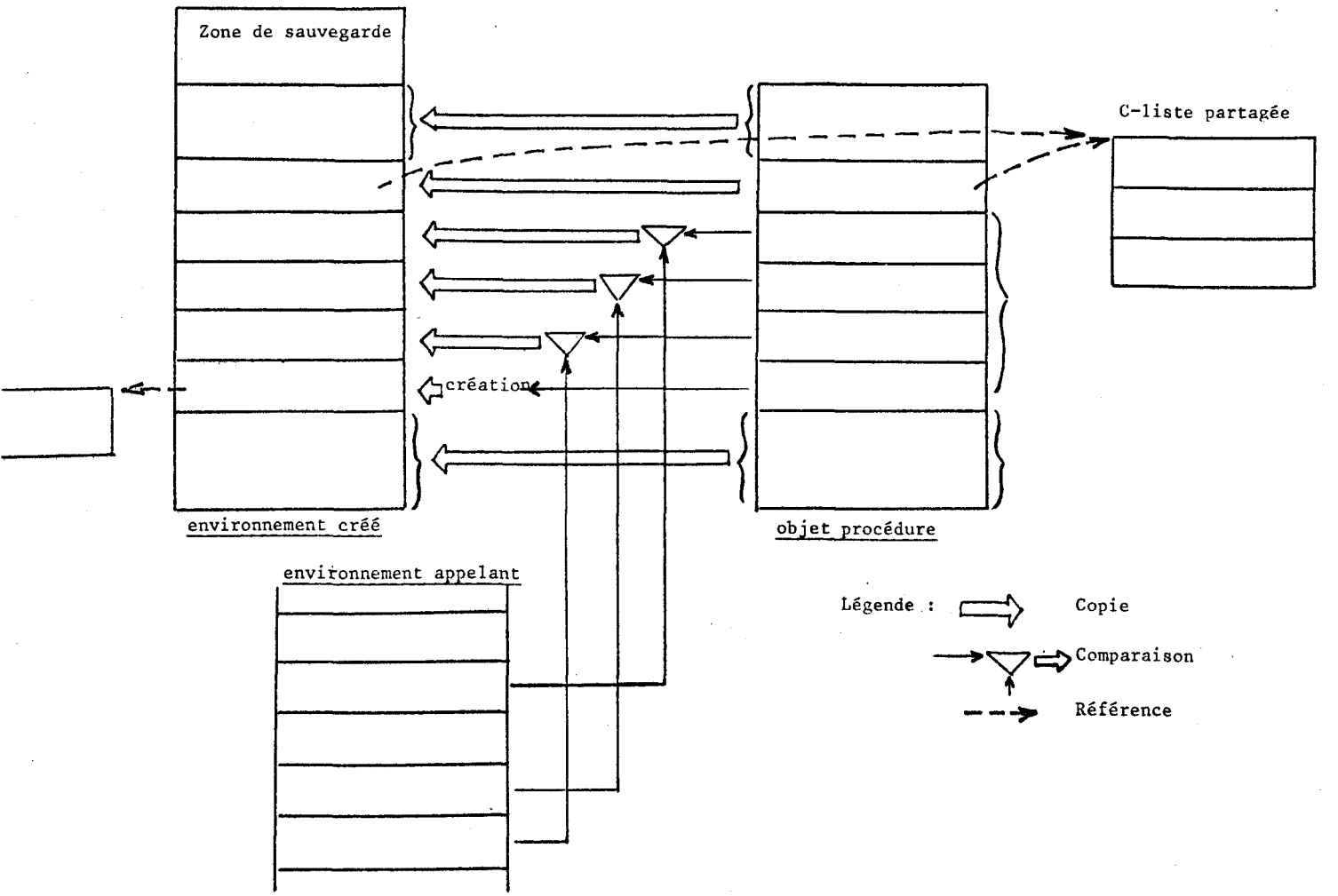


Figure 2 : Création d'un environnement dans Hydra

Lorsque la procédure est appelée un environnement est créé. Son bloc de données est initialisé avec le contenu du bloc de données de la procédure augmenté d'une zone de sauvegarde de l'état de l'environnement. Sa C-liste est formée en copiant les capacités de l'objet procédure. Eventuellement, il s'agit d'emplacements vides qui définissent une zone de travail. Lorsqu'il s'agit de la capacité d'une maquette de paramètre, selon le cas ou bien le paramètre est décomposé ou bien il est simplement contrôlé. De même lorsqu'il s'agit d'une maquette de création, l'objet qu'elle décrit est généré. Enfin les pages de EPCI sont chargées et celles de l'EPAI sont couplées et l'environnement est activé.

On remarquera que dans ce schéma, l'objet procédure apparaît comme une maquette d'environnement. En fait l'opération appel de procédure est essentiellement une création d'environnement à l'aide de cette maquette. Elle est ensuite suivie d'une activation de l'environnement. L'habitude de raisonner sur la maquette de l'environnement plus que sur l'environnement lui-même vient de ce que le plus souvent seul l'appel procédural est implanté. Comme chaque appel doit provoquer une nouvelle création d'environnement, celui-ci n'apparaît pas en tant que tel et au niveau de l'opération d'appel la création et le transfert de contrôle se confondent. L'exemple suivant va justifier cette remarque.

1.3. - Exemple : le problème des objets rémanents

On appelle objet remanent un objet créé lors d'un premier appel de procédure et réassocié ensuite à cette procédure lors des invocations ultérieur L'objet n'appartient pas en propre à la procédure elle-même mais au couple (appelant, appelé). Ainsi l'exécution de la même procédure par deux processus différents conduit à définir deux ensembles d'objets rémanents associés chacun à un couple (processus, procédure). La prise en compte de la protection conduit en fait à une définition plus rigoureuse : un objet rémanent appartient à un couple (domaine, procédure) (cf Chap. I.2). Ainsi dans Multics un ensemble de rémanents est associé à un triplet (processus, n° anneau, procédure) où le numéro d'anneau est celui dans lequel va s'exécuter la procédure appelée. Les rémanents sont maintenus dans le segment de liaison approprié. Cependant, l'opération appel étant définie sur l'objet procédure, la recherche du segment de liaison qui doit lui être associé est complexe. Elle requiert de maintenir une table des segments de liaison par domaine (par processus et par anneau). Ceci contribue à ralentir l'opération appel.

Dans le système que nous décrivons , chaque appel de procédure crèe un nouveau domaine. Ainsi la définition d'un nouveau domaine dépend du domaine depuis lequil il a été créé. Par ailleurs, le support des objets propres au domaine étant l'environnement, il suffit pour conserver les rémanents d'une procédure de conserver l'environnement créé lors de son appel. Cet environnement doit être conservé dans l'environnement depuis lequel la procédure a été appelée. Ceci nécessite de distinguer la création des environnements et le transfert de contrôle. Dans un premier temps, l'environnement est créé et une capacité est retournée à l'appelant. Par la suite, il suffit d'activer l'environnement pour chaque fois retrouver les objets rémanents.

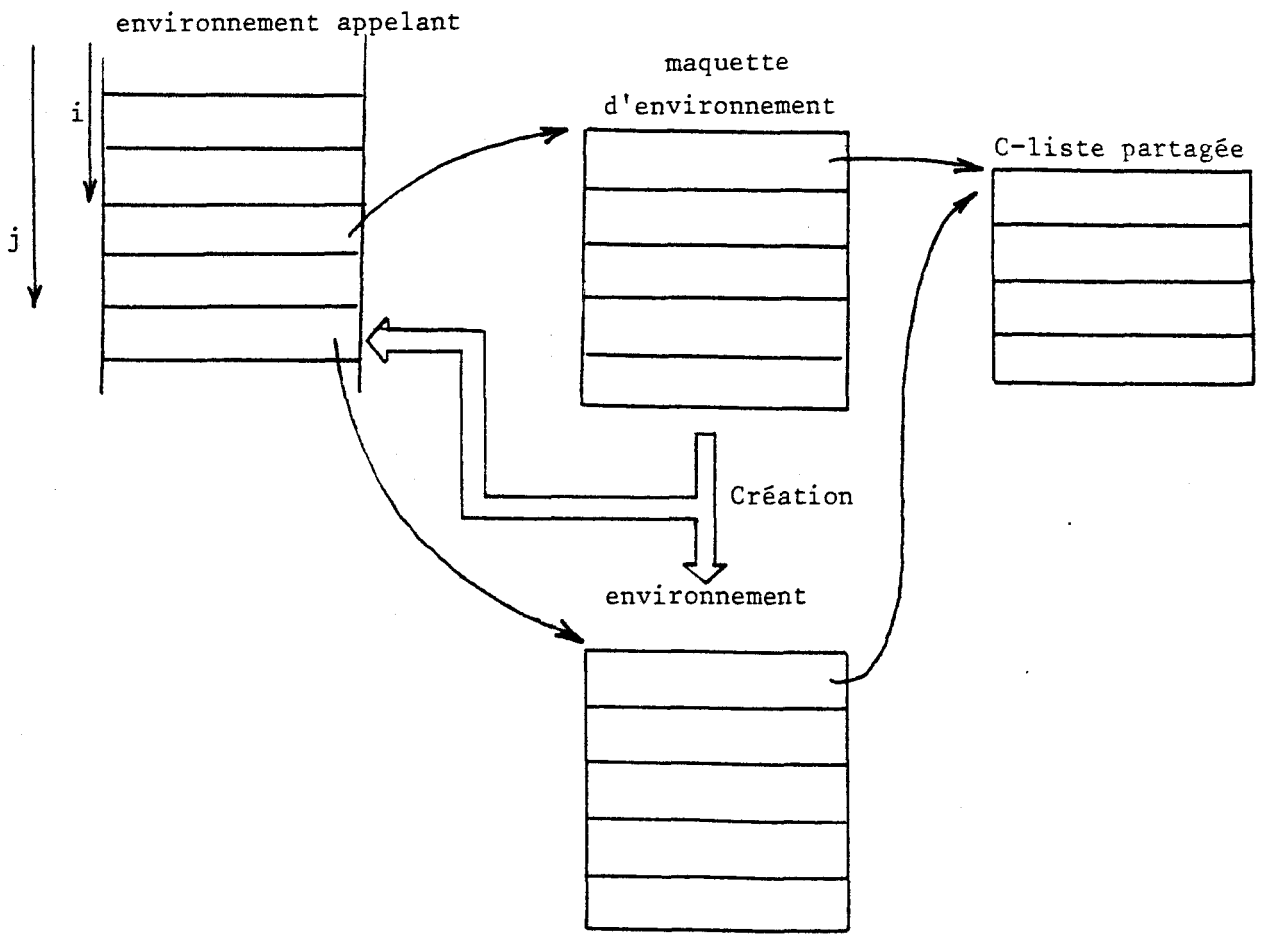


Figure 3 : Cas des rémanents

2. - L'objet environnement

2.1. - Généralités

A la suite des remarques précédentes, nous considérerons l'objet environnement, non l'objet procédure. Celui-ci utilisé pour faciliter la présentation n'a pas d'existence réelle dans le système. Il s'agit de fait d'une maquette d'environnement qui contient éventuellement d'autres maquettes : celles des objets locaux ou rémanents à créer en même temps que l'environnement. Les maquettes de paramètres ne sont pas incluses, l'exemple précédent en donne la justification ; puisque l'opération d'appel porte sur l'environnement c'est dans celui-ci que les paramètres seront traités. Par ailleurs les opérations réalisées sur les paramètres sont très simples et peuvent être des opérations de la machine utilisant les objets de type "type" appropriés. On remarquera que l'objet type est en fait une collection de maquettes, soit d'environnements, soit d'objets. L'opération appliquer consiste essentiellement à créer un objet qui éventuellement est un environnement.

Des paragraphes précédents, on a déduit deux opérations distinctes : créer-environnement et transférer_contrôle. Cette dernière opération sera détaillée ultérieurement. Il convient cependant de faire quelques remarques concernant leur utilisation.

La décision de mettre en place des opérations sophistiquées à un niveau bas du système conduit à des contraintes aux niveaux supérieurs, Ainsi la mise en place d'une seule opération d'appel procédural interdit de considérer le cas des rémanents puisque chaque appel crée un nouvel environnement. De même que l'insertion du transfert de paramètres dans l'opération de transfert de contrôle peut ne pas être en accord avec la réactivation d'une coroutine. Par exemple, la dissociation de l'opération créer-environnement et des opérations de contrôle permet d'envisager aussi bien la mise en place ultérieure d'un système dynamique où chaque appel provoque une création d'environnement que d'un système statique où les environnements sont créés lorsque l'utilisateur entre en session. De la même manière, il n'y a pas de règle précise concernant la structure d'une maquette d'environnement. Par exemple, la réalisation d'appels procéduraux peut se faire en créant un nouvel environnement à chaque appel. Alors la maquette d'environnement peut également contenir les maquettes des objets locaux. A l'opposé, les environnements peuvent être précréés ; alors, la création des objets locaux est réalisée par la procédure en exécution dans l'environnement. Nous nous intéressons ici aux mécanismes

élémentaires nécessaires à la mise en place de ces structures de contrôle indépendamment de leur utilisation.

La figure 4 schématise l'objet environnement. Deux emplacements de sa C-liste sont réservés, les autres sont définis suivant le contenu de sa maquette. Ces emplacements réservés sont destinés à contenir : a) la capacité de la C-liste commune à tous les environnements créés à partir de la même maquette et b) la capacité de la C-liste des paramètres reçus. Les paramètres transmis en retour le sont dans la même C-liste qu'à l'appel. Leur contrôle ou leur décomposition sont laissés à la charge de l'environnement appelé. On remarquera que la C-liste des paramètres peut être créée dès la création de l'environnement comme un composant interne de l'environnement. Il en est de même de la pile locale à l'environnement ; sa création comme un composant interne permet d'accélérer la mise en place de l'environnement.

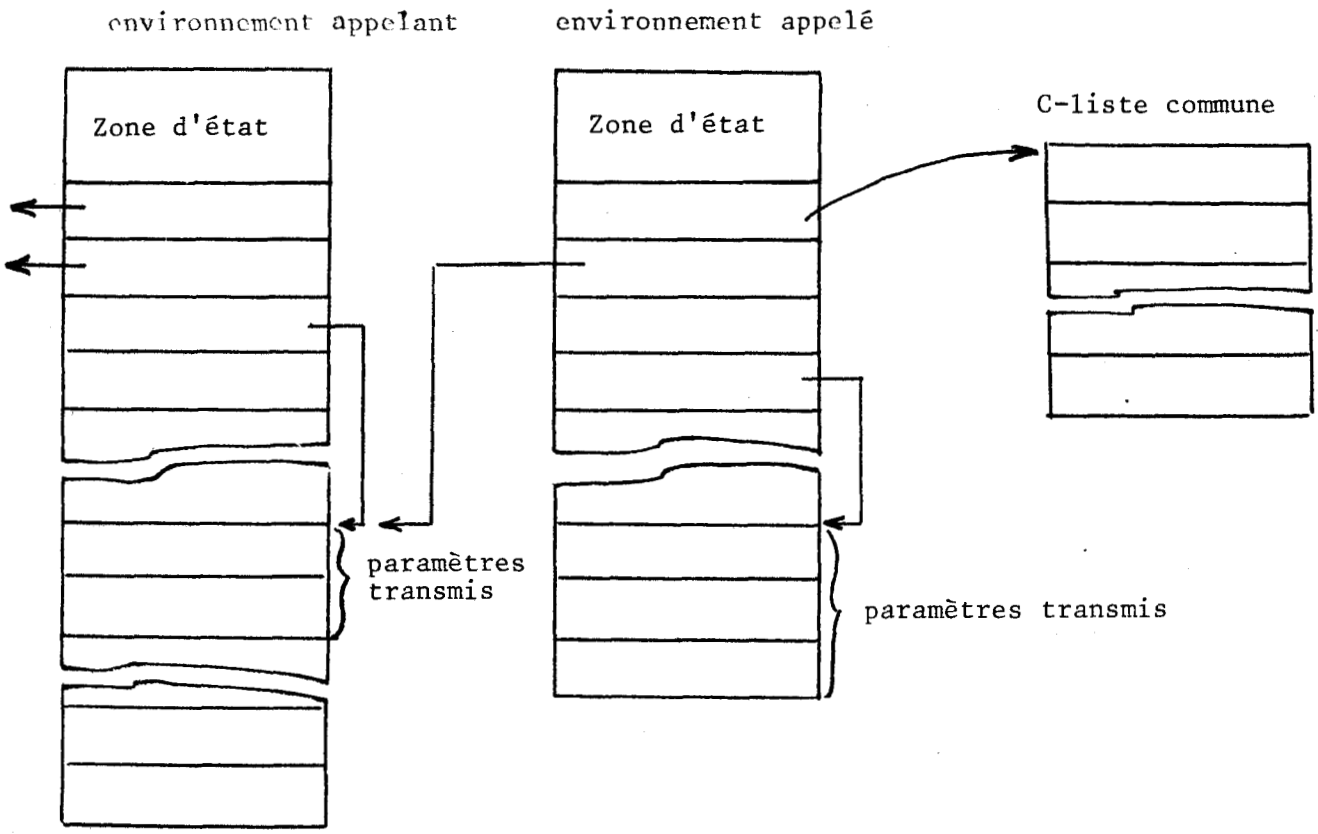


Figure 4 : Schéma d'un environnement

2.2. - Création d'un environnement

Quand le système est dynamique la création d'environnements est très fréquente ; il importe alors qu'elle soit rapide. Cette création consiste en premier lieu en une allocation d'espace. L'utilisation des maquettes qui permette de créer les composants d'un environnement sous contrôle du noyau et de regrouper ceux-ci dans un seul segment contribue à accélérer le mécanisme. Ceci est à comparer à Cm* ou à Hydra où une pile de données doit être créée pour chaque environnement. On se propose de définir ici le mécanisme d'allocation d'espace pour l'environnement lui-même. Nous étudierons deux méthodes d'allocation.

La première méthode repose sur l'utilisation d'une pile d'environnements. L'allocation d'espace consiste à réserver dans la pile l'espace nécessaire, elle repose sur la connaissance de la limite de l'espace utilisé par l'environnement appelant. Cette limite devient la base de l'espace alloué au nouvel environnement. Il suffit de deux registres pour décrire l'espace occupé par l'environnement et la liaison entre les environnements appelé et appelant consiste simplement en un pointeur vers la base de l'appelant pour permettre la restauration ultérieure de son espace. Le passage de paramètres peut être rapide si l'on utilise une technique de recouvrement telle que celle que décrit la figure 5. On remarquera que l'utilisation d'une pile ne permet de mettre en place que des appels procéduraux. L'exécution de coroutines dans le B6700 où une pile est utilisée se simulerait par la création d'un nouveau processus. D'autre part, pour éviter les débordements de pile, on est amené à réserver à celle-ci un espace relativement important. En fait la profondeur de pile utilisée est généralement faible et l'espace inutilisé est important. Comme la pile est éventuellement résidente afin que le mécanisme d'activation soit efficace, il en résulte une perte d'espace notable si l'on tient compte de ce phénomène pour l'ensemble des processus du système.

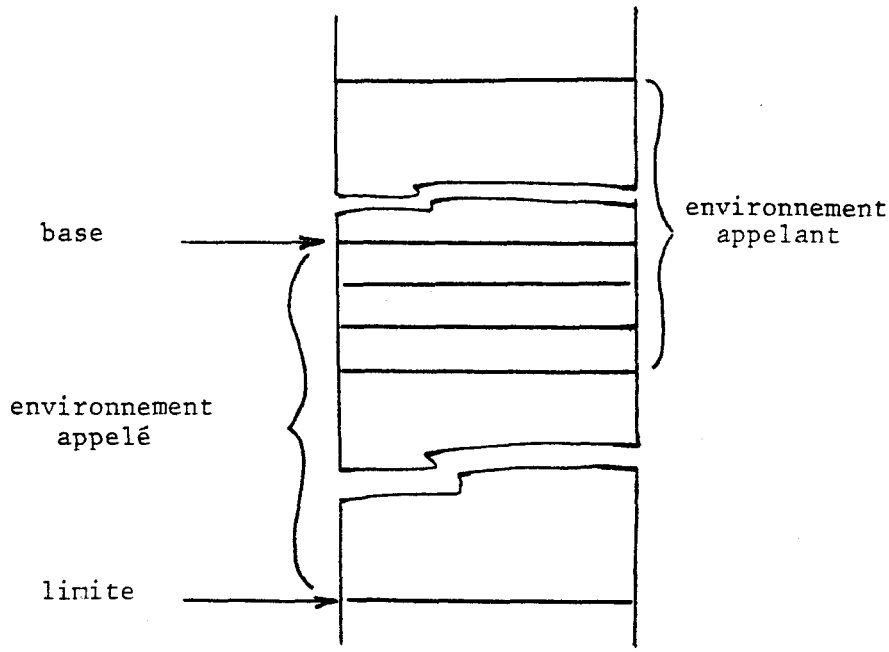


Figure 5 : Implantation de la pile d'environnements

La deuxième méthode est celle employée dans Cm* [KL78,JS78] et dans Ours [BV76]. Elle tient compte de la remarque précédente. La taille des environnements varie peu, aussi l'on peut utiliser un réservoir de supports d'environnements précréés et gérés en liste. L'allocation d'espace pour un environnement consiste alors en l'allocation d'un support pris dans cette liste. L'avantage d'une telle méthode est de permettre toute forme de contrôle d'exécution. Jones et Schiller [JS78] comparent les deux méthodes présentées ici dans le cas d'un système à capacités. L'inconvénient de la méthode du réservoir réside dans le fait que la réalisation des liens nécessite plusieurs copies de capacité, elle est donc moins efficace que la précédente. L'existence d'opérations adéquates fournies par la machine peut cependant contribuer à l'améliorer.

En principe le support alloué pourrait être renvoyé dans le réservoir lors de l'opération de retour si chaque appel provoquait la création d'un nouvel environnement. En fait, la capacité de cet environnement peut être retenue chez l'appelant. Le renvoi du support dans le réservoir ne sera possible, comme pour tout autre objet, que lorsque le compteur de capacités atteindra la valeur zéro. On rencontre ainsi un problème plus général des systèmes à capacités : certains objets peuvent en fait être utilisés pour décrire des ressources (c'est le cas des pages dans Hydra par exemple) ou même être des ressources (c'est le cas ici), il importe lorsque la dernière capacité est détruite que la ressource soit récupérée par son gérant. Une solution possible à ce problème est que chaque descripteur d'objet soit muni

d'un indicateur signalant ce cas et que le type correspondant soit muni d'une boîte aux lettres. Lorsqu'un effacement de la dernière capacité est requis, celle-ci est transmise dans la boîte aux lettres. On notera que celle-ci constitue en fait le réservoir des ressources.

Une remarque supplémentaire est nécessaire. Lorsque le support d'un environnement doit être réutilisé, il importe qu'il ne contienne plus de capacités. Cela compromettrait la protection mais aussi le ramasse-miettes. Si l'allocation a lieu dans une pile l'effacement du support doit avoir lieu soit au retour, soit lors de la réallocation ce qui est plus efficace car au cours de cette opération des écritures ont lieu de toute façon dans la pile. Cependant dans ce dernier cas les zones non réallouées de la pile contiennent encore des références. On remarque que ceci compromet l'efficacité de l'utilisation de la pile. Au contraire dans le cas du réservoir, les supports peuvent être renvoyé à une boîte aux lettres intermédiaire d'où ils sont extraits par un processus parallèle pour être réinitialisés. Ainsi l'inconvénient du maintien des liens entre environnements est compensé.

3. - Contrôle d'exécution

3.1. - Généralités

Une fois les paramètres préparés et l'environnement créé, le contrôle d'exécution concerne la transmission des paramètres et le transfert de contrôle. On remarque que le transfert de paramètres ne consiste qu'à recopier la capacité d'une C-liste. Le transfert de contrôle proprement dit peut être de deux sortes, ou bien l'environnement sur lequel il porte s'exécute en parallèle, ou bien le transfert est de type procédural. Plus précisément, nous distinguerons entre transfert série et transfert parallèle. On appellera transfert série tout transfert de contrôle provoquant la suspension de l'appelant.

Avant de détailler les opérations de transfert, il convient de remarquer que la notion de processus est assez peu significative ici. Ceci provient de l'indépendance des espaces d'objets des environnements. Plutôt que de considérer un processus, il convient de considérer l'environnement initial du processus, celui qui provoque l'exécution d'autres environnements. Ainsi l'élément qui à un instant donné possède le contrôle est le dernier environnement appelé. De plus la notion de processus conçu comme une suite linéaire d'environnements

liés par des relations de transfert série est relative à la notion de mono-processeur. L'évolution de la technologie conduit à penser à des multiprocesseurs où le nombre de processeurs peut être élevé et où l'unité de découpage des éléments exécutés en parallèle est plus relative à la notion d'environnement. Ainsi Cm* utilise la notion de "task force" traduite ici par tribu. Une tribu est un ensemble d'environnements coopérant par des opérations de transfert de contrôle série ou parallèle. Elle décrit un arbre d'environnements. Eventuellement d'autres formes de coopération peuvent prendre place entre environnements d'une même tribu ou entre tribus par échange de messages.

Le gérant des ressources processeur ne connait d'une tribu que des environnements, ceux qui sont actifs. Il en résulte que les opérations de transfert de contrôle interagissent avec ce gérant. Le transfert de contrôle à un environnement se traduit par son placement dans la liste des environnements actifs du gérant. Ainsi chaque environnement est conçu comme un élément indépendant et possède entre autres une zone de sauvegarde de son état.

3.2. - Transfert série

On distinguera ici l'appel de procédure et l'appel de coroutine. L'appel de coroutine sera considéré comme un transfert série puisqu'il provoque la suspension de l'appelant. Initialement, tous deux ne se distinguent pas ; ils correspondent à la séquence suivante exécutée sur l'environnement fils B :

- transférer $<B,p>$: ceci réalise le transfert des paramètres p ;
- lier $<A,B>$: ceci créé la relation père-fils entre A et B et permettra l'opération de retour ;
- activer $<B>$: place l'environnement B dans la liste des environnements actif du gérant ;
- suspendre $<A>$;

L'opération de liaison consiste à placer la capacité du père chez le fils, inversement l'opération de retour consistera à détruire ce lien et à réactiver A en suspendant B. Cependant, une distinction doit être faite entre procédure et coroutine. Dans le premier cas un nouvel appel doit provoquer la réexécution dans les mêmes conditions que précédemment, voire en créant un nouvel environnement. Dans le second cas, au contraire, l'appelé doit être replacé dans l'état où il se trouvait avant d'effectuer le retour. On a remarqué que chaque environnement est muni d'une zone de sauvegarde ; cependant, celle-ci étant utilisée par le gérant, les valeurs initiales ne peuvent plus y être contenues au

moment du retour. On doit donc considérer que l'environnement est muni d'une zone initiale distincte à l'aide de laquelle il est possible de restaurer son état initial, en particulier son compteur ordinal. Alors, le retour depuis une coroutine s'écrirait :

- détacher <*B*> : détruit le lien de retour dans l'environnement ^{B} ;
- activer <*A*> ;
- suspendre <*B*> ;

alors que le retour d'une procédure s'écrirait :

- détacher <*B*> ;
- activer <*A*> ;
- quitter <*B*> ; qui réinitialise l'environnement et le suspend sans sauvegarde.

On notera qu'il s'agit d'opérations rapides qui agissent sur des listes : ainsi l'opération activer peut être l'envoi de l'environnement dans une boîte aux lettres. L'opération la plus complexe est l'opération suspendre qui nécessite une sauvegarde inévitable. Quitter et suspendre activent le noyau pour qu'un nouvel environnement puisse être chargé sur le processeur.

3.3. - Transfert parallèle

Le transfert de contrôle parallèle se caractérise par les opérations bifurquer (fork) et rejoindre (join) qui permettent à une tribu de générer ses environnements coopérants. L'opération bifurquer <*B*> lance l'exécution parallèle d'un nouvel environnement tandis que l'opération rejoindre <*B*> permet à l'environnement qui l'exécute d'attendre la fin de l'exécution de *B*. Une opération additionelle est nécessaire, générer <*B*> qui permet d'initialiser une nouvelle tribu. Elle est semblable à bifurquer mais ne lie pas l'environnement activé ; lorsque celui-ci effectuera l'opération de renvoi du contrôle éventuellement il disparaitra. On remarquera qu'un environnement n'a pas connaissance de la manière dont il est activé. Le renvoi du contrôle par un environnement doit donc être indépendant du fait qu'il a été activé en parallèle ou en série. Il en résulte que l'opération rejoindre, qui place l'environnement qui l'exécute dans les mêmes conditions qu'un environnement ayant réalisé un transfert série, nécessite la mise en place d'une structure particulière.

On distinguera deux structures de contrôle : l'une série, l'autre parallèle schématisées par la figure 6. Il s'agit de deux structures de liste, l'une liant les environnements activés par un transfert série, l'autre les environnements

qui ont été activés depuis le même environnement par un transfert parallèle. Le transfert parallèle ne se distingue du transfert série que par la liaison utilisée et le fait que l'appelant ne se suspend pas :

- lier-parallèle <*A*,*B*> ;
- activer <*B*> ;

L'opération rejoindre <*B*> consiste à détacher *B* de la "liste parallèle" pour le placer dans la "liste série". Ainsi, lorsque *B* exécute le renvoi du contrôle tout se passe comme s'il avait été activé par un transfert série et son père est activé à nouveau. Il convient donc de modifier l'opération retour comme suit :

si "série" alors début détacher <*B*> ; activer <*A*> fin
sinon détacher <*B*> ;

L'opération rejoindre s'écrirait :

- détacher <*B*> ;
- lier <*A*,*B*> ;
- suspendre <*A*> ;

On notera que si le retour de contrôle depuis *B* se produit pendant l'exécution de cette séquence, *A* peut ne jamais être réactivé. Il convient de réaliser les séquences rejoindre et retour en exclusion mutuelle.

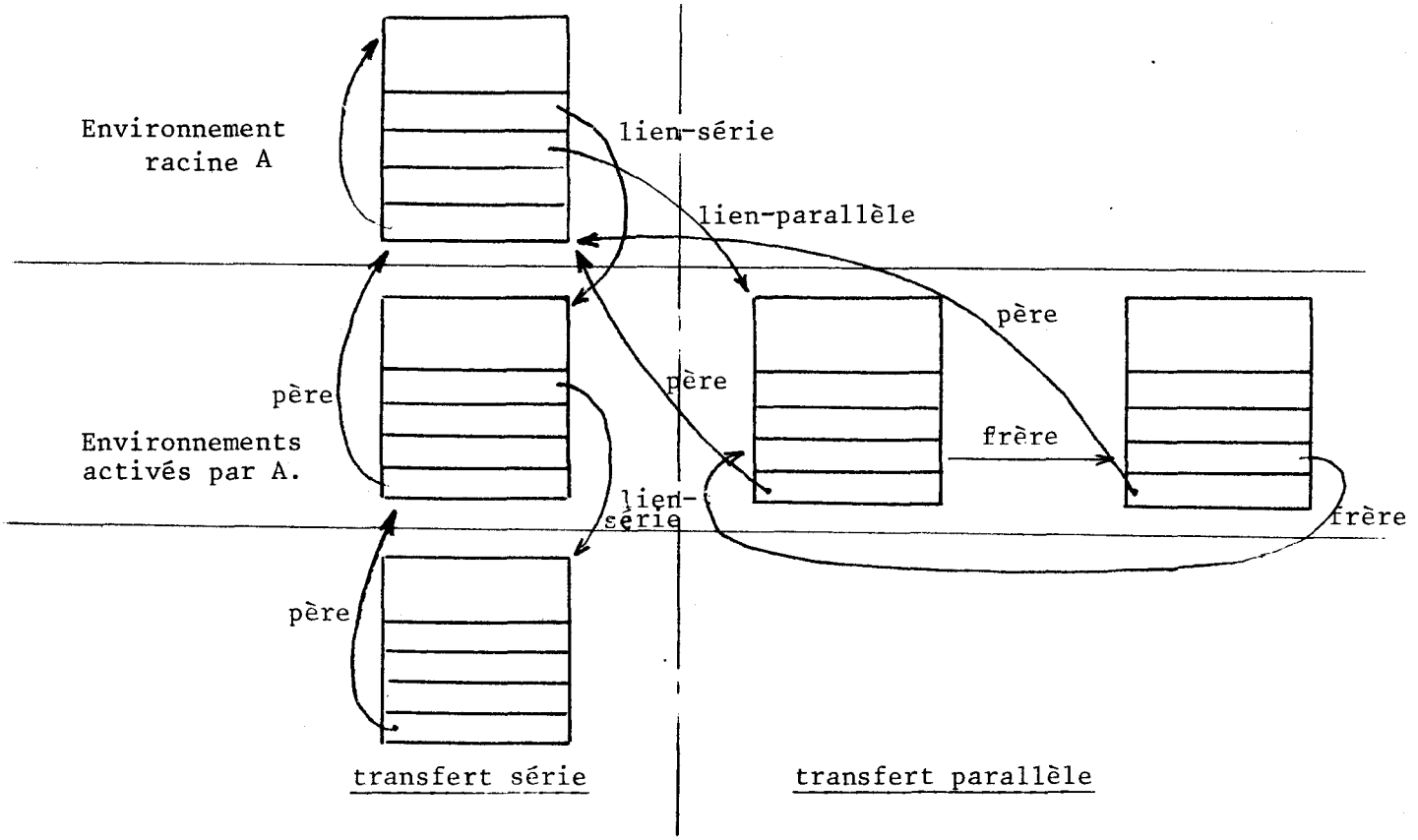


Figure 6 : Liaison des environnements dans Cm*

4. - Organisation du noyau

Pour exécuter un environnement (i.e. pour faire évoluer l'état des objets qu'il regroupe), il faut associer la C-liste correspondante aux registres de données usuels et aux opérateurs câblés de la machine. Pour cela, il suffit que cette C-liste soit repérée par un registre de la machine.

Le noyau est organisé de même en termes d'environnements. Cependant, l'objet environnement proprement dit, muni des opérations d'appel et de retour n'apparait qu'aux niveaux externes du noyau. Les appels d'environnements dans le noyau se traduisent par des chargements de C-listes sur les registres du processeur. L'exécution d'un environnement dans le noyau fait appel à des opérations plus internes telles que des opérations sur C-listes puisque justement c'est le noyau qui met en place le type environnement. Ainsi à ce niveau la protection n'est pas encore assurée.

Comme le noyau met en place les fonctions de base fréquemment utilisées, son appel doit être réalisé à l'aide de fonctions plus élémentaires fournies par la machine. De plus cet appel doit être efficace. De même que sur une machine conventionnelle l'appel du superviseur se réalise éventuellement par un changement de jeu de registres, de même, l'appel du noyau se réalisera par un changement de C-liste d'environnement. Ainsi dans Cm* chaque processeur repère en permanence deux C-listes, celle du noyau et celle de l'utilisateur. L'appel du noyau bascule le repère sur la C-liste appropriée. La mise en place des opérations décrites précédemment concernera justement l'établissement d'une nouvelle C-liste utilisateur. La figure 7 donne le schéma de l'environnement noyau de Cm*

Les notions de type, de module peuvent être utilisées pour l'écriture du noyau mais le seul élément physique de découpage est la C-liste. La définition du noyau interfère avec celle de la machine ; par exemple, certaines opérations d'un type sont réalisées par la machine alors que d'autres le sont par programme. C'est le cas du type "type" : l'opération appliquer qui provoque la création d'un environnement est programmée alors que les opérations décomposer et contrôler sont réalisées par la machine.

L'exemple précédent illustre le découpage du noyau : extérieurement l'utilisateur voit sans doute les opérations créer environnement, appel d'environnement, retour etc ... qui elles-mêmes se réalisent à l'aide des opérations activer, suspendre etc ... Il n'est pas dans notre intention de décrire le noyau d'un tel système, celui-ci se définit en accord avec le

type de système à mettre en place. Nous prendrons seulement l'exemple de Star OS le noyau CM*. La figure 8 esquisse son organisation en modules et montre comment ceux-ci s'articulent. On notera en particulier la gestion des environnements réalisée en deux modules.

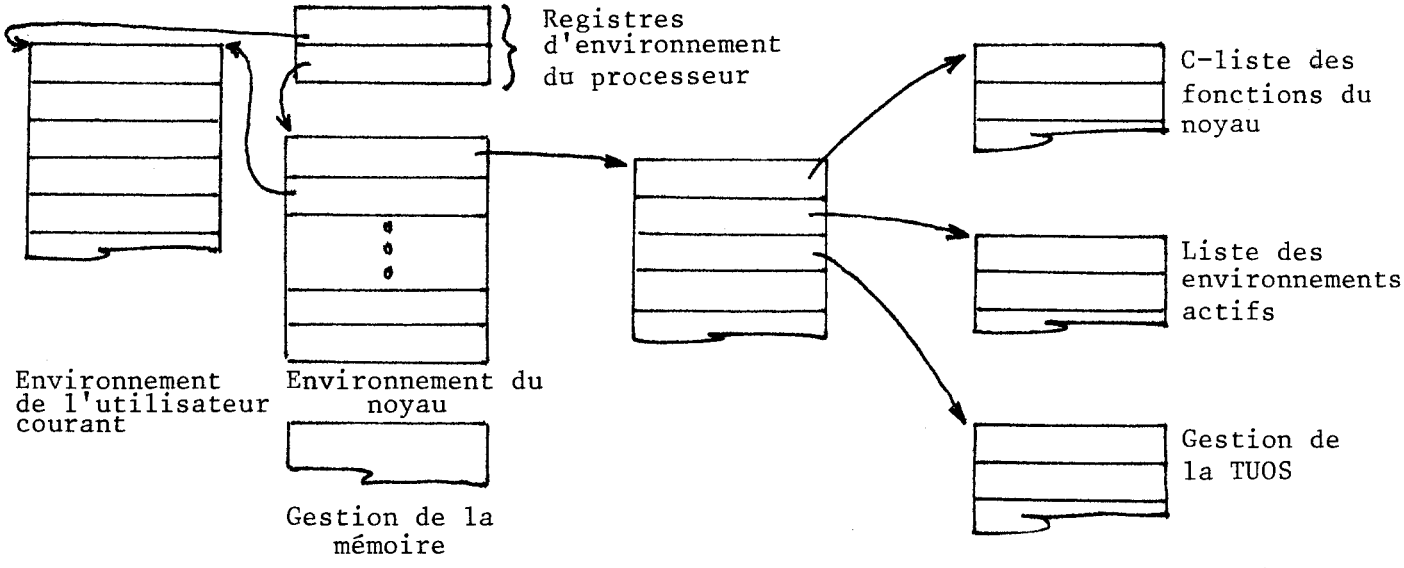


Figure 7 : Environnement Noyau dans Cm*

L'un met en place la structure des modules et utilise les opérations sur les types mis en place par le microprogramme. Tandis que le second qui met en place les opérations d'appel, de retour etc ... utilise les opérations construites par le module de multiplexage qui réalise l'activation effective d'un module sur un processeur. De même ce module utilise la gestion des messages conditionnels pour ranger un environnement dans la liste des environnements actifs d'un processeur [KL78, Jo77, Jo78].

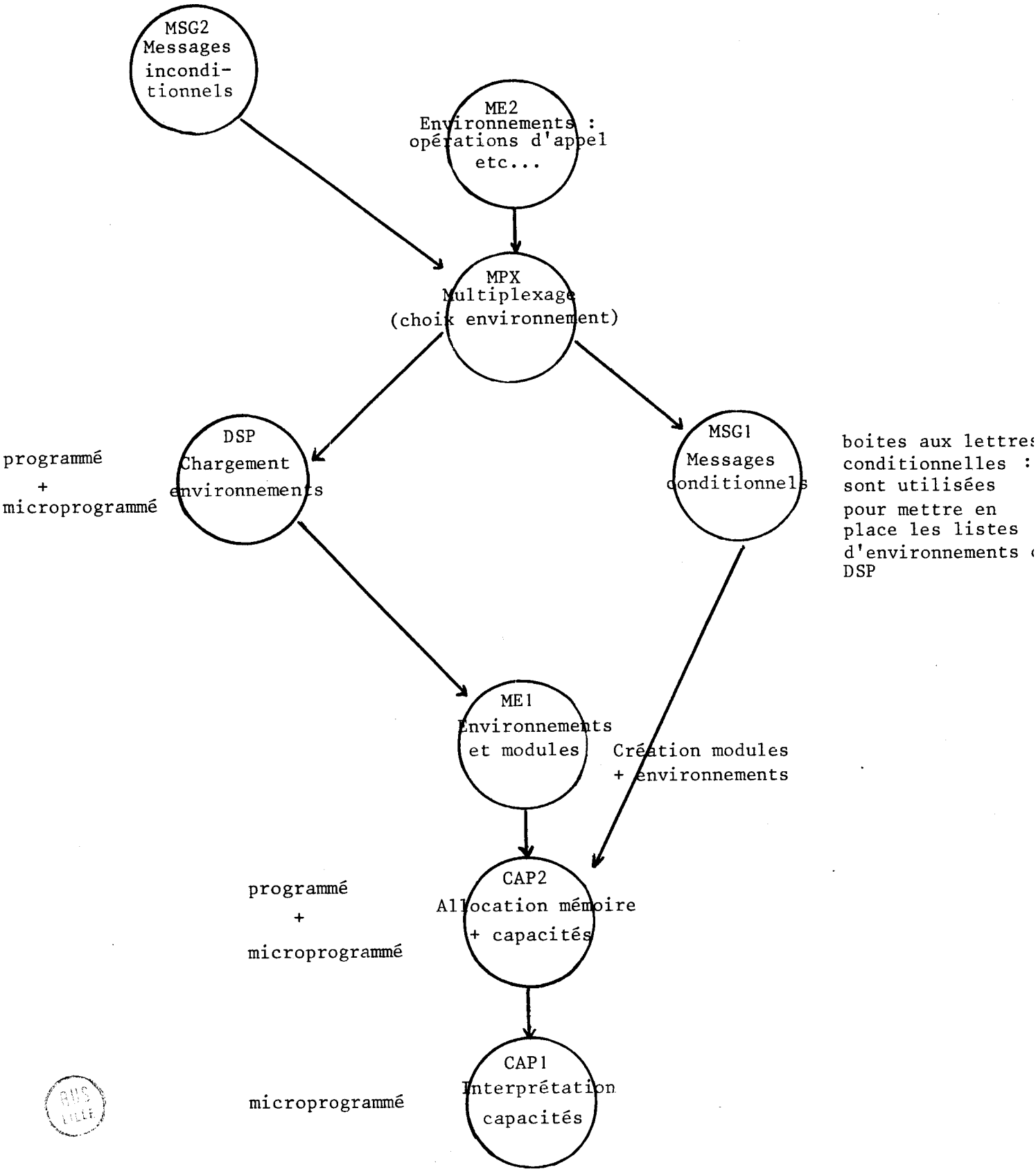


Figure 8 : Organisation du noyau de Cm*

5. - Adressage

La notion de C-liste utilisée comme unité de regroupement d'objets introduit une nouvelle dimension : l'espace des noms utilisés dans un environnement cesse d'être linéaire. L'environnement est la racine d'une arborescence de C-listes et l'accès à un objet peut requérir de cheminer à travers diverses C-listes. En particulier, des accès ont lieu à des objets dont la capacité est tenue dans la C-liste commune. En l'absence d'un mécanisme approprié, l'accès à un objet nécessiterait plusieurs opérations de chargement de capacité dans l'environnement jusqu'à ce que sa capacité y soit accessible. Hydra fournit une telle opération (l'opération "Walk") qui prenant en paramètre la description d'un chemin dans une arborescence de C-listes permet la capture de la capacité de l'objet cible dans l'environnement [Wu74a].

La C-liste commune étant l'ensemble d'objets le plus probablement accédé indirectement (elle contient entre autres les capacités des segments de code), un mécanisme à double indexation semble approprié. C'est ce type d'accès qu'utilise Cm*. La figure 9 schématise un environnement de Cm*. Le nom d'un objet se présente sous la forme (i,j) où i est un nom dans l'environnement et j un nom dans une C-liste secondaire :

- pour o<i<16 , i désigne une capacité dans l'environnement et j une capacité dans la C-liste désignée par i.
- (o,j) désigne l'environnement lui-même.
- pour i ≥16 , i désigne la capacité d'un objet quelconque dans l'environnement et j est ignoré.

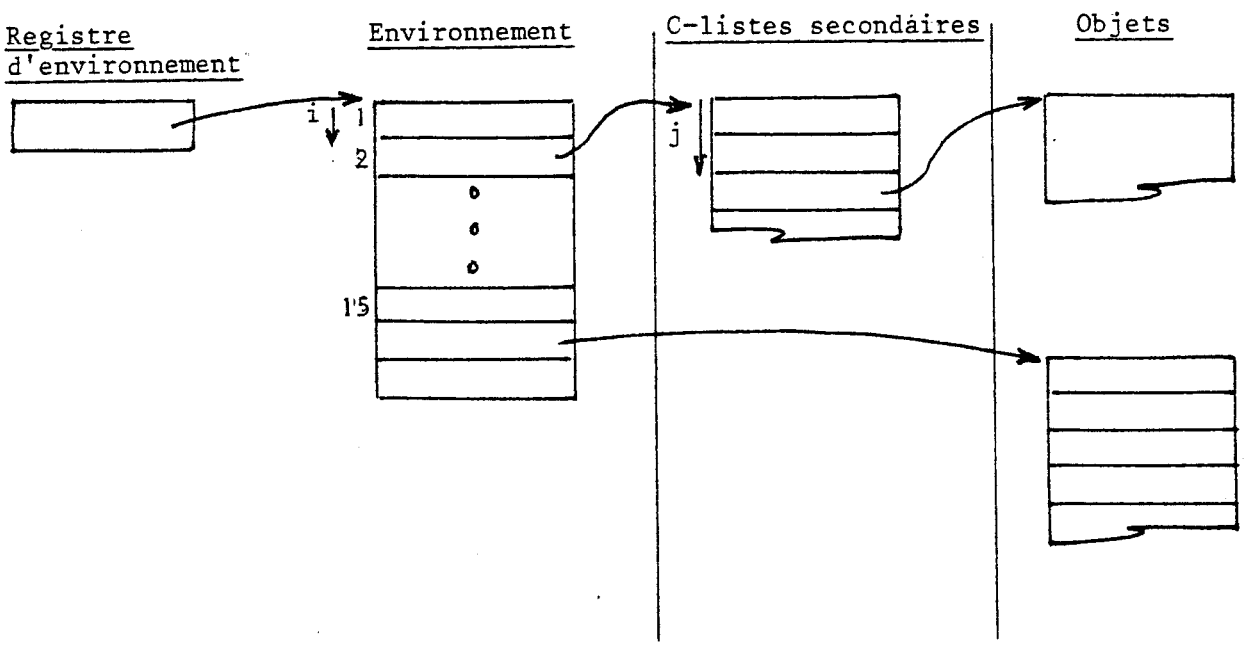


Figure 9 : Adressage dans Cm*

On sait par ailleurs que l'accès à un objet à travers sa capacité est de fait une indirection à travers son descripteur. Si l'accès est réalisé par indexation multiple, chaque indexation nécessite en plus une indirection dans la table active. Ceci contribue à ralentir les accès aux objets. On n'a pas l'expérience réelle de mécanismes accélérateurs. Les réalisations actuelles portent sur des machines dont la longueur de mot est faible (Hydra, Cm*, P.250) et où il convenait également d'étendre les possibilités de désignation . Aussi, on a utilisé des techniques de couplage. Cm* utilise des registres fenêtres que l'on charge à l'aide d'une capacité désignée par un nom (i,j). Lorsque cette capacité est chargée dans le registre elle est traduite en termes d'adresse dans la mémoire. Les mécanismes utilisés dans Hydra ou le système P.250 sont semblables. Le couplage n'est réalisé que pour les segments car ce sont les seuls objets que la machine peut interprêter. La figure 10 schématise le fonctionnement du mécanisme.

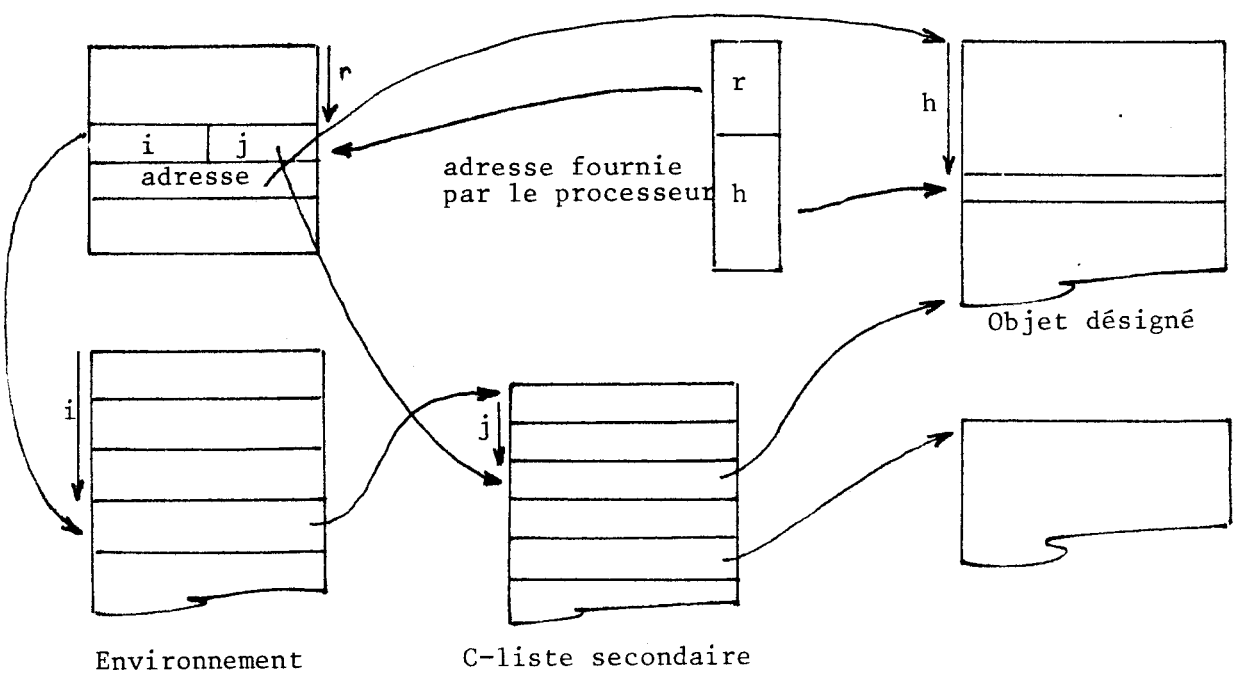


Figure 10 : Utilisation des registres fenêtres dans Cm*

Cette méthode requiert de l'utilisateur qu'il gère lui-même la carte d'implantation et qu'il connaisse constamment l'ensemble des objets qui lui sont visibles. De même qu'un compilateur est amené à réaliser la gestion des registres accumulateur de la machine, de même il pourrait réaliser cette gestion des noms. Cependant, on peut faire à ce mécanisme une critique plus importante qui est que chaque changement d'environnement nécessite la sauvegarde des noms contenus dans les registres fenêtres. Ceci contribue à pénaliser l'opération de changement d'environnement alors qu'au contraire,celle-ci étant fréquemment utilisée, il conviendrait qu'elle soit rapide.

L'utilisation d'une mémoire associative qui à un nom d'objet associe son descripteur est donc plus appropriée car elle n'est plus fonction de l'environnement courant. Le mécanisme d'adressage devant fonctionner également sans mémoire associative, il convient que celle-ci serve à l'accélération des accès dans la table active et non dans la table passive. L'accès à un objet se réalise alors de la manière suivante :

- recherche dans la table associative
- si présent alors accès
  sinon début recherche dans la table active
  si présent alors charger table associative
  sinon recherche dans table passive
  fin

De par le mécanisme de regroupement des objets, une technique de pagination peut être utilisée. Il convient donc qu'en fait la mémoire associative permette de localiser également la page et non uniquement le segment. La figure 11 schématise son fonctionnement où l'on a considéré le cas d'un accès à l'aide d'une capacité interne. L'accès à l'objet désigné se réalise alors comme une indirection.

Dans une première étape la capacité est utilisée pour localiser le descripteur interne de l'objet qui fournit son adresse de base dans le segment support. Ensuite, celle-ci doit être interprétée à nouveau puisque la mémoire associative contient une entrée par page.

Sur le plan de l'utilisation de la mémoire associative, le regroupement des objets n'améliore pas les accès. Cependant, il a permis de diminuer le nombre d'entrées dans les tables actives et passives et d'améliorer la gestion des transferts entre mémoires principale et secondaire. Il conviendrait sans doute de compléter la mémoire associative à l'aide d'un cache de sorte à accéder également plus rapidement au descripteur interne.

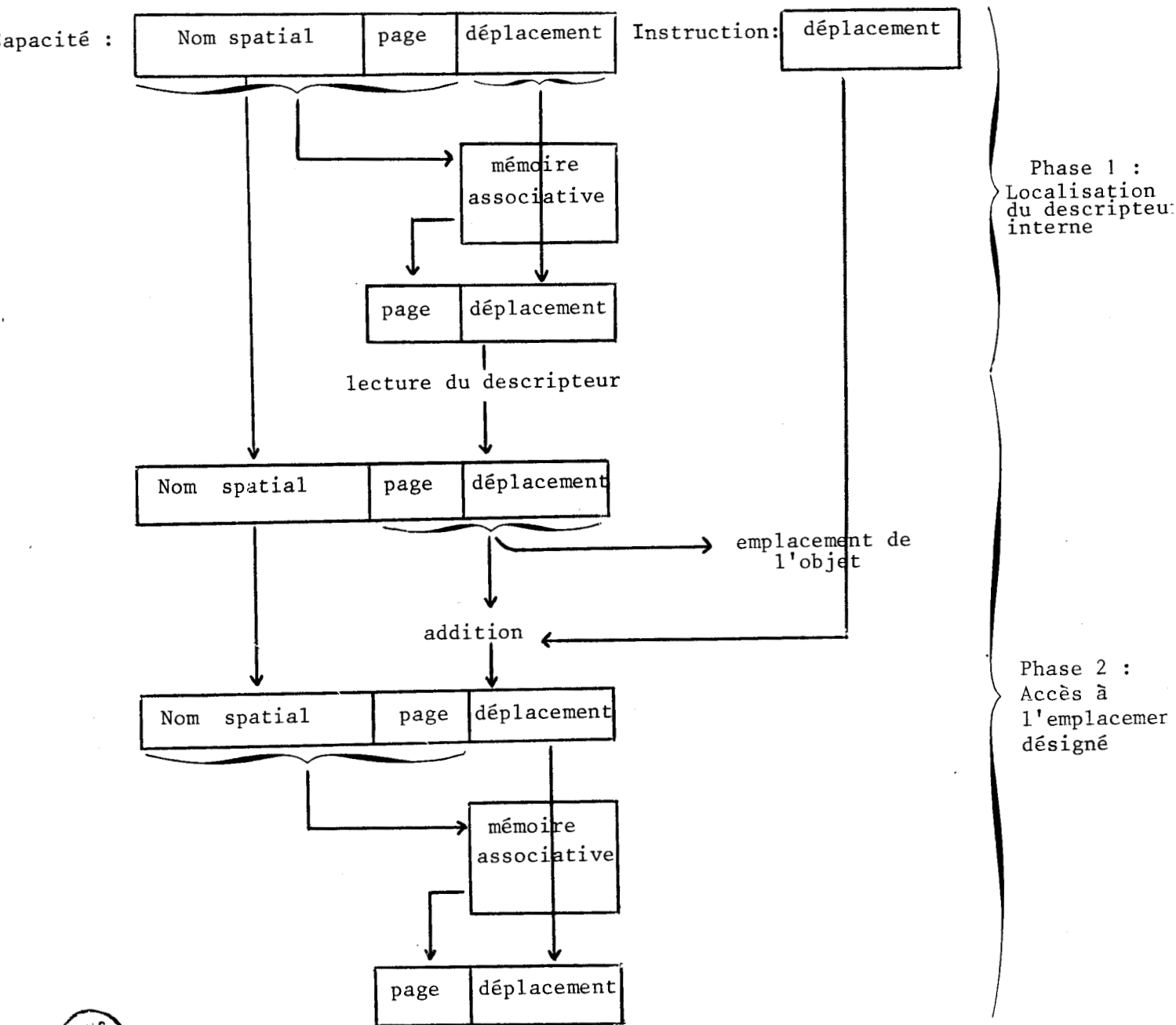


Figure 11 : Organisation de la mémoire associative

TROISIEME PARTIE

TRAITEMENTS DES ERREURS

CHAPITRE V

FIABILITE

1. - Généralités

D'une manière générale, il y a deux façons d'aborder le problème de la fiabilité. Une première méthode est illustrée par l'attitude de l'industrie aérospatiale : il ne peut y avoir de panne du tout de quelqu'élément que ce soit d'un système ; éventuellement, les composants du système sont doublés voire triplés de sorte que la fonction reste assurée. L'autre attitude est celle de l'industrie du téléphone : tant que le système complet continue à remplir son service, on accepte que des composants de ce système soient en panne.

Les deux exemples précédents illustrent deux cas extrêmes. Sans doute, dans un système complexe, aucun des deux ne s'applique au système complet mais plutôt à des parties de celui-ci. Cependant, jusqu'à il y a peu de temps, l'attitude absolutiste de l'industrie aérospatiale prévalait dans les systèmes informatiques. Wulf [Wu 75] fait remarquer que ces approches se caractérisent par le type de mission des systèmes auxquels on les applique. La mission d'un système aérospatial se compte en minutes, heures ou jours. A l'autre extrémité, le cahier des charges d'un système téléphonique précisera que ce système ne doit pas avoir plus d'une heure de panne en quarante ans. Nous nous placerons ici dans le contexte de la tolérance aux fautes (attitude de l'industrie du téléphone) et nous en appliquerons le principe au logiciel.

Il importe de distinguer entre faute et erreur. On appelle faute la panne initiale, l'erreur est le résultat apparent de la faute. Par exemple, la panne d'un circuit de la mémoire ne sera pas détectée, par contre une lecture provoquera une erreur de parité. On sait détecter les erreurs plus aisément que les fautes. L'intolérance aux pannes du logiciel consisterait à éviter les fautes de programmation et à s'assurer que les algorithmes sont corrects. Cependant, quand bien même on serait capable de prouver la correction du système complet, il faudrait s'assurer que l'implantation des algorithmes est correcte également. De plus, ceci repose sur le fait que la machine elle-même est supposée fiable. Cette attitude n'est pas jugée réaliste car elle doit porter sur l'ensemble tant matériel que logiciel. La réalisation d'un système fiable est coûteuse.

Par exemple la réalisation d'une machine fiable conduirait à multiplier le nombre de circuits et à ajouter des éléments de contrôle. Il est donc plus réaliste d'adopter une attitude de compromis et de tenir compte des conséquences possibles d'une faute. Traquer la dernière faute dans un programme est souvent difficile ; si elle provoque rarement des erreurs, il peut être plus efficace de ne pas s'en préoccuper. De même, il est plus important en général d'assurer la fiabilité des tables du système. Il en résulte qu'une attitude raisonnable consistera à traiter les erreurs plutôt que d'éviter absolument leur apparition. Cependant, ceci peut également donner lieu à un compromis. C'est ainsi qu'en général on considère acceptable le coût de détruire un processus utilisateur si ceci permet d'éviter l'écroulement du système complet et protège le reste de la communauté.

2. - Méthodologie

La construction d'un logiciel plus fiable consiste en premier lieu à éviter les fautes. Une approche appropriée est de concevoir les programmes de sorte qu'ils soient petits, simples et compréhensibles. Elle repose sur le principe de l'"information cachée", tel que le suggèrent les méthodes de construction modulaire [Pa 72a] ou de décomposition en objets abstraits [LZ 74] et que les mécanismes à capacités renforcent à l'exécution [Li 76]. Selon ces principes, un module ne connait ceux qu'il utilise que par leurs spécifications. L'utilisation d'une abstraction mise en place par un module n'a lieu qu'à travers les opérations fournies par ce module et les détails de sa réalisation sont inconnus de l'utilisateur. Ainsi, même si l'implantation d'un module est changée, elle n'a pas de répercussion sur celle des modules qui l'utilisent pourvu que ses spécifications soient inchangées. Les raisons pour lesquelles l'implantation d'un module peut être changée sont :

- la publication d'un nouvel algorithme plus efficace,
- la transformation des structures de données mises en place de sorte à répondre aux besoins de nouvelles classes d'utilisation.

Il résulte de ces principes que l'interdépendance entre modules est minimale et clairement définie par leurs spécifications. Ceci contribue à minimiser les fautes dues à une mauvaise utilisation des données.

Le traitement des erreurs, quant à lui, repose sur les principes suivants :

- détection : il doit être possible de détecter toute erreur (ou tout au moins presque toutes),
- localisation : les erreurs doivent être détectées le plus tôt possible de sorte à limiter le dommage causé et à éviter leur propagation,
- visibilité : lorsqu'une erreur est détectée, elle doit si possible rester cachée à l'utilisateur. En conséquence, elle doit être traitée dans le module qui la détecte,
- correction : l'information doit rester cohérente. Si l'erreur détectée ne peut être traitée, au moins le système doit être replacé dans un état cohérent ; l'erreur doit être signalée de sorte qu'elle puisse être corrigée à un niveau plus élevé du système, tout en restant cachée aux niveaux plus élevés que le niveau de la correction.

Il est difficile de dissocier ces principes. On notera que les principes de construction d'un système de protection s'appliquent ici aussi. Afin d'éviter la propagation des erreurs, il est important que les objets accessibles dans un module soient limités au strict minimum requis par le traitement que réalise le module. Ainsi le principe du moindre privilège contribue à la fiabilité du système. De même, afin de détecter le plus possible les erreurs il importe de construire les modules d'un système selon le principe de la méfiance mutuelle. Un module doit se méfier des résultats qu'il reçoit du module qu'il appelle. L'extension de type selon laquelle un objet ne peut être manipulé que par les opérations définies sur son type contribue à l'application de ces principes et limite la mauvaise utilisation des objets.

3. - Détection et correction

Toute erreur devrait être détectée ; en particulier la détection des erreurs produites par la machine est importante. Considérons l'exemple d'ILLIAC IV. Il n'y a aucune détection d'erreur, aussi des programmes de tests sont exécutés périodiquement pour vérifier le bon fonctionnement de la machine. Les utilisateurs sont conduits à supposer que la machine a bien fonctionné si le test suivant ne détecte aucune erreur. En fait, la plupart des erreurs de la machine sont transitoires. On peut donc douter d'une telle stratégie.

La détection des erreurs doit avoir lieu le plus tôt possible si l'on veut éviter qu'elles puissent en générer d'autres. Plus une erreur est détectée tôt et plus elle est facile à corriger. Le test des paramètres d'appel et de retour d'une procédure est donc important. Un mauvais paramètre d'entrée conduit souvent plus qu'à une mauvaise sortie, il peut remettre en cause l'intégrité du module même. Ainsi se justifie le mécanisme d'extension de type. Chaque module est totalement responsable de l'abstraction qu'il implante. Il en résulte qu'il doit maintenir l'intégrité de cette abstraction. L'exemple du code introduit à la fois dans les capacités et le descripteur d'un objet (cf. Chap. II-5.) permet de maintenir l'intégrité de l'objet en évitant son accès à l'aide d'une ancienne capacité. Enfin, chaque abstraction repose sur celles qu'elle utilise. Le contrôle qu'une procédure peut avoir sur le type invoqué est limité par le mécanisme d'extension de type. On est donc conduit à supposer que les abstractions qu'elle utilise sont à priori cohérentes. Aussi, il importe que chaque procédure vérifie que les fonctions internes qu'elle exécute fournissent des résultats corrects. En particulier, elle devrait vérifier avant que le contrôle soit rendu à l'appelant que le résultat est en accord avec ses propres spécifications. Ceci a pour résultat de maintenir également l'intégrité de la procédure elle-même. Tout module devrait ainsi corriger les erreurs détectées sur les abstractions qu'il met en place, de sorte que ces erreurs restent cachées aux utilisateurs des abstractions.

La méthode des "recovery blocks" fournit un exemple d'une telle stratégie [Hor 74]. Cette méthode suppose l'existence d'un language à structure de blocs du type Algol. Il conviendrait de discuter cette hypothèse par ailleurs, dans la mesure ou elle ne permet pas la mise en place d'abstraction. A chaque sortie d'un bloc, un test de vraisemblance est réalisé. Si le test échoue, alors toute l'information du bloc est remise dans l'état où elle se trouvait à l'entrée dans le bloc et un bloc alternant est exécuté qui réalise la même fonction. La même méthode est appliquée aux blocs alternants. Lorsque tous les blocs alternants d'un bloc principal ont échoué, la même méthode est appliquée au bloc englobant. Des tests de vraisemblance doivent être réalisés selon le même principe dans chaque procédure.

La détection autant que la correction des erreurs reposent sur l'utilisation d'une certaine redondance qui peut être spatiale ou temporelle. L'exécution des tests de vraisemblance est un exemple de redondance temporelle. Il peut s'agir de vérifier qu'un résultat se trouve dans une gamme de valeurs acceptables ou même de réexécuter partiellement la fonction testée. L'utilisation de double liens dans une structure de liste est un exemple où l'information utilisée pour détecter les erreurs dans la structure de la liste est

également utilisée pour la réparer. Il est hors du propos de cette thèse d'étudier de manière exhaustive les techniques de détection et de correction que l'on trouvera, par ailleurs, dans [Po 78].

On remarquera qu'ici encore un compromis est nécessaire. Il porte sur le coût de la correction en termes de perte de performance et de place de mémoire et le coût de l'erreur caractérisé par ses conséquences possibles. La correction des erreurs peut nécessiter une redondance plus importante que la détection. Par exemple, la méthode des "recovery blocks" utilise une redondance partielle pour réaliser les tests de vraisemblance et une redondance massive pour assurer la correction. Cette dernière consiste en la sauvegarde complète de l'état à l'entrée de chaque bloc. Si les conséquences possibles d'une erreur sont jugées moins importantes, on peut, éventuellement, se contenter de la détecter. Ainsi, les modules les plus internes du système s'attacheront sans doute à corriger toutes les erreurs car elles y sont toutes d'égale importance, tandis que les autres modules se contenteront parfois de les détecter. On ne prétend pas corriger toutes les erreurs mais seulement celles dont le coût de correction est jugé moindre que le coût des dommages qu'elles peuvent provoquer.

Considérons, par exemple, une structure de liste. On y distingue l'information qui met en place la structure, ce sont les pointeurs vers les éléments de la liste, et le contenu de la liste. On peut s'attacher à contrôler la structure de la liste plus que son contenu lui-même. Une erreur sur la structure peut provoquer la perte de plusieurs éléments, voire de la liste complète. Dans cet exemple, on corrigerait sans doute les erreurs relatives à la structure de la liste et détecterait celles qui concernent ses éléments.

On a noté, cependant, que la mise en place d'une abstraction repose sur celles qu'elle utilise. La non détection d'une erreur peut donc avoir des conséquences importantes car celle-ci peut se propager d'une abstraction à l'autre. Il importerait donc de détecter toutes les erreurs possibles. Ici encore, le coût de la détection peut être jugé trop élevé. Par exemple, le contrôle qu'une inversion de matrice a été réalisée correctement nécessiterait d'effectuer la multiplication du résultat par la matrice initiale et de vérifier que le résultat est la matrice unité. Le coût de cette vérification est vraisemblablement inacceptable car du même ordre que celui de l'opération elle-même. Le calcul de la seule diagonale est moins long. Il ne permet pas de détecter

toutes les erreurs mais vérifie la vraisemblance du résultat et détecte la plupart des erreurs. En général, toutes les erreurs ne seront donc pas détectées. Il importe cependant de détecter avec une probabilité acceptable les erreurs les plus probables ou celles jugées les plus importantes.

Lorsqu'une erreur ne peut être corrigée, elle met en jeu l'intégrité des utilisateurs de l'abstraction concernée. Elle doit donc leur être signalée. Ceci doit être fait dans des termes compréhensibles. De même que l'utilisateur d'une abstraction n'en a connaissance que par ses spécifications, de même les erreurs relatives à cette abstraction doivent être reportées dans les termes de l'abstraction. Ainsi, dans un système réalisant une mémoire virtuelle, une erreur de parité ne devrait pas être reportée comme "erreur parité, XYZ" où XYZ est une adresse physique. Reportée à l'utilisateur, elle devrait apparaître comme une erreur sur l'abstraction implantée à cette adresse. La même erreur sur un élément de la liste de l'exemple précédent serait signalée comme "élément n erroné".

4. - Signalisation

4.1. - Généralités

La signalisation d'une erreur a pour but de permettre sa correction par le module qui reçoit le signal (pour l'exemple de la liste considérée précédemment, celui-ci peut avoir conservé une copie de l'élément erroné). Si cela n'est pas possible, la signalisation de l'erreur permet aux modules utilisateurs de l'abstraction de rétablir un état cohérent. La signalisation doit se propager jusqu'à ce qu'un état global cohérent soit atteint. Ainsi, si l'abstraction erronée participe à la mise en place d'une autre abstraction, ceci provoque une nouvelle erreur reportée dans les termes de l'abstraction à ce niveau et ainsi de suite jusqu'à ce que l'erreur devienne transparente aux niveaux encore supérieurs.

Considérons l'exemple suivant. On suppose l'existence du type "liste" dont les objets sont formés d'une C-liste contenant les capacités des éléments de la liste et d'un segment décrivant la structure de la liste en termes des indices des éléments dans la C-liste. On suppose que le type des éléments d'une liste est défini à sa création à l'aide d'un objet type de sorte que les opérations du type liste puissent contrôler l'homogénéité de la liste.

Ainsi, le type sémaphore est réalisé à l'aide de la valeur du sémaphore et d'une liste dont les éléments dont des processus. Supposons qu'au cours d'une opération sur la liste d'un sémaphore une erreur de parité se produise, qu'elle modifie la capacité d'un processus de la liste et que le noyau ne soit pas capable de regénérer cette capacité. Imaginons également qu'il s'agit du processus concerné par l'opération en cours sur le sémaphore ; c'est-à-dire : s'il s'agit de l'opération V, c'est le processus qui doit être activé et dans le cas de l'opération P c'est le processus à placer en attente du sémaphore. L'action du gérant des listes (i.e. le module du type "liste") consiste à supprimer l'élément de la liste et à signaler l'erreur au gérant des sémaphores (i.e. le module associé au type sémaphore) dans les termes "élément courant perdu". Ainsi, la liste est maintenue dans un état cohérent. L'action du gérant des sémaphores dépend de l'opération en cours. S'il s'agit de l'opération P, il possède encore la capacité du processus et l'opération peut à nouveau être tentée sur la liste. S'il s'agit de l'opération V, ce n'est pas le cas. Alors l'action du gérant consiste à décrémenter la valeur du sémaphore de sorte qu'elle soit en accord avec l'état de la liste. Ensuite, l'opération V peut à nouveau être réalisée afin d'activer un autre processus et l'erreur être signalée au gérant des processus. On peut imaginer que l'action de ce dernier sera de trouver le processus qui n'appartient plus à aucune liste mais se trouve à l'état bloqué et de le réinitialiser sur l'opération P en décrémentant son compteur ordinal.

On remarquera que selon le principe de l'information cachée, il n'est pas toujours possible de corriger une erreur à l'extérieur du module qui implante l'abstraction erronée. Le gérant des sémaphores de l'exemple précédent n'était pas capable de regénérer la capacité du processus manquant. Tout au moins, il doit être possible aux modules concernés de rétablir un état cohérent tout comme le gérant des listes et le gérant des sémaphores de l'exemple. Lorsqu'une correction doit être dirigée de l'extérieur, elle doit être réalisée dans les termes de l'abstraction sur laquelle elle porte, donc à l'aide d'opérations prévues à cet effet et masquant l'abstraction à ses utilisateurs. La correction éventuelle d'une erreur est donc définie dans les spécifications de l'abstraction.

4.2. - Caractéristiques d'un mécanisme de signalisation

Un tel mécanisme a une portée plus générale dans un système que la simple signalisation des erreurs. De nombreux cas sont considérés comme anormaux vis-à-vis des spécifications d'un module sans pour autant être des erreurs. L'apparition d'un défaut de page en cours de l'exécution d'un programme en est un exemple. Il ne s'agira pas d'une erreur à moins que cela ne se produise dans l'environnement qui justement traite les défauts de page. On convient, en général, de définir des cas "anormaux" (ou exceptions) lorsque leur traitement a lieu en dehors de l'environnement où ils se produisent. Nous préférerons donc adopter le terme général d'exception pour désigner de tels évènements.

Alors le mécanisme de signalisation a pour but de localiser l'environnement le plus approprié au traitement d'une exception. Le traitement d'une exception dans un environnement peut varier à chaque instant selon le contexte d'exécution dans cet environnement (pour un programme de type Algol, le contexte serait défini ici par le bloc courant, la procédure interne en cours,...). Il en résulte que le mécanisme de signalisation doit réaliser une liaison dynamique entre exception signalée et le contexte approprié. Afin de permettre cette liaison, on suppose que les exceptions qu'un module est susceptible de signaler sont définies dans ses spécifications. Alors chaque exception signalée est identifiée sans ambiguité dans le système par son nom et l'abstraction sur laquelle elle porte. On appelle condition cette instance d'exception. On suppose de plus que l'aptitude d'un contexte à traiter une condition est définie explicitement dans ce contexte. A cet effet, pour chaque condition qu'un contexte peut traiter, il déclare un contrôleur. Cette déclaration a pour portée celle du contexte.

Détection et traitement ayant lieu dans des environnements différents, il importe que contrôleur et signaleur puissent échanger des paramètres. La seule identification d'une exception ne suffit en général pas à la traiter. Afin de préserver l'indépendance des environnements et les principes de structuration, on suppose qu'un contrôleur est une procédure de l'environnement où il est déclaré. Ainsi, les interactions du signaleur et du contrôleur d'une part, et celles du contrôleur avec le contexte où il est déclaré, d'autre part, sont clairement établies. A cause de ces contraintes, il serait donc important que la déclaration des contrôleurs et le contrôle de leur portée soient intégrés au langage utilisé [Le 77].

Alors que le problème du traitement des exceptions est abordé dans certains langages ou systèmes [PLI on condition, Pa 72c, 76, Go 75, DEC 74, Lam 74, KL 76], aucune méthode proposée n'est complètement satisfaisante. On se propose ici d'analyser l'une de ces propositions [KL 76] et d'en déduire les caractéristiques d'un schéma approprié.

[KL 76] présente un schéma répondant aux caractéristiques précédentes et basé sur la remarque qu'une erreur peut rarement être corrigée en dehors de l'environnement où elle est détectée. En effet, la correction totale d'une erreur doit permettre la continuation du traitement dans l'environnement détecteur ; elle nécessite une redondance d'information qu'en général le principe de l'"information cachée" ne permet pas de maintenir en dehors du module qui met en place l'abstraction erronée. L'exemple du sémaphore présenté précédemment illustre bien ce problème : le gérant des sémaphores n'a, en général, pas trace des processus que contient la liste associée au sémaphore. Souvent, le traitement d'une erreur en dehors de l'environnement où elle apparaît consiste donc à rétablir un état cohérent corrigeant l'effet de l'erreur et non l'erreur elle-même. Ainsi, le gérant des sémaphores rétablit un état cohérent où l'un des processus est supposé ne pas avoir exécuté d'opération P. Eventuellement, la propagation de la condition (appelée ici rétro-propagation) est poursuivie jusqu'à ce que de proche en proche, l'ensemble des abstractions concernées aient été rétablies dans un état cohérent. Alors le système est à nouveau considéré dans un état cohérent global et le traitement est repris (en fait réinitialisé). Il résulte de cette remarque que la signalisation de l'exception concerne l'environnement appelant et qu'elle agit comme une opération de retour réalisée après que l'environnement signaleur ait lui-même rétabli localement un état cohérent.

Afin de mettre en place ce schéma, chaque environnement est formé de trois régions :

- la région 'normale' est celle où prend place l'exécution lorsqu'aucun incident ne se produit,
- la région "incident" définit l'ensemble des contrôleurs mis en place dans l'environnement,
- la région "panne" pourvoit au cas où l'exception ne peut être traitée dans la région incident.

Selon ce schéma, le contexte de traitement d'une condition est implicitement défini, c'est celui où l'environnement signaleur a été invoqué. Chaque invocation d'une opération doit être précédée de l'établissement d'une région incident où sont décrites toutes les réactions aux conditions susceptibles d'être signalées par cette opération. L'exécution d'une opération retour anormal <*c*,*p*> où *c* définit l'exception signalée et *p* est un paramètre provoque un retour de cet environnement signaleur et l'activation de l'environnement appelant dans sa région incident. Lorsque la condition signalée n'a pas été prévue, la région panne définit l'action à entreprendre. Par défaut, il s'agit d'une rétropropagation vers l'environnement appelant.

Ce schéma appelle plusieurs remarques qui vont nous permettre de définir les caractéristiques d'un schéma plus approprié.

La signalisation d'une exception suppose que l'environnement signaleur ait rétabli un état cohérent car elle provoque un retour de contrôle depuis cet environnement. L'exemple du sémaphore montre qu'une telle contrainte peut s'avérer maladroite. Au cours de la réalisation d'une opération P il est possible de renvoyer le processus manquant au gérant des listes sans que l'opération sur la liste doive être réinitialisée. Le schéma proposé dans [Pa 72c] où les fonctions de traitement d'exception sont fournies à l'environnement appelé par son utilisateur permet ainsi un retour dans cet environnement et donc, éventuellement une reprise en séquence du traitement interrompu.

De même, en reposant sur le fait que chaque environnement rétablit un état cohérent local, ce schéma implique une programmation prudente et la prise en considération à chaque niveau d'abstraction de tous les cas d'exception possibles. Si à l'un des niveaux d'abstraction un état cohérent n'a pas été rétabli et que le contrôle a été rendu à l'environnement appelant, éventuellement l'état sera considéré cohérent à ce niveau alors qu'il ne l'est pas en ce qui concerne les abstractions qu'il utilise.

Comme beaucoup de schémas celui-ci ne considère que la hiérarchie des appels. Il en résulte que seuls les environnements qui n'ont pas terminé leur exécution sont susceptibles de traiter une condition. Parnas note qu'il conviendrait de prendre en compte la relation "utilise" entre environnements et abstraction et non la relation "appel". Tous les environnements qui "utilisent" un objet sont concernés par le traitement d'une condition signalée sur cet objet et non uniquement l'environnement qui avait appelé le signaleur.

Cette relation se concrétise ici par la possession d'une capacité de l'objet considéré. Ainsi tous les environnements possédant la capacité d'un objet, pourvu qu'ils soient munis d'un contrôleur adéquat, devraient être considérés a priori par le mécanisme de signalisation. La même remarque s'applique au problème du partage d'objets. La relation "utilise" est relative au partage d'objets entre environnements qu'ils appartiennent ou non à la même structure de contrôle.

L'exemple suivant, emprunté à Levin [Le 77], illustre la portée de la relation "utilise". Supposons l'existence d'un allocateur de ressources qui fournit une abstraction appelée "réservoir". Cette abstraction est partagée par différents utilisateurs qui, éventuellement, n'ont pas connaissance l'un de l'autre. Supposons que l'allocateur définisse une exception "réservoir - niveau bas" qu'il signale lorsque le réservoir ne contient pas assez de ressources pour satisfaire une requête. On peut supposer que chaque utilisateur est susceptible, en pareil cas, de libérer des ressources, soit qu'elles ne lui soient plus utiles, soit en compactant ses données. Ainsi, tous les utilisateurs du réservoir sont susceptibles d'éliminer la condition "réservoir - niveau bas" quand elle est signalée. Pourtant un seul d'entre eux a invoqué l'opération "allouer" qui a provoqué sa signalisation. Ceci illustre la différence entre la relation "appel" et la relation "utilise", lorsque des objets partagés sont en cause. Cette dernière semble plus appropriée en pareil cas. On peut encore supposer que la condition "réservoir - niveau bas" est signalée avant qu'il ne soit plus possible de satisfaire une requête, justement pour éviter que ceci se produise. Alors, il n'y aurait pas de raison d'interrompre l'allocateur après qu'il ait signalé la condition. De même, en supposant que les ressources demandées ne puissent pas être allouées, l'allocateur devrait simplement être suspendu puisque les utilisateurs sont susceptibles de rendre des ressources qui permettront de répondre à la demande. Ce n'est que lorsque la condition ne peut être éliminée et que le réservoir reste trop bas pour que l'allocation ait lieu qu'un retour doit être réalisé avec signalisation de la condition appropriée. Ainsi, ce retour doit rester sous contrôle du signaleur.

5. - Le schéma de Levin

5.1. - Définitions

Ainsi que l'a montré l'exemple précédent, il convient parfois d'associer contrôleur et condition sur la base de la relation "utilise". Cependant, il arrive tout aussi bien qu'une condition soit relative à l'invocation d'une opération. Considérons le module qui met en place la notion de fichier. L'opération écrire-fichier peut signaler deux exceptions : "fichier incohérent" et "fichier en lecture seule". La première exception reflète une incohérence détectée sur un fichier particulier tandis que la seconde est relative à une tentative d'écriture. La première concerne tous les utilisateurs du fichier, la seconde concerne le seul utilisateur qui a invoqué l'opération. Ainsi, Levin considère deux classes d'exceptions : "flow class condition" et "structure class condition". Les exceptions du type "flow" sont relatives à l'invocation d'une opération alors que celles du type "structure" concerne l'utilisation d'une abstraction. On remarquera que dans les deux cas, la relation "utilise" apparaît. En effet, les exceptions de la classe "flow" sont relatives à l'utilisation d'une instance d'opération (dans notre exemple, l'environnement créé à l'appel de écrire fichier) alors que celles de la classe "structure" sont relativesà l'utilisation d'une instance d'un type (dans notre exemple, le fichier). La notation "flow" ou "structure" définit la nature de l'instance.

Nous avions noté que le traitement d'une condition peut changer avec le contexte d'exécution dans chaque module. Ainsi dans un même environnement plusieurs contrôleurs peuvent exister sumultanément pour une même condition. Le mécanisme de signalisation doit donc identifier les contextes appropriés. On remarque que ceci se produit parce qu'en général la durée de vie d'un objet est supérieure à celle des contextes où il est utilisé. Ceci n'est pas le cas en ce qui concerne les conditions de la classe "flow". La durée de vie de l'objet concerné est celle d'une invocation de procédure dans le contexte où est déclaré le contrôleur. Le seul contexte à considérer est alors défini sans ambiguité. Ainsi la seule écriture de "raise <*condition*>" suffit à identifier le contrôleur à activer. Dans le cas des conditions de la classe "structure" la localisation des contextes concernés nécessitera en général de préciser "raise <*objet, condition*>" afin d'identifier la condition sans ambiguité. Il en sera de même de la déclaration du contrôleur. De plus, une règle d'éligibilité des contrôleurs ainsi localisés doit être définie. Ces règles d'identification et de portée se définissent en accord avec le langage

utilisé (ici Alphard [Wu 76]). L'indépendance des modules conduit à les considérer séparément et pour chacun d'eux seul est dit éligible le premier contrôleur rencontré dans un contexte englobant le contexte courant. Ce sera le seul contrôleur susceptible d'être activé dans chacun des modules. Ainsi, dans le programme de la figure 1 emprunté à Levin, la signalisation de la condition "p.pool-low" alors que le contrôle se trouve à la ligne 11 conduit à l'activation du contrôleur défini à la ligne 17. Par contre, si cette condition se produit alors que le contrôle est à la ligne 10, c'est le contrôleur défini à cette ligne 10 qui est activé.

```
1  module A
2    begin
3      shared p : pool
4      private   m, n : hairylist
5      function  useb (i : int) =
6        begin
7          private  z1, z2 : block
8          z1  ←  allocate (p,i)
9          if  z1 = nil then return fi
10         z2  ←  allocate (p, i + 3) [pool-low : release (z1) ; z1 ← nil]
11         if  z1 = nil
12             then return
13             else if z2 = nil then release (z1) ; return fi
14         fi
15            <fill in z1 and z2>
16         begin enter (m, z1) ; enter (m, z2) end  [pool-low :]
17       end  [pool-low : squeeze (m)]
18   end  [pool-low : (squeeze (m) ; squeeze (n)]
```

Figure 1 : portée des contrôleurs dans le schéma de Levin.

Levin fait encore remarquer que la durée de vie d'un contexte est toujours inférieure à celle de l'objet (puisqu'il s'agit d'une activation de fonction), et que selon les règles définies jusqu'à présent, il n'est pas possible de le contrôler pendant toute sa durée de vie. Un objet a une durée de vie supérieure à celle des contextes où il est utilisé parce qu'il subsiste dans des modules (et non plus dans des activations de procédures d'un module). Aussi, afin de permettre de le contrôler pendant toute sa durée de vie, les définitions doivent être étendues au module lui-même. Le module est alors considéré comme un contexte permanent (correspondant à son bloc le plus externe) et lorsqu'une condition est signalée, les modules même non actifs, pourvus d'un contrôleur, sont également considérés. Ainsi dans l'exemple de la figure 1, si le module A n'est pas en cours d'exécution, la signalisation de la condition "pool-low" conduit à l'activer sur le contrôleur déclaré à la ligne 18.

Lorsqu'une condition est signalée, plusieurs contrôleurs peuvent être lancés en exécution. Cependant, dans l'exemple de l'allocateur de ressources, il n'est peut-être pas nécessaire de les activer tous. Une bonne stratégie serait d'éliminer la condition "pool-low" dès que l'allocateur a reçu suffisamment de ressources pour satisfaire la demande et non d'attendre que tous les contrôleurs se soient exécutés. Levin définit trois stratégies de sélection des contrôleurs à activer :

- "broadcast" : tous les contrôleurs éligibles sont activés en parallèle avec le signaleur,
- "broadcast and wait" : tous les contrôleurs sont activés et le signaleur attend la fin de leur exécution,
- "sequential conditional" : un seul contrôleur est activé a la fois. De plus un prédicat est associé à la condition. Dès que celui-ci est vrai, la sélection des contrôleurs est interrompue et le contrôle est rendu au signaleur.

5.2. - Mise en place du mécanisme

On a noté que dans le cas de conditions, de la classe "flow", la localisation du contrôleur éligible est simple puisqu'il s'agit ou bien du contrôleur associé au contexte d'où l'appel a été réalisé ou bien du premier contrôleur rencontré dans un contexte englobant. Ainsi il suffit de lier entre eux tous les contrôleurs déclarés sur la même exception (dans une pile par exemple). Si de plus chaque contrôleur porte l'identification du contexte où il est déclaré, il est alors aisé d'identifier le contrôleur approprié dans la liste.

Dans le cas des conditions de la classe "structure", on notera que la définition des contrôleurs est basée sur la relation "utilise". Ainsi, il convient de retrouver tous les contextes qui utilisent l'objet sur lequel la condition est signalée et parmi ceux-ci de conserver ceux qui possèdent un contrôleur. Ceci suggère une implantation proposée par Levin. Pour chaque objet et chaque condition sur cet objet, on maintient en permanence une liste des contrôleurs. Comme chaque déclaration d'un contrôleur définit l'objet et l'exception concernés, il est aisé de l'insérer dans la liste appropriée. Chaque contrôleur porte également le nom unique du contexte et du module où il est déclaré, on identifie ainsi pour chaque module le seul contrôleur à activer. En fait, deux listes sont maintenues : l'une, statique, est relative aux contrôleurs déclarés au niveau des modules ; l'autre, dynamique, est relative aux contrôleurs déclarés au cours de l'évolution des contextes d'exécution issus des modules. Lorsqu'une condition est signalée, la liste dynamique est d'abord parcourue, puis la liste statique. Dans cette dernière ne sont retenus que les modules pour lesquels aucun contrôleur n'a été trouvé dans la liste dynamique. La figure 2 schématise l'organisation des listes. On y a représenté les modules à l'aide de rectangles et les contextes d'exécution dans ces modules à l'aide de rectangles imbriqués. On a supposé l'existence de deux processus partageant l'objet o. Sur cette figure, les contrôleurs C_{A21}, C_{B11}, C_{A12} seront activés lors de la signalisation de la condition α sur o. Le contrôleur C_{A11} est masqué par C_{A12}, il ne sera donc pas retenu.

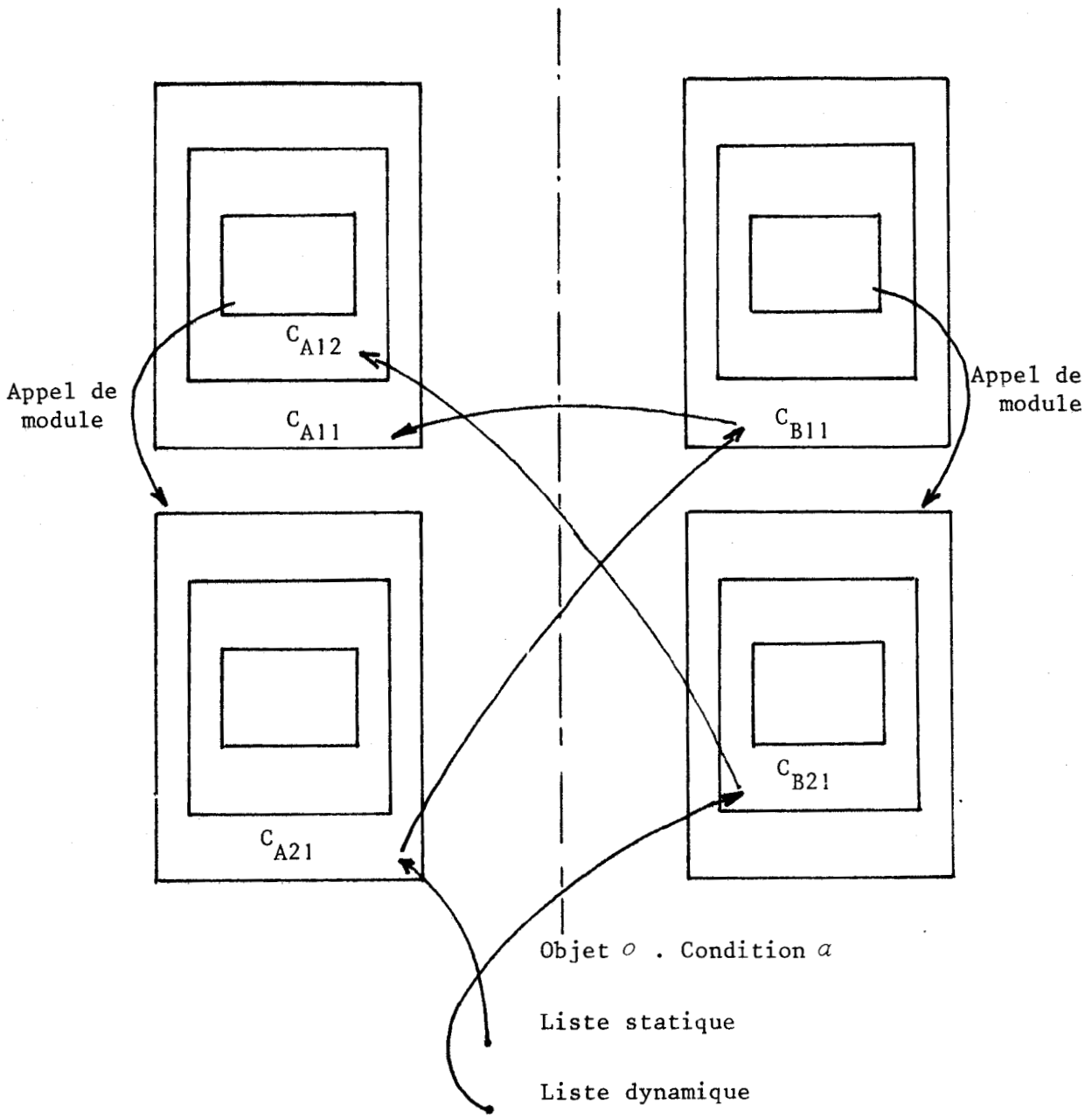


Figure 2 : Organisation des listes de contrôleurs dans le schéma de Levin.

6. - Discussion

La difficulté de mettre en place le schéma de Levin réside dans le maintien de la liste dynamique. Il intervient à chaque changement de contexte et s'avère peu efficace, en particulier dans un système à capacités où chaque copie de capacité est coûteuse. On rappelle qu'un mécanisme de signalisation efficace ne doit pas être coûteux lorsqu'il n'est pas utilisé. Or la gestion des listes prend justement place lorsqu'aucune condition n'est signalée. Pour améliorer le mécanisme, il convient d'éviter l'utilisation de capacités, ce qui implique qu'il soit mis en place dans les couches les plus internes du noyau.

On a déjà noté que les composants d'un objet n'ont parfois pas d'utilisation à l'extérieur de la structure de l'objet. Ils n'apparaissent que dans les environnements appelés successivement pour réaliser les opérations du type de l'objet. Il en résulte que la signalisation d'une condition de la classe "structure" sur le composant le plus interne d'un objet de type construit provoquera un comportement identique à celui d'une condition de la class "flow". Ce n'est qu'au niveau de composition le plus externe que la signalisation affectera éventuellement plusieurs contrôleurs : ceux fournis par les utilisateurs de l'objet. La figure 3 schématise ce comportement dans le cas d'une erreur de parité sur le segment support du fichier de l'exemple du chapitre III.

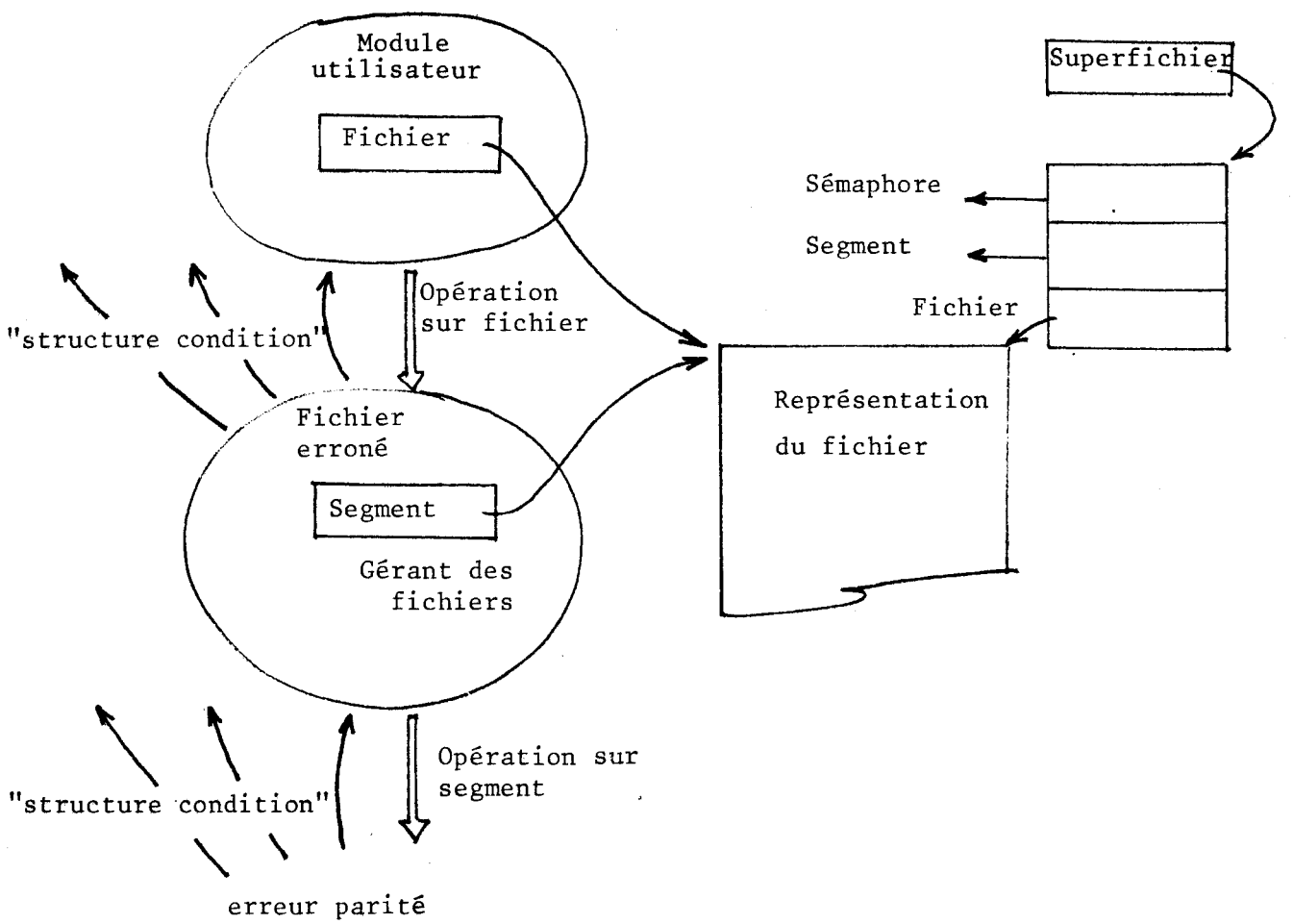


Figure 3 : Report d'erreur dans le cas d'un objet construit

L'erreur de parité est signalée sur le segment sous la forme d'une condition de la classe "structure" ; cependant, elle n'est recevable que dans l'environnement issu du module qui réalise les opérations sur le type fichier. Cette erreur ne pouvant y être corrigée, elle est transmise aux utilisateurs du fichier.

Il n'est en général pas possible de définir au niveau d'un type si l'objet manipulé est un composant interne ou un objet partagé. Il en résulte que même si la signalisation d'une exception sur le composant d'un objet se traduit par une rétropropagation de l'erreur, sa signalisation doit se faire selon le schéma relatif aux exceptions de la classe "structure condition". L'occurence d'un tel phénomène est fréquente de par la méthodologie de construction du système en termes de type. On notera cependant qu'en pareil cas, le maintien des listes et la recherche du contrôleur sont rapides dans la mesure où de telles listes ne contiennent qu'un seul élément.

Il importe également de noter que le plus souvent le rôle du contrôleur associé à une erreur traitée comme une exception de la classe "structure condition", se limite au rétablissement d'un état cohérent du module. Il est rare qu'à ce niveau une correction complète de l'erreur puisse être entreprise. Il en résulte que souvent une rétropropagation prendra place de toute façon jusqu'à atteindre un niveau où une alternative puisse être tentée.

Il résulte de ces remarques que souvent on mettra en jeu un mécanisme complexe pour produire le même effet qu'un mécanisme plus simple tel que celui présenté dans [KL 76]. Cependant, on a noté que la signalisation d'une erreur doit porter sur l'ensemble des utilisateurs de l'abstraction erronée. Le mécanisme de Levin est le seul qui permet l'application de cette méthode.

CHAPITRE VI

MAINTIEN DE LA COHERENCE

1. - Introduction

Nous avons noté au chapitre précédent que la fiabilité d'un module repose sur celle des modules qu'il utilise. Or, lorsqu'une erreur est détectée par un module et signalée au module appelant, l'information erronée lui est "cachée" et son action est souvent limitée à la restauration d'un état cohérent. La réalisation éventuelle d'une action de correction de l'abstraction erronée sera limitée par les opérations disponibles au niveau du module signaleur.

Cette méthode n'est par conséquent pas complètement satisfaisante. Elle apparaît plutôt orientée vers la détection des erreurs et la limitation des dommages causés et non vers la correction. Le plus souvent,la cohérence d'une abstraction repose sur le module qui la met en place et aucun contrôle n'est laissé à son utilisateur. De plus, souvent l'utilisateur d'un objet sera concerné par une séquence d'opérations et la cohérence de l'objet relativement à cette séquence et non relativement à une simple opération. Alors, lorsqu'une erreur est détectée, la correction nécessite que l'objet soit ramené à l'état cohérent qu'il possédait avant le début de la séquence d'opérations. Selon le schéma précédent, retrouver cet état nécessiterait de défaire la séquence des opérations réalisées. Souvent le jeu d'opérations définies sur un type inclut aussi l'inverse de ces opérations. Cependant, défaire une séquence d'opérations nécessite que chaque étape en ait été enregistrée et qu'une séquence inverse ait été définie. Ceci est une source probable de nouvelles erreurs et peut même être contradictoire avec d'autres contraintes de fiabilité. Par exemple, la réalisation de cette séquence inverse peut nécessiter des droits qui autrement n'auraient pas été nécessaires ; ceci est en conflit avec la limitation stricte des droits, introduite pour éviter des erreurs. De plus, un écroulement du système pendant une telle séquence peut laisser les objets utilisés dans un état incohérent. Ceci est irréparable si aucun état cohérent de ces objets n'a été sauvegardé.

Ces remarques suggèrent qu'il est important d'adopter une approche plus globale qui permette à un utilisateur d'un objet d'en contrôler la cohérence et qui permette d'assurer la conservation des états d'un objet.

Le mécanisme présenté ici permet de maintenir des copies d'objets. Le rétablissement d'un état cohérent consiste alors à rétablir la version appropriée des objets concernés.

2. - Structure des objets

Nous avons noté au cours des chapitres précédents de cette thèse que les objets de type construit, peuvent éventuellement être formés de composants eux-mêmes de type construit : la structure complète d'un objet peut constituer un graphe complexe. De plus, ce graphe peut évoluer durant la vie de l'objet. La structure de certains objets est totalement définie au moment de leur création tandis que d'autres objets se verront ajouter ou retirer des composants au cours de leur existence. On notera encore qu'il n'est pas interdit que des objets partagent leurs composants avec d'autres objets. Suivant le principe de l'"information cachée", la structure d'un objet et son évolution sont inconnus au niveau où il est utilisé. Comme cette évolution est probablement différente pour les objets d'un même type, il n'est pas clair ce que signifie "copier un objet" ni, a fortiori, comment le réaliser. Un modèle de composition des objets est nécessaire pour étudier ce problème. Nous considérerons deux structures de base : la structure graphique et la structure fermée.

Un objet présente une structure graphique si ses composants sont en fait des références à d'autres objets dont l'évolution est indépendante. La copie ou la sauvegarde d'un tel objet est simple et ne concerne que le premier niveau de composition. Un catalogue est un exemple d'objet à structure graphique (figure 1). C'est une table associative qui, à un nom symbolique, associe une référence à l'objet correspondant. En première approximation, on réaliserait un catalogue à l'aide d'une C-liste contenant les capacités (des références) des objets référencés ou celles d'autres catalogues. Tous ces objets peuvent être utilisés indépendamment ou à travers d'autres structures. La structure des objets contenus dans le catalogue n'est pas prise en compte à ce niveau ; ils sont utilisés et gérés ailleurs dans le système. La copie d'un catalogue consiste alors seulement à copier sa C-liste.

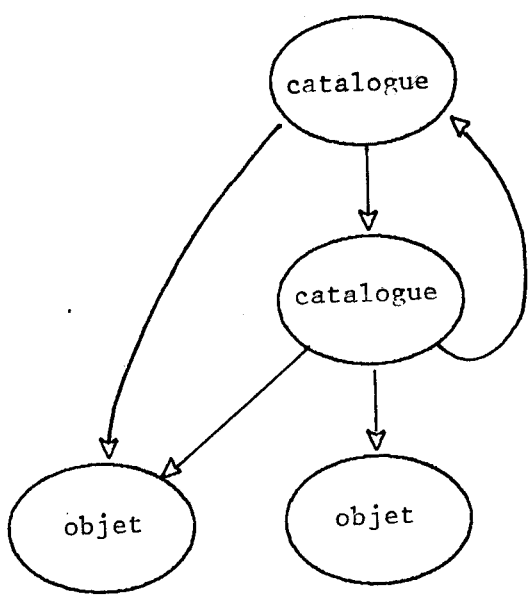


Figure 1 : Structure d'un catalogue

Au contraire, un objet serait dit de structure fermée si sa représentation contenait également celle de ses composants (des valeurs). Un fichier est un exemple d'une telle structure (figure 2). Un fichier serait formé d'un sémaphore (utilisé pour réaliser les opérations ouvrir et fermer) et d'un segment représentant le fichier proprement dit. Le sémaphore et le fichier ne sont pas des entités indépendantes.La sauvegarde d'un tel objet nécessiterait celle de tous ses composants.

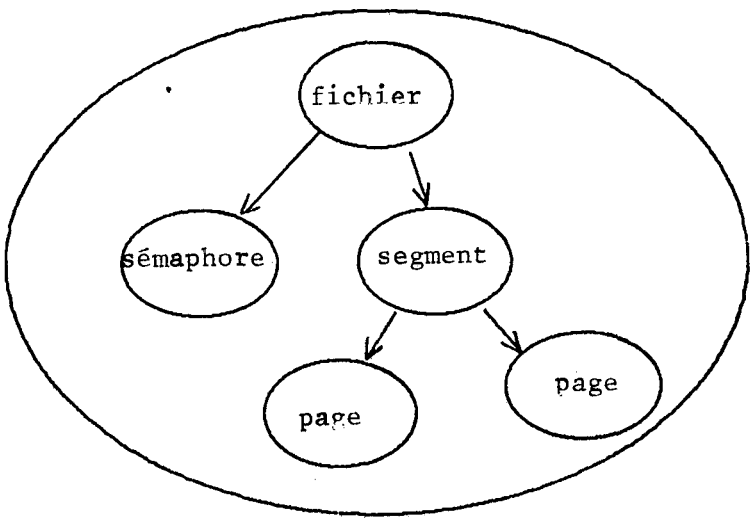


Figure 2 : Structure d'un fichier

Ces deux exemples représentent des extrêmes parmi les structures possibles d'objets. Il s'agit de deux structures pures. Même si des exemples de ces deux structuresde base peuvent être trouvées, dans la réalité la structure d'un objet procèdera de l'une et l'autre à la fois. Par exemple, un sémaphore peut être vu comme contenant un composant, sa valeur, et une référence au premier élément de la liste des processus en attente. De plus, la structure d'un objet évolue au cours de son existence. Heureusement cette évolution est connue de l'opération qui la réalise. Ainsi, il est possible de décrire complètement au niveau de l'objet si un composant est un "vrai composant" (= une valeur) ou une simple référence à un objet. Une telle description peut être maintenue par les opérations du type de l'objet.

Il convient de faire une remarque supplémentaire. Que la structure d'un objet soit graphique ou fermée peut également changer suivant la sémantique de l'opération considérée. Reprenons à nouveau l'exemple du fichier. Il sera considéré comme une structure fermée par les opérations du type fichier.

Cependant la sauvegarde du fichier ne provoquerait sans doute pas celle du sémaphore qui par conséquent ne sera pas considéré comme un composant vrai mais comme une référence.

Dans la suite de ce chapitre, nous considérerons le problème de la sauvegarde de l'état d'un objet. La nature d'une structure sera définie dans ce but. Il est clair maintenant que la sauvegarde d'un objet est complètement spécifiée par son type . Les opérations du type sont capables de maintenir pour chaque objet une description des composants à sauvegarder en même temps que lui.

3. Un mécanisme de sauvegarde

Nous avons remarqué en introduction le besoin de traiter la cohérence des objets d'une manière globale. On distinguera entre la cohérence verticale et la cohérence horizontale. La cohérence verticale est relative à l'utilisation séquentielle des objets tandis que la cohérence horizontale concerne le maintien des relations entre objets d'un même ensemble.

3.1 Cohérence verticale

On a noté qu'il importe de considérer la cohérence d'une séquence d'opérations. De même qu'il était possible de définir un état cohérent avant qu'une simple opération prenne place, de même il est possible de considérer l'état cohérent qui existe avant qu'une séquence d'opérations ne soit réalisée sur un objet donné. Alors, la correction consiste à restaurer cet état cohérent.

Puisque les objets peuvent être construits en plusieurs niveaux de composition, la cohérence d'un objet met en jeu celle de tous ses composants de proche en proche jusqu'à atteindre les composants de types primitifs.

Au niveau où nous nous plaçons, la réalisation des objets est cachée. Par conséquent, la définition et la sauvegarde d'états cohérents nécessitent l'intervention d'un mécanisme de base. Ce mécanisme devrait respecter le principe de l'"information cachée" et ne pas nécessiter la connaissance du type de l'objet à sauvegarder. Au contraire, il devrait être aussi indépendant que possible de l'objet et être réalisé par le noyau.

3.2 Cohérence horizontale

Un exemple simple suffit à présenter la cohérence horizontale. Considérons une base de données qui mémorise l'état des comptes courants des clients d'une banque. La transaction suivante enregistre un mouvement de 100 F d'un compte A sur un compte B :

```
lire A ;
A ← A - 100 ;
écrire A ;
lire B ;
B ← B + 100 ;
écrire B ;
```

la base de données doit être conservée cohérente globalement. Ici, ceci signifie que A + B doit avoir la même valeur avant et après l'exécution de la transaction. Supposons qu'une panne se produise après que A ait été modifié et que celle-ci nécessite la restauration de B et la reprise de la transaction. B ne peut être restauré seul car A a déjà été modifié et la contrainte sur A + B n'est pas satisfaite.

3.3 Introduction du mécanisme [LW 78]

La cohérence globale d'un ensemble d'objets nécessite de définir les liens entre ces objets. Dans le cas de la cohérence horizontale, l'aide du programme est nécessaire. On remarquera qu'il en était de même en ce qui concerne la structure des objets. A chaque niveau de composition de cette structure, les composants de l'objet sont liés par le même type de relation de cohérence.

Pour permettre l'expression de ces relations, on introduit la notion d'ensemble de cohérence. En supposant que S1 est un ensemble de cohérence forme de A + B, la transaction de l'exemple précédent s'écrirait :

```
ouvrir S1 ;
corps de la transaction ;
fermer S1 ;
```

Un ensemble de cohérence est un objet dont le type est réalisé par le noyau. Le but des ensembles de cohérence est de définir la portée des actions de reprise et les objets qu'elles impliquent. Il n'est pas nécessaire de contrôler l'accès à ces objets. Une fois qu'un ensemble a été ouvert ses objets sont utilisés librement à l'aide de leurs capacités.

Les opérations définies sur le type ensemble de cohérence sont : ouvrir, dans, fermer, restaurer. Les opérations ouvrir et dans permettent de déclarer la cohérence de l'état courant des objets de l'ensemble vis à vis du programme et déclenchent leur sauvegarde. Ainsi, dans l'exemple précédent, A et B seraient considérés ensemble et gardés cohérents l'un avec l'autre plutôt que d'être traités séparément. L'opération fermer déclare la fin de l'utilisation de l'ensemble. Elle signifie également que ses objets doivent à nouveau être

considérés comme cohérents vis à vis de la transaction. L'opération restaurer rétablit l'état cohérent que les objets de l'ensemble possédaient avant l'exécution de l'opération ouvrir. Par conséquent, elle ne peut être exécutée que si l'ensemble n'a pas été fermé. Afin de permettre l'identification des différentes versions d'un objet et la restauration d'un ensemble, les versions d'objets et les ensembles de cohérence sont datés avec la valeur de l'horloge à l'instant où l'ensemble est ouvert.

4. Cohérence verticale

Les ensembles de cohérence ont pour but de maintenir l'état cohérent d'un ensemble d'objets avant la réalisation d'une action. Cette action s'exprime probablement en termes d'actions plus élémentaires réalisées par des modules à des niveaux plus internes. Ces modules de leur côté ont déclaré des ensembles de cohérence qui éventuellement ont des intersections non vides avec les ensembles déclarés précédemment. On s'intéresse ici aux séquences d'actions réalisées par un utilisateur. Le paragraphe suivant étendra l'étude au cas de plusieurs utilisateur concurrents.

Reprenons la base de données de l'exemple précédent. L'utilisateur déclare A et B dans l'ensemble S1. Supposons maintenant qu'il invoque le gérant de la base de données pour réaliser une opération sur le compte A. Afin de permettre ses propres actions de reprise, le gérant déclare un nouvel ensemble S2 qui contient lui aussi A. En ce qui concerne le gérant, l'état de A avant qu'il soit appelé était cohérent. Si une reprise doit être effectuée, c'est cet état qui doit être rétabli. Par conséquent, lorsque l'ensemble S2 est ouvert, l'état de A doit être à nouveau sauvegardé. Considérons maintenant la transaction. Quand le gérant lui retourne le contrôle, l'état courant de A doit être celui qu'il possédait lorsque S2 a été fermé. Les mêmes exigences s'appliquent à tous les ensembles ouverts séquentiellement. Par conséquent, lorsque dans une même séquence une opération ouvrir est réalisée sur un ensemble qui recoupe un autre ensemble précédemment ouvert, une nouvelle version courante est fournie pour ses objets, qu'il en existe déjà une ou non, et la version précédente est sauvegardée. Il en résulte qu'entre l'ouverture et la fermeture d'un ensemble, il se peut que plusieurs versions aient été fabriquées pour certains de ses composants. Cependant, l'opération restaurer doit être capable de retrouver pour chaque objet de l'ensemble la version de référence à partir de laquelle une version courante avait été générée pour la première fois. Afin de permettre l'identification de ces versions, les ensembles sont marqués avec la valeur courante de l'horloge à l'instant où ils sont ouverts.

Il en est de même quand une nouvelle version est créée pour un objet. Ainsi, lorsqu'un ensemble doit être restauré, les versions à rétablir peuvent être identifiées en comparant la marque d'horloge de l'ensemble avec celle des différentes versions de ses objets. La version qui pour chaque objet doit être rétablie est celle dont la marque est à la fois plus petite mais la plus proche de celle de l'ensemble. La figure 3 schématise ce procédé.

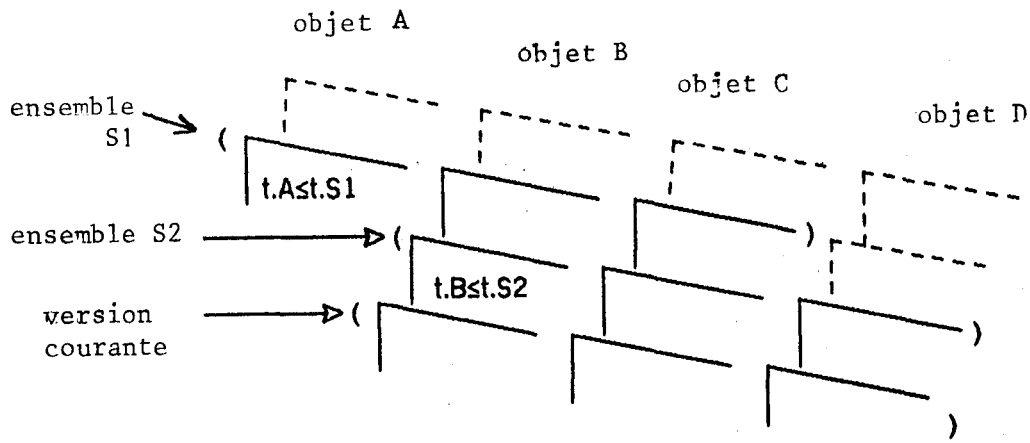


Figure 3.1 : Etats de deux ensembles imbriqués

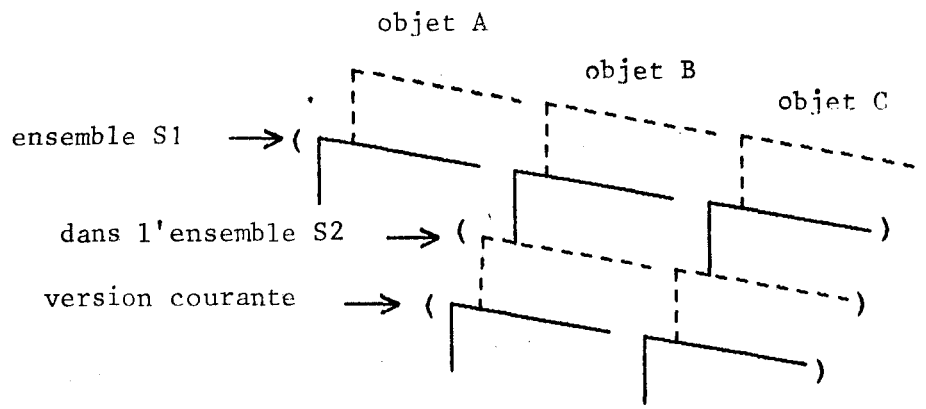


Figure 3.2 : Fermeture d'un ensemble englobé

De la même façon, l'opération fermer déclare que les versions créées pour le besoin de la transaction sont à nouveau cohérente relativement à cette transaction. Cependant, l'ensemble de cohérence continue de repérer la version précédente. Afin d'associer à l'ensemble de cohérence les nouvelles versions de ses objets, il est daté avec la valeur courante de l'horloge lorsqu'il est fermé. Alors, il est possible de retrouver les versions cohérentes les plus récentes vis à vis de la transaction à laquelle l'ensemble était associé. On remarquera encore que lorsqu'un ensemble a été fermé, de nouvelles versions cohérentes peuvent être déclarées pour certains de ses objets. Elles définissent un état partiel cohérent du système. La restauration des anciennes versions détruirait cet état cohérent. Aussi lorsqu'un ensemble a été fermé, l'opération restaurer ne peut plus être réalisée sur lui. Eventuellement des objets dont la représentation est la copie du ceux qui repère l'ensemble pourraient être fournis.

De même, à cause de l'opération fermer, lorsque des ensembles non disjoints sont ouverts l'un après l'autre, l'état cohérent que repère un ensemble est détruit par l'utilisation de ses objets dans un ensemble ouvert antérieurement (par abus nous dirons que ce dernier englobe l'autre ensemble). La figure 3.2 illustre cette remarque. Les versions quittées à la fermeture de l'ensemble englobé S2 deviennent les versions courantes de l'ensemble englobant S1. Si maintenant les objets de S1 sont modifiés, l'état repéré par S2 cesse d'être cohérent vis à vis de l'action qui avait été réalisée dans S2. L'utilisation correcte des ensembles de cohérence consiste à conserver cet état, et à faire les nouveaux accès aux objets de S1 à travers un nouvel ensemble S3. Les accès à ces objets ne se font pas dans S1 dont le rôle est de repérer un état antérieur, mais à travers S3.

Remarquons que la nouvelle version d'un objet peut être produite lors du premier accès à cet objet et non systématiquement à l'ouverture de l'ensemble qui le contient. Ceci peut être utilisé pour permettre la sauvegarde des états des ensembles englobés. Les objets de S2 peuvent être placés dans un état tel que le premier accès après que S2 a été fermé déclenche le même processus que lorsque S1 est ouvert. Ainsi la création de nouvelles versions est provoquée et l'état cohérent de S2 est sauvegardé.

Le même problème apparaît lorsqu'un ensemble englobant est restauré. Les états cohérents repérés par les ensembles qu'il englobe sont détruits. Cependant, on remarquera que l'imbrication des ensembles traduit celle des opérations. Leur effet doit être annulé lorsque la restauration porte sur une opération englobante. Ainsi la destruction des états cohérents repérés par les ensembles englobés lors de la restauration d'un ensemble englobant se justifie par le fait que celui-ci n'était pas fermé.

5. Cohérence horizontale

La cohérence horizontale est relative aux relations entre objets et est réalisée à l'aide des ensembles de cohérence. On l'étend ici au cas d'une utilisation parallèle des objets. A l'inverse d'une utilisation à travers des ensembles ouverts séquentiellement, lorsque des objets sont utilisés concurremment ils doivent exhiber la même version. Ceci conduit à des conflits qui résultent en un "effet de domino" [Ra 75].

Considérons à nouveau la base de données de l'exemple précédent. Une transaction t1 peut réaliser une mise à jour des comptes A et B pendant qu'une autre transaction t2 utilise les comptes B et C. Des règles permettent à t1 et t2 de s'exécuter en parallèle tout en restant cohérentes [Gr 75]. Cependant, si une panne dans la transaction t1 conduit à restaurer A, il se peut que la cohérence de la base de données nécessite que B soit également restauré à l'aide de l'ensemble S1 = {A,B}. Comme la restauration de B implique celle de C par l'ensemble S2 = {B,C} déclaré par la transaction t2, ceci conduit éventuellement à un phénomène cumulatif où il faut considérer la fermeture transitive de tous les ensembles qui se recoupent. Comme la restauration des ensembles pris ainsi en compte n'est pas déclenchée par leurs utilisateurs eux-mêmes, il en résulte des problèmes de reprises où toutes les transactions concernées doivent être réinitialisées.

La figure 4 illustre ce phénomène. Sur cette figure, les lignes horizontales indiquent les instants d'ouverture et de fermeture d'un ensemble et les lignes verticales la durée d'utilisation de cet ensemble. Lorsqu'une même figure recoupe plusieurs processus, elle schématise l'utilisation d'un même ensemble par ces processus. Si une erreur apparaît au point X dans le processus 4, alors l'ensemble qu'utilise le processus 3 doit être restauré. Ceci conduit à reprendre le processus 3 au point Y. Mais comme ce processus interagissait avec le processus 2 celui-ci doit être également réinitialisé, ce qui conduit à prendre en compte également le processus 1. Ce problème a été analysé dans [Ra 75].

Les utilisateurs concurrents doivent donc coopérer dans la définition de leurs actions de reprise. Ceci se traduit par la définition d'une structure commune appelée conversation [Ra 75]. Alors que dans notre exemple la transaction t1 utilisait l'ensemble de cohérence S1 = {A, B} et la transaction t2 l'ensemble S2 = {B, C}, la structure réelle de restauration est formée de S = Union (S1,S2). Les utilisateurs impliqués doivent avoir coopéré dans la déclaration de cette fusion de sorte qu'une structure commune permettant la reprise de leurs actions puisse être mise en place. Lorsqu'une opération ouvrir est tentée sur un ensemble qui recoupe un autre ensemble déjà ouvert par un autre utilisateur, cette opération est refusée. Dans ce but, quand un ensemble est ouvert, il est marqué

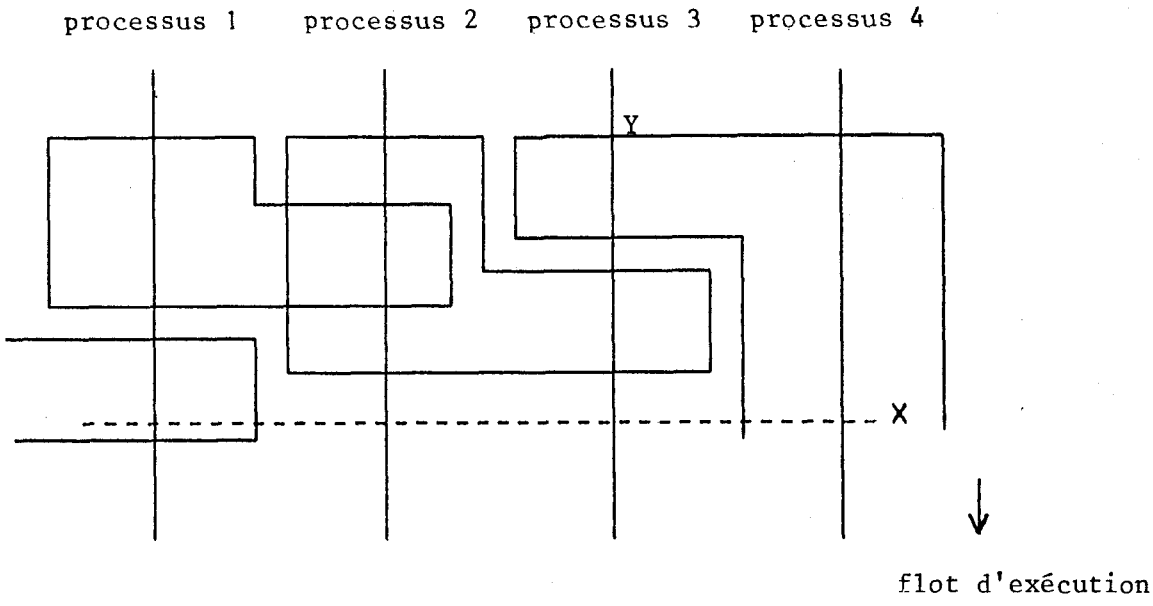


Figure 4 : Conversations entre processus parallèles

à l'aide de l'identification de son utilisateur (le nom d'un environnement par exemple). Cette marque est effacée lors de la fermeture de l'ensemble. La réalisation d'une conversation entre deux utilisateurs requiert que l'ensemble qui couvre tous les objets utilisés au cours de cette conversation ait été précédemment ouvert. Une conversation se définit donc par un ensemble de cohérence. Il est de plus nécessaire de connaître le point de reprise de chaque utilisateur impliqué dans la conversation. On lui impose de déclarer son entrée dans la conversation. Alors, ce point de reprise est celui où la déclaration a eu lieu.

L'ouverture d'une conversation se réalise à l'aide de l'opération dans. Elle s'écrit : dans S ouvrir Si qui déclare l'utilisation du sous-ensemble Si dans la conversation S. Par cette opération l'utilisateur déclare sa coopération avec ceux qui ont réalisé la même opération sur S. Alors l'ensemble de cohérence Si est lié à l'ensemble S et à tous les sous-ensembles déjà déclarés dans la même conversation. La restauration de Si déclenchera ainsi celle de S.

Dans l'exemple précédent, chaque transaction s'écrirait donc :

dans S ouvrir Si ;
corps de la transaction ti ;
fermer Si ;

On remarquera que lorsque la restauration d'un ensemble est provoquée, seul le demandeur est capable de contrôler son action de reprise. Ses interlocuteurs doivent être réinitialisés de façon à pouvoir s'exécuter à nouveau dans un état cohérent. Le point de reprise de chacun des utilisateurs impliqués dans une conversation est l'instruction qui suit l'exécution de l'opération dans. Il peut être sauvegardé dans le sous-ensemble Si déclaré par l'utilisateur.

L'opération fermer déclare la cohérence des objets d'un ensemble. Dans le cas d'une conversation cette cohérence n'est réelle que lorsque la conversation est fermée. Ce n'est le cas que lorsque tous les sous-ensembles précédemment ouverts sont fermés.

Finalement, la réalisation des actions de reprise peut affecter éventuellement des environnements qui ne coopéraient pas dans une conversation déclarée et provoquer leur incohérence. Pour éviter de tels effets de bord toutes les interactions entre utilisateurs doivent être déclarées à l'aide de conversations. Alors les structures de reprise des environnements concurrents s'imbriquent de telle manière que les dernières conversations déclarées sont des sous-ensembles des précédentes. Ainsi chaque opération dans exécutée dans un environnement déclare pour ensemble de couverture le sous-ensemble d'un ensemble précédemment ouvert.

6. Mise en place du mécanisme

La création d'un ensemble de cohérence prend en paramètre une liste d'objets et retourne une capacité pour l'ensemble créé. Quand il vient d'être créé, l'ensemble ne repère pas un état cohérent et n'est pas marqué à l'aide de la valeur courante de l'horloge.

A l'ouverture d'un ensemble, ses objets devraient être sauvegardés. On remarquera qu'il est peu probable qu'ils soient en mémoire à cet instant. Aussi, au lieu d'en faire une copie en mémoire secondaire on procède de manière inverse. Leur représentation est amenée en mémoire et déclarée comme étant la nouvelle version courante de l'objet, puis un nouvel espace leur est alloué en mémoire secondaire. La création d'une nouvelle version se réduit donc à l'allocation d'espace en mémoire secondaire. Lorsque la représentation de l'objet se trouve déjà en mémoire, elle est déclarée comme la nouvelle version puis renvoyée en mémoire secondaire de sorte à mettre à jour la version précédente.

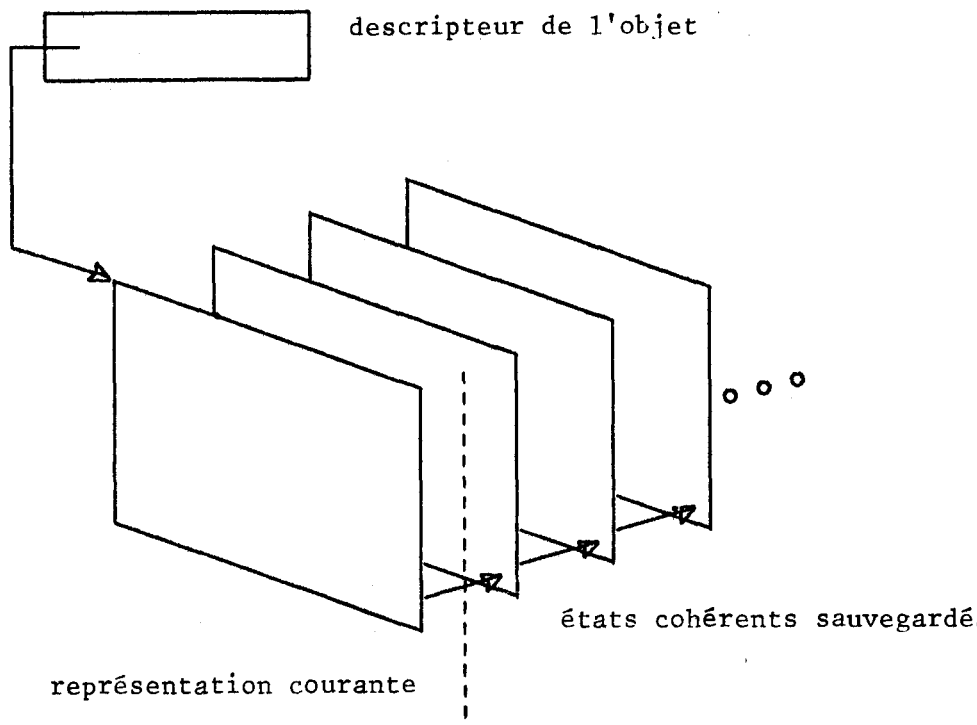


Figure 5 : Sauvegarde de l'état d'un objet

Rappelons qu'une capacité identifie un descripteur unique qui lui-même identifie la représentation de l'objet (en termes d'adresses). Ce descripteur repère la version courante de l'objet. C'est également la tête d'une liste de toutes les versions de l'objet. Ainsi la restauration de l'état antérieur d'un objet consiste à substituer les liens de cette liste. L'utilisation d'un indicateur adéquat dans le descripteur de l'objet permet de retarder le chargement de sa représentation en mémoire et la création d'une nouvelle version jusqu'à son utilisation réelle. Lors de l'ouverture d'un ensemble, les versions courantes présentes en mémoire sont renvoyées en mémoire secondaire et un indicateur de chargement est positionné de sorte à déclencher la création d'une version courante lors du 1er accès. Eventuellement, elle sera déjà présente en mémoire. La création de la version courante d'un objet provoque également son marquage avec la valeur courante de l'horloge de sorte qu'elle puisse être identifiée.

Suivant les remarques du paragraphe 2 de ce chapitre, quand une nouvelle version d'objet est créée, seuls ses "vrais" composants doivent être pris en compte. Ceux-ci sont ses composants internes et éventuellement d'autres composants externes (cf. chapitre III). Les composants internes sont créés automatiquement puisqu'ils se trouvent dans le même bloc de mémoire. En ce qui concerne les composants externes, il n'est pas besoin d'en créer immédiatement une nouvelle version. Seul le premier niveau de composition est considéré : il est défini par la TOL racine de l'objet. Celle-ci contient l'identification de ses vrais composants qui eux-mêmes contiennent l'identification de leurs propres vrais composants et ainsi de suite jusqu'au dernier niveau. On notera que ces composants peuvent être décrits à chaque niveau à l'aide d'un ensemble de cohérence.

Chaque niveau de composition est pris en compte à son tour et ses composants ne sont copiés que lorsqu'un accès y est réalisé puisqu'on ne garde trace que des modifications. Lorsqu'une nouvelle version d'objets est créée, le noyau provoque l'ouverture de l'ensemble de cohérence qui lui est associé et déclenche ainsi la création (différée jusqu'à ce qu'un accès ait lieu) d'une nouvelle version pour ses composants. Ainsi des nouvelles versions sont-elles créées récursivement à tous les niveaux de composition jusqu'à atteindre les niveaux primitifs. Les seules copies portent sur des C-listes sauf au dernier niveau de composition qui implique des segments.

On a noté qu'il est parfois nécessaire de faire intervenir l'identification d'un environnement utilisateur. Celui-ci n'intervient que temporairement et est donc rangé dans le descripteur de chaque objet de l'ensemble. Il permet de vérifier que les intersections d'ensembles sont relatives à une utilisation séquentielle et non pas à une utilisation parallèle. Lorsque le descripteur d'un objet contient déjà une telle identification, celle-ci devrait être identique à celle du demandeur sinon l'ensemble ne peut être ouvert qu'à l'aide d'une opération dans.
Celle-ci requiert uniquement que le sous-ensemble déclaré soit effectivement inclu dans l'ensemble qui définit la conversation. Le sous-ensemble est alors lié à son ensemble de couverture et aux sous-ensembles déjà ouverts.

Lors de la fermeture d'un ensemble toutes ces informations sont supprimées. De plus, le descripteur de l'ensemble est à nouveau marqué avec la valeur courante de l'horloge afin d'identifier les versions les plus récentes. L'utilisation ultérieure de ces versions récentes des objets sans déclaration d'ensemble de cohérence en détruirait la cohérence.

Afin de l'éviter, lorsqu'un ensemble est fermé, les descripteurs de ses objets sont placés dans le même état qu'à l'ouverture d'un ensemble. Ainsi, toute tentative d'utilisation provoque la création d'une nouvelle version et par conséquent la sauvegarde de celle que repère l'ensemble. La création de multiples copies que ce mécanisme peut engendrer peut être évitée à l'aide de l'indicateur de modification utilisé habituellement pour gérer la mémoire virtuelle. De cette manière, il n'y aurait pas de création d'une nouvelle version courante lorsque la précédente n'a pas été altérée.

Quoique ce mécanisme puisse sembler complexe, la gestion des ensembles de cohérence devrait être assez simple dans la mesure où elle couvre les actions habituelles d'une gestion de mémoire virtuelle. Cependant, elle utilise beaucoup plus d'espace en mémoire secondaire. Remarquons que de toute manière une technique de ramasse miettes est nécessaire. Elle devrait assurer la réorganisation de la mémoire secondaire en récupérant l'espace alloué aux versions qui ne sont plus utilisées. Chaque fois qu'un ensemble est marqué avec la valeur courante de l'horloge celle-ci définit des versions inutilisables. Cependant, la marque d'horloge n'étant identique pour aucun objet ni ensemble l'identification de ces versions nécessite une grande prudence.

CONCLUSION

Rappelons tout d'abord quels étaient nos objectifs. Il s'agissait de définir des mécanismes qui permettent d'assurer la protection des objets et la fiabilité du système. Nous avons :

- d'abord rappelé les problèmes liés au contrôle de l'accès aux objets et défini une solution à ces problèmes,
- analysé la mise en place de cette solution et précisé les mécanismes sur lesquels elle repose,
- dégagé les problèmes nouveaux que soulève la réalisation d'un système à l'aide de ces mécanismes et proposé des solutions à ces problèmes,
- analysé le problème de la fiabilité des systèmes et dégagé une méthode de structuration qui, en utilisant les mécanismes précédents, permet de construire des systèmes "tolérant les fautes",
- dégagé les besoins du traitement d'erreurs et proposé des mécanismes adaptés à sa réalisation.

Rappelons brièvement les principes qui sont à la base des mécanismes proposés. La protection et la fiabilité reposent sur le maintien à l'exécution du découpage modulaire du système. Le principe de "méfiance mutuelle" conduit à un découpage aussi fin que possible du système et à la réalisation de modules dont les espaces d'adressage sont indépendants. Les mécanismes proposés sont basés sur l'utilisation de la notion de capacité qui permet de réaliser la sélectivité et le contrôle des accès aux objets. L'utilisation explicite de la notion de type permet d'associer un objet aux opérations définies sur son type. Ainsi le type d'un objet identifie les seules opérations qui sont autorisées à manipuler sa représentation. Ceci permet d'assurer l'intégrité des objets. Les opérations d'un type sont totalement responsables du maintien de l'intégrité des objets de ce type. Un autre aspect important des mécanismes proposés réside dans l'extensibilité du système. Des nouveaux types peuvent être construits dans les termes des types déjà existants sans remettre en cause l'intégrité des objets de ces types. Ceci a été réalisé en introduisant la notion de maquette.

Les mécanismes que nous avons définis pour mettre en place les concepts rappelés ci-dessus permettent le partage généralisé des objets. Nous avons montré comment ces mécanismes peuvent être étendus au cas des réseaux de machines. Ils permettent alors, dès les niveaux les plus internes du système,

de rendre invisible la configuration du réseau et contribuent ainsi à rendre plus aisé le développement de systèmes dans un tel contexte. Ces mécanismes définissent également une architecture particulièrement adaptée aux langages à structure de type tels que CLU et Alphard. Ceci a été obtenu en étendant la notion de maquette déjà présente dans Hydra et en introduisant la notion de composant interne d'un objet. Ce mécanisme devrait favoriser le maintien à l'exécution de structures fines qui, ne nécessitant pas d'être protégées, ne seraient pas implantées à l'aide des mécanismes à capacités à cause de la lourdeur relative qu'introduisent les indirections à travers les tables du système. Ce même mécanisme nous a permis de simplifier la mise en place des environnements et d'accélérer les changements fréquents d'environnements qu'impliquent la protection. Efin, les mécanismes à capacités permettent également d'éviter la propagation des erreurs hors des modules où elles se produisent. Ceci permet de mettre en place une méthode de construction tolérant les fautes. L'introduction de la notion d'ensemble de cohérence, nous a permis de compléter cette méthode et de fournir à l'utilisateur la possibilité de contrôler la cohérence des abstractions qu'il manipule et de gérer les reprises sur erreur dans son programme.

Cependant, la réalisation des architectures à capacités présente encore des problèmes. Nous avons exposé certains d'entre eux et avons proposé des solutions. Ainsi, nous pensons que la généralisation de la notion de maquette et l'introduction de la notion de composant interne contribuent à résoudre les problèmes relatifs à la gestion des supports et des transferts qui résultent de l'existence de nombreux objets de petite taille. Par contre le problème de la récupération de l'espace occupé par les objets lorsque ceux-ci cessent d'être référencés reste posé. Nous avons montré que l'utilisation de compteurs de référence ne permet pas d'éviter la réalisation d'un ramasse-miettes, or, celle-ci s'avère complexe. Les techniques actuelles ne sont pas satisfaisantes et l'on doit envisager de réaliser la récupération de l'espace pendant que le système est actif. Le schéma proposé par Bishop nous a permis d'exposer ce problème. Toutefois, ce schéma s'avère à notre avis trop lourd pour être applicable. On notera que la notion de composant interne apporte là aussi une solution partielle. Lorsqu'un objet cesse d'être référencé, il en est de même de ses composants internes. Ainsi, seuls les objets externes doivent être identifiés par le ramasse-miettes. Nous avons noté dans le chapitre III que peu d'objets sont partagés de manière permanente aussi, une utilisation judicieuse du mécanisme de composition devrait permettre de limiter le nombre des

objets concernés par le ramasse-miettes. Il reste cependant que l'on ne peut éviter la réalisation d'un ramasse-miettes et que le nombre d'objets concernés est trop important pour que les techniques habituelles soient encore utilisables.

On notera encore l'inadaptation des schémas actuels de signalisation des exceptions. Nous avons remarqué au chapitre V la nécessité de définir les contrôleurs qui traitent une exception donnée au niveau du module, ou au niveau du contexte dans un module, et d'activer ces contrôleurs indépendamment des relations d'appel qui existent entre modules. Ce modèle, défini par Levin, ne semble pas réalisable tel quel sur un système à capacités. Le maintien des listes de contrôleurs s'y avère trop lourd lorsque les déclarations de contrôleurs sont fréquentes. Ainsi ce modèle doit être adapté pour que sa réalisation devienne satisfaisante.

Par ailleurs, si la notion d'ensemble de cohérence que nous avons introduite semble adaptée aux besoins du traitement des erreurs par l'utilisateur d'une abstraction, leur utilisation n'a pu être expérimentée. En particulier, l'ouverture et la fermeture fréquentes d'ensembles de cohérence laisse craindre une prolifération de versions anciennes d'objets et donc une occupation importante d'espace en mémoire secondaire. Il convient ici encore de réaliser un ramasse-miettes approprié qui permettent d'éliminer les anciennes versions d'objets lorsqu'elles cessent d'être référencées par un ensemble de cohérence. En fait, cette fonction doit être incluse dans le ramasse-miettes introduit au chapitre III et contribue à le rendre plus complexe. En particulier, la décision d'éliminer un objet dépend désormais des références que contiennent les anciennes versions d'autres objets. Nous avons proposé également que la réalisation de la récupération de l'espace occupé par les versions non accessibles d'un objet soit couplée aux mécanismes d'archivage du système. On notera à ce propos que cet archivage nécessite le transfert de représentations d'objets sur des supports amovibles. La protection nécessite qu'il n'y ait pas de copies de capacités sur un tel support. En effet, il serait alors possible de créer ou modifier des capacités et donc de contourner les mécanismes de protection. Il n'existe pas à l'heure actuelle de réalisation d'un tel mécanisme d'archivage.

Pour conclure, nous remarquerons que les réalisations actuelles en sont encore au stade du développement. Ainsi l'expérience acquise est encore limitée. En particulier, nous n'avons pas encore l'expérience de l'utilisation d'un système à capacités pour maintenir à l'exécution les structures mises en place par un langage possédant les notions de module, d'objet, de type. L'étude d'un tel langage (Alphard) est en cours à CMU où il existe par ailleurs des systèmes

à capacités (Hydra et Cm*), il sera intéressant de juger comment ces systèmes s'adaptent au langage.

La notion de capacité a été introduite pour définir un mécanisme de protection. Cependant, ce mécanisme n'a pas encore été réellement expérimenté sur ce plan puisqu'il n'a pas été mis à la disposition de l'utilisateur. En particulier, il n'existe pas actuellement de langage permettant d'exprimer les contraintes de protection (citons toutefois l'étude de Jones et Liskov [JL 78]). De même, certains problèmes de protection tel que celui de l'étanchéité sont encore ouverts.

Enfin, les architectures de machines actuelles sont inadaptées à la réalisation de systèmes à capacités. De plus, les réalisations actuelles utilisent des machines qui introduisent des contraintes supplémentaires telles que : l'absence de segmentation, la taille des mots, la taille de l'espace d'adressage. Même la modification de ce matériel ne permet pas de disposer d'un environnement suffisamment adapté. Il est donc difficile de faire la part des problèmes propres à l'utilisation de la notion de capacité et de ceux qui proviennent simplement du matériel utilisé.

BIBLIOGRAPHIE

[Alm 78] Almes, G., G. Robertson, An Extensible File System for Hydra, Proceedings of the *Third Conference on Software Engineering*, Atlanta, (Mai 1978).

[Ban 78] Banino, J.S., J. Ferrié, C. Kaiser, D. Lanciaux, *Contrôle de l'accès aux objets partagés dans les systèmes informatiques*, Monographie AFCET (1978).

[Bat 70] Batson, A.P., Shy-Ming Ju, D.C. Wood, Measurements on Segment Size, *Com. ACM*, (Mai 1970)

[Bat 77] Batson, A.P., R.E. Brundage, Segment Sizes and Lifetimes in Algol 60 Programs, *Com. ACM*, (Janvier 1977).

[Bet 70] Betourne, C., J. Boulenger, J. Ferrié, C. Kaiser, S. Krakowiak, J. Mossière, Process Management and Resource Sharing in the Multi Acess System ESOPE, *Com.ACM*, (Décembre 1970).

[Bi 77] Bishop, P.B., *Computer Systems with a Large Adress Space and Garbage Collection*, Ph. D. Thesis, M.I.T., (Mai 1977).

[Bra 73] Branstad, D.K., Privacy and Protection in Operating Systems, *Computer*, (Janvier 1973).

[BV 76] Briat, J., J.P. Verjus, Implication de certaines propriétés d'un noyau de système (MAS) sur son langage d'écriture, *BIGRE n°4*, (Octobre 1978).

[CM 77] Fuller, S.H., A.K. Jones, and I. Durham, Eds., The Cm* Review Report, *Carnegie-Mellon University, Dept. of Computer Science, Tech. Report*, (Juin 1977).

[Coh 75] Cohen, E., D. Jefferson, Protection in the Hydra Operating System. Proceedings of the Fifth Symposium on Operating Systems Principles, *Operating Systems Review* 9, 5 (1975).

[Cos 74] Cosserat, D.C., Adata model based on the capability protection mechanism *International workshop on Protection in Operating Systems*, IRIA (Août 1974).

[Cro 75] Crocus, *Systèmes d'exploitation des ordinateurs - Principes de conception*, Dunod éditeur, (1975).

[Dar 77] Darondeau, P., *Objets et Types dans un système Multi-langages*, Thèse d'Etat, Université de Grenoble (1978).

[DEC 74] Digital Equipment Corporation, BLISS-11 Programmer's Manual, Maynard, Mass., (1974).

[Den 74] Denning, D.E., P.J. Denning, G.S. Graham, Selectively confined subsystems, *International workshop on Protection in Operating Systems*, IRIA, (Août 1974).

[Den 76] Denning, P.J., Fault Tolerant Operating Systems, *Computing Surveys*, (Decembre 1976).

[DVH 66] Dennis, J.B., E.C. Van Horn, Programming semantics for multiprogrammed computations, *Comm. ACM*, (Mars 1966).

[Eng 72] England, D.M., Architectural features of System 250, *Infotech State of the Art Report 14 - Operating System*, (1972).

[Eng 74] England, D.M., Capability concept mechanisms and structure in System 250, *International workshop on Protection*, IRIA, (Août 1974).

[Fab 68] Fabry, R.S., Preliminary description of a supervisor for a computer organized around capabilities, *Quaterly report 18, Institute of Computer Research, University of Chicago*, (1968).

[Fe 74] Fabry, R.S., The case for capability-based computers, *Comm. ACM*, (Juillet 1974).

[Fe 74] Ferrié, J., C. Kaiser, D. Lanciaux, B. Martin An extensible structure for protected systems design, *International workshop on Protection in Operating Systems*, IRIA, (Août 1974) - *The computer Journal*, (Novembre 1976).

[Fe 75] Ferrié, J., *Contrôle de l'accès aux objets dans les systèmes informatiques*, thèse d'Etat, Paris VI, (Septembre 1975).

[Feu 73] Feustel, E. A. On the Advantages of Tagged Architecture, *IEEE Computers*, (Juillet 1973).

[FL 76] Ferrié, J., D. Lanciaux, Le système Plessey 250, *Rapport Laboria n° 168*, (Avril 1976).

[Fu 78] Fuller, S.H., J.K. Ousterhout, L. Raskin, P.I. Rubinfeld, P.J. Sindhu, R.J. Swan, Multi-processors : An Overview and Working Example, *Proc. of the IEEE*, Vol.66,No.2, (Février 1978).

[Go 75] Goodenough, J.B., Exeption Handling : Issues and a Proposed Notation, *Com. ACM*, (Décembre 1975).

[Gou 78] Goullon, H., R. Isle, K.P. Loehr, Dynamic Restructuring in an Experimental Operating System, Proc. *3rd International Conference on Software Engineering*, Atlanta, (Mai 1978).

[Gr 72] Gray, J., B. Lampson, B. Lindsay, H. Sturgis, The control structure of an operating system, *IBM Research*, RC 3949, (Juillet 1972).

[Gr 75] Gray, J.N., R.A. Lorie, G.R. Putzolu, I.L. Traiger, Granularity of Locks and Degrees of Consistency in a Shared Data Base, *IBM Research Report*, R.J. 1654, (Septembre 1975).

[Gra 68] Graham, R.M., Protection in an information processing utility, *Comm. ACM*, (Mai 1968).

[Gra 72] Graham, G.S., P.J. Denning, Protection - Principles and practice, *AFIPS Conf. Proc., 40, SJCC*, (1972).

[GR 75] Guiboud Ribaud, S., *Mécanismes d'adressage et de protection dans les systèmes informatiques. Application au noyau Gemau*, Thèse d'Etat, Université de Grenoble (1975).

[Ha 76] Habermann, A.N., L. Flon, and L. Cooprider, Modularity and Hierarchy in a Family of Operating Systems, *Com. ACM*, (Mai 1976).

[Ha 78] Habermann, A.N., P. Feiler, L. Flon, L. Guarino, L.W. Cooprider, B. Schwanke, Modularization and Hierarchy in a Family of Operating Systems, *Tech. Report, Department of Computer Science, Carnegie-Mellon University,* (Février 1978).

[Ho 74] Hoare, C.A.R., Monitors : An operating System Structuring Concept, *Com. ACM,* (Octobre 1974).

[Hor 74] Horning, J.J., A program Structure for Error Detection and Recorvery, *Proc. Conf. On Operating Systems : Theoretical and Pratical Aspects,* IRIA (1974).

[IBM 70] IBM Corporation, PL/I (F) Language Reference Manual, Form GC28-8201, IBM Corporation, (1970).

[Jan 74] Janson, P.A., *Removing the dynamic linker from the security Kernel of a computing utility,* M.S. Thesis, Project MAC, M.I.T., (Mai 1974).

[Jan 76] Janson, P.A., *Dynamic linking and environment initialization in a multi-domain process*, Ph.D., MIT, Project MAC, TR 132, (1974).

[JL 78] Jones, A.K., B.H. Liskov, A Language Extension for Expressing Constraints on Data Access, *Com. ACM.* (Mai 1978).

[Jo 73] Jones, A.K., *Protection in programmed systems,* Ph.D. Thesis, Carnegie-Mellon University, (Juin 1973).

[Jo 77] Jones, A.K., R.J. Chansler, I. Durham, P. Feiler, K. Schwans, Software Management of Cm*, a Distributed Multiprocessor, *Proc. National Computer Conference,* (1977).

[Jo 78] Jones, A.K., R.J. Chansler, I. Durham, P. Feiler, D.A. Scelza, K. Schwans, S.R. Vegdahl, Programming Issues Raised by a Multiprocessor, *Proc. of the IEEE,* Vol.66,No.2, (Février 1978).

[JS 78] Jones, A.K., L. Schiller, Dynamic Support for Small Domains, *Research Report, Carnegie-Mellon University,* (Novembre 1977).

[KL 76] Kaiser, C., D. Lanciaux, Un Modèle Uniforme pour le Traitement des erreurs dans une Architecture à Domaines, *Rapport Laboria n° 173,* (Juin 1976).

[KL 78] Kaiser, C., D. Lanciaux, Le système Cm*, *Fiches du groupe Cornafion,* (Octobre 1978).

[Lam 69] Lampson, B.W., Dynamic protection structures, *AFIPS Conf. Proc. 35 Fall Joint Computer Conference,* (1969).

[Lam 71] Lampson, B.W., Protection, *Proc. Fifth Annual Princeton Conference on Information Science and Systems, Princeton University,* (Mars 1971).

[Lam 73] Lampson, B.W., A note on the confinement problem, *Comm. ACM,* (Octobre 1973).

[Lam 74] Lampson, B.W., J.G. Mitchell, E.H. Satterthwaite, On the Transfer of Control Between Contexts, *Lecture Notes in Computer Science,* Vol.19, B. Robinet (ed.), Springer-Verlag, N.Y., (1974).

[Lam 76] Lampson, B.W., H.E. Sturgis, Reflections on an Operating System Design. *Comm. ACM,* (Mai 1976)

[Lam 77] Lampson, B.W., H.E. Sturgis, Crash Recovery in a Distributed Data Storage System, *Xerox report*, (1977).

[Le 77] Levin, R., *Program Structures for Exceptional Condition Handling*, Ph. D. Thesis, Carnegie-Mellon University, (Juin 1977).

[Li 76] Linden, T.A., Operating System Structures to Support Security and Reliable Software, *Computing Surveys*, (Décembre 1976).

[Lin 73] Lindsay, B., Suggestions for an extensible capability based-machine architecture, *International workshop on Computer Architecture*, Grenoble, (Juin 1973).

[Lis 77] Liskov, B., et Al. Abstraction mechanisms in CLU, *Computation Structures Group Memo 144-1, M.I.T. Dept of Elect. Eng. and Computer Science*, (Janvier 1977).

[LSW 78] Lanciaux, D., L. Schiller, W.A. Wulf, Supporting Small Objetcts in a capability System, *Research Report, Carnegie-Mellon University*, (Février 1978).

[LW 78] Lanciaux, D., W.A. Wulf, Error Recovery in Capability Systems, *Research Report, Carnegie-Mellon University*, (Juin 1978).

[LZ 74] Liskov, B., S. Zilles, Programming with abstract data types, *SIGPLAN Notices*, Vol 9, N° 4, (Avril 1974).

[Mo 73a] Morris, J.H., Protection in programming languages, *Comm. ACM*, (Janvier 1973).

[Mo 73b] Morris, J.H., Types are not sets, *ACM Symposium on Principles of Languages*, Boston, (Octobre 1973).

[Mos 77] Mossière, J., *Méthodes pour l'écriture de systèmes d'exploitation*. Thèse d'Etat, Université de Grenoble (1977).

[Pa 72a] Parnas, D.L., On the Criteria to be Used in Decomposing Systems into Modules, *Com. ACM*, (Décembre 1972).

[Pa 72b] Parnas, D.L., A Technique for Software Module Specification, *Com. ACM*, (Mai 1972).

[Pa 72c] Parnas, D.L., Response to Detected Errors in Well-structured Programs, *Carnegie-Mellon University, Department of Computer Science Report*, (1972).

[Pa 76] Parnas, D.L., H. Wurges, Response to Undesired Events in Software Systems, *T.H. Darmstadt*, (1976).

[Po 78] Pollack, F.J., *A Design Methodology for Fault-Tolerant Software*, Ph. D. Thesis, Carnegie-Mellon University, (1978).

[Pr 73] Price, R.W., *Implications of a Virtual Memory Mechanism for Implementing Protection in a Family of Operating Systems*, Ph. D. Thesis, Carnegie-Mellon University, (Juin 1973).

[Ra 75] Randell, B., System Structure for Software Fault Tolerance, *IEEE Transactions on Software Engineering*, (Juin 1975).

[Red 74] Redell, D.D., R.S. Fabry, Selective revocation of capabilities, *International workshop on Protection in Operating Systems*, IRIA, (Août 1974).

[Sal 74] Saltzer, J.E., Protection and the control of information sharing in Multics, *Comm. ACM*, (Juillet 1974).

[Sch 72] Schroeder, M., *Cooperation of Mutually Suspicious Subsystems*, Ph. D. Thesis, Massachusetts Institute of Technology, (1972).

[Sp 73] Spier, M.J., T.N. Hastings, D.N. Cutler, An Experimental Implementation of the Kernel/Domain Architecture, *A.C.M. Operating System Review*, (Octobre 1973).

[Sp 74] Spier M.J., H. Hastings, D.N. Cutler, A storage mapping technique for the implementation of protective domains, *Software-Practice and Experience, vol.4*, (1974).

[Wal 73] Walker, R.D.H., *The Structure of a Well-Protected Computer*, Ph.D. Thesis, Cambridge, (Décembre 1973).

[Wal 74] Needham, R.M., R.D.H. Walker, Protection and Process Management in the CAP computer, *International workshop on Protection in Operating Systems*, IRIA, (Aout 1974)

[Wu 74] Wulf, W.A., et. al., Hydra : The Kernel of a multiprocessor Operating System, *Com. ACM*, (Juin 1974).

[Wu 75] Wulf, W.A., Reliable Hardware-Software Architecture, *SIGPLAN Notices, 10, 6*, (1975).

[Wu 76a] Wulf, W.A., London, R.L., and Shaw, M., Abstraction and Verification in Alphard, *New Directions in Programming Languages* - 1975, S.A. Schuman (ed.), IRIA (1976).

[Wu 76b] Wulf, W.A., R.L. London, M. Shaw, Abstraction and Verification in Alphard : Introduction to Language and Methodology *Carnegie-Mellon University and USC Information Sciences Institute Technical Reports*, (1976).

[Wu 76c] Wulf, W.A., R.L. London, M. Shaw, An introduction to the Construction and Verification of Alphard Programs, *IEEE Transactions on Software Engineering*, (Décembre 1976).