N° d'ordre : 1008

UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

THÈSE

présentée à

L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

pour obtenir

LE TITRE DE DOCTEUR DE 3ème CYCLE

Dergham MICHAEL

L'EXPRESSION DES CONTRAINTES DE PROTECTION DANS UN LANGAGE DE HAUT NIVEAU, ADA, DANS LE CADRE DU PROJET OMPHALE.

Thèse soutenue le 26 novembre 1982 devant la Commission d'Examen

Membres du Jury	P.	BACCHUS	Président
	C.	CARREZ	Rapporteur
	V.	CORDONNIER	Examinateur
	E.	DELATTRE	Examinateur

PROFESSEURS 1ère CLASSE

M.	BACCHUS Pierre	Mathématiques
M.	BEAUFILS Jean-Pierre (dét.)	Chimie
M.	BIAYS Pierre	G.A.S.
M.	BILLARD Jean	Physique
M.	BONNOT Ernest	Biologie
M.	BOUGHON Pierre	Mathématiques
M.	BOURIQUET Robert	Biologie
M.	CELET Paul	Sciences de la Terre
M.	COEURE Gérard	Mathématiques
M.	CONSTANT Eugène	I.E.E.A.
M.	CORDONNIER Vincent	I.E.E.A.
M.	DEBOURSE Jean-Pierre	S.E.S.
M.	DELATTRE Charles	Sciences de la Terre
M.	DURCHON Maurice	Biologie
M.	ESCAIG Bertrand	Physique
M.	FAURE Robert	Mathématiques
M.	FOURET René	Physique
M.	GABILLARD Robert	I.E.E.A.
M.	GRANELLE Jean-Jacques	S.E.S.
M.	GRUSON Laurent	Mathématiques
M.	GUILLAUME Jean	Biologie
M.	HECTOR Joseph	Mathématiques
M.	HEUBEL Joseph	Chimie
M.	LABLACHE COMBIER Alain	Chimie
M.	LACOSTE Louis	Biologie
M.	LANSRAUX Guy	Physique
M.	LAVEINE Jean-Pierre	Sciences de la Terre
M.	LEBRUN André	C.U.E.E.P.
M.	LEHMANN Daniel	Mathématiques
Mme	LENOBLE Jacqueline	Physique
M.	LHOMME Jean	Chimie
M.	LOMBARD Jacques	S.E.S.
M.	LOUCHEUX Claude	Chimie
M.	LUCQUIN Michel	Chimie

M.	MAILLET Pierre	S.E.S.
M.	MONTREUIL Jean	Biologie
M.	PAQUET Jacques	Sciences de la Terre
M.	PARREAU Michel	Mathématiques
M.	PROUVOST Jean	Sciences de la Terre
M.	SALMER Georges	I.E.E.A.
Mme	SCHWARTZ Marie-Hélène	Mathématiques
M.	SEGUIER Guy	I.E.E.A.
M.	STANKIEWICZ François	Sciences Economiques
M.	TILLIEU Jacques	Physique
M.	TRIDOT Gabriel	Chimie
M.	VIDAL Pierre	I.E.E.A.
M.	VIVIER Emile	Biologie
M.	WERTHEIMER Raymond	Physique
M.	ZEYTOUNIAN Radyadour	Mathématiques

PROFESSEURS 2ème CLASSE

M.	AL FAKIR Sabah	Mathématiques
M.	ANTOINE Philippe	Mathématiques
M.	BART André	Biologie
Mme	BATTIAU Yvonne	Géographie
M.	BEGUIN Paul	Mathématiques
M.	BELLET Jean	Physique
M.	BKOUCHE Rudolphe	Mathématiques
M.	BOBE Bernard	S.E.S.
M.	BODART Marcel	Biologie
M.	BOILLY Bénoni	Biologie
M.	BONNELLE Jean-Pierre	Chimie
M.	BOSQ Denis	Mathématiques
M.	BREZINSKI Claude	I.E.E.A.
M.	BRUYELLE Pierre (Chargé d'enseignement)	Géographie
M.	CAPURON Alfred	Biologie
M.	CARREZ Christian	I.E.E.A.
M.	CHAMLEY Hervé	E.U.D.I.L.

M.	CHAPOTON Alain	C.U.E.E.P.
M.	COQUERY Jean-Marie	Biologie
Mme	CORSIN Paule	Sciences de la Terre
M.	CORTOIS Jean	Physique
M.	COUTURIER Daniel	Chimie
Mle	DACHARRY Monique	Géographie
M.	DEBRABANT Pierre	E.U.D.I.L.
M.	DEGAUQUE Pierre	I.E.E.A.
M.	DELORME Pierre	Biologie
M.	DEMUNTER Paul	C.U.E.E.P.
M.	DE PARIS Jean-Claude	Mathématiques
M.	DEVRAINNE Pierre	Chimie
M.	DHAINAUT André	Biologie
M.	DORMARD Serge	S.E.S.
M.	DOUKHAN Jean-Claude	E.U.D.I.L.
M.	DUBOIS Henri	Physique
M.	DUBRULLE Alain	Physique
M.	DUEE Gérard	Sciences de la Terre
M.	DYMENT Arthur	Mathématiques
M.	FLAMME Jean-Marie	E.U.D.I.L.
M.	FONTAINE Hubert	Physique
M.	GERVAIS Michel	S.E.S.
M.	GOBLOT Rémi	Mathématiques
M.	GOSSELIN Gabriel	S.E.S.
M.	GOUDMAND Pierre	Chimie
M.	GREVET Patrice	S.E.S.
M.	GUILBAULT Pierre	Biologie
M.	HANGAN Théodore	Mathématiques
M.	HERMAN Maurice	Physique
M.	JACOB Gérard	I.E.E.A.
M.	JACOB Pierre	Mathématiques
M.	JOURNEL Gérard	E.U.D.I.L.
M.	KREMBEL Jean	Biologie
M.	LAURENT François	I.E.E.A.
Mle	LEGRAND Denise	Mathématiques
Mle	LEGRAND Solange	Mathématiques (Calais)
Mme	LEHMANN Josiane	Mathématiques

M. LEMAIRE Jean	Physique
M. LENTACKER Firmin	G.A.S.
M. LEVASSEUR Michel	I.P.A.
M. LHENAFF René	G.A.S.
M. LOCQUENEUX Robert	Physique
M. LOSFELD Joseph	I.E.E.A.
M. LOUAGE Francis	E.U.D.I.L.
M. MACKE Bruno	Physique
M. MAIZIERES Christian	I.E.E.A.
Mle MARQUET Simone	Mathématiques
M. MESSELYN Jean	Physique
M. MIGEON Michel	E.U.D.I.L.
M. MIGNOT Fulbert	Mathématiques
M. MONTEL Marc	Physique
Mme NGUYEN VAN CHI Régine	G.A.S.
M. PARSY Fernand	Mathématiques
Mle PAUPARDIN Colette	Biologie
M. PERROT Pierre	Chimie
M. PERTUZON Emile	Biologie
M. PONSOLLE Louis	Chimie
M. PORCHET Maurice	Biologie
M. POVY Lucien	E.U.D.I.L.
M. RACZY Ladislas	I.E.E.A.
M. RICHARD Alain	Biologie
M. RIETSCH François	E.U.D.I.L.
M. ROGALSKI Marc	M.P.A.
M. ROUSSEAU Jean-Paul	Biologie
M. ROY Jean-Claude	Biologie
M. SALAMA Pierre	S.E.S.
Mme SCHWARZBACH Yvette (CCP)	M.P.A.
M. SCHAMPS Joël	Physique
M. SIMON Michel	S.E.S.
M. SLIWA Henri	Chimie
M. SOMME Jean	G.A.S.
Mle SPIK Geneviève	Biologie
M. STERBOUL François	E.U.D.I.L.
M. TAILLIEZ Roger	Institut Agricole

M.	TOULOTTE Jean-Marc	I.E.E.A.
M.	VANDORPE Bernard	E.U.D.I.L.
M.	WALLART Francis	Chimie
M.	WATERLOT Michel	Sciences de la Terre
Mme	ZINN JUSTIN Nicole	M.P.A.

CHARGES DE COURS

M.	TOP Gérard	S.E.S.
M.	ADAM Michel	S.E.S.

CHARGES DE CONFERENCES

M.	DUVEAU Jacques	S.E.S.
M.	HOFLACK Jacques	I.P..A
M.	LATOUCHE Serge	S.E.S.
M.	MALAUSSENA DE PERNO Jean-Louis	S.E.S.
M.	OPIGEZ Philippe	S.E.S.

Je tiens à remercier

Monsieur le Professeur Pierre BACCHUS, qui m'a fait l'honneur d'accepter de présider ce Jury,

Monsieur le Professeur Christian CARREZ qui a dirigé cette thèse. Il m'a accueilli dans son équipe de recherche et par ses remarques et conseils, tant sur le fond que sur la forme, m'a aidé à mener ce travail à son terme,

Monsieur le Professeur Vincent CORDONNIER pour m'avoir accueilli dans son laboratoire, soutenu dans les moments les plus difficiles et pour avoir accepté de faire partie du Jury,

Monsieur Eric DELATTRE, Assistant à L'Université des Sciences et Techniques de Lille 1, qui a aussi accepté d'examiner ce travail. Ses conseils, les nombreuses discussions que nous avons eues, m'ont été précieux.

Mes remerciements vont aussi à tous les membres du Laboratoire d'Informatique et tout particulièrement à Michel MERIAUX, Attaché de Recherches au CNRS, pour ce qu'il a fait pour moi et la liste est longue ...

Je remercie également Madame Françoise TAILLY qui a courageusement et avec beaucoup de sympathie réalisé la frappe de ce document, ainsi que Madame Henriette DEBOCK qui en a très gentiment assurée l'impression.

Enfin 1, ma Femme, qui m'a accompagnée tout au long de ce travail et qui a su me supporter avec beaucoup de patience, mérite d'être remerciée particulièrement.

Enfin 2, après tant d'années en France dont quelques unes dans le Nord, je voudrais remercier tous les amis que j'ai eu la chance de rencontrer ; qu'ils sachent que j'en garderai toujours un chaleureux souvenir.

A mes Parents

A mes Soeurs et Frères

A Roueida

A Rime

L'EXPRESSION DES CONTRAINTES DE PROTECTION DANS UN LANGAGE DE HAUT NIVEAU ADA, DANS LE CADRE DU PROJET OMPHALE.

INTRODUCTION

Ce travail s'inscrit dans le cadre du projet de recherche OMPHALE [DEL 79], ayant pour objet la conception, la réalisation et l'évaluation d'une architecture matérielle et logicielle adaptée à la programmation modulaire et structurée.

L'obtention d'un système sûr et fiable, objectif final de la recherche, a conduit les concepteurs à doter le système de mécanismes de protection, l'ensemble de ces mécanisme allant du cablé au programmé, qui permettent de contrôler l'accès à tous les objets du système.

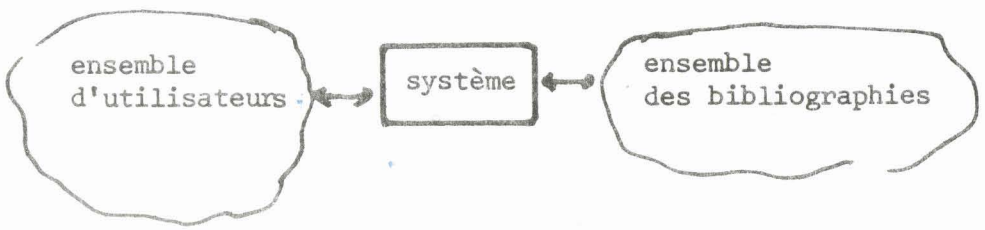
Plus précisément, ces mécanismes permettent :

- de maintenir l'intégrité des objets. Il s'agit de faire en sorte que seules les opérations définies sur un objet puissent être appliquées.

- d'assurer la sélectivité des accès. Un système (matériel et logiciel) est considéré comme un outil partagé entre plusieurs utilisateurs ayant des droits différents sur les objets du système. Ces droits s'exprimant en terme d'opérations applicables sur les objets, il faut pouvoir faire varier ces droits en fonction des utilisateurs.

Citons deux exemples bien connus :

1°) - L'exemple pris par Wulf [WUL 74] pour HYDRA et étudié par GEIB pour l'IAPX 432 [INTEL 81][GEI 82] peut se résumer par le schéma suivant :



Le système doit assurer le contrôle de l'accès aux bibliographies par les utilisateurs en fonction des droits accordés à chacun d'entre eux.

2° - L'exemple d'un système de courier dans une prison, introduit par Ambler [AMB 77], que nous présenterons et étudierons au chapitres I et III.

Ainsi, le "pouvoir d'un processus" se définit, d'une part, par les objets accessibles par le processus, d'autre part, par les opérations applicables sur ces objets.

Deux approches de solution ont été développées :

- celle basée sur les anneaux de protection, l'exemple le plus connu est celui de MULTICS.

- celle basée sur les domaines de protection.

Pour des raisons qui sont exposées dans [DEL 79] et en particulier, le respect du "principe du moindre privilège" qui limite à tout instant l'accès aux seuls objets nécessaires, l'architecture d'OMPHALE, et les mécanismes de protection proposés sont essentiellement basés sur l'adressage par capacité et les domaines de protection. Au départ, les promoteurs de ces notions voyaient en elles un moyen d'établir une protection des entités (appelées objet) composant le système, c'est à dire la possibilité de contrôler la validité des accès aux objets... Puis, les travaux menés par [FER 74] et [WUL 74] cherchaient à établir une certaine correspondance entre la notion de type, issue des études sur les langages supportant les types abstraits, et les notions proposées. Ces langages, comme SIMULA, CLU et ADA et bien d'autres, indépendemment de toute architecture contribuent à résoudre certains problèmes de protection, en permettant une programmation sur laquelle des contrôles peuvent être effectués dès la compilation.
[FER 74] différencie ces contrôles de protection statiques des contrôles dynamiques, répétés systématiquement à chaque accès durant l'exécution, car le contrôle statique se révèle insuffisant et incomplet en particulier en cas de panne de matériel par exemple, qui annulera la preuve de correction d'un programme.

OMPHALE est, plutôt, une machine orientée contrôle dynamique.
Les recherches tant dans le logiciel que dans le matériel ont généralement été menées séparément, ce qui a eu pour conséquence de limiter la portée des résultats obtenus. Une recherche coordonnée matériel-logiciel apparaît mieux adaptée pour atteindre les objectifs de fiabilité annoncés.

Ce travail reprend et poursuit les travaux de Férié, Wulf et Jones-Liskov dans le cadre du projet OMPHALE.

Après avoir rappelé dans le chapitre I les notions essentielles de protection dans les systèmes, l'apport de différents langages existants et les extensions des langages à l'aide des capacités etc..., le chapitre II rappelle l'architecture du système, les mécanismes de protection utilisés, et précise les caractéristiques que devrait avoir un langage adapté.
Le chapitre III justifie le choix du langage ADA et étudie la contribution de celui-ci à la résolution de certains problèmes de protection, on y étudie comment ADA permet de résoudre des problèmes types de protection, puis on présente les problèmes qui restent ouverts.

Ce choix étant effectué. Le chapitre IV, dans sa première partie, étudie une implémentation comparable sur une machine existante (L'IAPX 432, système orienté objet dont l'adressage repose sur des capacités regroupées) conçue pour être programmée entièrement en langage de haut niveau. Dans la seconde partie on étudie comment l'implémentation d'ADA peut être fait sur le système et nous montrons ainsi qu'OMPHALE peut supporter un langage de haut niveau adapté aux problèmes de protection.

CHAPITRE I

LA PROTECTION DANS UN SYSTÈME
ET
SON EXPRESSION DANS LES LANGAGES DE PROGRAMMATION

INTRODUCTION.

La protection des objets, initialement considérée comme un problème de système, s'est avérée être nécessaire à la réalisation de logiciel fiable. Ce chapitre, reprenant [CARREZ 79] étudie les moyens d'exprimer cette protection dans les langages évolués.
Tout d'abord, on présente un bref aperçu des problèmes liés à la protection des objets dans un système. En essayant de décrire un exemple dans différents langages actuels, on peut exhiber quelques mécanismes de ces langages, qui peuvent être exploités pour obtenir une certaine prtoection sur les objets dans un programme. Deux propositions d'extension sont ensuite étudiées. D'une part, les types qualifiés [LISKOV et JONES 76] utilisés conjointement avec des règles de liaison permettent un contrôle statique à la compilation des droits d'accès aux objets. D'autre part, les manageurs de capacité qui permettent [KIEBURTZ et SIL 78] de relier dynamiquement des objets entre eux tout en assurant le contrôle des droits d'accès.
Les problèmes liés à la révocation de ces droits sont également abordés.

I.1 - NOTIONS DE PROTECTION.

Il est couramment admis que la protection (ensemble des mécanismes allant du cablé au programme qui permettent de contrôler l'accès à tous les objets du système) a pour but :

- de maintenir *l'intégrité des objets* : Ils ne doivent être manipulés que conformément à leurs spécifications ; il s'agit de faire en sorte que seules les opérations définies sur un objet puissent être appliquées.

- d'assurer la *sélectivité des accès* : Un agent ne peut exécuter une opération sur un objet que s'il a le droit de le faire.

Parmi les problèmes classiques de protection à résoudre, mentionnons :

* **La méfiance mutuelle entre sous-systèmes**, un sous-système étant constitué d'un ensemble de procédures agissant sur des données (objets propres).

L'intégrité des objets d'un sous-système donné ne peut être assurée que si ceux-ci ne sont accessibles que par l'intermédiaire de ces procédures c'est à dire que leur représentation est inaccessible de l'extérieur du sous-système Inversement, une procédure de ce sous-système ne doit pas perturber l'appelant, c'est-à-dire avoir des actions erronées ou malveillantes sur les objets de l'appelant. Ce problème connu sous le nom de *"problème de cheval de Troie"* ne semble pas totalement soluble . Sa solution nécessite :

- l'isolation réciproque des sous-systèmes
- le contrôle de l'accès au sous-système en des points d'entrée spécifiés qui définissent les opérations du sous-système
- le contrôle des objets transmis en paramètre,
- la possibilité de restreindre les droits d'accès aux objets qu'il transmet en paramètre.

Le problème de la méfiance mutuelle se présente même lorsque les objets ne sont pas partagés. Sa solution, de par la modularité obtenue, aide à la construction de logiciel fiable et évite dans une certaine mesure la propagation des erreurs.

* La révocation des droits acquis.

Ce problème résulte du passage d'objets en paramètres à des sous-systèmes. La révocation est la possibilité pour un agent, qui a transmis tout ou partie de ses droits sur un objet à d'autres agents, de leur retirer sélectivement ces droits.

* L'étanchéité.

C'est empêcher un sous-système de divulguer de son propre chef des informations qui lui ont été transmises en paramètres.
Ces deux derniers problèmes sont posés lorsque les objets sont partagés.

Deux études sont présentées ici. La première consiste à montrer, sur un exemple, quels mécanismes, existant dans certains langages de programmation (qui n'ont pas été prévus dans ce but) peuvent résoudre la méfiance mutuelle.

La seconde expose quelques tentatives d'extension des langages pour tenir compte des problèmes de protection en utilisant la notion de capacité.

I.2 - MECANISMES D'EXPRESSION DANS LES LANGAGES EXISTANTS.

Les mécanismes de protection offerts par les langages de programmation sont assez variés.

Exemple : Système de Courier dans une Prison (SCP) ([Ambler 77]).

1°) - Les prisonniers communiquent par l'intermédiaire du SCP en échangeant des messages qui sont transmis par les gardiens. Chaque message consiste en une lettre signée enfermée dans une enveloppe cachetée comportant l'adresse du destinataire, et qui ne peut être ouverte que par lui.

2°) - Pour éviter que les gardiens ne puissent connaître *"qui envoie à qui"*, les prisonniers décident de passer par l'intermédiaire d'un facteur non partisan, c'est à dire qu'ils enferment les enveloppes dans un paquet étiquetté à l'adresse du facteur, lequel ouvre les paquets, trie les lettres, remet dans un même paquet celles à destination d'un même prisonnier, le ferme, l'étiquette à l'adresse du prisonnier.

On peut définir 6 procédures qui sont les suivantes :

P1	:	écriture des lettres et mise en paquet	(prisonnier)
G1	:	transfert des paquets des prisonniers au facteur	(gardien)
F1	:	ouverture des paquets, tri des lettres	(facteur)
F2	:	reconstitution des paquets	(facteur)
G2	:	distribution des paquets	(gardien)
P2	:	ouverture des paquets, lecture des lettres	(prisonnier)

La structure de données est en figure 1 et la table des accès en figure 2.

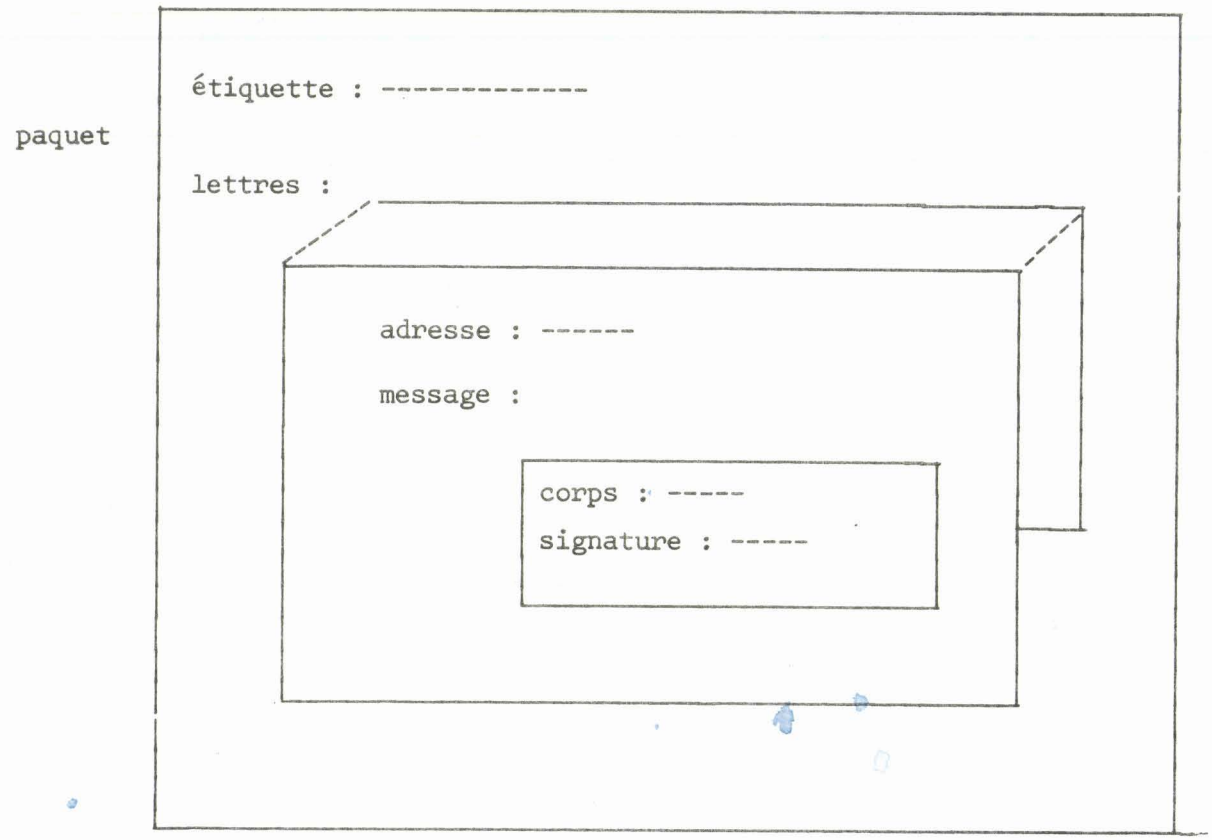


Figure 1. Structure de données.

	étiquette	adresse	corps	signature
P1	E	E	E	E
G1	L	/	/	/
F1	L	L	/	/
F2	E	L	/	/
G2	L	/	/	/
P2	L	L	L	L

Figure 2. Table des accès.

(L = accès en lecture, E = accès en écriture, / pas d'accès).

Ambler a proposé des solutions en PASCAL, Concurrent PASCAL, CLU, GYPSY, EUCLID

Nous rappelons les solutions en PASCAL, CLU et nous proposons une solution en SIMULA.
Deux solutions en ADA sont proposées dans le chapitre III, après avoir exposé ce qu'ADA apporte comme solutions aux problèmes de protection.

LA SOLUTION EN PASCAL.

Les moyens de protection de Pascal sont assez faibles : la solution proposée fait intervenir la possibilité de passer une procédure en paramètre. Pour protéger un objet X des actions d'une procédure P, il suffit que cette procédure ne soit pas dans la portée de X. La manipulation de X par P sera cependant possible sous contrôle des procédures décrivant les opérations autorisées sur X, qui sont passées en paramètres à P.

Cette solution prendra la forme suivante :

```
type paquet ---
type boîte aux lettres : array [id-prisonnier] of paquet ;
procédure P1 (pq : paquet) ;
procédure G1 (procédure retirer, ajouter) ;
procédure P2 (---) ;
procédure PRINCPALX ;
        var boîtes : boîte  aux lettres ;
        procédure retirer-boîte ;
        procédure ajouter-boîte ;
   loop ;
         P1 (boîtes [i]) ; % tous accès %
         G1 (retirer boîte, ajouter boîte) ;
         F1 F2 ;
         G2 ;
         P2 ;
   end  ;
end PRINCIPALX.
```

LA SOLUTION EN SIMULA.

Ambler a indiqué dans son article une solution en concurrent Pascal, où il utilise :

- le mécanisme d'"*entry*", pour rendre visible explicitement certaines fonctions d'accès (procédure). En effet elles sont toutes cachées à priori.

- L'attribut "*decoy*", pour cachéer sélectivement des identificateurs dans leurs portées, i.e. dans les contextes où l'on veut interdire leur utilisation.

SIMULA 67 ne fournit pas ce mécanisme de protection. Cependant, une extension [PAL 76] introduit les spécifications suivantes :

- les attributs peuvent être déclarés PROTECTED ; ils ne sont alors visibles que dans la classe où ils ont été déclarés "protected" et dans le corps des sous-classes de cette classe.

- Les attributs peuvent être déclarés HIDDEN ; ils sont alors invisibles dans les sous-classes de la classe où apparait cette spécification.

Notons que les spécifications HIDDEN et PROTECTED peuvent être combinées :

- les attributs déclarés HIDDEN PROTECTED ne sont alors visibles que dans le corps de la classe où figure la spécification.

- Les attriburs déclarés NOT HIDDEN PROTECTED sont alors rendus explicitement visibles de l'extérieur de la classe ; c'est pratiquement le mécanisme d'"*entry*" en C-PASCAL.

Nous avons cherché à fabriquer le "decoy" par la "*redefinition*" des objets associée aux règles de visibilité dans SIMULA, langage à structure de bloc comme ALGOL 60.

La solution obtenue est alors une traduction immédiate en SIMULA de celle en C-Pascal.
Les "*message*", "*lettre*", "*paquet*" sont définis dans des classes de portée globale : tmessage, tlettre, tpaquet.

Voici un extrait des déclarations de ces classes et la solution sera de la forme suivante :

```
CLASS tmessage ;
      NOT HIDDEN PROTECTED lire-corps, écrire-corps ;
      corps ;
      sign ;
      string procédure lire-corps ; begin lire-corps := corps end ;
            procédure écrire-corps (s) ; string S ; begin corps := S end ;

  begin end ;
end CLASS tmessage ;

CLASS tlettre ;
      NOT HIDDEN PROTECTED lire-adresse, écrire-lettre ;
      integer adresse ;
      ref (tmessage)  message ;
      integer procédure lire-adresse ;
              begin
                  lire-adresse := adresse ;
              end ;
      procédure écrire-lettre (m, a) ref (tmessage)  m , integer a ;

              begin
                message := m ;
                adresse := a ;
              end ;

      begin end ;

end CLASS  tlettre ;
```

```
CLASS tpaquet
      NOT HIDDEN PROTECTED lire-étiq, écrire étiq ;
      integer etiq ;
      ref (tlettre)  lettre ;
      lettres : sequence of t lettre ;

      integer procédure lire-étiq ;
              begin
                lire-étiq := étiq ;
              end ;
      procédure écrire-étiq (l, a) ref (tlettre)  l, integer a ;


              begin
                étiq := a ;
              end ;


      begin end ;
end CLASS  tpaquet ;

CLASS BP ; sequence of  tpaquet ;
CLASS BL ; array [1 .n] of  tpaquet ; integer n ;
ref (BL) boîte ;
```

```
procédure FACTEUR (mb) ; ref (BL) mb ;
      CLASS  tmessage
                          co vide
      end CLASS  tmessage ;
      procédure F1F2 (mb) ref (BP) mb ; --- end F1F2 ;
  begin
      F1F2 (mb) ;
  end FACTEUR ;


procédure XG (boîte) ;
      CLASS  tmessage ;
                          co vide
      end CLASS  tmessage ;
      CLASS  tlettre
                          co vide
      end CLASS tlettre ;
      ref (BP) mb ;
      procédure G1 (boîte, mb) ; ref (BL) boîte ; ref (BP) mb ;
      ¦
      ¦
      end G1 ;

      procédure G2 (boîte, mb) ; ref (BL) boîte, ref (BP) mb ;
      ¦
      ¦
      end G2 ;

 begin
  ¦
  ¦
   G1 (boîte, mb) ;
   FACTEUR (mb) ;
   G2 (boîte, mb) ;
 end XG ;
```

```
procédure PRISON1  (box) : ref  (tpaquet) box ;
      integer boîte ;
      procédure P1 (b) ref  (tpaquet) b ; --- end ;
  begin
      P1 (box) ;
  end ;

procédure PRISON2  (box) ; ref  (tpaquet) box ;
      integer boîte ;
      procédure P2 (b) ref  (tpaquet) b ; --- end ;
  begin
      P2 (box) ;
  end ;

begin
  loop
     for i := 1 to n do PRISON1  (boîte [i]) ;
        XG (boîte) ;
     for i := 1 to n do PRISON2  (boîte [i]) ;
     end loop ;

  end ;
```

Remarques :

- 1 - La procédure F1F2 est déclarée dans une procédure englobante FACTEUR dans laquelle nous avons redéfini une classe tmessage (vide) qui ne contient aucune opération. La classe originale tmessage est donc cachée, et par conséquence F1F2 ne pourra pas utiliser les opérations définies sur les messages, ni les procédures ayant un paramètre de cette classe. Comme elle connait toujours la classe tlettre et la classe tpaquet, elle peut donc appeler la procédure lire-adresse mais ne peut pas appeler la procédure écrire-lettre.

- *2* - De même pour les procédures G1, G2 sont déclarées dans une procédure englobante XG dans laquelle nous avons redéfini les classes tmessage et tlettre qui ne contiennent aucune opération. Les classes originales tmessage et tlettre sont alors cachées, et par conséquence G1 et G2 ne peuvent pas utiliser les opérations définies sur les messages et lettres, ni les procédures ayant un paramètre de ces classes. Comme elles connaissent toujours la classe tpaquet, elles peuvent donc appeler la procédure lire-étiq mais elles ne permet pas donc d'appeler la procédure écrire-étiq (1, a) car 1 est de type tlettre.

- *3* - Les variables qui composent les objets d'une classe sont HIDDEN PROTECTED et ne sont accessibles que par l'intermédiaire des opérations de cette classe.

- *4* - Les procédures des prisonniers peuvent appeler toutes les procédures définies dans toutes les classes.

LA SOLUTION EN CLU.

Comme on l'a dit dans le rapport technique, l'unité de base de ce langage est le module qui est :

- une procédure qui fournit une opération abstraite
- ou un cluster qui fournit un objet abstrait.

Une procédure peut retourner plusieurs valeurs comme résultat. Le résultat peut être assigné à un n-uplet.
Par exemple :

```
Pile - élément := Dépiler (pile)
où Dépiler = opération (S : type pile)
             retourne (type pile, type-élément).
```

Le "Cluster" contient une représentation de l'objet, qui est utilisée pour réaliser l'abstraction d'un ensemble d'opérations portant sur cette représentation. La représentation n'est connue que dans le "cluster" et ne peut être référencée que dans les opérations du "cluster". Seules peuvent être exportées

des opérations et celle-ci sont explicitement spécifiées dans une liste d'exportation. Ainsi on peut résumer en disant un "cluster" masque "complètement" la représentation des données qu'il contient, seuls les noms d'opérations sont exportables. Leurs noms figurent en tête de "cluster" dans la liste d'exportation.

Une variable en CLU est un identificateur susceptible de dénoter des objets d'un certain type précis au moyen d'une instruction d'affectation.
Un objet est une structure possédant une valeur de type spécifié.

Les objets sont crées dynamiquement par une opération *create* définie pour chaque type (explicitement ou implicitement). La création d'un objet fournit un nouvel objet (unique) qui n'a jamais existé auparavant.
En plus, CLU distingue deux formes d'égalité :

1°) - Deux variables sont égales (equal) si et seulement si elles dénotent toutes les deux le même objet.

2°) - Deux variables sont similaires (similar) si elles dénotent des objets ayant la même valeur.

A cause de l'unicité des objets, si equal (X, Y), alors il existe Z qui a crée l'objet pointé à la fois par X et Y et une séquence d'affectations telles que :

X := --- := Z et Y := --- := Z.

donc il est possible de savoir si deux variables nomment le même objet.
Cela signifie qu'il est impossible de fabriquer un objet et de le trouver égal à un autre objet quelconque.
On peut alors contrôler l'utilisation des fonctions d'accès par l'intermédiaire d'une *clé*, objet crée de façon unique. Cette *clé* ne pourra pas être refabriquée.
Ainsi la création d'une lettre a pour paramètre une clé qui est mémorisée dans l'objet lettre créé, et qui devra être fournie en paramètre des opérations contrôlées sur cet objet.

La solution prend la forme de la figure suivante :

```
type clé
type lettre = cluster is

              end

type boîte = cluster is

              end

type boîte aux lettres = array of type boîte
procédure FIF2
procédure G1
procédure G2
procédure P1
procédure P2
procédure PRINCIPALE
        clé-M : type clé ( ) ;
        clé-L : type clé ( ) ;
        boîtes : boîte aux lettres ;
     loop P1 (clé-L, clé-M)
          G1 ( — )
          FIF2 (clé-M, —)
          G2
          P2 (clé-L, clé-M, boîte [i])
     end loop

end PRINCIPALE.
```

Toutes les définitions de types ont dans leur portée P1, G1, F1F2, G2, P2.
La solution repose sur deux faits :

- 1 - Tous les accès à des champs critiques sont réalisés uniquement par des opérations d'accès des types.

- 2 - L'accès aux opérations des types est contrôlé par des clés.

Si une fonction d'accès est présentée avec une clé définie dans la structure de donnée alors l'accès est autorisé, puisque les clés ne peuvent pas être forgées, elles peuvent seulement avoir été données à la fonction par l'extérieur et de façon contrôlée.
Deux clés sont utilisées clé-M et clé-L (facteur et prisonnier) elles correspondent à la capacité d'accéder à Message type et Lettre type.
Elles sont ainsi crées et mémorisées dans les paquets ou dans les lettres. Notons que G1 ou G2 n'ayant pas la clé M ne peuvent ouvrir les paquets et ainsi accéder aux lettres des prisonniers.

La procédure PRINCIPALE démarre en créant les 2 clés uniques.
En conséquence, toute déclaration de lettre enveloppe est créée pour contenir la clé nécessaire à P1, G1, F1F2, P2, G2 selon la protection requise. Enfin on peut noter que le problème de la sélectivité d'accès aux objets et aux opérations est obtenue en construisant une clé ne pouvant pas être refabriquée.

I.3 - UTILISATION DES CAPACITÉS DANS LES LANGAGES.

Plusieurs approches ont été proposées pour étendre les langages et leur fournir des mécanismes de contrôle des accès tout en assurant l'indépendance des modules utilisant ces accès. Pour cela, un module doit pouvoir définir, d'une part les accès dont il a besoin sur les objets, d'autre part les accès qu'il autorise depuis l'extérieur. Ceci est assuré en utilisant le mécanisme de capacité. Une capacité est un lien vers un objet qui contient des informations complémentaires, permettant le contrôle de l'utilisation du lien. En général, il s'agit des droits définissant les opérations que le propriétaire de la capacité est autorisé à effectuer sur cet objet en utilisant ce lien.

I.3.1 - LES TYPES QUALIFIES.

Jones et Liskov ont introduit la notion de type qualifié pour exprimer le contrôle des accès aux objets, ce contrôle pouvant être fait à la compilation.

I.3.1.1 - Contrôle d'accès.

Pour contrôler les accès aux objets dès la compilation on ajoute une composante à un type : un type spécifie également une liste (ensemble) de droits. Un droit est un nom qui représente un accès légal aux objets de ce type, souvent *un droit correspond à l'utilisation d'une opération du type.*

L'idée de base est que pour utiliser légalement l'une des opérations du type, un utilisateur doit posséder les pouvoirs correspondant aux droits définis pour les objets passés en paramètres à cette opération.
Un exemple est donné dans la page suivante pour le type "*Mémoire Associative*", les opérations de ce type permettent d'en :

créer un nouvel exemplaire vide de taille fixée
ajouter un couple nom-valeur à la M.A.
remplacer la valeur associée à un nom donné
supprimer un couple nom-valeur

Pour invoquer l'une de ces opérations, l'utilisateur doit présenter un pouvoir pour appliquer l'opération au paramètre M.A, dans cet exemple, le nom de droit demandé est le même que le nom de l'opération. L'opération de création retourne tous les droits pour l'objet qu'elle a crée. Ces opérations utilisent également des objets de type entier, pour simplifier un seul droit, le droit d'utiliser, contrôlera l'utilisation de toutes les opérations sur les objets entier (integer).

TYPE : Mémoire Associative
droits : "ajouter", "remplacer", "chercher", "supprimer"
opérations : créer : entrée : entier (taille de la mémoire à créer)
sortie : M.A
droits :"ajouter", "remplacer", "chercher", "supprimer"
sont données
ajouter : entrée : M.A ; droit "ajouter"
entier ; droit d'utiliser (le nom)
entier : droit d'utiliser (la valeur)
effet : (ajouter modifie son paramètre "M.A")
remplacer : entrée : M.A ; droit "remplacer"
entier ; droit d'utiliser (le nom)
entier : droit d'utiliser (la nouvelle valeur)
effet : (remplacer modifie son paramètre "M.A")
chercher : entrée : M.A ; droit "chercher"
entier ; droit d'utiliser (le nom)
sortie : entier ; droit d'utiliser (la valeur)
supprimer : entrée : M.A.; droit "supprimer"
entier ; droit d'utiliser (le nom)
effet : (supprimer modifie son paramètre "M.A")

I.3.1.2. Notations et règles.

L'approche du contrôle d'accès est basée sur un modèle sémantique dans lequel les objets sont partagés en variables, chaque objet a un type qui détermine les accès légaux à l'objet. Toute variable est déclarée d'un type qualifié $Q = T(r_1 \ldots r_n)$ où T définit le type proprement dit (noté base (Q)) et $(r_1 \ldots r_n)$ les droits (notés droits (Q)) qui sont utilisables à travers la variable sur l'objet qu'elle repère.
Donc :

$$\text{base } (Q) = T \text{ et droit } (Q) = (r_1 \ldots r_n)$$

Exemple :

M.A {chercher}
M.A {ajouter, remplacer}
M.A {ajouter, remplacer, chercher, supprimer} = M.A {all}

sont des types qualifiés dérivés du type de base M.A.
Les types Q sont utilisés dans les déclarations des variables et dans la spécification des paramètres formels d'une procédure.

Un exemple de la *déclaration d'une variable* est la suivante :

V : M.A {ajouter, remplacer}

Le sens de cette déclaration est : V est une variable qui est utilisable pour référencer un objet M.A, avec les seuls droits "ajouter" et "remplacer" Une variable est considérée comme un couple (id-objet, T.Q). id-objet est un nom unique interprété par le mécanisme d'adressage pour sélectionner un objet unique. Quand une variable est crée, son T.Q (type qualifié) définit le type et les droits sur l'objet.
Son T.Q ne peut être modifié, mais l'objet nommé par une variable (via l'id-objet) peut être changé par une opération de liaison (binding). De ce fait, deux variables peuvent contenir le nom d'objet et donc se partager celui-ci (ces variables sont comme les variables pointeurs typés et l'opération de liaison est une opération d'assignation de pointeur).
Par ailleurs, une capacité est un lien vers un objet qui contient des informations complémentaires, permettant le contrôle de l'utilisation du lien.

En général, il s'agit des droits définissant les opérations que le propriétaire de la capacité est autorisé à effectuer sur cet objet utilisant le lien.

Une variable est une capacité au sens des systèmes.

Intuitivement, les restrictions d'utilisation d'un objet sont exprimées le long du chemin d'accès. Ainsi, en employant un chemin plutôt qu'un autre pour désigner un objet, on change les droits d'accès à cet objet, par exemple :

a : M A {chercher, ajouter}

b : M A {chercher}

tous les deux désignent le même objet. En utilisant b, il est impossible de modifier cet objet, puisque la seule opération permise sur cet objet est *chercher* ; par contre en utilisant a, l'objet peut être modifié par application de l'opération *ajouter*.
L'établissement d'une liaison est en fait la création d'un nouveau chemin d'accès à un objet, et on doit s'assurer que les droits obtenus par ce nouveau chemin ne sont pas augmentés.

Soient Q_1 et Q_2 deux T.Q. on dira $Q_1 \geq Q_2$ si et seulement si

1°) - Base (Q_1) = base (Q_2) c'est à dire Q_1 et Q_2 ont la même base ou encore ils sont dérivés du même type

et

2°) - droits (Q_1) $\supseteq$ droit (Q_2)

Une première règle de liaison peut alors être définie :

soient V : T_V c'est à dire T_V étant le T.Q de la variable V

E : T_E c'est à dire T_E étant le T.Q de l'expression E

V ← E est valide si $T_E \geq T_V$ (voir l'exemple) (1)

On peut alors accéder par l'expression E au même objet que pour la variable V mais avec les droits (E).

On note que cette règle assure qu'une variable repère un objet dont le type est le type de base du T.Q de cette variable.
La forme d'une expression détermine ses T.Q.
Une expression peut être

1°) - une variable

2°) - un appel de procédure.

Exemple pour (1) (une variable)

Considérons deux variables a et b

a : MA{chercher, ajouter}

b : MA{chercher}

On peut représenter ceci par

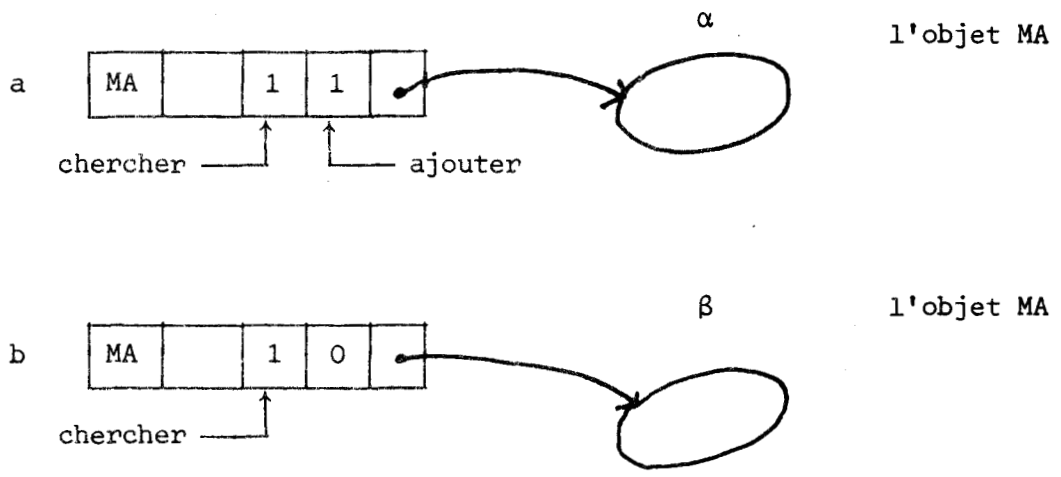


Figure 1 - L'état initial.

* b ← a est valide

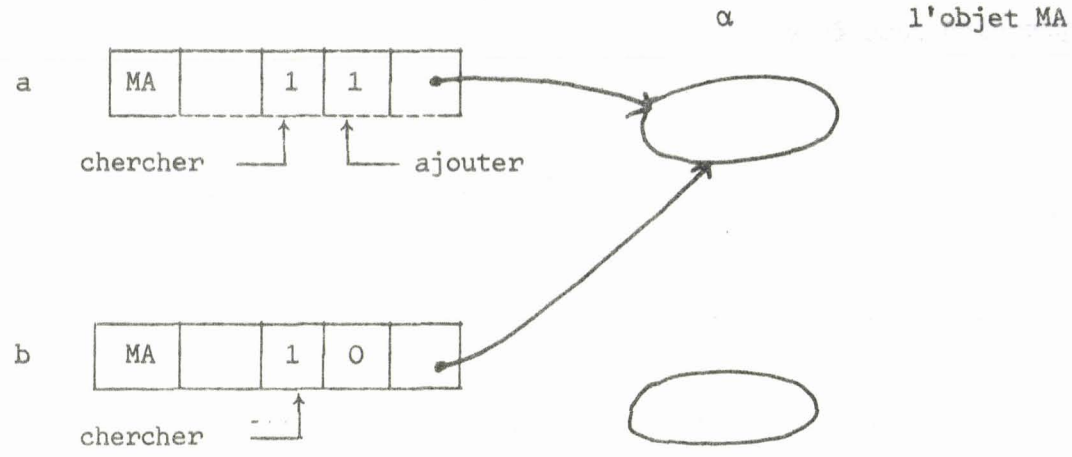


Figure 2 - montre le résultat de b ← a.

a et b représentent le même objet MAα.

Un nouveau chemin d'accès (de b à α) est crée comme résultat de cette liaison, mais il n'y a pas de nouveaux droits. En réalité le chemin d'accès via b possède moins de droits que l'ancien chemin d'accès.

* a ← b n'est pas valide

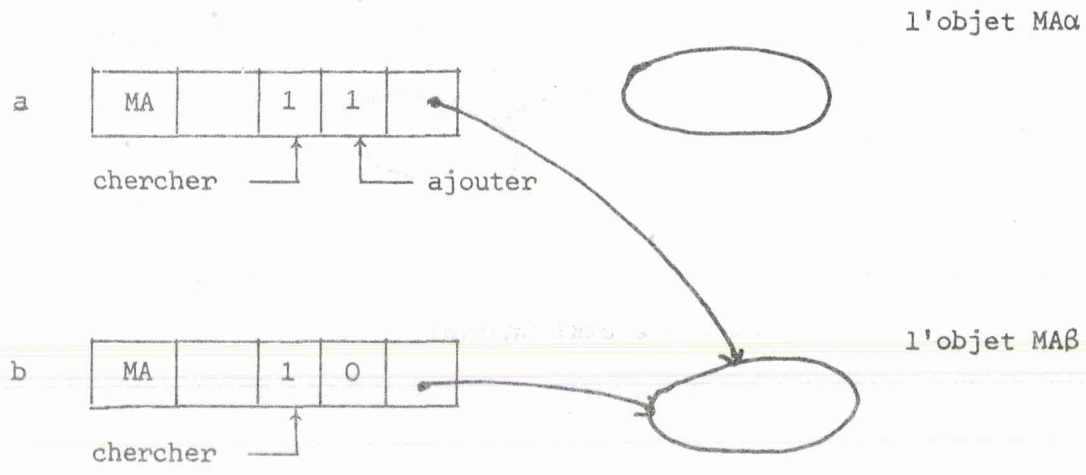


Figure 3 - montre que a ← b n'est pas valide.

Si cette liaison était réalisée, le nouveau chemin d'accès de a à β aura plus de droits que l'ancienne, c'est pourquoi cette liaison est *interdite*.

- Un exemple de la *spécification des paramètres formels d'une procédure*.

Quand l'expression est un appel de procédure, le T.Q de l'expression est déterminé par la définition de la procédure.
Une définition d'une procédure a la forme suivante :

```
procédure <nom de procédure> (<spécifications formelles>)
    returns <spécification résultat>
    <corps>
end <nom de procédure>
```

où <spécifications formelles> spécifient le nom et le T.Q pour chaque paramètre formel, et <spécification résultat> spécifie le T.Q retourné par la procédure. Le T.Q de l'appel de l'expression est le type spécifié dans <spécification résultat>.
La validité d'un appel de procédure peut être contrôléeen connaissant seulement la spécification des paramètres formels et du résultat.
Puisque tout paramètre formel est considéré comme une variable locale de la procédure, cette variable est créée à chaque appel. Les accès aux paramètres effectifs sont limités à ceux définis pour les paramètres formels donc l'appel de procédure est valide si les liaisons entre paramètres effectifs et formels sont valides.
Par exemple :

```
procédure P(x : T1 {f,g}) returns T2{K}
        a : T1 {f, g, h} ;
        b : T2 {K} ;
        c : T1 {f, h} ;
        d : T2 {K, l}
```

l'instruction b ← P(a) est valide car l'appel P(a) est valide (x ← a est valide), et T.Q de l'appel de l'expression est $T_2(K)$ et par conséquent la liaison à b est valide.
L'instruction b ← P(c) n'est pas valide car x ← c n'est pas valide.
L'instruction d ← P(a) n'est pas valide car le T.Q de d n'est pas ≤ au T.Q du résultat de P.

En résumé [Jon 76] présente un modèle de protection statique où des droits sont associés à chaque identification d'objet. Les droits, en nombre fini, sont déclarés dans le type abstrait et à chacun d'eux est associée une sémantique. A chacun des paramètres du type considéré, d'un opérateur d'accès est associée une liste (éventuellement vide) de tels droits. Pour que le compilateur autorise l'appel d'un opérateur, les identificateurs des objets passés en paramètres doivent avoir été dotés des droits correspondants.

Considérons l'exemple du type abstrait FICHIER :

deux droits sont nécessaires pour exprimer la protection

LEC : autorise la lecture
ECR : autorise l'écriture.

Dans le modèle proposé apr JONES, le type fichier s'écrit ainsi :

```
TYPE FICHIER
   Droits LEC, ECR ;
          procédure CREER FICHIER,
          procédure LIRE (F : FICHIER (LEC)),
          procédure ECRIRE (F : FICHIER (ECR)),
          procédure FUSION (F1 : FICHIER (LEC, ECR), F2 : FICHIER (LEC)),
          procédure MELANGE (F1 : FICHIER (LEC, ECR), F2 : FICHIER (LEC, ECR))
```

avec FUSION fusionne dans F1 les fichiers F1 et F2 et MELANGE modifie F1 et F2 selon certaine loi.
Ainsi avec les déclarations

```
F1 : FICHIER (LEC, ECR) ;
F2 : FICHIER (LEC) ;
```

Le compilateur interdit les écritures

MELANGE (F1,F2) et F1 := F2

et autorise

FUSION (F1,F2) et F2 := F1

(Ainsi si F1 et F2 désignant le même objet suivant l'identificateur utilisé, les droits d'accès à l'objet ne sont pas les mêmes).

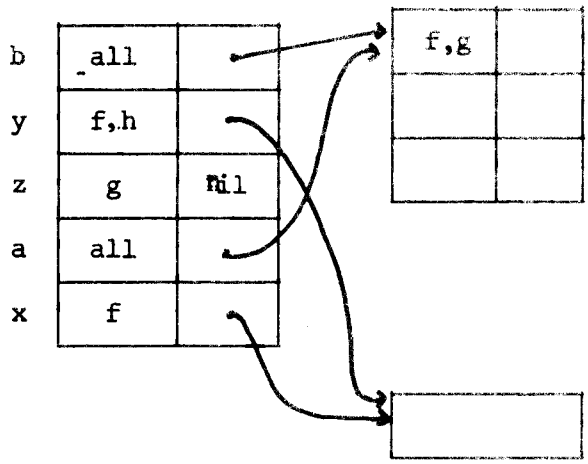
I.3.1.3 - Partage des objets structurés.

Les objets structurés posent des problèmes qui nécissitent une extension de règle exposée précédemment. Pour simplifier, nous considérons la structuration en tableau à une dimension d'objets de même type qualifié.

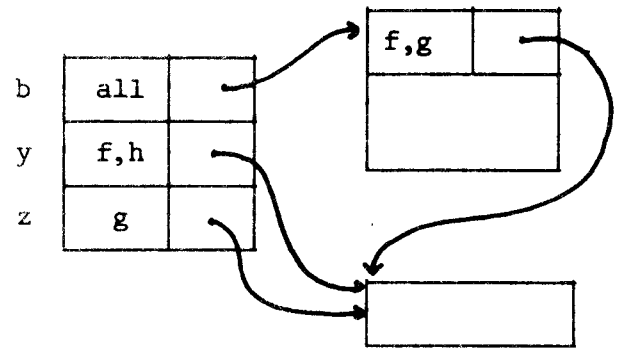
Considérons l'exemple suivant :

```
PROC P (a : array [T{f}]{all}, x : T {f})
    UP-date (a, 1, x)
 fin P

PROC Q (b : array[ T {f, g}] {all}, y : T{f, h})
     z : T{g}
     P(b, y)
     z ← fetch (b, 1)
 fin Q
```

Deux problèmes se posent. Le premier est relatif aux procédures d'accès aux éléments du tableau : ici, UP-date relie le premier élément de a à l'objet repéré par x et fetch retourne l'objet relié au premier élément de b : ces procédures seraient à définir pour tous les types qualifiés possibles. Le second se déduit de la figure suivante, qui montre que l'on acquiert le droit sur l'objet repéré par y, et donc que l'appel de P par Q est invalide.



avant appel de P



après appel de Q

L'appel P ne doit pas être autorisé

Pour résoudre ce problème, les auteurs introduisent le ?type pour indiquer que la base, ou les droits, ne sont pas complètement connus.
Un tel type est associé à un paramètre formel de la procédure, et la spécification complète du type sera déterminée pour chaque appel.

Plaçons-nous dans le contexte des déclarations :

```
Proc H (a : array [?R] {all}, b : array [?S] all}, c : S) returns R
        where R ≥ T {f, g}, S ≥ T  {f,g} ;
      x : array [T{f,g,h}]{all}
      y : array [T {f,g}]{all}
      u : T{f,g,h}
      v : T{f,g}
```

Examinons l'instruction u ← H(x,y,v). Tout d'abord, une analyse des ?type est faite, en fixant R = T{f,g,h} et S = T{f,g} ce qui est conforme aux contraintes spécifiées après where. Les ?types étant ainsi complètement spécifiés on peut contrôler l'appel comme précédemment, ce qui montre que l'instruction est valide puisque le type de v ≥ S et le type de u ≤ R.
Notons que u ← H(y,x,v) n'est pas valide car il entraîne R = T{f,g} et S = T{f,g,h}, et qu'alors le type de v n'est plus ≥ S et le type de u n'est plus ≤ R.

Pour faire en sorte que la validité du coprs de procédure soit indépendante des appels eux-mêmes ; il y a lieu de définir les règles de liaison qui font intervenir de ?types : ainsi dans le contexte x : Q_1 et y : Q_2, l'instruction y ← x est valide si :

1°) lorsque Q_1 et Q_2 sont structurés, alors ils sont identiques jusqu'aux droits sur la structure, c'est-à-dire $Q_1 = T[Q]\{d_1\}$ et $Q_2 = T[Q]\{d_2\}$ avec $d_1 \supset d_2$ par exemple :

Q_1 = array[T{f,g}]{all} et Q_2 = array[T{f,g}]{fetch}

ou

Q_1 = array [S]{all} et Q_2 = array [S]{fetch} où S est un ?type

2°) Lorsque Q_1 et Q_2 sont non structurés mais font intervenir un ?type alors la liaison doit être valide quel que soit l'élément pouvant représenter le ?type. Ainsi soit le ?type R ≥ T{f,g} :

i) $Q_1 = R$ et $Q_2 = R$

ii) $Q_1 = R$ et Q_2 n'est pas un ?type et $Q_2 \leq T\{f,g\}$

iii) $Q_2 = R$ et $Q_1 = T\{all\}$

Notons que ces mécanismes ne permettent que de réduire les droits. L'augmentation des droits sur un objet n'est possible qu'à l'entrée d'une procédure qui implémente une opération du type de l'objet.
Elle obtient alors l'accès à la représentation de l'objet passé en paramètre, et en ce sens, augmente ses pouvoirs.

[A. SILBERSCHATZ 81] propose d'étendre la règle de liaison citée plus haut en permettant au programmeur de pouvoir spécifier un ordre partiel "*plus dynamique*" sur les ?types utilisés. Mais ceci impose des contraintes additionnelles sur les paramètres effectifs qui doivent être contrôles de même que la possession des droits minimums.
Le contrôle de l'ordre partiel proposé doit être fait à l'endroit où la procédure est reliée à chaque appel. Avec le mécanisme original proposé plus haut chaque exemplaire d'un appel de procédure demandait que l'éditeur de lien vérifie que les droits de chaque paramètre effectif forment un surensemble des droits

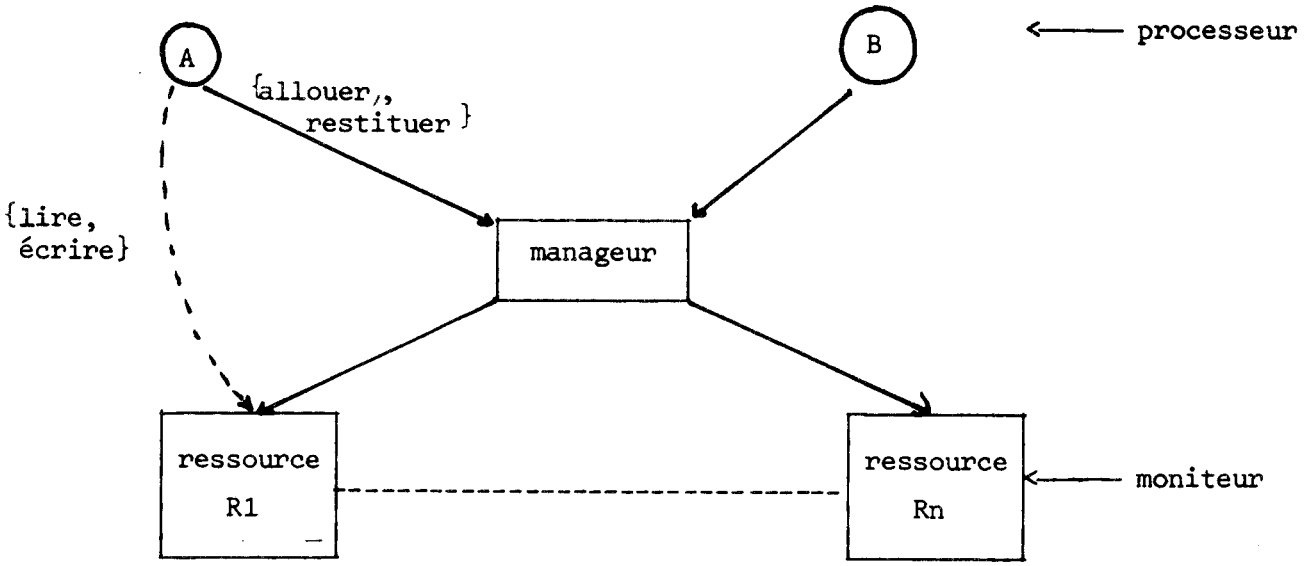
correspondant de ?type. Cette extension demande *en plus que, l'éditeur de lien vérifie que les droits des paramètres effectifs forment le même ordre partiel que celui des paramètres formels de ?type.*
Ceci n'est pas plus complexe que de tester si les paramètres effectifs reliés aux paramètres formels de ?type ont les droits adéquats.

A ce sujet on peut noter que le mécanisme de "spécification" de ADA permet de tout reporter au compilateur.

Il est important de noter que, comme dans le mécanisme original, aucun contrôle des droits d'accès n'est nécessaire à l'exécution. Si toutes les procédures sont compilées comme une unité, le compilateur exécute tout le contrôle des droits ; et si les procédures sont compilées séparément et liées entre elles, le contrôle doit être fait par l'éditeur de lien. Un contrôle à l'exécution est cependant utile, car le contrôle statique se révèle incomplet, par exemple en ce qui concerne le problème du partage des données en temps réel. Par exemple, la manipulation d'informations dans une base de données est beaucoup mieux contrôlée, si aucun des programmes qui lient la base n'est autorisé à y écrire. De plus, si les informations sont "sensible", alors la sécurité de la base sera plus grande si on peut contrôler quel programme lit quelle donnée.

I.3.2 - Les manageurs de capacité.

SILBERSCHATZ et Al. ont d'abord introduit [SIL 78] la notion de manageur, pour permettre la gestion dynamique d'une ressource (allocation, restitution) par un moniteur d'un type particulier (le manageur) sans que les accès à la ressource proprement dite aient à transiter par le manageur. La figure suivante montre les accès recherchés dans ce mécanisme. Lors de la demande d'allocation par le processus A, le manageur lui retourne un lien vers la ressource R1. Tant qu'il possède ce lien, A pourra accèder à R1 directement sans passer par le manageur.



Acquisition d'une ressource sans hiérarchie des moniteurs.

Les traits pleins représentent les *accès statiques*, les traits pointillés les *accès dynamiques* obtenus du manageur.

Le type manageur est ajouté à Concurrent Pascal et on définit un manageur comme un moniteur, mais avec l'entête :

```
type M_type = manager of T_type ref : T_type
```

Il est possible alors de déclarer un manageur particulier par :

```
var M : M_type
```

supposons qu'un autre programme contient la déclaration :

```
var T : T_type from M
```

Cette déclaration en fait ne fournit pas d'exemplaire d'objet T-type, mais spécifie son manageur M. Le programme peut alors accèder aux opérations de M à travers T, et obtenir ainsi éventuellement un lien vers un des objets T-type, gérés par M.

Ensuite l'accès aux opérations sur cet objet se fera à travers T. Il faut noter que c'est T-type-réf dans l'entête du manageur qui permet à celui-ci d'accéder explicitement à la variable T, et par conséquent de modifier dynamiquement le lien entre T et un objet T-type.

Ce mécanisme peut être étendu de telle sorte, que ce lien devienne une capacité, c'est-à-dire contienne en plus des droits d'accès à l'objet, qu'il soit statique ou dynamique. Pour cela, chaque définition de type détermine des familles de droits, en tant qu'ensemble d'opérations du type. Un manageur définira une famille avec d'une part des familles du type qu'il gère et les opérations qu'il définit. Il pourra ainsi empêcher un utilisateur d'accéder à certaines opérations du type qu'il gère.

Exemple :

```
a)  type T _ type = moniteur
b)       rights f1 = {P1, P2}
c)              f2 = {P3}
                ---- définition de P1, P2, P3 ...
d)       end monitor
e)  type M _ type = manager of T _ type _ ref : T _ type
f)       rights f3 = f1 + {P4, P5}
g)       var X array [1..N] of
h)            record --- exemplaire : T _ type (f2)
i)            end ;

         définition de P4, P5
j)  end
```

Le moniteur définit deux familles de droits f1 et f2. Le manageur en f) définit une seule famille de droit f3 en augmentant la famille f1 du type géré, et interdit ainsi l'accès par P3 aux objets qu'il gère. Il s'est par contre réservé cet accès en ligne h sur exemplaire. Un programme peut alors déclarer :

```
var T : T _ type (f3) from M
var A : T _ type (f1 + f2)
```

Par T, il pourra accéder à l'un des objets gérés par M, en utilisant les opérations P1, P2, P4, P5, mais ne pourra utliser P3 sur un tel objet. Par contre, il peut utiliser P1, P2 et P3 sur l'objet A, qui n'est pas géré par un manageur.

Bien que le problème ne soit pas mentionné par SILBERSCHATZ, le manageur doit pouvoir contrôler sélectivement les droits qu'il accorde en fonction de l'identité du demandeur. La méthode la plus simple consisterait à empêcher l'union de familles dans la déclaration d'une variable gérée par un manageur. Celui-ci peut alors définir une procédure d'établissement de lien par famille, qui accordera ou non le lien suivant l'identité du demandeur.

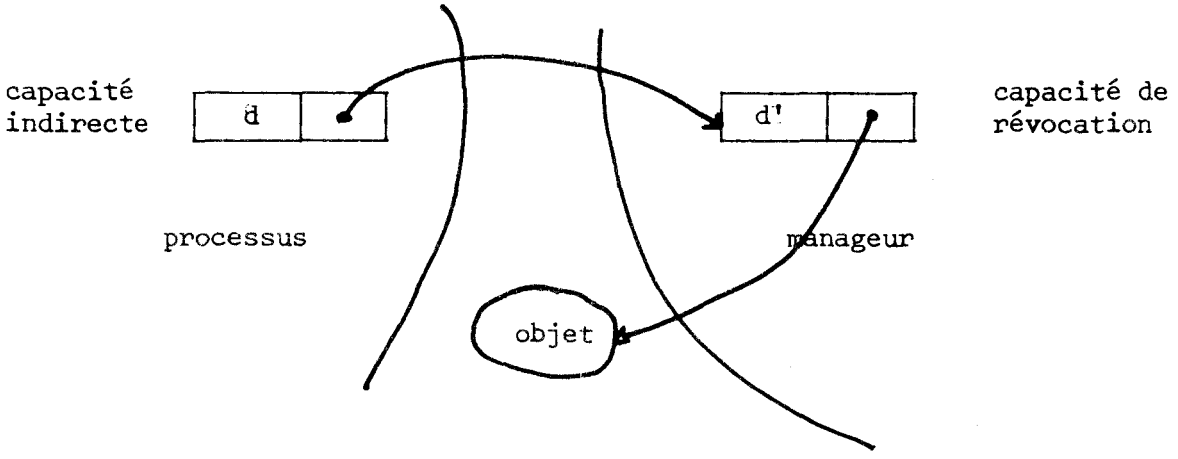
Le manageur s'exécutant en parallèle, la révocation prend ici tout son intérêt. Par exemple, s'il désire imposer qu'un processus ne puisse conserver une ressource pendant trop longtemps, il doit alors pouvoir invalider la capacité qu'il a précédemment fournie, alors qu'il ne peut accéder à cette capacité que lors des appels des usagers. Pour cela, les auteurs proposent l'utilisation d'une clé de validation. La mémorisation d'une même clé dans l'objet géré et dans la capacité transmise permet les accès tant qu'il y a égalité. La modification de la clé dans l'objet, par le manageur, invalidera alors les accès ultérieurs, et permet ainsi la révocation. Une telle révocation pose deux problèmes :

1°) La révocation ne peut être immédiate car l'utilisateur peut être en train d'exécuter une opération sur la ressource. S'il s'agit d'un moniteur l'opération de modification de la clé associée au moniteur doit être une opération implicite du moniteur. Le principe d'exclusion mutuelle d'accès au moniteur permettra la révocation à un moment opportun. S'il s'agit d'une classe, le problème n'est plus aussi simple, puisqu'il n'y a pas exclusion mutuelle. Il suffit alors d'attacher à chaque classe révocable, un moniteur implicite, appelé en tête(prologue) et en queue (épilogue) de chaque procédure de la classe et qui maintiendront un compteur d'activité.

Une révocation pour un moniteur peut entraîner une étreinte fatale. En effet un processus peut attendre un signal qui doit être émis par un autre processus, dont la capacité pour émettre ce signal a été révoquée. De plus, l'appel d'une procédure par l'intermédiaire d'une capacité révoquée donne lieu à erreur. Pour résoudre ces problèmes, toute capacité révocable contiendra une "étiquette de traitement d'exception", qui peut être fournie au manageur,

en paramètre des opérations établissant ou modifiant cette capacité. Enfin, tout moniteur révocable dispose d'une opération implicite d'éveil, qui sur ordre du manageur va parcourir toutes les files d'attente de ce moniteur, et réveiller les processus pour lesquels la capacité a été révoquée, en les orientant vers le traitement d'exception associé.

La méthode de la clé de validation proposée par SILBERSCHATZ présente l'inconvénient de ne pas être sélective, c'est à dire qu'elle agit sur toutes les capacités pour un même objet. On peut utiliser une implémentation différente, analogue à la réalisation du système [CAP-3 78]. Le manageur ne fournit plus des capacités directes pour l'objet, mais une capacité pour une autre capacité dite de révocation, qu'il conserve chez lui, et qui est reliée à l'objet. Tout accès à un objet par une capacité indirecte n'est autorisé que s'il fait partie des droits de la capacité indirecte, et de la capacité de révocation comme le montre la figure suivante.



Capacité de révocation.

Ainsi la révocation peut être sélective pour les opérations et pour les demandeurs.

Dans ce cas, il est nécessaire d'introduire dans le langage des variables du type "capacité de révocation", pouvant être manipulées par le manageur.

*

* *

CHAPITRE II

OMPHALE

II.1 - INTRODUCTION.

Notre travail se situe dans le projet OMPHALE. L'appellation OMPHALE constitue tout à la fois le nom d'un projet, celui d'un système opératoire et celui d'une machine.

Le projet OMPHALE a pour objectif la conception, la réalisation et l'évaluation d'une architecture matérielle et logicielle adaptée à la programmation modulaire et structurée. L'obtention d'une fiabilité acceptable des traitements, l'objectif final de la recherche, amène à doter le système de mécanismes de protection cablés et un "noyau" de logiciel correct chargé de gérer les calculs des utilisateurs. De plus, elle conduit à envisager l'expression même des contraintes de protection dans un langage de haut niveau. (Ce dernier point constitue une partie de notre travail ; chap. I).

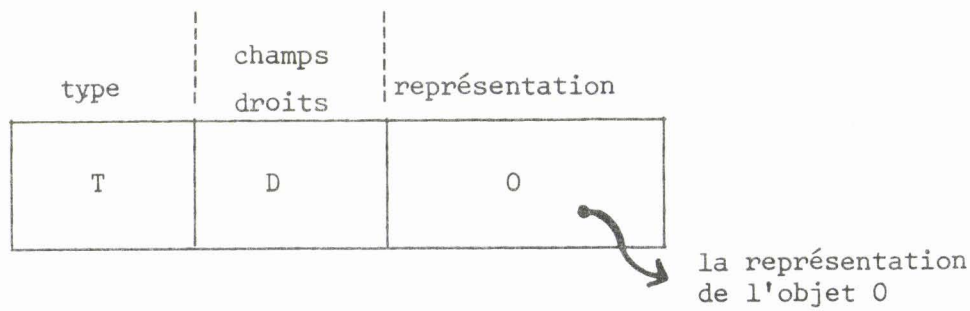
Dans ce chapitre on présente rapidement l'architecture matérielle logicielle du système. Les lecteurs intéressés peuvent consulter la thèse de [E. DELATTRE 79] pour plus de détails.

Le système OMPHALE structure en "domaine" les objets qui le composent -un système peut être considéré comme un ensemble d' "objets"- Un adressage par "capacités" relie la notion logicielle de domaine à la localisation physique des objets en mémoire. Les concepts d'objet , de capacité et de domaine proviennent de recherche portant sur la protection dans les systèmes opératoires Hydra (WULF 74), Plessy 250 ; ENGLAND 76, iAPX 432 (INTEL 81), etc... . L'étude de systèmes à capacités a fait apparaître que deux concepts de structuration basés sur les types et les domaines de protection augmentent la sécurité des systèmes opératoires. Le premier facilite une définition plus rigoureuse des objets du système, tandis que le second permet de restreindre à tout instant la visibilité aux seuls objets nécessaires à l'application d'une opération. La notion de type permet d'envisager une structuration possible du système opératoire où l'on regrouperait dans un module ou sous-système les opérations associées à un type

donné, voire les objets de ce type. Ces opérations seraient alors exécutées dans un domaine de protection. Il en résulterait une meilleure abstraction du système. Rappelons que ces notions d'abstraction sont des éléments de définition de langages modernes comme ALPHARD (WU 74), CLU (LIS 74), ADA (IC 80) ou dans des langages plus anciens comme SIMULA (DAHL 67).

II.2 - STRUCTURE DE DOMAINE.

Nous rappelons *l'adressage par capacités* introduit par **FABRY** (74) : une capacité est une sorte de pointeur constitué de trois champs : un champ "représentation" contient un nom, début de la chaîne d'accès qui mène à la représentation de l'objet en mémoire ; un champ "droits" renferme un codage des droits du possesseur de la capacité sur l'objet repéré ; un champ "type" porte le nom du type de l'objet repéré



Un objet n'est accessible qu'à partir d'une capacité qui le repère et à condition que les droits qu'elle porte soient suffisants.

L'architecture d'OMPHALE est fondée sur le concept de Domaine.

Au départ les promoteurs de cette notion voyaient en elle un moyen d'établir une "protection" des entités (appelées objets) composant le système c'est à dire la possibilité de contrôler la validité des accès aux objets. La protection repose sur la règle suivante : un agent ne peut déclencher une opération sur un objet que s'il est implicitement autorisé par la possession d'un droit (pouvoir) relatif au type de l'opération et à cet objet à un instant donné.

Un agent exécute des opérations dans un domaine de protection, il n'a accès qu'à une liste limitée d'objets sur lesquels il possède des droits. Ainsi le Domaine devient l'unité de protection du système. C'est la structure de contrôle de la validité d'une opération sur un objet et de la concordance de leur type respectif. En appliquant la règle de protection citée ci-dessus, on peut alors dire qu'un objet A ne peut être manipulé par un autre objet B que si l'objet B possède un droit sur l'objet A, droit relatif au type de l'opération désirée. Un domaine constitue alors la liste des objets manipulables (les "composantes" du domaine) accompagnée de la liste des droits correspondants.

Par ailleurs, un Domaine peut être vu comme l'unité d'exécution du système. C'est le résultat de la conservation jusqu'à l'exécution comprise de la modularité des programmes.

Il faut ajouter que tout objet appartient à une classe d'équivalence qui définit son type. Cette classe est le sous-ensemble des objets uniquement manipulables par quelques opérations constitutives du type. Le typage induit une partition de l'ensemble des objets du système.

Dès lors, un domaine matérialise le contexte d'exécution d'une opération d'un type. L'exécution d'un domaine constitue donc un moment dans le déroulement d'un processus. Un processus pourra utiliser pendant son exécution plusieurs domaines pour réaliser l'ensemble des opérations. Il est alors nécessaire de contrôler les changements de domaine. Ceci est réalisé en faisant du domaine un objet du système de type DOM, dont l'emploi est contrôlé par les mécanismes de protection (par exemple des instructions privilégiées (Appel-domaine et Retour-domaine) .

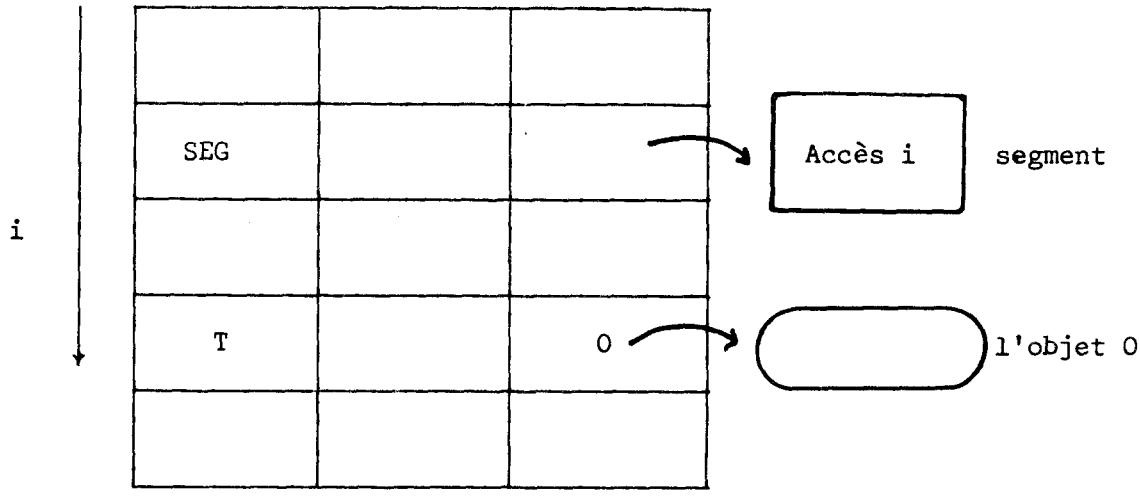
De plus, un domaine constitue, un "environnement fermé" -chaque agent du système ne possède que les droits (les capacités) qui lui ont été explicitement accordées- et dispose de toutes les informations nécessaires à son exécution (autonomie des domaines). Mais la protection est d'autant meilleure, et, par suite, la fiabilité du logiciel plus grande, que les domaines sont composés d'un nombre limité d'objets [LIND 76] : c'est à dire qu'ils renferment peu d'objets.

Alors un agent ne possédera pas plus d'objets que ceux nécessaires pour le déroulement de sa tâche immédiate. C'est le principe du "moindre privilége". L'application de ce principe se superpose à la volonté de travailler sur de programmes hautement modulaires dont la conséquence majeure est l'obtention de domaine de petite taille (hypothèse retenue pour OMPHALE).

II.3 - RÉALISATION D'UN DOMAINE.

L'adressage par capacités permet d'implémenter en une liste de capacités la structure de domaine. Un domaine sera représenté par une liste des capacités ou C-liste pour les objets qui le composent. L'index i d'une capacité dans cette liste constitue le *nom local* de l'objet repéré dans le domaine considéré.

Par ailleurs, une opération est représentée dans le système par un segment de code (ensemble ordonné de valeurs élémentaires exécutables) qui désigne les objets par des noms locaux dans le domaine dans lequel l'opération est exécutée, comme le montre la figure suivante.

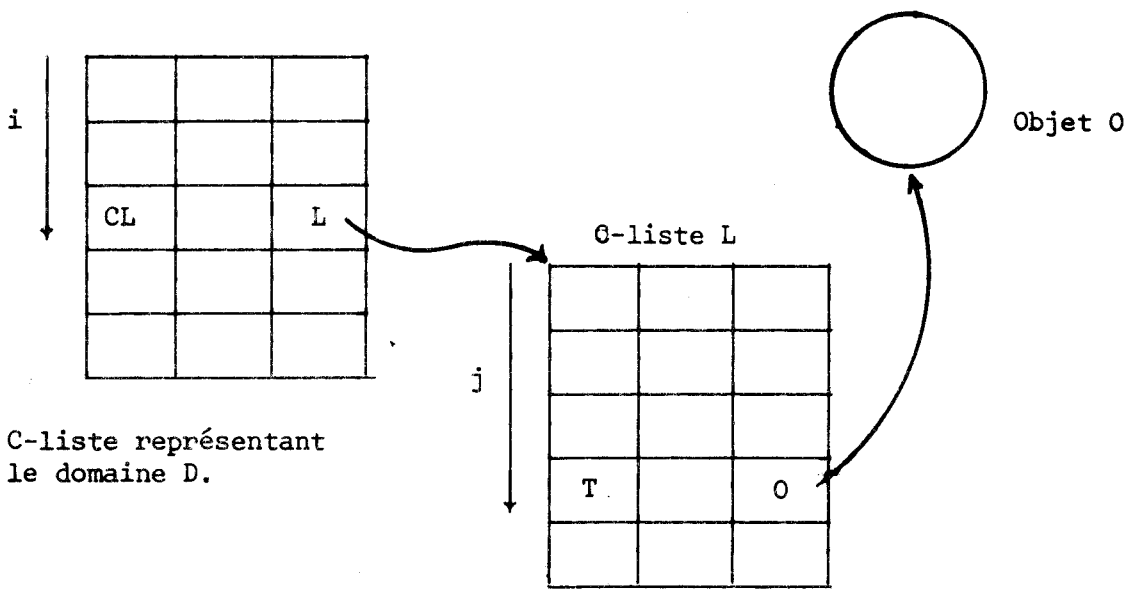


C-liste representant le domaine.

Ainsi, un domaine est matérialisé par une liste de capacités, ou "C-liste" : celle qui repère tous ses composants.

COMPLEMENTS SUR L'ADRESSAGE.

Les C-listes ne servent pas seulement à représenter des domaines. Elles sont aussi utilisées comme structures de regroupement des références à des objets. Une C-liste peut contenir une capacité pour une autre C-liste et la C-liste représentant un domaine devient la racine d'une arborescence de C-listes. Pour des raisons qui sont exposées dans [DEL 79], les noms locaux d'OMPHALE ne traitent que les deux premiers niveaux de l'arborescence, comme beaucoup de machines à capacité : un objet O peut être désigné par deux index, le premier i, étant l'index dans la C-liste D d'une C-liste L ; le second j, étant l'index dans la C-liste L d'une capacité pour l'objet, selon le schéma suivant :



Dans ce cas, le nom de l'objet O dans le domaine est noté i, j, les mécanismes d'adressage contrôlent que le champ type de la capacité désignée par i renferme le nom CL.

II.4 - EXÉCUTION D'UN DOMAINE.

Dans la plupart des systèmes, une opération est représentée par une procédure (l'objet procédure n'existe pas dans OMPHALE - voir mais cela ne remet pas en cause les résultats présentés ici). La procédure comprend un segment de code et les objets qui lui sont associés en permanence -ou objets permanents-. D'autres objets sont nécessaires à l'exécution d'une procédure (CROCUS (75)) : les objets locaux qui sont créés à chaque activation de la procédure ; les objets passés en paramètres ; et les objets rémanents, qui sont réutilisés à chaque activation de la procédure.
Ces objets constituent l'environnement d'exécution de la procédure, ils sont strictement nécessaires à son exécution. Ainsi, un domaine est confondu avec un environnement et est représenté par la C-liste des objets composant l'environnement d'exécution d'une opération d'un type.

Le noyau comprenant le matériel et le logiciel de base fournit, d'une part les opérations des types primitifs ainsi que les mécanismes associés tel que l'adressage par capacités.
Par exemple :

- n ← CREER-SEGMENT (ℓ)

crée un segment de ℓ valeurs élémentaires nulles. Une capacité pour le segment est placée à l'emplacement de n (nom local dans un domaine).

- n ← CREER-C-LISTE (c)

crée une C-liste de c emplacements nuls. Une capacité pour la C-liste est placée à l'emplacement, n.

D'autre part, les mécanismes de construction de type : les objets primitifs étant créés par le noyau, seuls les objets construits sont créés par interprétation d'une maquette, car le type d'un objet détermine le type de ses composants.

La création d'une maquette est effectuée par un domaine appelé par le compilateur. Ce domaine reçoit en paramètres un segment dont le contenu, déduit du texte-source, décrit les commandes d'interprétation, et les capacités pour les objets au moment de l'interprétation de la maquette. Ce processus de création d'une C-liste, interprétation d'une maquette et la composition ont été décrites dans [LAN 78] et [DEL 79] . Il est nécessaire de connaître l'organisation de l'objet-type (structure de regroupement de maquettes de création des objets) qui représente le type dans le système.

II.4.1 - Mise en place des domaines à partir de l'objet-type.

Dans les systèmes manipulant des procédures, chaque appel d'une procédure déclenche la création puis l'activation d'un nouvel environnement plus précisément, l'appel procédural se découle, généralement, comme suit :

1°) Première étape : création de l'environnement :

Les objets permanents de la procédure sont inclus dans l'environnement : les objets locaux sont crées, les objets rémanents et les objets paramètres sont insérés dans l'environnement.

2°) Deuxième étape : activation de l'environnement :

Le contrôle est transféré au nouvel environnement.

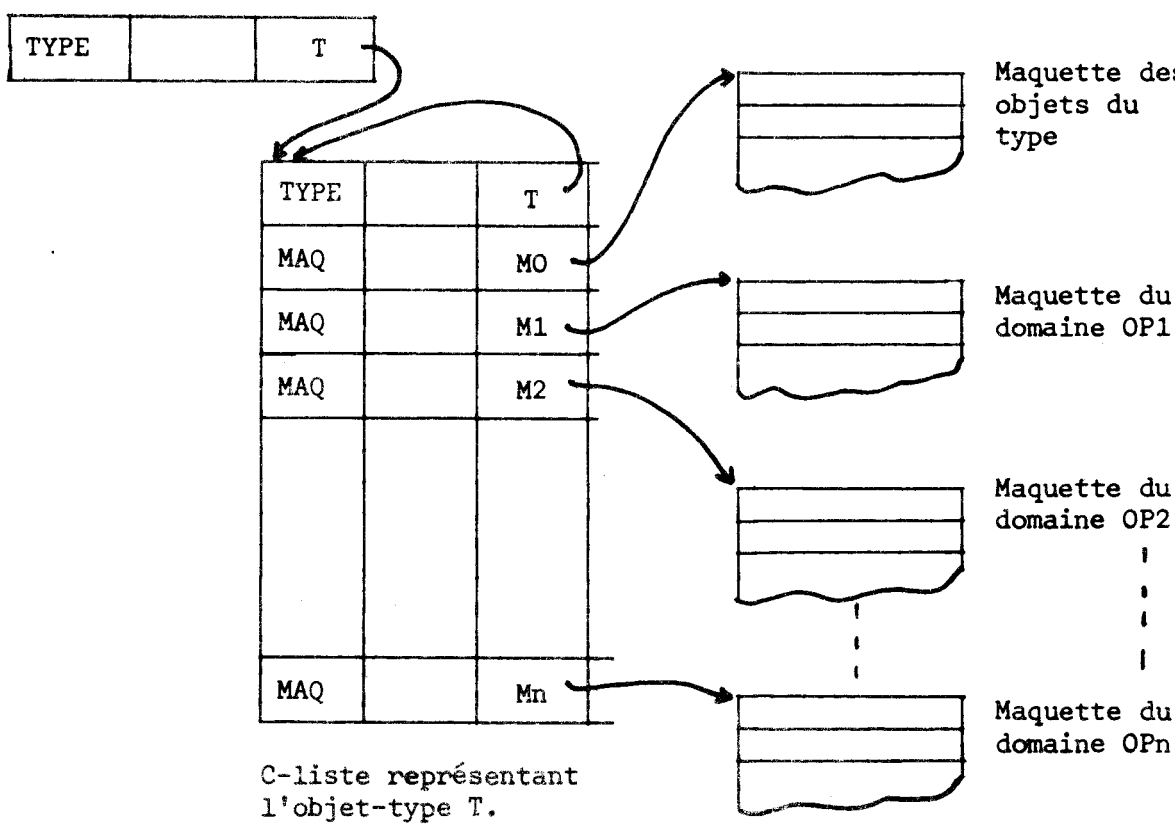
Lanciaux souligne les insuffisances d'un tel schéma [Lan (78)] ;

- l'incorporation des objets rémanents à l'environnement est souvent une opération complexe et couteuse en temps de calcul. Ainsi, par exemple, le système MULTICS [CROCUS (75)] consulte un segment de liaison pour retrouver les rémanents.

- La fourniture d'un seul appel procédural manque de généralité. Il propose de dissocier création de l'environnement et activation de l'environnement.

OMPHALE considère que l'opération de création de domaine (l'interprétation de sa maquette) est dissociée de toutes les opérations de transfert de contrôle. Les capacités pour les objets permanents d'une opération sont copiées de la maquette au domaine. Les objets locaux peuvent être crées à chaque activation du domaine ou en même temps que le domaine, au gré de l'utilisateur. Durant toute leur vie, les objets rémanents restent enfermés dans le domaine qui les a créés, ainsi ils sont facilement retrouvés à chaque activation du domaine qui les supporte.

Par conséquent, la structure des objets du type T est représentée par la maquette qui permet de les créer par interprétation. Il est également possible de représenter les opérations OP_1, OP_2, ..., OP_n du type T par des maquettes ; un domaine D_i réalise l'environnement d'exécution d'une opération OP_i ; comme la création et l'activation d'un domaine sont dissociées, l'opération OP_i peut être représentée par une maquette du domaine D_i. Dès lors, l'objet-type peut être vu comme un objet construit composé de n+1 maquettes : une maquette par opération du type, plus une maquette des objets du type. Par convention, la première capacité dans la C-liste représentant l'objet-type repère l'objet-type lui-même, la seconde capacité désigne la maquette des objets du type. Ainsi, l'objet-type possède la structure suivante :



C-liste représentant l'objet-type T.

Les objets permanents d'une opération OPi sont alors représentés par des capacités figurant dans la maquette du domaine D_i sous forme de descripteurs munis de la commande de copie. Une capacité de type SEG pour un segment de code ou une capacité de type TYPE permettant de décomposer les objets du type T entrent dans cette catégorie. La maquette permet ainsi de se passer d'objet procédure. Afin que les capacités repérant les objets permanents ne puissent pas être diffusées à l'extérieur de domaine D_i, elles ne possèdent pas les droits de copie et de transfert. De même, la capacité de type CL résultant d'une décomposition est privée des droits C et T, pour éviter une divulgation de la structure d'un objet du type hors du domaine D_i qui le manipule. Les descripteurs munis de la commande de création et inclus dans la maquette du domaine D_i correspondant soit à des objets locaux, soit à des objets rémanents, suivant qu'un nouveau domaine est crée à chaque appel de l'opération OPi, ou que le même domaine D_i est réactivité. Les descripteurs portant la commande d'annulation sont vraissemblablement associés à de futurs objets paramètres.

Enfin, l'interprétation de l'une des maquettes délivre, soit un objet de type, soit un domaine. Ce dernier réalise l'environnement d'exécution de l'une des opérations du type ce que nous allons voir dans le paragraphe suivant.

II.4.2 - Opérations de création.

L'opération de création des objets joue un rôle central dans le système : non seulement, elle est bien évidemment utilisée pour créer les objets mais, de plus, elle participe à la mise en place des domaines. Dans les deux cas, il s'agit :

- de créer une liste de capacités nulles
- d'interpréter la maquette
- de créer une capacité soit de type T, soit de type DOM, par composition.

Ceci se réalise par des opérations fournies par le noyau par exemple.

• La création d'un objet de type T s'effectue par l'opération

c <- ∿ CREER_OBJET (t, p)

t désigne une capacité pour l'objet-type T.
p désigne une capacité sur la C-liste des paramètres d'interprétation de la maquette.
La capacité résultante est placée en c.

- Par contre *l'opération de création de domaine* s'écrit :

d <- ∿CREER_DOMAINE (O, i, P)

O désigne une capacité pour un objet de type T
i désigne un numéro d'opération.

Ici, la i+1ème maquette de l'objet-type T, correspondant à l'opération OPi, est interprétée. La capacité pour l'objet-type DOM qui permettra de créer une capacité de type DOM figure dans le domaine même qui réalisera l'opération de création et n'a pas besoin d'être placée dans l'objet-type T. La capacité résultante est transférée à l'emplacement d.

- *L'opération d'interprétation de la maquette* s'écrit :

nc <-∿ INTERPRETER-MAQUETTE (T, n, P)

T désigne le nom unique de l'objet à créer
n désigne le numéro de maquette
nc désigne le nom local de l'emplacement destiné à la capacité pour l'objet nouvellement crée.
L'exécution de l'opération

c <- ∿ CREER-OBJET (t, P)

déclenche l'extraction de T dans la capacité pour l'objet-type de nom local t, l'appel

nd <- ∿ INTERPRETER-MAQUETTE (T, 1, P)

et l'exécution de

c <- ∿ CREER-DOMAINE (O, i, P)

provoque l'obtention du nom unique T à partir de la capacité pour l'objet $O_{(T)}$ de nom local O et l'activation

n< - ∿ INTERPRETER-MAQUETTE (DOM, i+1, P)

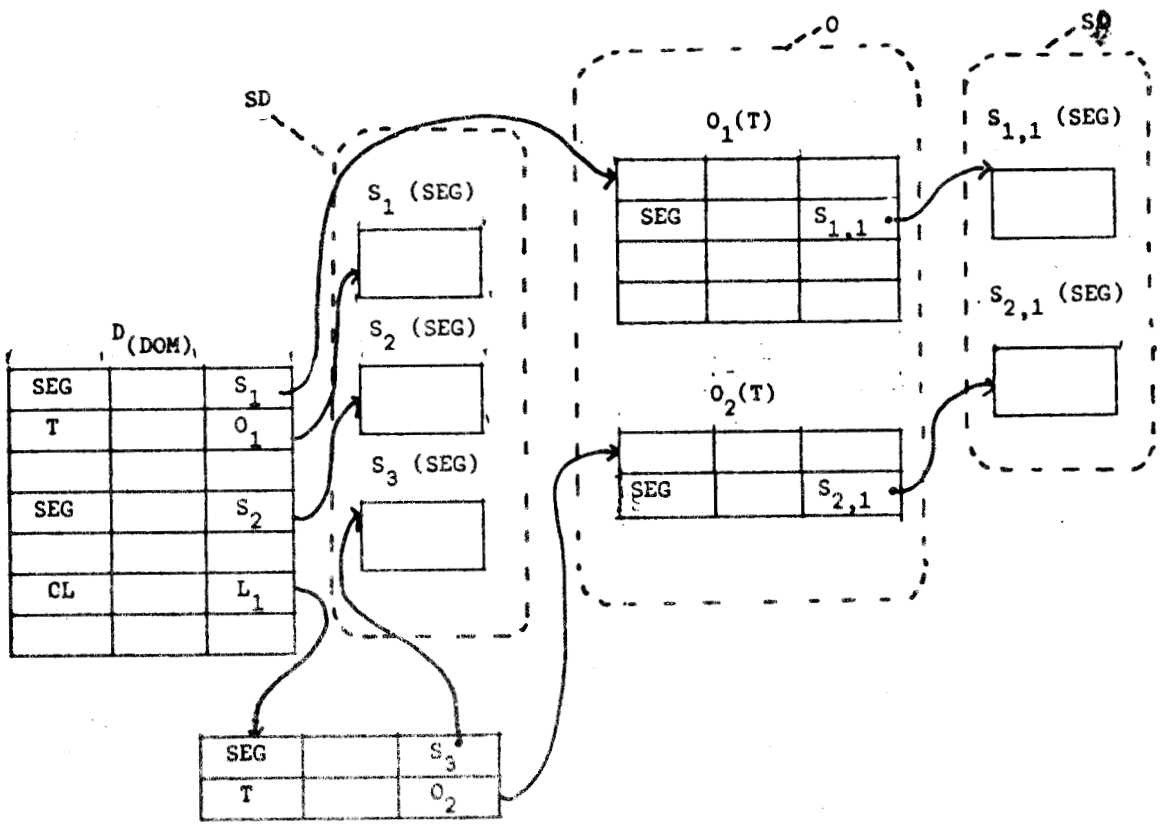
crée une capacité de type DOM de nom local nc.

II.4.3 - Les objets d'un domaine.

Soit un domaine D réalisant une opéraiton du type T. L'ensemble des objets du domaine D est constitué :

- du domaine D lui-même
- du sous-ensemble O des objets $O_i(T)$ de type T référencés dans D
- du sous-ensemble SD des segments S_j référencés dans D
- du sous-ensemble SO des segments $S_{i,k}$ comportant des objets $O_i(T)$ référencés dans D.

L'ensemble des segments du domaine D est l'union SD U SO.
Le schéma suivant visualise ces ensembles



Au cours de son exécution, un domaine ne peut manipuler que les segments SD U SO, la C-liste qui le représente et les C-listes, soit référencées dans cette première C-liste, soit constituant les représentations des objets de l'ensemble O, ces dernières à condition que les capacités aient été décomposées. Ces limitations proviennent de ce qu'un domaine réalisant une opération d'un type T ne peut décomposer que les objets du type T. En fait l'ensemble des objets d'un domaine doit être défini plus largement. Il est des cas où un domaine est autorisé à manipuler des objets de plusieurs types différents. Par exemple, un domaine manipulant des objets du type T1 construit par récursivité croisée. $-T_1$ est construit sur T_2, qui est, lui-même, construit sur T_1. peut accéder à la fois aux objets de type T_1 et aux objets de type T_2. Aussi, le sous-ensemble O doit être défini comme regroupant les objets O_i (T_j) référencés dans D, les types T_j étant les types manipulables par D.

Pour assurer une protection effective l'architecture du système OMPHALE doit respecter les deux règles suivantes :

1°) l'intégrité des capacités doit être assurée : il faut veiller à ce qu'aucune capacité ne puisse être créée de toutes pièces par un usager ; il faut éviter l'altération arbitraire d'un champ d'une capacité par un programme malveillant.

2°) Il faut assurer qu'à un moment donné les seuls objets physiquement accessibles soient les composants du domaine en cours d'exécution (voir plus haut).

Pour cette dernière règle, la notion d'objet "accessible" doit être précisée, ce que nous ferons dans le chap IV. Mais, tout d'abord, le seul type d'accès qu'un processeur est capable d'effectuer est, soit une lecture, soit une écriture en mémoire. Bien plus, toute opération, aussi complexe soit-elle se réduit, en définitve, à des lectures / écritures en mémoire. Or, il serait inefficace de contrôler individuellement la validité de chaque accès à chaque mot mémoire. Ce sont les segments (type primitif) qui sont protégés. Comme les accès aux segments sont effectués par le matériel, ils doivent être contrôlés par le matériel. Les opérations qui définissent le type primitif segment sont :

- écriture/lecture d'un mot
- création/destruction d'un segment
- exécution d'un segment de code.

Un autre type primitif est la C-liste. Une des opérations définissant ce type est la COPIE d'une capacité d'une C-liste à une autre C-liste.
Ces deux types primitifs (segment et C-liste) servent de base à la "construction" de tous les autres types : les opérations d'un type T1 manipulent un certain nombre de segments et de C-listes ; de même pour les opérations d'un type T2 : puis les opérations d'un type T3 utilisent des objets des types T1 et T2 via leurs opérations plus d'autres segments et C-listes ; etc...

Quant à l'opération elle-même, elle est matérialisée par un segment de code, composant du domaine qui réalise le contexte de l'opération. Cependant, un domaine réalisant une opération de type T n'a pas besoin de toute l'arborescence représentative de la structure d'un objet de type T, mais seulement de :

- la C-liste matérialisant le domaine
- les segments et C-liste propres au domaine et, en particulier, le segment de code matérialisant l'opération effectuée
- le ou les C-liste(s), et, éventuellement, le ou les segment(s) matérialisant les objets de type T composants du domaine.

A uninstant donné l'espace d'adressage du processeur doit être limité strictement à cette liste d'objets.

Finalement, une opération du type T se réduit à :

- des lectures/écritures dans le segment du domaine, des manipulations de capacités entre les C-listes du domaine
- et des appels à des domaines traitant des objets "moins construits" ("plus élémentaire").

II.5 - OMPHALE-SYSTÈME RÉPARTI.

A la différence de HYDRA ou du système 250, qui sont implémentés sur des multiprocesseurs dotés d'une mémoire commune partagée, OMPHALE est un multiordinateur chaque ordinateur, appelé "site", comprend un ou plusieurs processeurs, une mémoire locale (dans le sens où elle n'est pas accessible à des processeurs de sites différents) et des périphériques. Les sites communiquent par messages via un sous-système de communication (SSC).

La plupart des domaines peuvent s'exécuter sur n'importe quelle site et sont mobiles d'un site à un autre (les objets sont aussi, individuellement mobiles). Le choix du site d'exécution d'un domaine donné est guidé par la recherche d'un compromis, appelé "régulation de la charge", entre deux objectifs opposés : d'un côté, équilibrer les charges individuelles des sites, afin d'améliorer le temps de réponse ; de l'autre, minimiser la charge du SSC, afin de ne pas le saturer.

II.6 - CARACTÉRISTIQUES DEMANDÉES À UN LANGAGE DE HAUT NIVEAU POUR OMPHALE.

Après avoir rappelé brièvement les concepts essentiels d'OMPHALE, nous pensons qu'un langage de haut niveau pour OMPHALE doit se caractériser par les points suivants :

- un domaine étant considéré comme l'unité d'exécution du système. C'est le résultat de la conservation jusqu'à l'exécution comprise de la modularité des programmes.

  Le langage doit s'adapter à cette notion de modularité à l'exécution.

- Des mécanismes de protection basés essentiellement sur l'adressage par capacité, assurent que les objets ne sont pas accessibles à l'extérieur de domaines autorisés à les manipuler. Cela rend nécessaire la définition de la notion de "droit" dans le langage en terme d'opérations applicables sur les objets.

- Le système est constitué d'objets fortement typés. Un type est représenté par un objet-type. Ce concept de l'objet-type est le même que celui de type abstrait de donnée dans les langages modernes.

Enfin le mécanisme de protection dans OMPHALE est un mécanisme de contrôle à l'exécution destiné à être implanté au niveau de la machine. Ceci doit être complété par la définition d'une stratégie de protection dans un langage supportant la notion de types abstraits, et les autres caractéristiques cités ci-dessus.

Notre choix a été orienté vers ADA, tout en sachant que du point de vue de la protection, ce langage tel qu'il est défini dans [MR (80)] se comporte plutôt mieux que bien d'autres langage et qu 'il est encore loin des propositions de JONES et LISKOV (Chap. I), [CARREZ (82)].

CHAPITRE III

ADA ET LA PROTECTION

III.1 - INTRODUCTION

L'étude d'OMPHALE (chap. 2) a orienté notre choix vers le langage ADA en particulier pour les raisons suivantes :

. **langage algorithmique** : définition de types, structures de contrôle modernes ;

. **langage modulaire** : définition de modules (packages), compilation séparée des unités de programme, paramétrisation (unités génériques) ;

. **langage temp réel** : traitement du parallélisme (tâches), entrées/sorties, contrôle de temps, traitement des exceptions ;

. **langages d'écriture de système** : spécification des représentations.

En outre, ADA présente des caractéristiques dont certaines figurent dans plusieurs langages proposés et conçus dans un passé récent (le langage LIS, Ichbiah (76) qui élabore le concept de modularisation de SIMULA et reprend une structure voisine de PASCAL. De plus ADA, certainement, a été influencé aussi par d'autres langages comme CLU et ALPHARD.

Il possède d'autres caractéristiques : fiabilité, facilité de modification, indépendance vis-à-vis des machines, facilité de mise en oeuvre et de construction d'utilitaires.

Enfin, d'un point de vue pratique, ADA possède de bonnes chances d'être largement diffusé comme produit dans un avenir très proche, et on possède déjà une documentation importante sur le langage et son environnement.

Notre choix se portant donc sur ADA, nous étudions dans ce chapitre comment ce langage répond aux problèmes de protection évoqués au chap. I.

III.2 - RAPPELS SUR LE LANGAGE.

Doit-on le rappeler, ADA n'est pas un langage fait pour résoudre les problèmes de protection ; mais par le mécanisme de modularité qu'il offre (paquetage, type privé et limité etc...) il participe à résoudre certains de ces problèmes. En particulier, le langage permet des vérifications sur le programme dès la compilation ; mais rappelons d'abord quelques caractéristiques du langage ADA ;

* Tous les objets définis dans un programme ADA sont typés et ce langage est fortement typé, dans le sens où tout identificateur utilisé dans un programme doit être défini par une déclaration. Cette déclaration impose des restrictions dans la manière dont un identificateur peut être utilisé ; ceci peut être contrôlé à la compilation. En l'absence de toute autre déclaration, les seules opérations applicables sont l'affectation et toutes les opérations de comparaison.

* La sémantique du langage fait que seuls le type et les opérations associées sont connus à l'extérieur d'un paquetage. La réalisation des opérations elle-même ainsi que les types et les données qui leur sont nécessaires ne sont pas visibles à l'extérieur.

* Ces protections sont assurées par le compilateur. Deux autres niveaux supplémentaires peuvent être en jeu à l'aide des types privés et limités. Ils permettent de cacher la structure même des objets de ces types à l'extérieur du paquetage ; nous avons ainsi un véritable type abstrait de données.

* le langage est modulaire, c'est-à-dire qu'un programme est composé de modules (unités de bibliothèque ou sous-unités) et il admet la conception descendente comme technique de développement du programme. Cette conception descendente appelée "modularité verticale" dans [LIN (76)] se caractérise par les points suivants :

- plusieurs opérations d'un module peuvent être appelées depuis un autre module. C'est le cas d'un paquetage en ADA.

- des données d'un module peuvent être préservées entre les appels successifs (problèmes des variables rémanentes). C'est le cas des objets déclarés dans le corps d'un paquetage.

- les interactions entre modules sont définies explicitement et contrôlées ; en particulier, la représentation des objets d'un module n'est pas directement accessible depuis les autres modules.

  En ADA, les interactions sont définies par les données et sous-programmes déclarés dans un paquetage et donc contrôlés par le compilateur.

III.3 - MÉCANISMES DE ADA UTILISABLES POUR LA PROTECTION.

III.3.1 - Unités de bibliothèque.

Les unités de bibliothèque (sous-programmes ou paquetage) sont développées d'une manière indépendante. Elles sont réalisées en deux parties, une partie spécification visible par un utilisateur de bibliothèque, une partie réalisation qui est entièrement cachée. Un programme utilisateur (en particulier un système) peut être construit en utilisant ces unités de bibliothèque.

L'interface avec une unité de bibliothèque est parfaitement définie à l'aide de la partie spécification et peut donc être vérifiée voire contrôlée, complètement dès la compilation. Elle dépend de la nature de l'unité de bibliothèque :

- si l'unité de bibliothèque est un sous-programme : la structure de blocs et les règles de visibilité font que le programme utilisateur n'a pas accès aux objets et aux types déclarés dans le sous-programme La communication entre les deux se fait par appel au sous-programme avec passage de paramètres correspondants dont les types sont vérifiés lors de la compilation.

- si l'unité de bibliothèque est un paquetage : le programme utilisateur n'a accès qu'aux seuls objets et types déclarés dans la partie publique du paquetage. Le paquetage n'a aucun accès aux objets du programme utilisateur. La communication entre les deux se fait par appel aux sous-programme déclarés dans la partie publique du paquetage et passage des paramètres correspondants à ces sous-programmes. Les types de paramètres sont également vérifiés lors de la compilation. La communication peut aussi se faire à l'aide

des objets déclarés dans la partie publique du paquetage.

Ainsi ADA apporte des solutions au problème de la méfiance mutuelle cité dans le chapitre I. La coopération entre sous-systèmes (modules) mutuellement méfiant exige :

- l'isolation réciproque des sous-systèmes : celle-ci est réalisée en ADA grâce aux unités de bibliothèque.

- L'accès au sous-système en des points spécifiés : c'est effectivement le cas lorsque l'unité de bibliothèque est un sous-programme ou lorsqu'elle est un paquetage puisque seuls les sous-programmes de la partie publique d'un paquetage sont visibles.

- Le contrôle des paramètres reçu par le sous-système : tous les types des paramètres sont contrôlés dès la compilation.

- La possibilité pour un sous-système de restreindre les droits d'accès aux objets qu'il transmet en paramètres à d'autres sous-systèmes : ceci n'est que partiellement possible en ADA ; deux solutions sont possibles.

I.3.1.1 - Utilisation des types privés et limités privés.

1°) - Principe et utilisation.

Un paquetage possédant la déclaration de tels types a tous les droits d'accès sur les objets du type. Les autres paquetages ayant des objets de ces types n'ont pas accès à leur représentation. Les seules opérations autorisées sur ces objets sont celles mises à leur disposition par le paquetage propriétaire du type, ainsi que, pour les objets du type privé non limité, les opérations d'affectation et de comparaison (égalité, inégalité). Pour illustrer l'utilisation de ces types nous prendrons l'exemple d'un système de gestion de mots de passe dans lequel non seulement la structure interne des mots de passe ne sera pas communiquée à l'utilisateur mais encore on lui interdit toute opération autre que celles qui sont nécessaires.
Avant de présenter l'exemple, on rappelle que la partie déclaration du corps du paquetage peut contenir en outre des déclarations de variables locales au paquetage. Ces variables forment les objets *rémanents*, c'est à dire qu'elles gardent leurs valeurs entre deux appels à une procédure du paquetage. Ces variables ne sont pas accessibles de l'extérieur du paquetage, elles ne le

sont qu'à partir des procédures du paquetage.

Exemple.

Nous constituons un paquetage de gestion de ressources (par exemple des fichiers). L'utilisateur dispose de trois opérations possibles (OP1, OP2, OP3 ... qui donneront lieu à trois procédures) :

OP1 : NOUVELUT

immatriculation d'un nouvel utilisateur.
Celui-ci fournit le mot de passe qui sera répertorié (et qu'il devra fournir à toute utilisation du système) et reçoit un numéro d'utilisateur.

OP2 : GETRESSOURCE :

l'utilisateur se fait attribuer une ressource.
Il fournit : - son numéro d'utilisateur,
- son mot de passe
et il reçoit : un nom de ressource.

OP3 : ACCESRESSOURCE

pour utiliser une ressource qu'il s'est fait attribuer,
l'utilisateur indique :
- son numéro
- son mot de passe
- le nom de la ressource.

Biensûr, tout desaccord entre numéro et mot de passe sera sanctionné.

```
Package GESTION-RESSOURCES is
type  PASSWORD is string (1 .. 10) ; -- type visible
type  RESSOURCE is limited private ;
function NOUVELUT              (MOT-PASSE : in PASSWORD) return INTEGER ;
Procédure GETRESSOURCE  (NUT : in INTEGER ; MOT-PASS : in  PASSWORD
                                    NOMR ; out RESSOURCE) ;
Procédure ACCESRESSOURCE  (NUT : in INTEGER ; MOT-PASS : in PASSWORD
                                    NOMR : in RESSOURCE) ;
Private
        type RESSOURCE is
           record
              NOM : string (1..10) ;
              NOMBRE : INTEGER ;
           end record ;
end GESTION-RESSOURCES ;


Package body GESTION-RESSOURCE is
        MAX-UT : constant INTEGER := 100 ;    -- nombre Maxi d'utilisateur
        MAX-RES : constant INTEGER R := 10 ; -- maxi de ressource par utilisateur
        NB-UT : INTEGER ;                     -- nombre d'utilisateur immatriculé
        type UTIL is
           record
              MOT : PASSWORD                  -- mot de passe
              NBRES : 1..  MAX-RES ;          -- nombre de ressource attribué
           end record ;
        UTILIS : array (1..MAX-UT) of UTIL ;
        RESS : array (1..MAX-UT, 1..MAX-RES) of RESSOURCE ;
```

Toutes ces informations internes au corps du paquetage, et donc inconnues à l'extérieur, vont préciser la gestion qui va être faite. Il faut que ce soit inaccessible à l'utilisateur, sinon, il pourrait tricher : avoir plus de ressources que le maximum permis, accéder aux mots de passe des autres etc ...

Enfin, voici l'esquisse des procédures :

```
Function NOUVELUT  (MOT-PASS : in PASSWORD) is
    begin
          NB-UT := NB-UT + 1 :
          if NB-UT > MAX-UT then raise exception voulue ;
          end if ,

          UTILIS (NB-UT). MOT := MOT-PASS ;
          UTILIS (NB-UT). NBRES := 0 ;
          return NB-UT
    end NOUVELUT  ;

procedure  GETRESSOURCE (NUT : in INTEGER : MOT-PASS : in PASSWORD ;
                         NOMR : out RESSOURCE) is
          NR :1 .. MAX-RES ;
    begin
          if NUT not in UTILIS range then raise erreur accès ;
          end if :        -- pas le bon numéro
          if UTILIS (NUT). MOT ≠ MOT-PASS then raise erreur accès ;
          end if ;        -- pas le bon mot de passe
          NR := UTILIS (NUT). NBRES +1 ;
          NOMR := fabrication de fichier
          UTILIS (NUT).NBRES := NR ;
          RESS (NUT, NR) := NOMR  :
    end  GETRESSOURCE ;

procedure  ACCESRESSOURCE (----------) is
    begin
          For I in 1..UTILIS (NUT).NBRES loop
                if RESS (NUT, 1) = NOMR then ---

          ---
           |
           |
           |
           |

end  ACCESRESSOURCE ;
end GESTION-RESSOURCE ;
```

Une utilisation pourra se présenter de la façon suivante :

```
procédure USAGER is
      USE GESTION-RESSOURCE ;
      MES RESSOURCES : array (1..5) of RESSOURCE ;
begin
         mon code : INTEGER := NOUVELUT (      ) ;
         GETRESSOURCE  (       ) ;
         ACCESRESSOURCE  (      ) ;
end USAGER ;
```

2°) - Critiques et propositions [CARREZ (82)]

Considérons l'exemple suivant :

```
with P2 ;
package P1 is
---
end P1 ;
```

L'introduction d'un paquetage P1 dans la portée de P2 pose le problème de la protection des objets de P2 vis à vis des actions de P1. Si celle-ci est totale vis à vis des objets internes à P2 (ils ne seront pas importés), elle est existante pour ceux de la partie visible (sauf s'ils sont d'un type privé). Si P2 doit avoir une certaine méfiance vis à vis de P1, il faudra déclarer les objets sensibles dans le corps de P2 et définir les procédures de consultation de ces objets. Il aurait été intéressant de pouvoir définir dans un paquetage des objets "semi-constants", c'est à dire qui sont des constants à l'extérieur du paquetage et des variables dans le corps. Cela permettrait aussi d'éviter les modifications non synchronisées de ces variables dans un environnement multi-tâches.

L'absence d'affectation et d'égalité sur les objets de type privé limité à l'extérieur du paquetage entraine l'absence de ces opérations sur tous les objets construits avec ce type comme composant, puisque l'égalité (par exemple) sur les objets construits vérifie l'égalité sur les composants.

L'exemple suivant montre l'utilisation possible de tels objets construits :

```
Package P is
     type POSITION is limited private ;
     type FILE is
          record
               F-POS : POSITION ;
               STATUS : INTEGER ;
          end record ;
          ----
end P :
```

A l'extérieur de P il est possible de déclarer des objets de type FILE. Le composant POS de ces objets ne peut être consulté ou modifié que dans le corps de P. Par contre le composant STATUS peut être consulté ou modifié à l'extérieur de P. Remarquons ici encore que la possibilité de définir des types "semis-constant" (l'affectation n'est pas disponible pour les objets de ce type à l'extérieur du paquetage) permettant la consultation de composants sans autoriser leur modification.

Rappelons que l'égalité peut être redéfinie pour les types privés limités ainsi que pour les types construits à partir d'eux (ce sont les seuls cas de redéfinition de l'égalité) [MR 67 ADA (80)]. Notons cependant que la redéfinition de l'égalité pour un type n'entraine pas la redéfinition implicite pour les types construits à partir de lui. Ainsi la redéfinition de l'égalité pour POSITION de l'exemple cité au-dessus n'entrainerait pas celle de FILE.

On peut donc reprocher à ADA l'absence totale de *sélectivité* dans la protection des objets d'un paquetage : le programmeur qui désire une protection sélective doit augmenter considérablement le nombre de sous-programmes de son paquetage.

I.3.1.2 - Utilisation des types dérivés.

Rappelons que les propriétés essentielles des types dérivés en ADA sont :

- les types dérivés sont de la même classe de type que celle dont il dérive
- l'ensemble des valeurs possibles est une copie des valeurs possibles du type père

- les opérations applicables sur le type père (c'est à dire sous-programmes ayant un paramètre ou un résultat du type père) sont hérités par le type dérivé. [And (79)] propose que la révocation des droits acquis soit implantée au moyen de mécanisme de types étendus proposé par les langages modernes tels SIMULA, ALPHARD, CLU et ADA.

En effet, la déclaration d'un nouveau type en ADA produit toujours un type distinct. Le nouveau type hérite des opérations applicables sur le type de base. Si le nouveau type est un type composé il hérite des opérations applicables sur les types de base de ses composants.

Le mécanisme proposé est basé sur le fait que les types abstraits permettent de définir un type avec des opérations associées. A l'aide des types dérivés, il est possible ensuite d'étendre à volonté les opérations associées. Ces opérations peuvent être distribuées à des utilisateurs potentiels en leur donnant accès ou non aux unités de bibliothèques réalisant ces opérations.

Cette possibilité est malheureusement statique. La révocation de droits obtenus par un utilisateur, n'est possible que lors d'une nouvelle exécution du programme si l'édition de lien est dynamique ce qui est le cas dans OMPHALE. Par contre, il est probablement possible de faire une certaine révocation dynamique des droits ainsi qu'une amplification dynamique des droits à l'aide des types dérivés sans toutefois pouvoir résoudre le problème de courrier dans la prison (chapitre I).

- L'expéditeur et le destinataire disposent de droits sur les lettres en particulier celui de les lire.

- Le facteur ne doit pas avoir accès à leur contenu.

Il y a donc révocation de droit de lecture entre l'expéditeur et le facteur et puis amplification entre le facteur et le destinataire. Cette solution bien qu'elle soit beaucoup plus progressive et nuancée, ne permet pas d'interdire la lecture.

III.3.2 - Les sous-unités.

Une *sous-unité* est le corps d'un sous-programme, le corps d'un paquetage ou le corps d'une tâche.

Le développement du programme en sous-unités est de nature hiérarchique sous forme d'arbre. Les règles de visibilité et de portée des identificateurs en ADA permettent des vérifications, dès la compilation, de leurs utilisations dans le texte du programme. Typiquement, une sous-unité ne peut pas référencer des objets ou types d'une sous-unité non visible.

Par ailleurs, ADA fournit un outil de compilation séparée des sous-unités composant le programme, unité de bibliothèque et sous-unités. Ce dernier point (la compilation séparée des sous-unités) est fort critiquable [CAR (82)] : En effet comme toute unité de compilation, une sous-unité A d'une unité mère B peut être précédée de clause with. L'unité C (ou les unités) mentionnée dans la clause with est ajoutée au paquetage standard, puis le contexte de la souche de A dans B est pris en compte. Tout se passe comme si B avait fait l'inclusion de C. Il est alors possible que B masque l'identificateur C qui n'est donc pas visible directement dans A. Il sera visible par STANDARD.C sous réserve que B n'ait pas aussi masqué STANDARD (dans ce cas il est entièrement inaccessible de A !). Ainsi, la compilation séparée d'une sous-suite doit rester pour le programmeur une facilité d'exploitation et non une technique de "modularisation". Notons enfin que toute déclaration dans l'unité englobante peut masquer les identificateurs introduits par une clause use attachée à la sous-unité et accroître ainsi la confusion du programmeur.

III.3.3 - Contrôle du passage des paramètres.

Le contrôle du passage des paramètres peut avoir des conséquences assez importantes pour la protection. En effet, ADA fournit trois modes de passage des paramètres :

- mode in (paramètre-donnée) : dans ce cas le paramètre ne peut pas être modifié dans le sous-programme. Les fonctions utilisent obligatoirement ce mode.

. mode out (paramètre-résultat) : le paramètre n'a pas de valeur à l'entrée et il est modifié dans le corps de la procédure

. mode in out (à la fois donnée et résultat) : tout est possible.

Le mode de passage est défini dans la spécification par le programmeur. Dans la version actuelle du langage, rien ne permet de contrôler le mode de passage lors d'un appel de sous-programme, excepté la possibilité d'imposer le mode in en fournissant une expression comme paramètre effectif. [CAR 82] propose de rajouter ce qui a été initialement prévue dans la première version du langage [ADA 79] :

:= mode in attendu
:=: mode in out attendu
=: mode out attendu

III.4 - AUTRES PROBLÈMES DE PROTECTION.

* Le problème du "cheval de Troie" est résolu, en partie en ADA de la façon suivante :

. l'isolation réciproque des sous-système est assez bien résolue grâce aux unités de bibliothèque, qui peuvent limiter l'accès aux seuls objets transmis en paramètres

. Les opérations permises sur les objets passés en paramètres sont obligatoirement celles définies par le type de ces objets, par contre ADA ne permet pas d'assurer une sélectivité de ces opérations.

* Le problème de l'étanchéité (le sous-système ne doit pas mémoriser ou divulguer l'information qui lui est transmise en paramètre) n'est pas résolu par ADA.

III.5 - DEUX SOLUTIONS ADA AU PROBLÈME DU COURRIER DANS UNE PRISON.

Nous avons présenté au chapitre I le problème du courrier dans une prison ; bien que la solution en CLU proposée puisse être traduite en ADA, nous proposons ici deux autres solutions exploitant les possibilités spécifiques offertes par ADA. La première est basée sur les notions de "sous-type" et de "renommage" ; la seconde utilise les propriétés des discriminants, associés aux types limités privés.

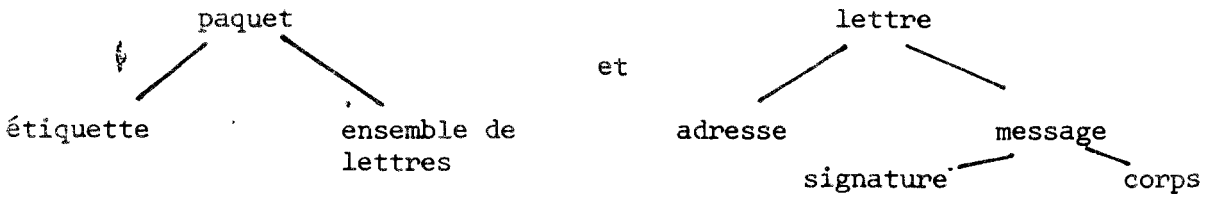
Rappelons tout d'abord quelques hypothèses :

- le facteur est non-partisan ; cependant on lui interdit de lire le contenu des lettres.

- Le tri et la distribution du courrier se font correctement le facteur et les prisonniers ne devront donc pas vérifier les étiquettes de paquets, ni les adresses de lettres.

- On suppose que les prisonniers et le facteur peuvent créer (et détruire) des paquets vides.

La structure de donnée prend la forme de l'arbre suivant :



Nous supposons pour simplifier que :

. l'identification d'un prisonnier ou du facteur est définie par un type discret TIDP du paquetage STANDARD que nous n'expliquons pas ici.

. Le corps du message est une chaîne de caractères.

Les structures des données sont définies par des spécifications de paquetage de bibliothèque qui peuvent être rendues accessibles d'un agent par le biais de clause with. Ces clauses devront être mises en tête des unités de compilation, nous supposons qu'à ce niveau, un contrôle est effectué sur les autorisations d'accès à ces spécifications, ce contrôle étant fait par l'environnement de programmation par exemple, ou par le chef de projet.

TCL définit le contenu d'une lettre
TLETTRE définit une lettre
TCP définit le contenu d'un paquet
définit un paquet.

Les actions autorisées sur ces structures de données sont :

pour le gardien

connaître le destinataire d'un paquet	DESTINATAIRE

pour le facteur

ouvrir les paquets	OUVRIR
ferme les paquets	FERMER
extraire une lettre d'un paquet	EXTRAIRE
savoir s'il y a des lettres dans un paquet	NONVIDE
ajouter une lettre à un paquet	AJOUTER
connaître le destinataire d'une lettre	DESTINATAIRE

Pour les prisonniers

En plus des opérations permises au facteur, accéder au type TCL pour lire et écrire les lettres	
fermer une lettre	FERMER
ouvrir une lettre	OUVRIR

III.5.1 - Première solution.

La première solution consiste à rassembler dans un premier paquetage POUR-PRISONNIER l'ensemble des types et des oéprations, et à définir deux paquetages POUR-FACTEUR et POUR -GARDIEN qui redéfinissent le sous-ensemble de types et opérations accessibles respectivement au facteur et au gardien par des clauses de sous-types et de renommage. Ainsi, le paquetage FACTEUR n'est aurotisé à importer que le paquetage POUR-FACTEUR qui lui fournit les opérations permises et elles seules. En particulier il n'a pas accès à la repré-

sentation des lettres car TLETTRE est limité privé, ni à leur contenu car il ne connaît pas TCL, et enfin ne peut les ouvrir, car il ne dispose pas de l'opération. Par contre, sa connaissance du type TLETTRE et du type contenu des paquets (TCP) lié avec les opérations EXTRAIRE, AJOUTER, lui permet de transférer les lettres d'un paquet à l'autre.

Nous ne donnons ici que les spécifications des paquetages, et renvoyons en annexe A, la solution complète, y compris la réalisation du corps de ces paquetages.

```
package POUR_PRISONNIER is
    type TCL is record
        MESSAGE:string  ;
        SIGNATURE:TIDP  ;
    end ;
    type  TLETTRE is limited private ;
    procedure FERMER  (CL:TCL;D:TIDP;L:out TLETTRE);
    procedure OUVRIR  (L:TLETTRE;CL:out TCL);
    function DESTINATAIRE (L:TLETTRE) return TIDP;
    type TCP is private;
    AUCUNE:constant TCP;
    function NONVIDE(CP:TCP) return BOOLEAN ;
    procedure EXTRAIRE(CP:in out TCP;L:out TLETTRE) ;
    procedure AJOUTER (CP:in out TCP;L:    TLETTRE) ;

    type TPP is private;
    AUCUN : constant TPP;
    procedure FERMER(CP:in out TCP;D:TIDP;P:out TPP) ;
    procedure OUVRIR(P:in out TPP;CP:out TCP) ;
    function DESTINATAIRE (P:TPP) return TIDP;
private
    type TCP is access TLETTRE ;
    type TLETTRE is record
        DEST : TIDP;
        CONTENU : TCL ;
        SUIVANTE: TCP ;
    end ;
    AUCUNE : constant TCP:= null ;
    type TP is record
        DEST : TIDP;
        CONTENU: TCP ;
    end ;
    type TPP is access TP;
    AUCUN : constant TPP :=null;
end ;
```

```
with POUR_PRISONNIER;
package POUR_FACTEUR is
    subtype TLETTRE is POUR_PRISONNIER.TLETTRE;
    function DESTINATAIRE (L:TLETTRE) return TIDP renames
             POUR_PRISONNIER.DESTINATAIRE ;
    subtype TCP is POUR_PRISONNIER.TCP;
    AUCUNE :TCP renames POUR_PRISONNIER.AUCUNE;
    function NONVIDE(CP:TCP) return BOOLEAN ;
    procedure EXTRAIRE(CP:in out TCP;L:out TLETTRE) renames
             POUR_PRISONNIER.EXTRAIRE;
    procedure AJOUTER (CP:in out TCP;L:    TLETTRE) renames
             POUR_PRISONNIER.AJOUTER ;

    subtype TPP is POUR_PRISONNIER.TPP;
    AUCUN: TPP renames POUR_PRISONNIER.AUCUN;
    procedure FERMER(CP:in out TCP;D:TIDP;P:out TPP) renames
             POUR_PRISONNIER.FERMER ;
    procedure OUVRIR(P:in out TPP;CP:out TCP) renames
             POUR_PRISONNIER.OUVRIR ;
end;


with POUR_PRISONNIER;
package POUR_GARDIEN is
    subtype TPP is POUR_PRISONNIER.TPP;
    AUCUN : TPP renames POUR_PRISONNIER.AUCUN;
    function DESTINATAIRE (P:TPP) return TIDP renames
             POUR_PRISONNIER.DESTINATAIRE ;
end;

with POUR_GARDIEN    ;
package GARDIEN is
   procedure ENVOYER ( P : POUR_GARDIEN.TPP ) ;
   procedure RECEVOIR ( D :TIDP ; P : out POUR_GARDIEN.TPP ) ;
end ;

with POUR_FACTEUR,GARDIEN ;
package FACTEUR is end ;

with POUR_PRISONNIER,GARDIEN ;
package PRISONNIER is end ;
```

III.5.2 - Seconde solution.

La deuxième solution consiste à éclater les types et opérations dans différents paquetages, et à limiter les accès à ces différents paquetages suivant les besoins.
On trouve ainsi :

- un paquetage CONTENU-LETTRE qui définit le type TCL comme publique, et dont l'accès par with est supposé interdit au facteur et au gardien,

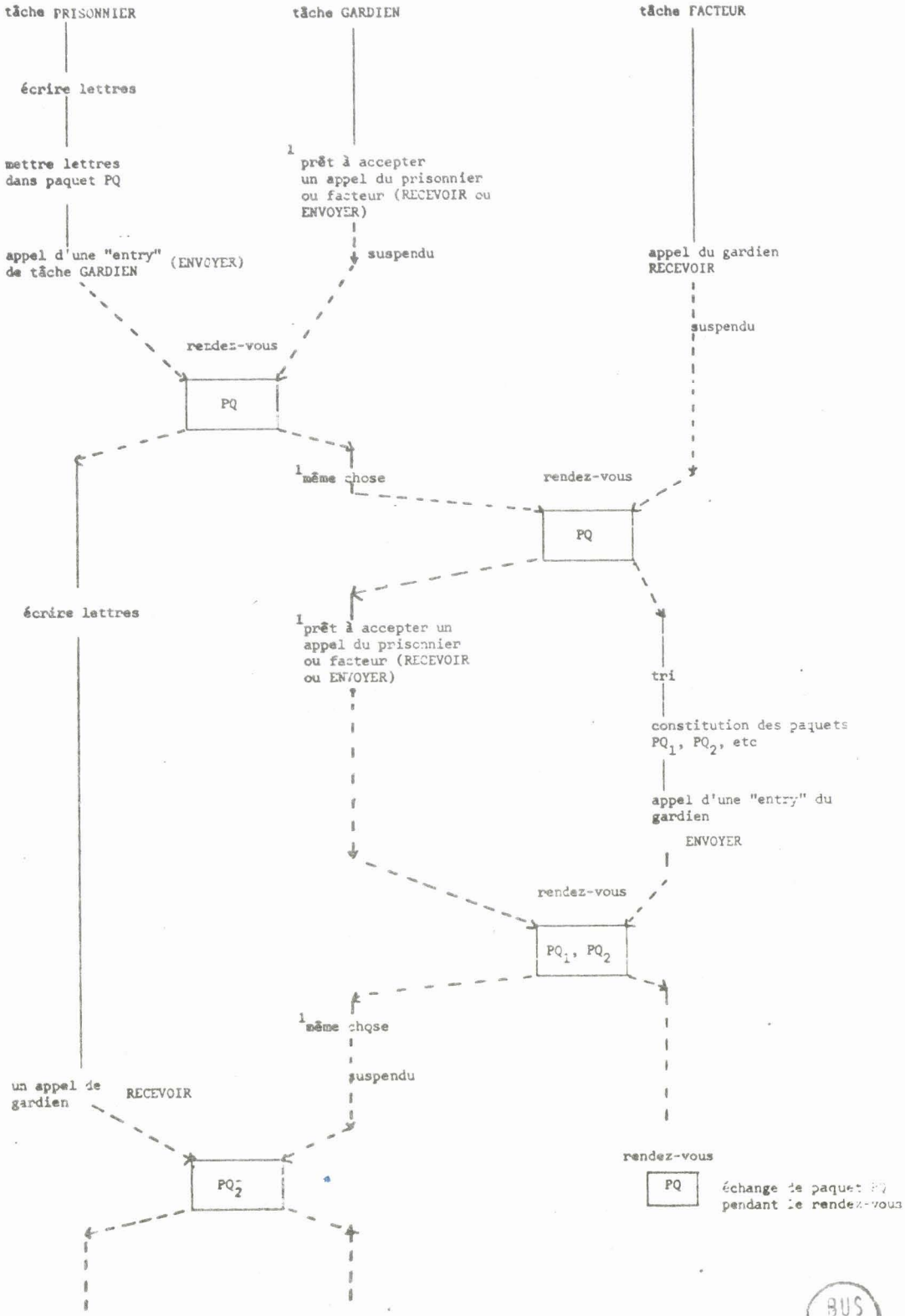
- un paquetage LETTRES qui définit le type TLETTRE limité privé, avec ses opérations OUVRIR et FERMER ainsi que le type TCP simplement privé, avec ses opérations NONVIDE, EXTRAIRE, AJOUTER. Ce paquetage est supposé interdit par with au gardien. On peut noter ici que les procédures OUVRIR et FERMER sur les lettres, bien qu'accessibles du facteur ne sont pas utilisables par lui, car il ne connait pas le type TCL et ne peut donc fournir de paramètre de ce type,

- un paquetage PAQUET qui définit le type TPP et ses opérations OUVRIR, FERMER. Ce paquetage est supposé accessible à tous, mais le gardien ne pourra utiliser ces deux opérations car il ne peut fournir de paramètre du type LETTRES.TCP.

Remarquons que les fonctions DESTINATAIRE pour les lettres ou les paquets n'apparaissent plus dans cette solution. En effet, nous utilisons ici des types limités privés avec discriminant pour identifier le destinataire. La valeur de ce discriminant peut donc être consultée à l'extérieur du paquetage. Comme un discriminant ne peut être modifié que par affectation globale à la structure à laquelle il appartient, et que de plus ces structures sont de type limité privé (cette affectation globale est interdite à l'extérieur des paquetages), la valeur de ces discriminants est donc protégée en écriture à l'extérieur. Notons que ces discriminants ont ainsi la caractéristique de "semi-constant" [CAR 82] à l'exportation.

Nous ne donnons ici que les spécifications des paquetages, et renvoyons en annexe B, la solution complète. Nous remarquons que les réalisations des corps des paquetages sont pratiquement les mêmes que pour la première solution.

```
package CONTENU_LETTRE is
    type TCL is record
       MESSAGE:string  ;
        SIGNATURE:TIDP  ;
    end ;
end ;


with CONTENU_LETTRE;
package LETTRES is
    type  TLETTRE(DEST:TIDP) is limited private ;
    procedure FERMER  (CL:CONTENU_LETTRE.TCL;D:TIDP;L:out TLETTRE);
    procedure OUVRIR  (L:TLETTRE;CL:out CONTENU_LETTRE.TCL);
    type TCP is private;
    AUCUNE:constant TCP;
    function NONVIDE(CP:TCP) return BOOLEAN ;
    procedure EXTRAIRE(CP:in out TCP;L:out TLETTRE) ;
    procedure AJOUTER (CP:in out TCP;L:    TLETTRE) ;
 private
    type TCP is access TLETTRE ;
    type TLETTRE (DEST : TIDP) is record

       CONTENU : CONTENU_LETTRE.TCL ;
        SUIVANTE: TCP ;
    end ;
    AUCUNE : constant TCP:= null ;
end ;

with LETTRES ;
package PAQUET is
   type TP (DEST:TIDP) is limited private ;
   type TPP is access TP ;
   procedure FERMER(CP:in out LETTRES.TCP;D:TIDP;P:out TPP) ;
   procedure OUVRIR(P:in out TPP;CP:out LETTRES.TCP) ;
 private
   type TP (DEST : TIDP) is record
      CONTENU: LETTRES.TCP ;
   end ;
end ;

with PAQUET   ;
package GARDIEN is
   procedure ENVOYER ( P : PAQUET.TPP ) ;
   procedure RECEVOIR ( D :TIDP ; P : out PAQUET.TPP ) ;
end ;

package FACTEUR is end ;

with PAQUET,LETTRES,CONTENU_LETTRE,GARDIEN ;
package PRISONNIER is end ;
```



Exemple de communication.

III.5.3 - Remarque.

Toutes les solutions (sauf CLU) proposées par Ambler (cf. Chap I) prennent la forme d'un programme principal séquentiel.

Par contre, dans les deux solutions que nous proposons, chaque "acteur" peut être associé à une tâche (procédure) indépendante, les différentes tâches se déroulant en parallèle. L'échange des paquets, c'est à dire la synchronisation et la communication peut être réalisée d'une manière très simple à l'aide du mécanisme de rendez-vous (cf. Schéma). Si cet aspect paraît étrangers aux problèmes de protection, il met par contre en évidence la présence des trois agents ayant des pouvoirs différents sur des objets qu'ils se communiquent.
D'une manière générale, pour définir un système sûr, qui assure la protection, il faut être en mesure de faire varier le "pouvoir d'un processus (agent)" qui se définit, d'une part par les objets accessibles par l'agent, d'autre part, par les opérations applicables sur ces objets. Dans les solutions proposées, la démarche utilisée pour assurer la protection dans le système est plutôt orientée sur les opérations applicables sur les objets qu'une protection sur les objets eux-mêmes. Celle-ci pourrait être obtenue, cependant, de façon plus sélective en empêchant les agents d'ouvrir les paquets ou lettres qui ne leur sont pas destinés en utilisant un mécanisme de clé individuelle du même genre que celle de la solution CLU.

III.6.- CONCLUSION.

ADA peut résoudre le problème de la méfiance mutuelle, mais résoud incomplètement celui du cheval de Troie, et pas du tout celui de l'étanchéité. Néanmoins, il faut constater qu'il assure mieux la protection que les autres langages, même s'il ne résoud pas tous les problèmes.

Il convient maintenant d'étudier comment ADA peut être supporté d'une part, par une architecture existante orientée objet (iAPX 432) d'autre part, par l'architecture à structure de domaine qui nous intéresse plus précisément, à savoir OMPHALE.

*

* *

CHAPITRE IV

IMPLÉMENTATION DU LANGAGE ADA
SUR
L'iAPX 432 ET OMPHALE

PARTIE I

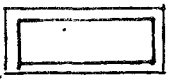
ADA ET L'iAPX 432

IV.1 - ADA ET L'iAPX 432

IV.1.1 - Présentation de l'iAPX 432.

l'iAPX 432 [INT (81)] a une architecture orientée objet. La mémoire associée doit être considérée comme une mosaïque de petits espaces adressables appelés "objets", chacun d'eux consistant en un bloc contigü de mémoire ; l'iAPX 432 peut accéder à 2^{24} objets dans un espace virtuel de 2^{40} octets.

Un objet peut être :

- **Un objet simple** représenté par  , formé par un bloc linéaire non structuré de données.

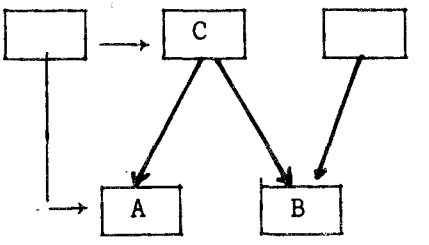
- **Un objet d'accès** (Access object), représenté par  , qui est un tableau linéaire de descripteurs d'accès (capacités) aux autres objets (une C-liste).

L'accès à un objet est toujours contrôlé par l'intermédiaire d'une capacité, qui contient les droits d'accès à l'objet ; le matériel reconnait deux types de segments de base :

- des segments de données associés aux objets simples
- des segments d'accès associés aux objets d'accès.

Un objet peut exister physiquement comme sous-partie d'un autre objet, même si les deux objets sont logiquement différents. Un objet qui est physiquement contenu dans un objet parent est appelé RAFFINEMENT de l'objet parent.

Dans l'exemple ci-contre, bien que A et B soient des sous-parties physiques de l'objet C, ils ont tous les privilèges des objets, comme s'ils étaient distincts physiquement de C et l'un de l'autre.



L'iAPX 432 intègre dans le silicium, de la même manière que par exemple l'IBM 38 :

- un jeu d'instructions de base, qui s'apparentent à celles du langage ADA

- un noyau de base du système d'exploitation, système qui opère sur les objets directement reconnus et traités par le matériel (les objets sont crées et manipulés par des micro-programmes intégrés dans le silicium).

Intel développe une couche logicielle (iMAX 432) permettant de coopérer avec la partie "silicium" du système d'exploitation.

IV.1.2 - Structure des programmes.

Chaque programme s'exécutant seul ou simultanément avec d'autres programmes est défini, du point de vue de sa priorité, sa protection, son espace mémoire et son état (en attente ou en exécution) par une structure de données, chargée en même temps que le programme correspondant et appelée "*objet-programme*". On accède directement à l'objet-programme par la logique du micro-processeur pour assurer l'exécution du programme correspondant.

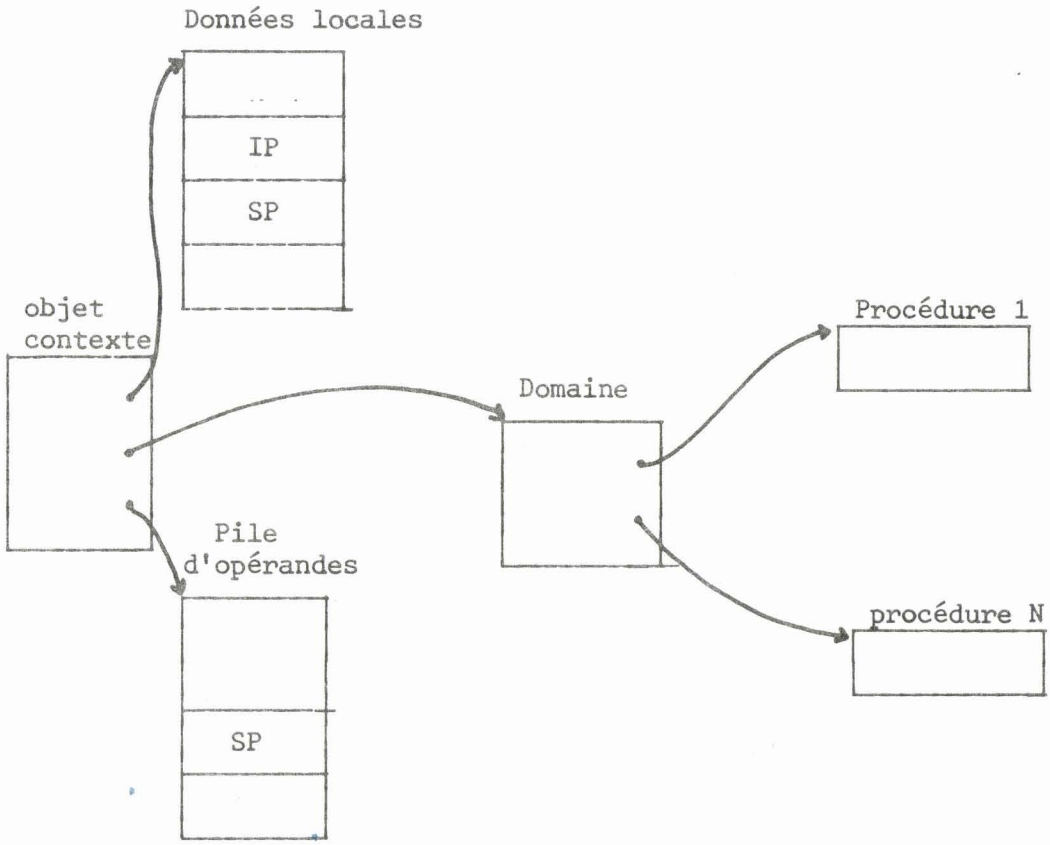
Chaque programme est constitué par une ou plusieurs "*procédures*" que la structure matérielle de l'iAPX 432 reconnaît du fait qu'elles sont représentées par un objet dit "*contexte*" qui n'est rien d'autre qu'une structure de données directement accessible par le matériel et définissant parfaitement la procédure.

Une procédure est un ensemble d'instructions réalisant une opération définie ; elle est appelée de façon externe en lui présentant les paramètres nécessaires au traitement voulu ; en retour, la procédure fournit le résultat du traitement effectué.

Une procédure peut être partagée entre plusieurs programmes, sans que son code ne soit dupliqué, autant de fois qu'il y a de programmes ; mais chaque fois que la procédure est appelée par un programme différent, un exemplaire ("*instance*") nouveau de cette procédure est créé et est représenté par un contexte associé au couple programme-procédure. Ce contexte est reconnu par le matériel iAPX 432.

D'autres types d'objets interviennent dans la structure d'un programme du l'iAPX 432. Il s'agit de l'objet instruction et de l'objet donné.

A ce stade, on peut schématiser ces notions par la figure suivante :



Un programme, matérialisé par l'objet-programme qui le définit parfaitement vis à vis de matériel, utilise un objet contexte qui décrit les procédures (segments de code exécutable) constituant le programme.., ainsi que les groupes de données intervenant dans le traitement.

L'objet programme est créé à l'initialisation, il est ensuite reconnu et manipulé par le matériel. Il contient toutes les informations nécessaires sur le programme correspondant.

IV.1.3 - Correspondances ADA <-> iAPX 432.

IV.1.3.1 - Paquetage.

Un paquetage ADA constitue une méthode pratique pour le regroupement d'un ensemble d'objets de données avec un ensemble de procédures (opérations) qui travaillent sur eux.

Le concept correspondant en l'iAPX 432 est *l'objet-domaine*, qui comme le paquetage ADA est une collection d'objets-données et d'objets-procédures. En déclarant une portion de paquetage privée, un programmeur peut empêcher l'accès de l'extérieur du paquetage à certaines procédures et à certaines données. Le reste est considéré comme publique, accessible de l'extérieur du paquetage. De même l'iPAX 432 spécifie des zones publiques (communes) et privées dans un domaine et peut protéger la partie invisible du domaine contre les accès non autorisés. Un usager ayant un accès total à un domaine aura une capacité qui autorise l'accès à tout le domaine ; par contre un utilisateur avec un accès restreint au domaine aura une capacité qui sera un raffinement de la partie visible du domaine contenant des données publiques.

De même que le paquetage ADA fournit une méthode d'organisation pour grouper des données et des procédures liées entre elles, l'objet domaine constitue une unité reconnaissable pour grouper des procédures déjà compilées et des données qui leur sont associées. Cette unité peut être construite par compilation séparée d'un paquetage ADA et stockée dans un fichier d'objets en attendant un accès ultérieur. De plus l'architecture de l'iAPX 432 garantit qu'un objet-domaine ne peut pas être confondu avec un autre type d'objet.

IV.1.3.2- Procédure.

Un autre objet qui présente une correspondance directe avec une construction ADA est l'objet-procédure correspondant à la procédure ou à la fonction ADA. Un objet procédure consiste en un ensemble d'instructions iAPX 432 exécutables, accompagnées de l'information de contrôle nécessaire pour former l'objet "contexte" qui représente le stockage local de variables associées à l'activation d'une procédure, accompagné de quelques informations de contrôle qui permettent à la procédure d'accéder à des variables globales et de revenir à la procédure d'où elle a été appelé (appelante). Un principe de base de cette approche est que des contextes différents alloués dynamiquement existent pour chaque activation de la procédure au moment de l'exécution. Un certain

nombre de langages modernes de programmation dont ADA sont des langages à structure de blocs. Les sous-programmes sont imbriqués hiérarchiquement les uns dans les autres. Le niveau lexical correspond au niveau d'imbrication des sous-programmes depuis le niveau hiérarchique le plus haut (notion de programme principal ayant le niveau 1) jusqu'au niveau le plus interne (sous-programmes locaux) ; ADA insiste sur la nécessité sémantique que :

- le retour d'une procédure P ne peut pas se produire avant le retour de toutes les procédures appelées par P.

- Des références vers des données stockées dans des contextes différents sont interdites.

Donc une simple pile (LIFO) suffit (en l'absence de tasking) pour la gestion mémoire des contextes. Les règles de visibilité de ces langages sont telles qu'une unité peut voir et accéder les objets déclarés dans les unités englobantes. Une méthode classique d'implémentation de cette accessibilité à l'exécution utilise une pile d'activation. Les concepteurs de l'iAPX 432 ont reconnus l'efficacité de cette pile en tant que base pour l'environnement au moment de l'exécution de langages à structure de bloc comme ADA. En conséquence l'architecture iAPX 432 supporte cette technique de réalisation et d'implémentation.

IV.1.4 - Exemple d'implémentation.

Nous présentons maintenant un exemple emprunté à [S. Zeigler (81)], présentant une implémentation de paquetages et de procédures ADA à l'aide de domaines et de contextes de l'iAPX 432. A cet exemple (page 8) peut être associé le schéma de la figure [1 - (page 9)] montrant la correspondance paquetage/domaine.

Au centre de cette implémentation se trouve l'OBJET D'ACCES DE CONTEXTE (OAC) et l'OBJET DE DONNEES DE CONTEXTE (ODC) qui représentent respectivement les variables de références locales (accès) et les variables de données locales d'un enregistrement d'activation (E.A). Ensemble, ces deux objets sont appelés OBJET CONTEXTE (OC).

```
procédure MAIN is
    package P is
        I,J : INTEGER ;
          K : BOOLEAN ;
          procedure ALPHA (...) ;
          procedure BETA (...) ;
          ----
    end P ;
    package Q is
          M,N : INTEGER
          procedure ONE (...) ;
          procedure TWO (...) ;
    end Q ;
    package body P is
          procedure ALPHA (...) is
          ----
              Q. TWO (...) ;
          ----
              end ALPHA
          procedure BETA (...) is ... ;
    end P ;
    package body Q is
          procedure ONE (...) is ... ;
          procedure TWO (...) is
          ----
              procedure INTWO (...) is ... ;
              ----
              INTWO (...) ;
              ----
          end  TWO
    end Q ;

end MAIN ;
```

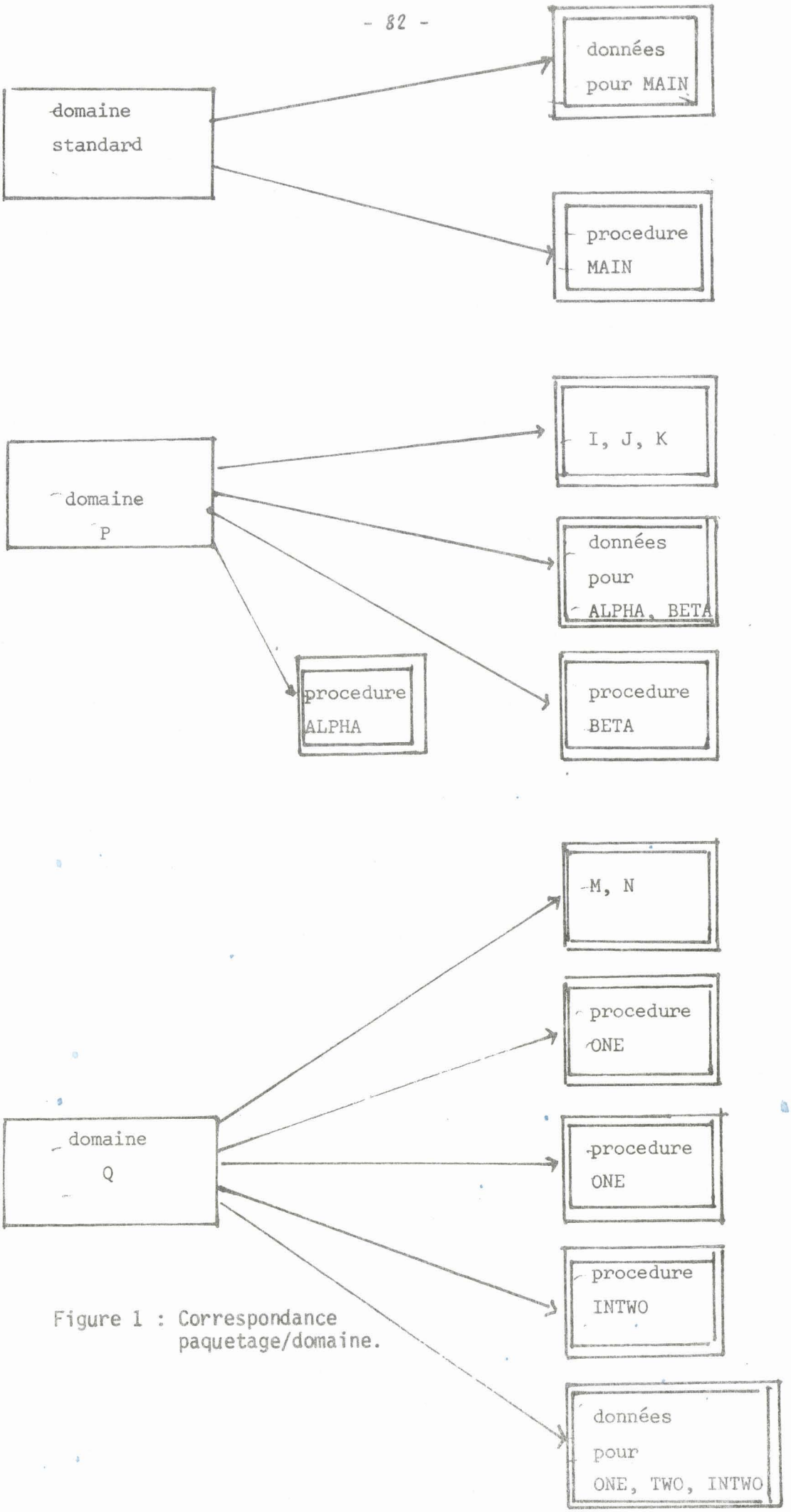


Figure 1 : Correspondance paquetage/domaine.

Il est possible à la procédure P. ALPHA d'appeler la procédure Q. TWO. A cette forme d'appel de procédure inter-paquetage en ADA correspond dans l'iAPX 432 à une activation d'objet procédure inter-Domaine invoqué par une instruction d'appel de procédure. Cette instruction demande trois paramètres pour activer un objet procédure :

- Le premier désigne l'objet domaine dans lequel l'objet procédure réside.

- Le deuxième donne l'index d'un objet procédure particulier dans l'objet domaine.

- Le troisième désigne l'objet contenant les paramètres effectifs pour l'objet procédure appelé .

Deux instructions spéciales (iAPX 432), *l'appel de procédure* et le *retour de procédure* sont utilisées pour créer de nouveaux objets-contextes et détruire les objets-contextes qui ne sont plus nécessaires ; donc elles provoquent un changement d'environnement. Les figures 2 et 3 (pages 11-12) montrent la fonction de ces deux instructions et l'effet qu'elles ont sur l'environnement d'un programme au moment de l'exécution.

La transition entre les figures 2 et 3 se fait par les étapes suivantes :

1°) - Les paramètres effectifs destinés à la procédure INTWO sont séparés en paramètres de référence et paramètres de données

. les paramètres de données sont placés dans l'objet de données de paramètres effectifs de Q. TWO (c'est un raffinement de l'objet de données de contexte (ODC) de la procédure Q. TWO (l'appelante).

. Les paramètres de référence sont placés dans un objet d'accès de paramètres effectifs de Q. TWO (c'est un raffinement de l'objet d'accès de contexte (OAC) de la procédure Q. TWO) avec une capacité vers l'objet de données de paramètres effectifs, cette capacité permet au programme d'accéder aux paramètres de données et aux paramètres de référence avec une capacité unique désignant l'objet d'accès de paramètres effectifs.
Cette capacité unique est fournie comme paramètre à l'instruction d'appel de procédure et est placée dans le nouvel objet d'accès de contexte (OAC) créé pour INTWO.

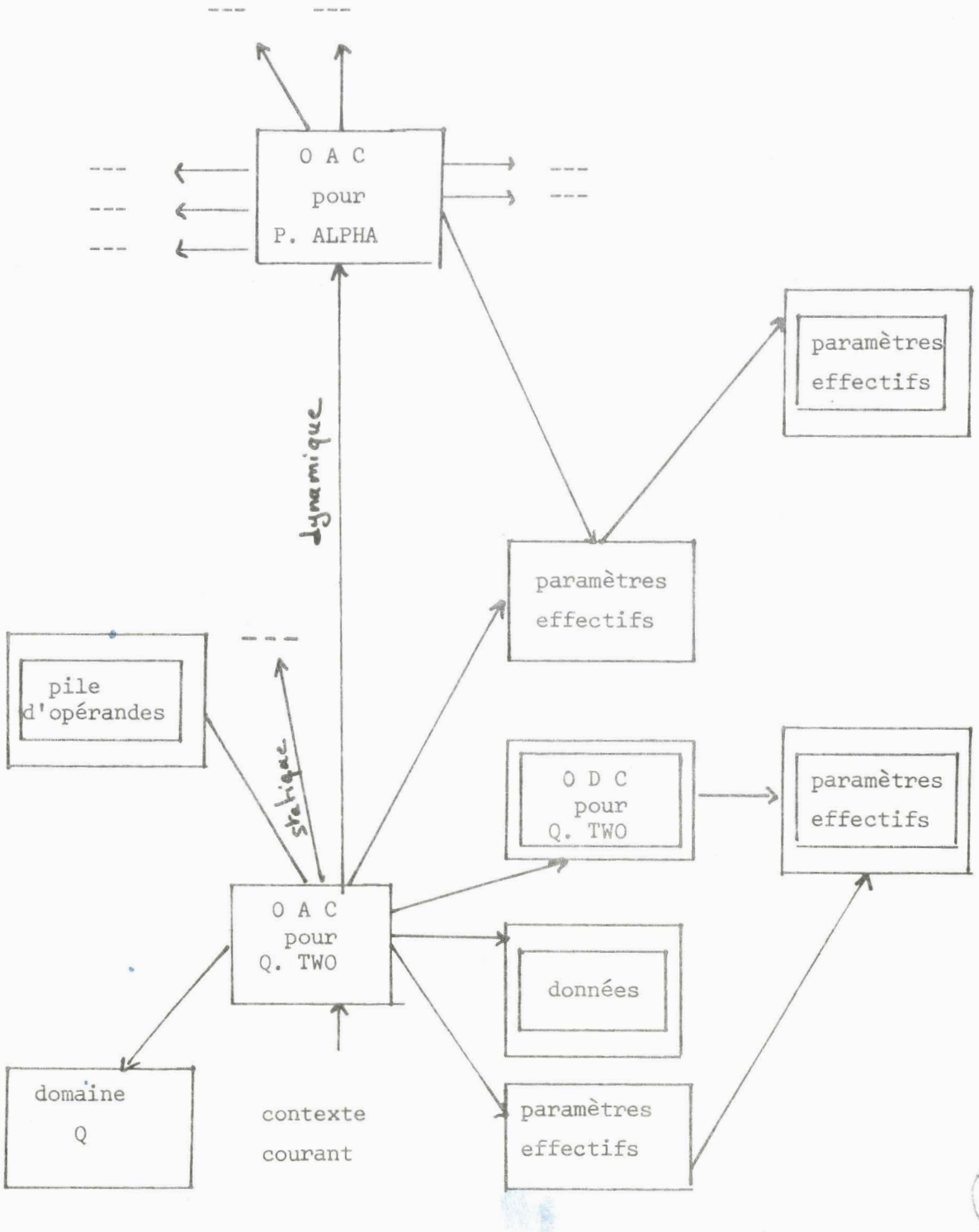


Figure 2.

La figure (2) illustre, pour la structure de programme de la figure (1), une portion de l'environnement au moment de l'exécution lorsque le programme principal a appelé la procédure P. ALPHA qui a appelé la procédure Q. TWO mais avant l'appel de la procédure INTWO.

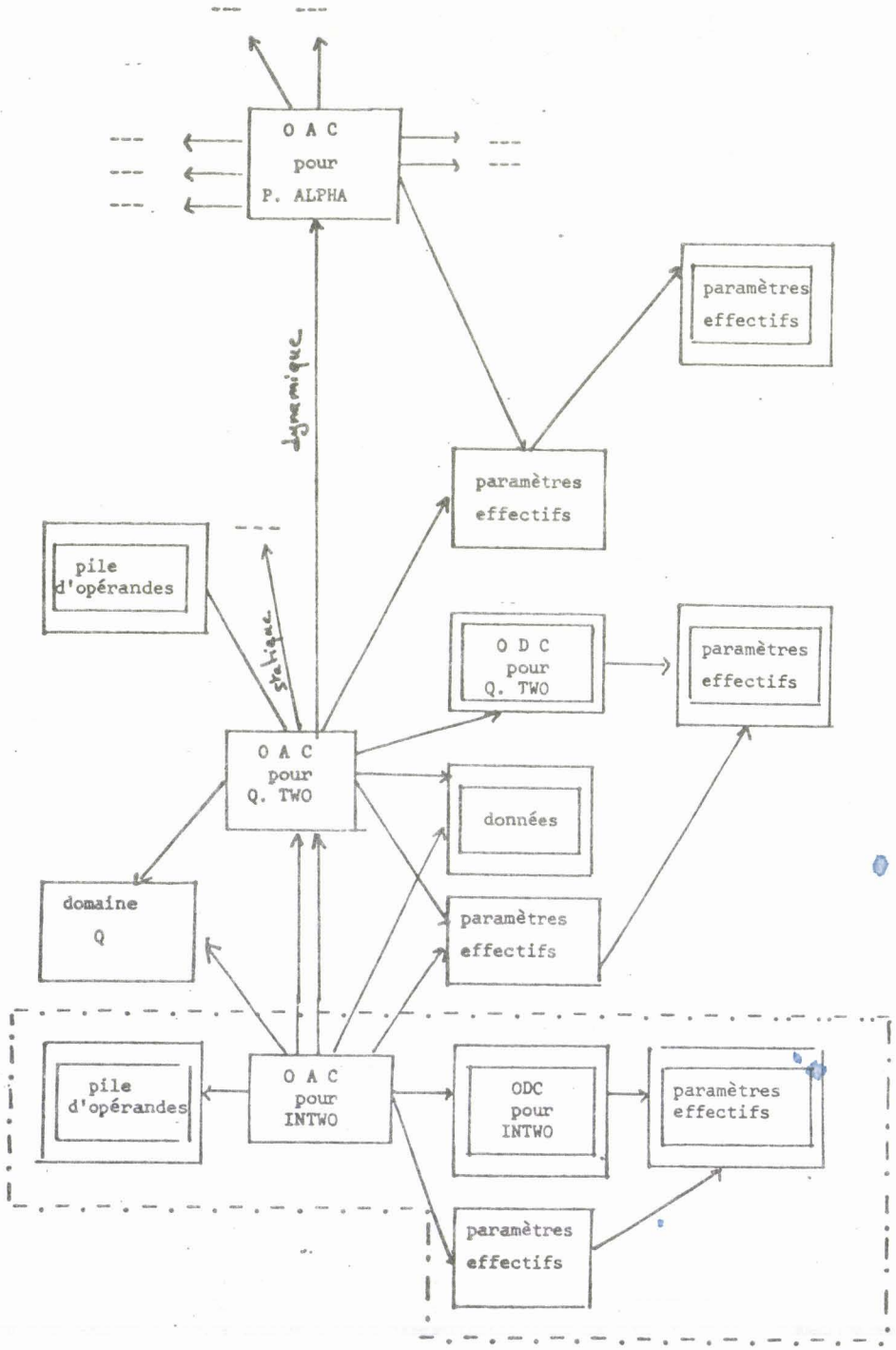


Figure 3.

La figure (3) illustre l'effet de l'appel de la procédure INTWO, l'effet de retour à la procédure Q.TWO à partir de la procédure INTWO est également donné dans la figure (2).

2°) - L'OAC et l'ODC pour la procédure INTWO appelée sont alloués. Ces objets fournissent respectivement le stockage local pour des variables contenant des informations de référence et les données. Un capacité spéciale de l'OAC désigne l'ODC qui est alloué comme emplacement pour des variables locales ne contenant que des données.

3°) - L'objet pile d'opérandes locaux est créé. Une capacité le référençant est placée dans l'OAC.

4°) - Une capacité référençant un objet contenant toutes les valeurs de données constantes apparaissant dans le domaine (paquetages) est placée dans l'OAC, Cette capacité existe déjà dans l'objet domaine engendré par le compilateur ADA.

5°) - a) - Une capacité référençant l'objet domaine courant (le paquetage Q) est placée dans l'OAC da la procédure INTWO (appelée).

b) - Une autre capacité référençant l'OAC de la procédure Q.TWO (appelante) est également placée dans l'OAC de la procédure INTWO (appelée). Cette capacité est habituellement appelée le lien dynamique.

6°) - Un lien statique pour l'accès aux variables globales est stocké dans l'OAC de la procédure INTWO ; dans notre exemple puisque INTWO est imbriquée dansla procédure Q.TWO, le lien statique référence l'OAC de la procédure Q.TWO (englobante).

7°) - D'autres objets raffinement peuvent être créés pour passer des paramètres de données et des paramètres de référence à n'importe quelle procédure appelée par INTWO.

Les étapes 2 à 5 sont réalisées par l'instruction d'appel de procédure et le compilateur ADA générerait le code pour 1, 6 et 7.

EN RESUME.

L'architecture de l'iAPX 432 est basée sur l'adressage par capacités, l'accès aux objets se fait par l'intermédiaire de capacités ; un objet ne sera accessible que si une capacité pour cet objet peut être désignée par l'instruction en cours. Dans ce système, il faut donc pour chaque phase du déroulement d'un programme que l'ensemble minimal des capacités pour les objets nécessaires à cette phase soit dynamiquement créé et que seules les capacités de cet ensem-

ble puissent être désignées (principe du moindre privilège). Cela est réalisé par les objets domaines et les objets contextes.

Un objet domaine (implémentant la structure statique d'un paquetage ADA) est fabriqué par le compilateur comme une C-liste de capacités pour des segments d'instructions et des segments de données liés sémantiquement.

Un domaine est composé de deux parties, l'une publique et l'autre privée qui sont analogues à l'interface et le corps d'un paquetage ADA. Seuls les objets pour lesquels on trouve des liens dans la partie publique d'un objet domaine peuvent être accessibles aux autres domaines connectés ; car un domaine contient aussi des liens aux autres domaines et il reste une partie d'un réseau de domaines connectés. Suivant le principe de la programmation modulaire, les domaines sont indépendants les uns des autres. Les liaisons entre domaines sont réalisées par les messages transmis à l'appel.

Un objet contexte (supportant l'environnement d'exécution d'une procédure) est crée dynamiquement par le matériel à l'appel d'une procédure d'un domaine. Il regroupe les capacités nécessaires à une exécution de la procédure. Les objets contextes supportent donc l'allocation dynamique de la mémoire à chaque activation. Le déroulement d'un processus iAPX 432 peut être vu comme une succession de basculements (changement de contexte) d'un domaine à un autre (avec passage de messages). Chaque basculement entraîne soit la création dynamique d'un nouvel objet contexte quand une procédure est appelée ou la destruction dynamique d'un contexte s'il s'agit d'un retour.

*

* *

PARTIE II

ADA ET OMPHALE

IV.II - ADA ET OMPHALE

De même que nous avons étudié l'implémentation de quelques objets d'ADA sur l'iAPX 432, nous étudions ici (incomplètement) comment OMPHALE apporte des solutions pour l'implémentation de ce langage.
Nous nous intéressons plus particulièrement aux paquetages et aux procédures, qui, comme nous l'avons montré dans le chapitre III, assurent la modularité de ADA, et donc une certaine protection.

IV.II.1 - Compilation-Exécution de ADA.

Nous rappelons ici quelques remarques concernant la compilation, et l'exécution en ADA, ainsi que l'exécution sur Omphale.

Tout programme se présente comme un ensemble d'unités de compilation qui peuvent être des procédures ou des paquetages, celles-ci sont stockées en bilbiothèque sous forme *d'unités de bibliothèque*. Par ailleurs, une procédure peut contenir d'autres procédures un paquetage peut contenir des procédures, etc...

Considérons un paquetage (unité de bibliothèque).

```
interface ┌ package P is
          │    type(s) T        }
          │    constantes       }   ressources "disponibles pour"
          │    variables V      }   d'autres unités de bibliothèque
          │    procédure P1     }
          │    procédure P2     }
          └ end P

          ┌ package body P is
          │          réalisation concrète des ressources
          │
          │ begin
          │          initialisation
          │
          └ end
```

Considérons également une procédure

```
procedure Q
begin


end
```

[PAR 76] introduit la *relation de dépendance* entre P et Q (on dit que Q dépend de P), si le bon fonctionnement de Q nécessite la présence de P. Dans ADA "modulaire", cette relation de dépendance prend deux formes distinctes :

- inclusion
- importation

1°) - **INCLUSION** : Q inclus P dans son texte

```
procédure Q is
        package P        ---- dans ce cas P n'est plus une unité
        end              ---- de bibliothèque
        package body P


        end P
begin
end
```

2°) - **IMPORTATION** : with

Q utilise des ressources fournies par P (donc décrites dans son interface) ; on dit aussi que Q "importe" ces ressources, et que P les "exporte"

```
with P ;
procedure Q is
begin
end
```

Cette relation de dépendance impose donc un ordre (partiel) pour la compilation : ainsi la procédure Q qui importe le paquetage P ne peut être compilée qu'après la compilation de la spécification de P.

Par ailleurs l'interface d'un paquetage doit être compilée et élaborée avant le corps de ce paquetage. L'élaboration du corps de paquetage aura lieu, lorsque sa présence devient nécessaire, elle consiste en l'élaboration des déclarations internes, suivie de l'exécution de la partie initialisation.

Avant l'exécution d'un programme principal, toutes les unités de bibliothèque doivent être complètement élaborées. Ces unités de bibliothèque sont celles mentionnées dans les clauses with du programme principal.
Ces élaborations sont effectuées dans un ordre respectant l'ordre partiel défini par les clauses with de ces unités de bibliothèque elles-mêmes, et ainsi de suite de manière transitive.

Ce programme se déroule dans un environnement de données prédéfinies ; les définitions de ces données sont regroupées dans un paquetage STANDARD, et tout se passe comme si le bloc le plus externe de ce programme était écrit :

```
with STANDARD
    procedure PROGRAMME-PRINCIPAL is ---
```

mais, nous notons que STANDARD est un paquetage prédéfini, il n'est pas une unité de bibliothèque (il ne peut pas être recompilé etc...).

IV.2.2 - Correspondance ADA-OMPHALE.

1°) - Procédure.

Exécuter un programme sur OMPHALE, c'est faire activer par (le système) un domaine "principal", qui est la racine d'une arborescende de domaines, un domaine est à la fois l'unité de protection et l'unité d'exécution ; cette arborescence représente l'ensemble des domaines constituant le programme à exécuter.

Ce domaine principal peut activer ses fils, et aussi créer d'autres fils à partir de maquettes qu'il interprète.

L'unité d'exécution en ADA est la procédure. Nous cherchons à établir une certaine correspondance entre un domaine d'OMPHALE et l'environnement (le contexte) d'exécution d'une procédure ADA.

Une procédure ADA simple comporte

- des paramètres formels, c'est à dire des identifications dont on précise le type et le mode de passage. Ces paramètres sont manipulés dans le corps de la procédure.

- Des objets locaux à la procédure.

- Une séquence d'instructions exécutables.

Nous supposons ici que cette procédure n'importe aucun objet. L'exemple traité sera le suivant :

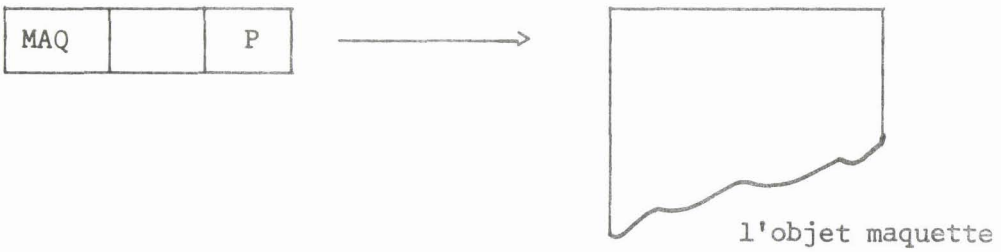
```
procédure P (X : in T1, Y : out T2, Z : inout T3) is
    M : T ;
    begin
        Y := X ;
        M := Y ;
        Z := F(Z) ;
    end ;
```

La correspondance ADA-OMPHALE s'établit pendant toutes les phases de la compilation à l'exécution en passant par l'édition de liens et le chargement.

Pour sa part, le compilateur assure partiellement cette correspondance de la façon suivante :

- création d'une maquette (objet primitif protégé) ; la création d'une maquette est effectué par un DCCM (Domaine de Compilateur pour la Création des Maquettes) appelé par le compilateur. Ce domaine reçoit en paramètre un segment dont le contenu, déduit du texte source, décrit les commandes d'interprétation et les capacités pour les objets au moment de l'interprétation de la maquette.

- celui-ci fournit en retour une capacité référençant un objet, de type MAQ, composé de n descripteurs.



L'éditeur de liens déclenche l'interprétation des maquettes et produit un objet de type construit sur le type DOM ; nous verrons cela plus loin.

Par ailleurs, la structure d'arborescence d'OMPHALE, s'apparente à la structure de bloc du langage ADA en ce qui concerne l'accessibilité aux objets, ce que nous allons étudier dans le paragraphe suivant.

Accessibilité.

Supposons qu'un programme "principal" utilisant la procédure P_1 qui elle-même déclare une procédure P_2 de la manière suivante :

```
procédure P1 is
    A : T1 ;              -- déclaration locale de P1
    B : T1 ;
procédure P2 (M : in T2, N : out T2, L : in out T3) is
    X : T2 ;              -- déclaration locale de P2
    Y : T2 ;
    begin
        N := M ;
        X := N ;
    end P2

begin
  |
  |
end P1.
```

D'après les règles de portée et d'accessibilité dans ADA on peut remarquer :

- A, B sont accessibles dans la séquence d'instruction de la procédure P_2

- L'objet P_1 détermine une ressource de calcul qui peut être appelé (invoqué) dans la partie instruction (corps) d'une autre unité de programme par exemple "principal".

- L'objet P_2 est une déclaration locale à la procédure P_1 et il est inaccessible à l'extérieur du coprs de la procédure P_1

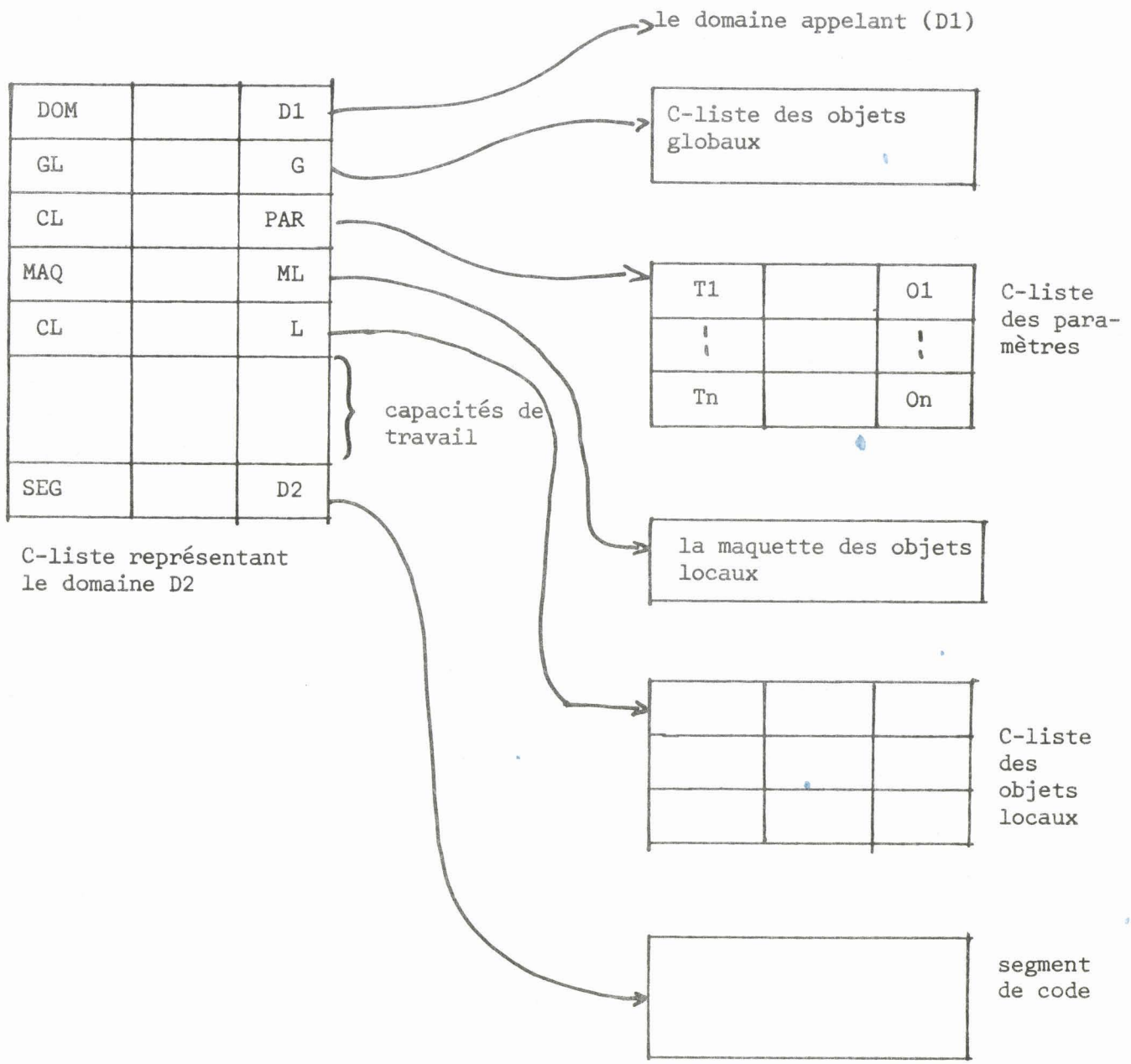
- La procédure P_2 peut être appelée à la fois dans la partie d'instruction (corps) de P_1 et à l'intérieur de la procédure P_2 (recursivité), mais elle est inaccessible dans les déclarations locales de P_1 qui précédent la déclaration de la procédure P_2.

- Les déclarations locales de P_2 (qui sont X et Y) ne sont pas accessibles dans la partie instruction (corps) de la procédure P_1, car les mots clés procédure ... end P_2 sont des limites de portée qui cachent les objets locaux déclarés dans la procédure P_2 donc ils sont inaccessibles dans l'environnement englobant.

Avant d'étudier comment ces règles sont supportées par OMPHALE, voyons comment un domaine est organisé à l'exécution :

Organisation d'un domaine.

La figure suivante représente l'organisation d'un domaine D_2, supposé en cours d'exécution après activation par un domaine D_1. Toutes les opérations de *transfert de contrôle*, permettant de programmer le transfert des paramètres, l'activation d'un domaine et l'établissement de liaison appelant/appelée entre domaines etc..., et les autres opérations permettant d'allouer des quanta de temps ou de mémoire à un domaine, sont toutes largement développées dans [DEL 79] (chap. 6.2 et 6.4). Nous pensons qu'elles ne nous concernent pas directement et nous les admettons telles quelles. La C-liste représentent le domaine D_2 contient entre autre :


le domaine appelant (D1)
DOM
D1
GL
G
CL
PAR
MAQ
ML
CL
L
capacités de travail
SEG
D2
C-liste représentant le domaine D2
C-liste des objets globaux
T1
O1
Tn
On
C-liste des paramètres
la maquette des objets locaux
C-liste des objets locaux
segment de code

Le noyau fournit une opération de création des obejts locaux, notée :

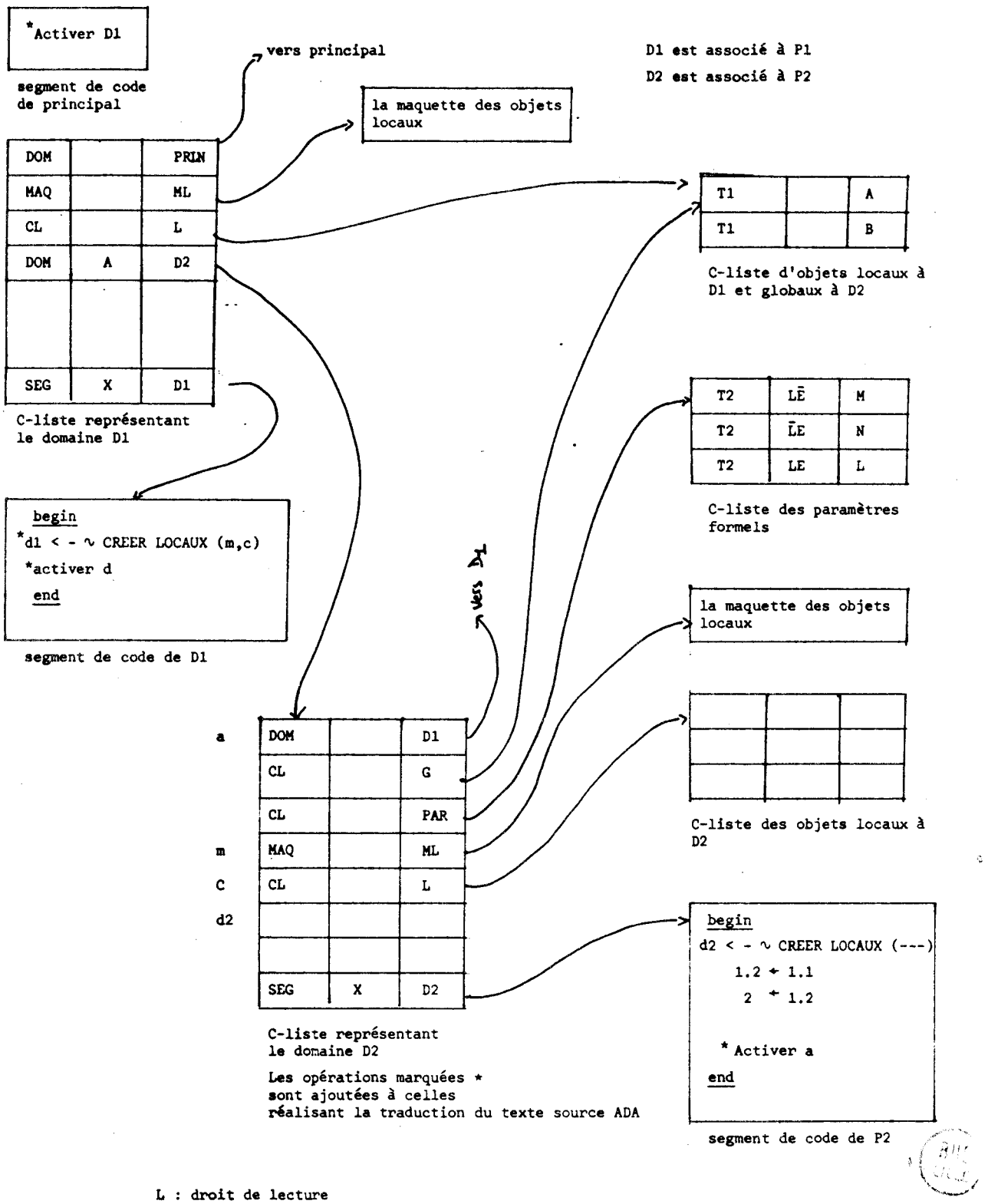
$$d \leftarrow \sim \text{CREER\_LOCAUX}\ (m, c, P_1, \ldots, P_2)$$

où m est le nom local dans le domaine appelé d'une maquette d'une C-liste des objets locaux.

c est le nom local dans le domaine appelé de la C-liste à créer, $P_1, \ldots, P_q$ sont les noms locaux des capacités repérant les paramètres d'interprétation de la maquette.

L'opération de création des locaux est déclenchée par le domaine appelé. Une C-liste des paramètres d'interprétation de la maquette des locaux est constituée et est envoyée, ainsi que les index m et c au site possédant le domaine appelé. Le noyau de ce site provoque l'interprétation de la maquette m. La capacité résultante est placée en c .
En fin d'exécution de l'appelé, les objets locaux sont détruits et le domaine est réinitialisé.

L'exemple donné au-dessus sera implémenté comme le suivant (97), en supposant que la création des locaux n'exige pas la transmission de paramètres d'interprétation.

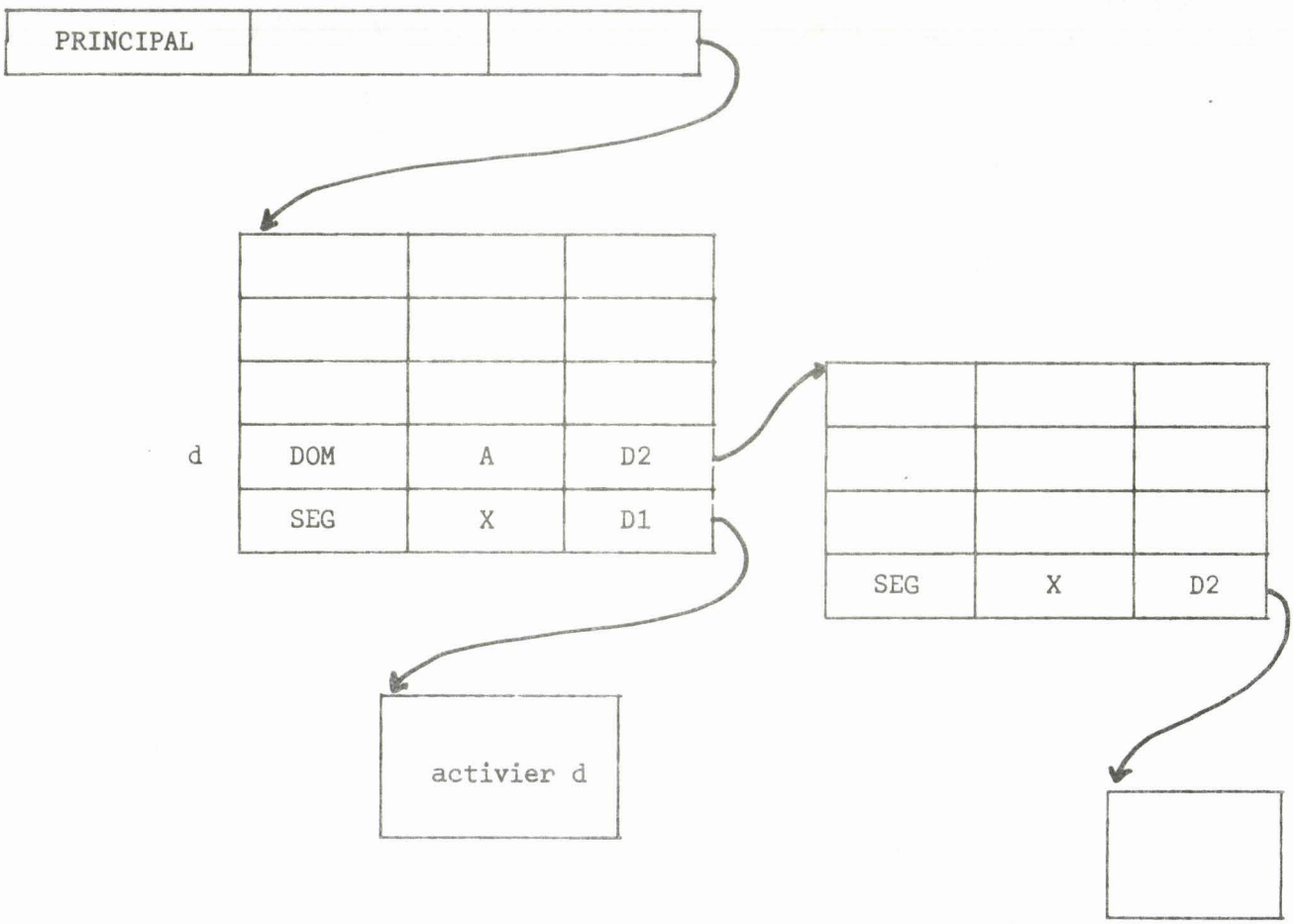


Les opérations marquées * sont ajoutées à celles réalisant la traduction du texte source ADA

L : droit de lecture

E : droit d'écriture

Ainsi avant l'exécution nous avons, en résumé, le schéma suivant.



Le chargeur décompose la capacité de type PRINCIPAL et la compose en capacité de type DOM, en suite il active le domaine D1 qui pourrait activer son fils D2 sous réserve que la maquette de celui-ci soit déjà interprétée et que D2 soit donc activable.

2°) - Paquetage.

Avant d'étudier la correspondance du paquetage nous rappelons la distinction entre un type ADA et un type OMPHALE.

La notion de type dans ADA est la même que dans les autres langages de programmation typés, PASCAL, ALGOL 68 par exemple. Par définition, un type caractérise un ensemble de valeurs, que peuvent prendre les objets de type, et un ensemble d'opérations applicables à ces valeurs. Ainsi ADA permet de

manipuler des objets simples (de type de base) et des objets construits.

Dans OMPHALE, on a intérêt, pour des raisons d'efficacité, à manipuler des objets plus "gros" que les entiers ou les réels, ce qui a conduit le système à s'intéresser particulièrement aux objets construits (pile, liste, fichier, ...). La notion de type dans ce système s'apparante à celle de type abstrait dans les langages de très haut niveau, qui correspond au paquetage ADA, la structure de données qui implémente le type est un objet appelé "objet-type". De plus, le typage en OMPHALE repose sur une approche de structuration forte de ses types où le type d'un objet détermine entièrement les types (et les droits portant sur) de ses composants, et les règles de protection lient très étroitement la structure des objets aux opérations de type.

On peut remarquer que cette approche manque de généralité par rapport aux possiblités offertes par ADA.

Prenons le cas d'un paquetage simple comportant deux procédures. On pourrait établir une certaine équivalence entre ce paquetage et un objet-type d'OMPHALE, ce que nous allons illustré sur un exemple et nous envisageons plus loin (IV.II.4) une extension pour traiter le cas, général, de la définition de plusieurs types dans le même paquetage.

L'exemple traité sera le suivant :

```
package PILE is
    procedure E (EL : in T1) ;
    procedure D (EL : out T1) ;
end ;
```

Nous complétons cette exemple dans la suite de cette partie.

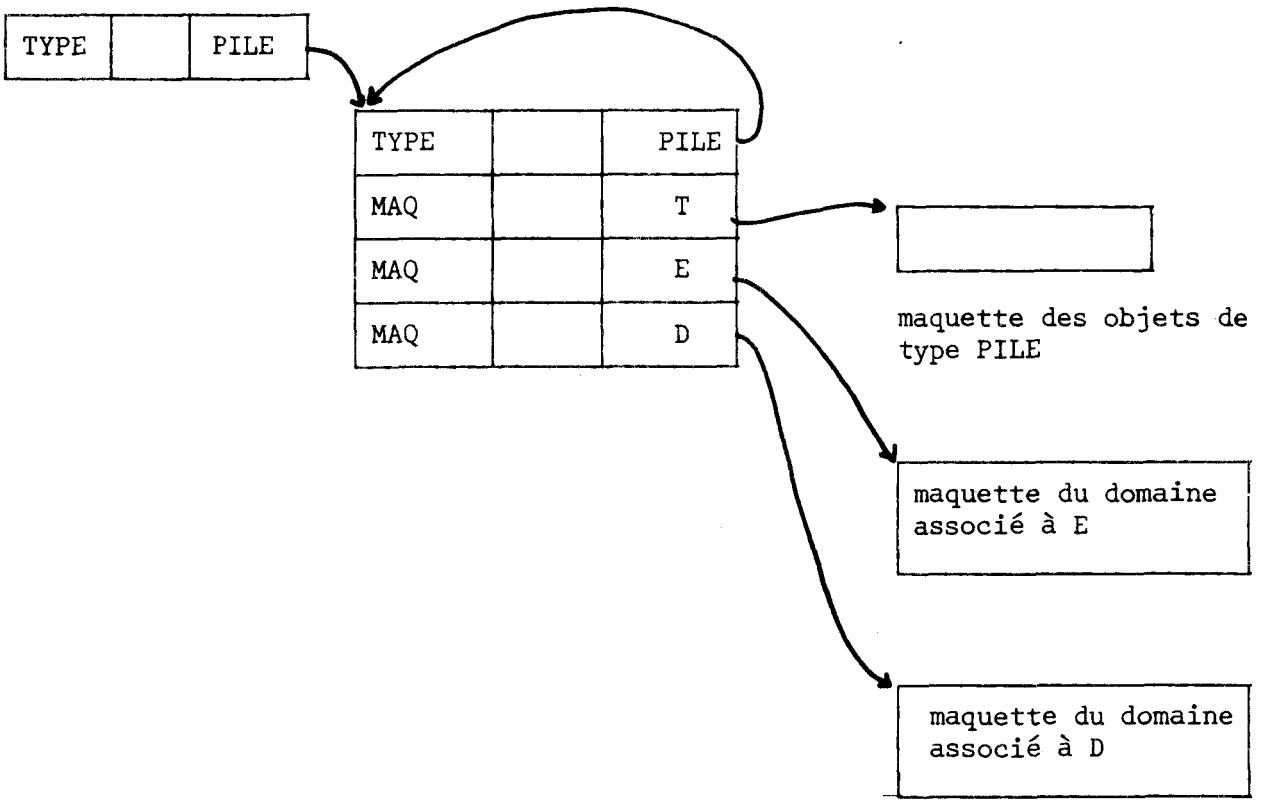
Dans l'état actuel d'OMPHALE, une première approche consiste à associer à ce paquetage un objet-type (lui-même associé à un type et non à plusieurs), qui est une structure de regroupement de maquettes.

Comme pour les procédures, la correspondance s'établir pendant toutes les phases menant de la compilation à l'exécution.

Pour sa part, le compilateur assure, partiellement, cette correspondance de la manière suivante :

- création des maquettes : la création des maquettes s'effectue par un domaine, le DCCM, appelé par compilateur. Le DCCM reçoit en paramètre un segment dont le contenu, déduit du texte source, décrivant les commande d'interprétation et les capacités pour les objets au moment de l'interprétation.

- Celui-ci fournit en retour une capacité référençont une C-liste cette C-liste représentant l'objet-type :



C-liste représentant l'objet type PILE.

Dans OMPHALE on est amené à confondre le nom de la pile avec celui de type T.

Dans ce cas nous ne pouvons créer qu'un seul objet-type PILE.
Si on veux créer plusieurs objet-type PILE on doit ajouter un type T dans le paqeutage et alors les opérations E, D aurons un paramètre de type T en plus. Nous verrons cela dans le paragraphe suivant.

IV.2.3 - Interprétation des maquettes.

Un mécanisme analogue à la création et l'interprétation de maquettes est utilisé dans ADA pour les objets génériques : soit l'exemple suivant :

```
générіc
    type T is private
    procédure ECHANGE (X, Y : in out T) is
        Z : T ;
    begin
        Z := X ;
        X := Y ;
        Y := Z ;
    end ECHANGE
```

ADA fournit une "maquette" (objet générique) ou encore un modèle non exécutable pour réaliser l'opération d'échange de deux objets. A partir de cette maquette, on peut, à l'aide d'une instruction new et en fournissant les paramètres (ici un type T =>TC (type connu) et un nom ECHANGETC, par exemple), créer une procédure exécutable pour échanger deux objets de type TC :

```
procedure ECHANGE TC is new ECHANGE (T =>TC)
```

le même mécanisme est utilisé pour les paquetages, voyons cela sur un exemple.

```
générique
        type ELEM is private
            package PILES is
                type TPILE (TAIL : NATURAL) is limited private
                procedure E (P : in out TPILE ; EL : in ELEM ; B : out BOOL) ;
                procedure D (P : in out TPILE ; EL : out ELEM ; B : out BOOL) ;

            private
                type TPILE (TAIL : NATURAL) is
                  record
                    INDEX : INTEGER rang 0 .. N := 0 ;
                    TAB : array (1..TAIL) of ELEM ;
                  end record ;
        end ;
            package body PILES is
              procedure E (P : in out TPILE ; EL : in ELEM ; B : out BOOL) ;
                begin
                    if INDEX =N then B := FALSE ;
                                else
                                begin
                                     INDEX := INDEX + 1 ;
                                     TAB (INDEX) := EL ;
                                     B := TRUE ; -- j'ai pu empilé
                                end ;
                    end if ;
                end ;
              procedure D (P : in out TPILE ; EL : out ELEM ; B :out BOOL) ;
                begin
                    if INDEX ≠ 0 then begin
                                      EL := TAB (INDEX) ;
                                      INDEX := INDEX-1 ;
                                      B := TRUE ;
                                      end ;
                                 else B := FALSE ; -- impossible
                    end if ;
                end ;
            begin
                N := 10 ;
            end ;

end ;
```

La création d'un exemplaire (une pile d'entier PE par exemple) de ce paquetage, et son utilisation se font de la manière suivante :

```
déclare
    package PE is new PILES (INTEGER)
        P : PE.TPILE (100) ;
    begin
            PE.E (P, 20, B) ;
    end ;
```

Notons cependant que le langage considère qu'une telle création d'exemplaires à partir d'unités génériques fait partie de l'élaboration des déclarations ; il impose donc une généricité explicite (instruction new).
En ce qui concerne le mécanisme proposé d'association procédure-domaine, ce choix parait donc cohérent avec l'esprit de ADA ; sauf que, dans OMPHALE, cette opération est nécessairement constitutive d'un type ; cependant la généricité dans OMPHALE doit être traitée en amont de l'interprétation des maquettes

Dans OMPHALE, les objets primitifs, i.e. de type primitifs : SEG, C-liste, et MAQ sont crées par le noyau ; seuls les objets construits sont crées par interprétation de maquettes, car le type d'un objet détermine le type de ses composants.

Comme nous venons de le voir le compilateur fournit des capacités référençant des maquettes d'objets et des maquettes de domaines non encore interprétées. Deux approches d'interprétation sont possibles : statique et dynamique, que nous allons étudier sur l'exemple de la pile, mais auparavant nous rappelons que dans (MIC 82) nous avons présenté d'autres solutions possibles ; une de ces solutions montre d'une part, l'utilisateur d'un tel paquetage dispose des opérations, Empiler, Dépiler et deux variables booléennes pile-vide et pile-pleine, accessibles. D'autre part, l'existance dans le corps du paquetage d'une partie initialisation qui sera exécuter implicitement au moment de l'élaboration du paquetage.
Or le concept d'objet fortement typé d'OMPHALE impose que les variables pile-vide et pile-pleine doivent être encapsulées dans des domaines réalisant des opérations de type associé, au paquetage ; et l'opération d'initialisation doit être considérée comme une opération constitutive du type et elle doit être explicitement activée (déclenchée) par l'utilisateur ou au moins le compilateur, quand il compile une procédure U utilisant un tel paquetage, doit générer une opération

d'activation de la partie initialisation dans le code de cette procédure.

Pour ces raisons, et pour ne pas augmenter le nombre de domaines, surtout pour ne pas créer des domaines qui réalisent des opérations très simples (tester une variable booléenne), nous proposons d'implémenter le paquetage PE créé précédemment et utilisé par la procédure suivante :

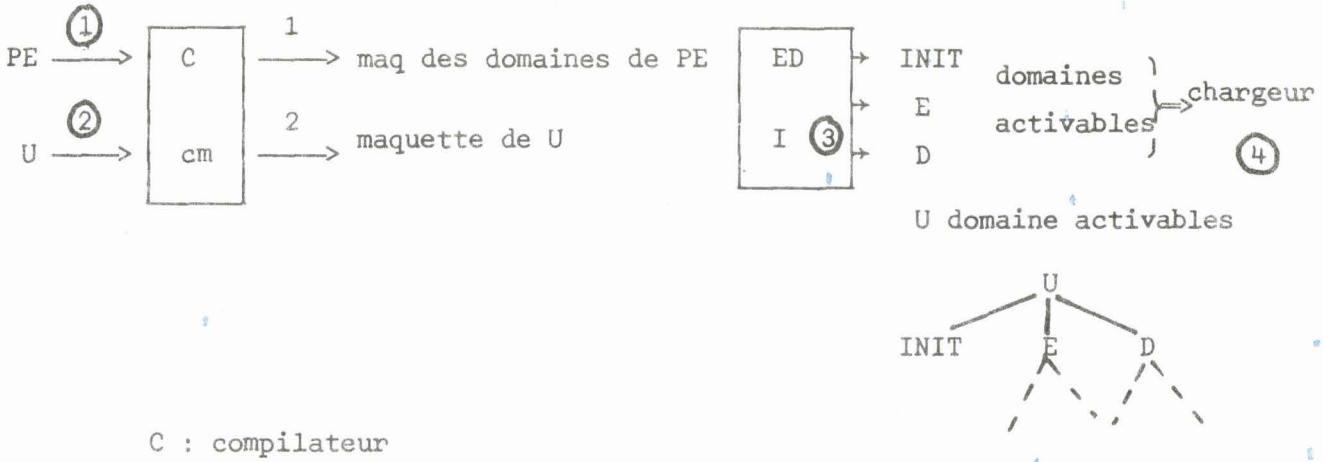
```
with PE
procedure U is          --- U comme USAGER constituant
                            le domaine principal
   X : INTERGER ;
   A : boolean
 begin

   créer P
   PE.E (X =>..., A, B)
     if A = TRUE then    -- OK
                 else    -- empilement impossible ;
                         -- pile pleine...
     end if ;

   PE.D (X, A, B)
     if A = TRUE then    -- OK ; X contient la valeur dépilée
                 else    -- non OK ; pile vide
     end if ;

end U ;
```

Texte source de la procédure U.

1°) - Interprétation statique.

L'interprétation statique consiste à interpréter les maquettes et fournir une capacité vers le domaine associé l'environnement d'exécution de la procédure U, qui est matérialisé par une C-liste de capacité référençant les objets composants ce domaine.

De même quand il s'agit d'un paquetage l'éditeur de liens va interpréter les maquettes associées au paquetage ce qui fabrique des domaines activables. Ceci revient à construire toute l'arborescence, y compirs les domaines qui ne seront éventuellement pas activés. En résumé



C : compilateur
cm : création des maquettes
ED : éditeur de liens
I : interprétation des maquettes.

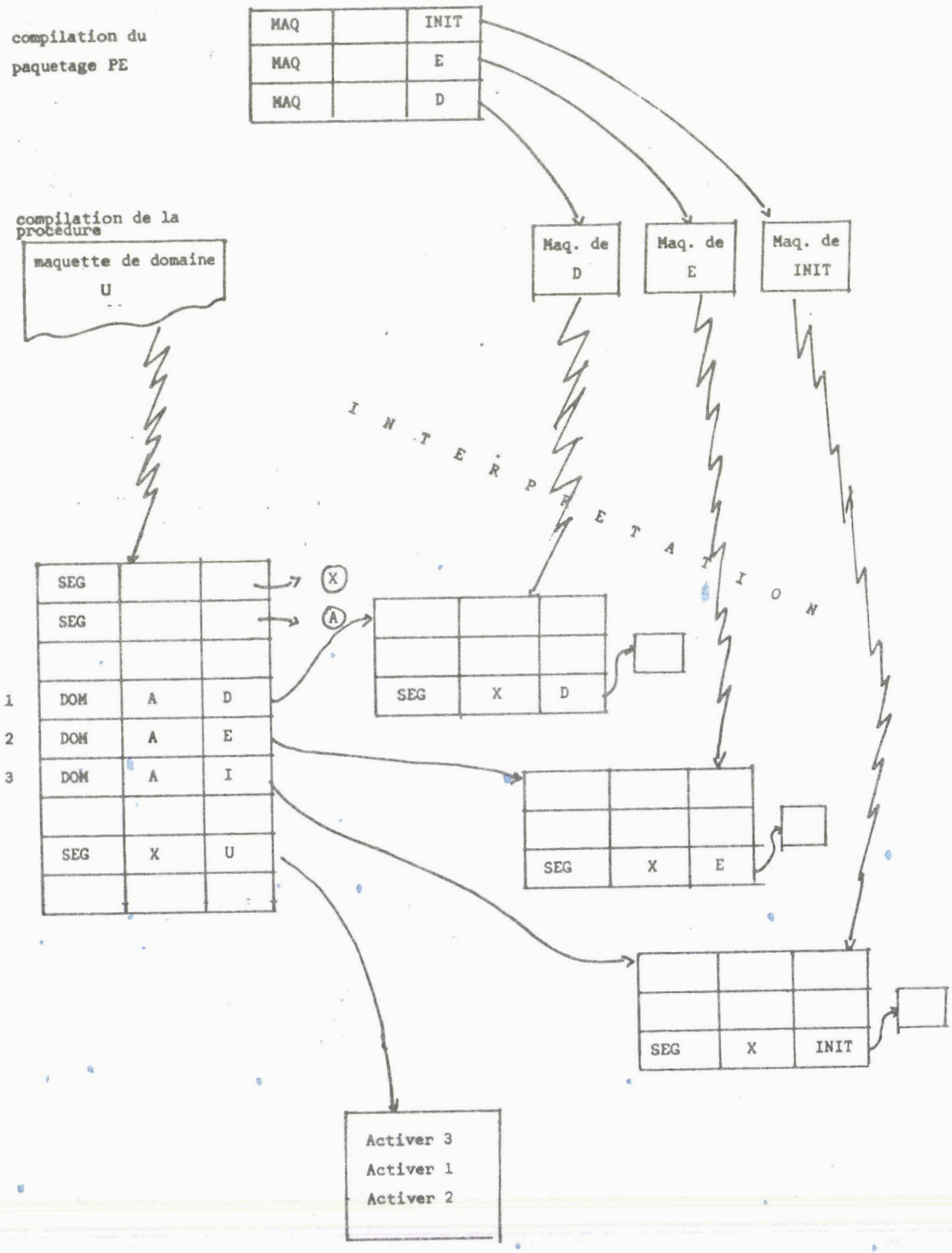
Le schéma suivant montre, partiellement, le domaine U, associé à l'environnement d'exécution de la procédure U. On peut remarquer :

(1) Compilation du paquetage qui va produire les maquettes des domaines

(2) Compilation de la procédure U qui va produire une maquette

(3) Interprétation des maquettes et constitution de l'arborescence

(4) Chargement.


compilation du
paquetage PE
MAQ
INIT
MAQ
E
MAQ
D
compilation de la
procedure
maquette de domaine
U
Maq. de
D
Maq. de
E
Maq. de
INIT
I N T E R P R E T A T I O N
SEG
SEG
X
A
1 DOM A D
2 DOM A E
3 DOM A I
SEG X U
SEG X D
SEG X E
SEG X INIT
Activer 3
Activer 1
Activer 2

2°) - Interprétation dynamique.

Quand il s'agit d'une maquette de domaine fournie par le compilateur l'élaboration (à l'exécution), consiste précisément en l'interprétation de cette maquette à l'aide des opérations d'interprétation définies, et fournit une capacité vers le domaine associé à l'environnement d'exécution de la procédure.

S'il s'agit d'un paquetage, l'interprétation dynamique consiste à conserver à l'exécution le paquetage sous forme d'un objet-type, avec ses maquettes non interprétées. Ces maquettes ne doivent être élaborées et chargées qu'en cas de besoin (c'est à dire le plus tard possible). A la compilation d'une unité (procédure), le compilateur doit alors mettre dans le code de cette procé dure des opérations (définies dans le noyau) qui permettront à l'exécution, de créer les objets et les domaines à partir des maquettes associées au paquetage et d'activer ces domaines. Cette approche, qui consiste à ne rendre statique, que ce qui est nécessaire, et de compléter au fur et à mesure, éventuellement à l'exécution, correspond mieux à l'esprit du système OMPHALE.

Le schéma suivant montre globalement l'implémentation de l'exemple donné.

Nous remarquons les différentes étapes de la procédure U utilisant le paquetage PE antérieurement compilé :

(1) Le domaine associé à l'environnement d'exécution de la procédure U contient entre-autres :

- une capacité vers l'objet-type PE
- une capacité vers son segment exécutable
- des capacités vers les C-listes nécessaires pour l'interprétation des maquettes de PE.

(2) L'opération ⊠ <- ∿ CREER-OBJET (t,p) permet de créer un objet de type PE, accessible par la capacité c, à partir

- de l'objet-type accédé par la capacité du nom local t
- des paramètres d'interprétation accédé par la capacité du nom local P.

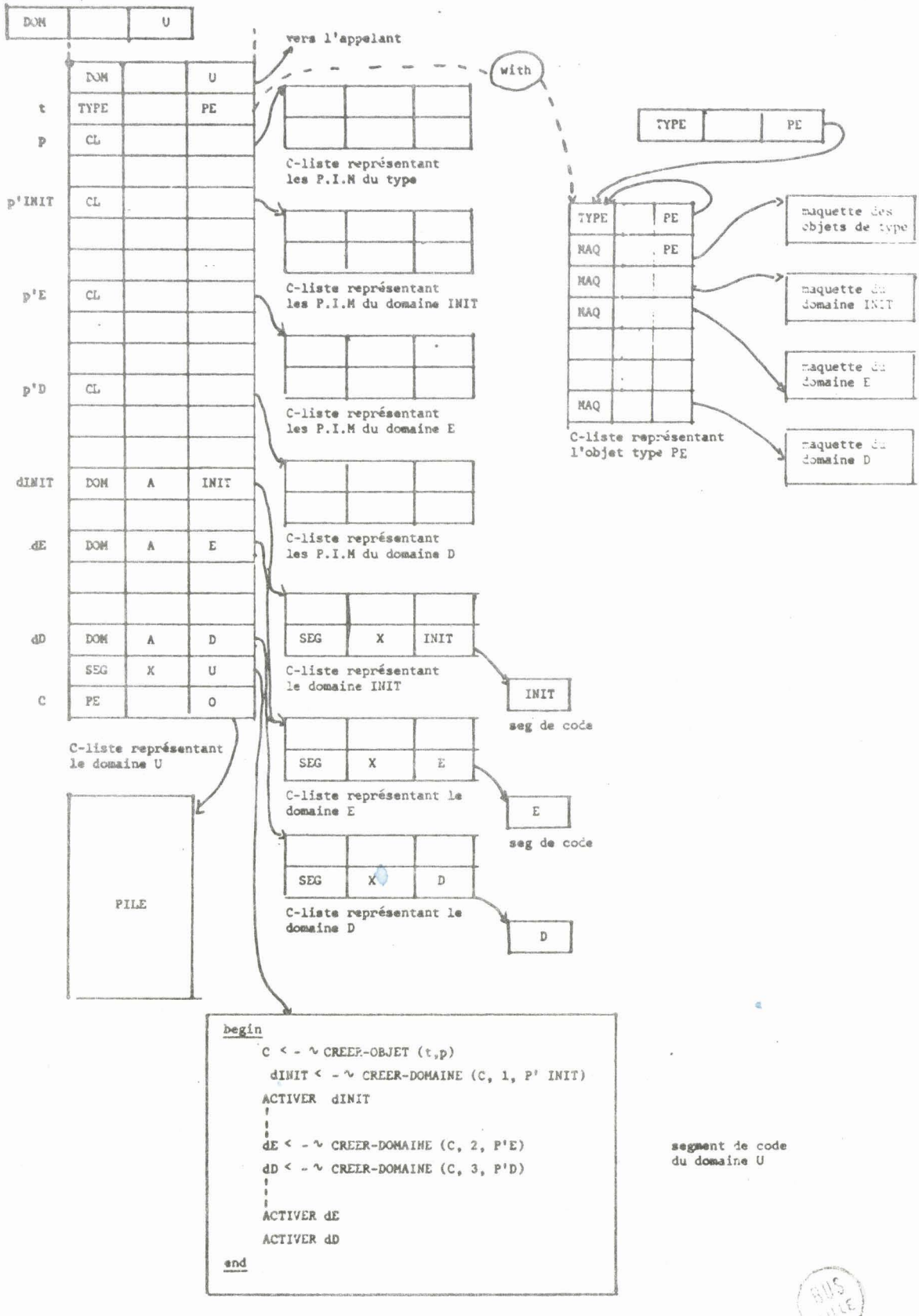
(3) Un objet de type PE étant crée, il faut créer les domaines correspondants aux différentes opérations sur ce type.
Ainsi dINIT <- ∿ CREER-DOMAINE (c, 1, p' INIT) permet de créer un domaine d'initialisation (accessible par dINIT) à l'aide

- de la capacité c permettant d'accéder à l'objet de type PE crée précédemment

- des paramètres d'interprétation de l'opération 1 associée, accédés par la capacité du nom local p'INIT.

Ce domaine étant crée, il peut (doit) être activé, à volonté, par l'opération usuelle ACTIVER dINIT qui transfère le contrôle à ce domaine nouvellement crée.

(4) Il en est de même pour dE et dD, ou tout autre domaine.



P.I.M : Paramètre d'interprétation de la maquette.

Remarques.

① En ce qui concerne le langage, nous avons déjà indiqué que l'importation des spécifications d'un paquetage par une procédure se fait soit par la clause with, soit par l'inclusion. Bien qu'il soit possible d'imaginer dans les deux cas une édition de liens statique ou dynamique, il est clair que :

- l'inclusion induit une édition de lien statique : le paquetage est décrit dans la procédure elle-même, qui est un tout

- le with, inclusion d'un paquetage extérieur, antérieurement compilé, nous oriente vers l'approche dynamique. Cependant, rien n'interdit d'utiliser une solution statique.

② Omphale étant pour sa part une machine essentiellement orientée vers le contrôle dynamique, il paraît préférable d'utiliser l'importation par la clause with, et donc l'interprétation dynamique décrite auparavant Remarquons cependant que, si ce mécanisme a de nombreux avantages du point de vue de la protection, il peut être lourd à mettre en oeuvre ; en particulier, les opérations du noyau (CREER-OBJET et CREER-DOMAINE) figurant alors dans le code de l'utilisateur, doivent être soigneusement contrôlées quant à leurs effets. Ces opérations sont rajoutées par un compilateur supposé correct et non pas par l'utilisateur.

③ Les variables et constantes déclarées dans la spécification du paquetage PE sont considérées comme décrites dans la maquette des objets du type. Elles peuvent être accédées par la procédure U utilisant PE.

Par contre, les objets rémanents, i.e. ceux déclarés dans le corps du paquetage ne peuvent pas être accédés par la procédure ; cependant, nous considérons qu'ils sont aussi décrits dans la maquette des objets de type.

④ Les objets locaux à la procédure U, ici X, A, sont crées, lors de l'élaboration du domaine associé à U.

(5) L'exemple montre le choix, que nous avons proposé d'effectuer l'initialisation du paquetage à l'aide d'une procédure d'initialisation qui inclut l'ensemble des initialisations déduites du texte source à l'aide des facilités offertes par le langage. Ceci impose que la première activation après la création de l'objet de type PE soit celle de cette procédure (domaine). Dans le cas où aucune initialisation n'est à faire, c'est à dire si les initialisations sont faites dans le texte des procédures alors il n'y a aucun problème, car ces initialisations seront faites au moement de l'activation des domaines correspondants ; dans ce cas la capacité de la procédure INIT pointera vers nil.

(6) Seules les spécifications du paquetage sont visibles de la procédure U : en effet, le domaine associé à U ne voit ni les variables locales au paquetage, ni la réalisation des procédures de celui-ci (le domaine ne peut accéder à ces procédures qu'en activant les domaines qui leur sont associés). L'implémentation proposée correspond donc aux principes de ADA en ce qui concerne l'accessibilité.

(7) La création d'un objet du type doit précéder toute création (et toute activation) de domaine du type. De même tout domaine doit exister pour pouvoir être activé. Bien entendu, dans Omphale, les droits contrôlent l'existence et l'accès aux domaines. Une erreur de programmation (activation d'un domaine qui n'existe pas par exemple) se traduit par une tentative de violation des règles d'accès et donc par un déroulement .

IV.2.4 - Paquetage implémentant plusieurs types.

Pour simplifier, dans l'exemple traité précédemment, nous nous sommes limités à un seul type dans un paquetage, car la définition originelle d'OMPHALE imposait que les opérations soient attachées à un type unique. Or ADA permet d'avoir plusieurs types dans le même paquetage, et les procédures d'un tel paquetage peuvent manipuler les représentations des objets de ces différents types ; par exemple dans (III.5) le paquetage LETTRES définit deux types TLETTRE et TCP. Si les procédures FERMER, OUVRIR, AJOUTER n'accèdent à la représentation que d'un seul type et peuvent alors être considérées comme des opérations de ce type, par contre EXTRAIRE accède à la représentation des deux types, ce qui est incompatible avec la définition de OMPHALE :

```
procedure EXTRAIRE (CP :in out TCP ;L: out TLETTRE) ;
   CPL : TCP := CP ;
   begin
     CP:=CP.all.SUIVANTE ;   -- acces a la representation TCP
     L :=CPL.all
     L.SUIVANTE :=null ;     -- acces a la representation TLETTRE
   end ;
```

Le compilateur peut alors définir une opération interne du type TLETTRE qui réalise l'accès nécessaire à EXTRAIRE et qui sera appelé par EXTRAIRE : celle-ci peut alors être considérée comme une opération du type TCP.

Cependant, une telle implémentation est assez artificielle, et couteuse.

IV.2.4.1 - Modifications à apporter à OMPHALE.

Pour permettre de lever cette contrainte, faisons quelques remarques.

1 - Les capacités des maquettes des domaines des opérations d'un type ont été placées dans la C-liste de l'objet-type T, car les opérations étaient attachées à ce type unique. Mais il n'y a pas de raisons fonctionnelles de maintenir cette dépendance directe.

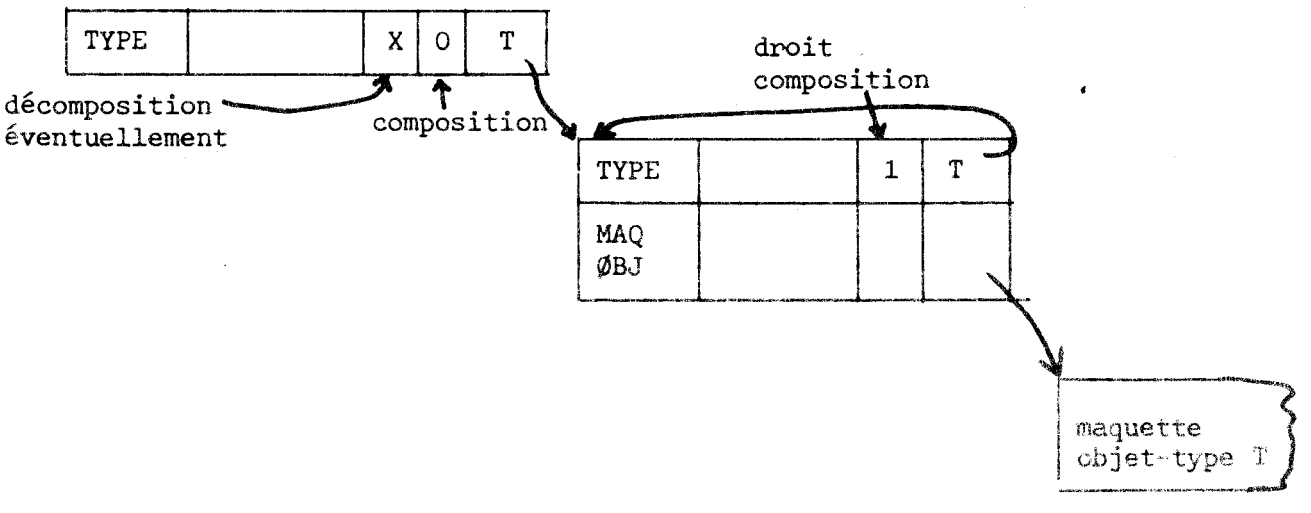
2 - La capacité de type TYPE présente en tête de la C-liste de l'objet type est la seule capacité pour cet objet-type à posséder le droit de composition. Cette unicité est assuré par le fait qu'elle ne possède pas le droit de copie ni le droit de transfert. Ceci a pour but d'interdire de pouvoir encapsuler sous le type T une C-liste quelconque : rappelons que l'opération de composition est une opération du type TYPE qui demande comme paramètre une capacité pour la C-liste de représentation de l'objet à composer et une capacité pour l'objet-type T avec le droit de composition. Il s'ensuit alors que seules les opérations du type ayant déjà accès à la représentation de l'objet-type lui-même pourront effectivement appeler cette opération de composition. C'est pourquoi la représentation de l'objet-type T doit renfermer cette capacité pour lui-même avec les droits énoncés.

3 - L'association à une maquette de domaine d'une capacité pour l'objet type DOM avec le droit de composition n'est par contre pas nécessaire, puisqu'au contraire cette capacité ne sera connue que des opérations manipulant les domaines et donc appartenant au noyau.

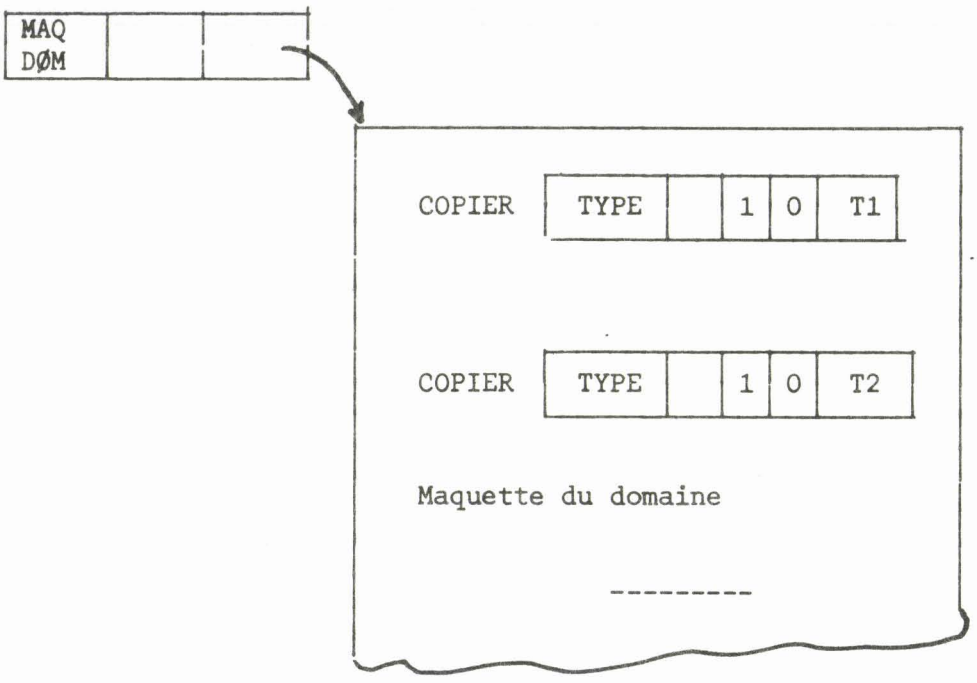
4 - Pour qu'un domaine ait accès à la représentation d'un objet du type T, il faut qu'il dispose d'une capacité pour l'objet-type T avec le droit de décomposer. Ceci était supposé implicitement réalisé pour les seules opérations du type par l'inclusion de cette capacité, avec commande de copie lors de l'interprétation dans les maquettes des domaines correspondants. Rien n'oblige cependant cette capacité à être unique, au contraire la maquette d'un domaine peut comporter des capacités avec droit de décomposition, pour chacun des objets-type T_i, que le domaine est autorisé à décomposer.

Nous aurons donc, les représentations suivantes :

1 - Objet type.



2 - Maquette domaine autorisé à décomposer T1 et T2.



IV.2.4.2 - Représentation d'un paquetage ADA en OMPHALE

La représentation d'un paquetage ADA en OMPHALE, après compilation, peut être représenté par une C-liste contenant les capacités suivantes :

1°) une capacité pour chaque objet-type T exporté par ce paquetage.

2°) Une capacité pour la maquette des objets rémanents (ou plusieurs s'il est nécessaire de classer ces objets par catégorie).

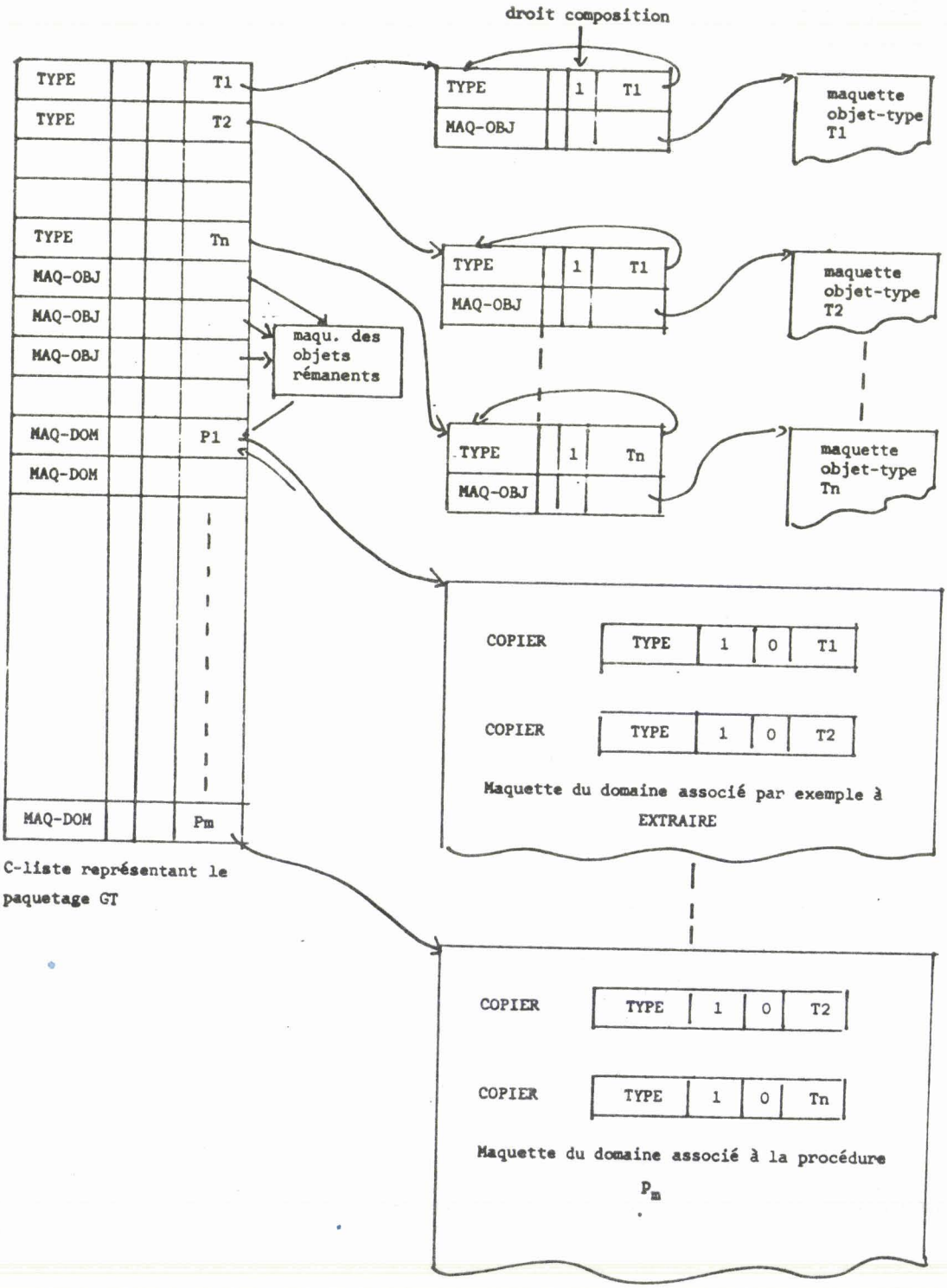
3°) Une capacité pour chaque maquette de domaine pour les procédures publiques ou privées du paquetage.

L'élaboration de la déclaration du paquetage, i.e. sa transformation en vue de l'exécution peut consister à créer une C-liste déduite de la précédente par interprétation des maquettes. Voyons cela sur un exemple du paquetage comportant n types T1,...,Tn et m opérations par exemple :

```
package GT is              ---comme groupe de types
     type T1 ;             --- T1 comme TCP
     type T2 ;             --- T2 comme TLETTRE
     type Tn ;
     procédure P1 (CP : T1 ; Li T2) ;  -- P1 comme EXTRAIRE
     procédure P2 (                ) ;
     procédure Pm (L : T2, N : Tn) ;
end ;
```

La figure suivant montre globalement la C-liste représentant ce paquetage. On peut remarquer sur le schéma , d'une part, que la structure proposée pour représenter un paquetage implémentant plusieurs types est cohérante avec le système, dans le sens où elle est composée d'une C-liste, objet primitif du système, d'autre part, que la maquette d'un domaine représentant une opération du paquetage contient autant de descripteurs que de types dont il a besoin. Chaque descripteur commence par la commande COPIER et est muni de droit de décomposition. Dans le cas où nous avons un seul type dans le paquetage, on revient naturellement à OMPHALE et dans ce cas l'objet type sera un cas particulier de ce que nous venons de proposer. Cependant, la modification proposée, correspond plus au concept de paquetage d'ADA

Enfin puisqu'il s'agit, dans cette structure, essentiellement de maquettes, nous pensons que le DCCM devrait pouvoir créer de tels objets et trouver les emplacements qui conviennent pour placer les descripteurs dans les maquettes.
En conclusion, dans OMPHALE [DEL (79)] un objet type représentait les opérations permises sur un seul type. Notre proposition consiste à élargir cet ensemble à l'union des opérations permises sur chacun des types.



C-liste représentant le paquetage GT

IV.2.5 - Conclusion.

OMPHALE est un système orienté objets fortement typés : un objet construit est dit fortement typé (son type est dit fort) si ses n composants sont astreints à vérifier des spécifications "sévères" relatives :

- aux types des composants
- aux droits requis sur les composants.

Cette hypothèse peut sembler fort restrictive et trop rigide en pratique. Il est d'ailleurs envisagé de l'assouplir au stade de l'implémentation du système [DEL 79].

ADA, répétons le, n'est pas un langage fait pour résoudre les problèmes de protection ; c'est un langage de portée générale. Cependant, nous avons montré que ADA, par les mécanisme de modularité qu'il offre, peut être adopté comme langage d'implémentation d'OMPHALE, sous réserve d'éventuelles extensions en particulier, en ce qui concerne la manipulation des droits.

IV.3 - COMMENTAIRES.

Les principes du système OMPHALE ont été établis en 1977-1978. Ils cherchaient à résoudre des problèmes de protection dans les systèmes (répartis en particulier), en utilisant les types construits (abstraits) de données.
ADA, apparu en 1979-1980, confirme certains aspects de cette approche, notamment en ce qui concerne la modularité. Le système iAPX 432, quant à lui, annoncé en juin 1981, cherche lui aussi à résoudre certains problèmes de protection, avec une architecture différente de celle d'OMPHALE. On pourrait comparer ces deux machines (mode de répartition, adressage, protection, etc...), mais nous nous contentions ici d'indiquer comment les aspects modulaires de ADA peuvent trouver leur correspondance sur l'une ou l'autre machine, aucune n'étant une machine -langage ADA-.

iAPX 432	ADA	OMPHALE
domaine	paquetage-un type paquetage n types	objet-type (maquette) groupe d'objet-type (maquettes)
contexte	procedure	domaine

ANNEXES

SOLUTIONS ADA DU SYSTÈME DE COURRIER DANS UNE PRISON.

A : Première solution

B : Deuxième solution

ANNEXE A

PREMIÈRE SOLUTION.

```
package POUR_PRISONNIER is
    type TCL is record
        MESSAGE:string  ;
        SIGNATURE:TIDP  ;
    end ;
    type  TLETTRE is limited private ;
    procedure FERMER  (CL:TCL;D:TIDP;L:out TLETTRE);
    procedure OUVRIR  (L:TLETTRE;CL:out TCL);
    function DESTINATAIRE (L:TLETTRE) return TIDP;
    type TCP is private;
    AUCUNE:constant TCP;
    function NONVIDE(CP:TCP) return BOOLEAN ;
    procedure EXTRAIRE(CP:in out TCP;L:out TLETTRE) ;
    procedure AJOUTER (CP:in out TCP;L:    TLETTRE) ;

    type TPP is private;
    AUCUN : constant TPP;
    procedure FERMER(CP:in out TCP;D:TIDP;P:out TPP) ;
    procedure OUVRIR(P:in out TPP;CP:out TCP) ;
    function DESTINATAIRE (P:TPP) return TIDP;
private
    type TCP is access TLETTRE ;
    type TLETTRE is record
        DEST : TIDP;
        CONTENU : TCL ;
        SUIVANTE: TCP ;
    end ;
    AUCUNE : constant TCP:= null ;
    type TP is record
        DEST : TIDP;
        CONTENU: TCP ;
    end ;
    type TPP is access TP;
    AUCUN : constant TPP :=null;
end ;
```

```
with POUR_PRISONNIER;
package POUR_FACTEUR is
    subtype TLETTRE is POUR_PRISONNIER.TLETTRE;
    function DESTINATAIRE (L:TLETTRE) return TIDP renames
             POUR_PRISONNIER.DESTINATAIRE ;
    subtype TCP is POUR_PRISONNIER.TCP;
    AUCUNE :TCP renames POUR_PRISONNIER.AUCUNE;
    function NONVIDE(CP:TCP) return BOOLEAN ;
    procedure EXTRAIRE(CP:in out TCP;L:out TLETTRE) renames
              POUR_PRISONNIER.EXTRAIRE;
    procedure AJOUTER (CP:in out TCP;L:    TLETTRE) renames
              POUR_PRISONNIER.AJOUTER ;

    subtype TPP is POUR_PRISONNIER.TPP;
    AUCUN: TPP renames POUR_PRISONNIER.AUCUN;
    procedure FERMER(CP:in out TCP;D:TIDP;P:out TPP) renames
              POUR_PRISONNIER.FERMER ;
    procedure OUVRIR(P:in out TPP;CP:out TCP) renames
              POUR_PRISONNIER.OUVRIR ;
end;


with POUR_PRISONNIER;
package POUR_GARDIEN is
    subtype TPP is POUR_PRISONNIER.TPP;
    AUCUN : TPP renames POUR_PRISONNIER.AUCUN;
    function DESTINATAIRE (P:TPP) return TIDP renames
             POUR_PRISONNIER.DESTINATAIRE ;
end;

with POUR_GARDIEN   ;
package GARDIEN is
   procedure ENVOYER ( P : POUR_GARDIEN.TPP ) ;
   procedure RECEVOIR ( D :TIDP ; P : out POUR_GARDIEN.TPP ) ;
end ;

with POUR_FACTEUR,GARDIEN ;
package FACTEUR is end ;

with POUR_PRISONNIER,GARDIEN ;
package PRISONNIER is end ;
```

```
package body POUR_PRISONNIER is
    procedure FERMER (CL:TCL ; D:TIDP ; L:out TLETTRE);
       begin
         L:=(DEST => D ; CONTENU => CL ; SUIVANTE =>null );
       end ;
    procedure OUVRIR (L:TLETTRE ; CL :out TCL ) ;
       begin
         CL := L.CONTENU ;
       end ;
    function DESTINATAIRE (L:TLETTRE) return TIDP
       begin
         return L.DEST;
       end;
      function NONVIDE (CP : TCP )return BOOLEAN ;
       begin
         return CP =null ;
       end ;
    procedure EXTRAIRE (CP :in out TCP ;L: out TLETTRE) ;
       CPL : TCP := CP ;
       begin
         CP:=CP.all.SUIVANTE ;
         L :=CPL.all
         L.SUIVANTE :=null ;
       end ;
    procedure AJOUTER (CP:in out TCP ; L:TLETTRE ) ;
       CPL : TCP := new TLETTRE '(L) ;
       begin
         CPL.all.SUIVANTE := CP ;
         CP := CPL ;
       end ;

   procedure FERMER (CP:in out TCP ; D:TIDP ; P:out TPP ) ;
     begin
       P:= new TP' (DEST =>D , CONTENU => CP ) ;
       CP:= AUCUNE ;
     end ;
   procedure OUVRIR ( P:in out TPP ; CP : out TCP ) ;
     begin
       CP:= P.all.CONTENU ;
       P := null ;
     end;
   function DESTINATAIRE (P:TPP) return TIDP;
     begin
       return P.all.DEST;
     end;
end ;
```

```
package body GARDIEN is
  task GARDIEN is
      entry RECEPTION ( P : POUR_GARDIEN.TPP ) ;
      entry EMETTRE   ( D : TIDP ; P : out POUR_GARDIEN.TPP ) ;
  end ;
  procedure ENVOYER(P:POUR_GARDIEN.TPP) renames GARDIEN.RECEPTION ;
  procedure RECEVOIR(D:TIDP;P:out POUR_GARDIEN.TPP)renames GARDIEN.EMETTRE;
  task body GARDIEN is
     type TLB ;
     type TPLB is access TLB ;
     type TLB  is record
          BP : POUR_GARDIEN.TPP ;
          BS : TPLB ;
     end ;
     type TB is record
          F,L : TPLB ;
     end ;
     BOITE : array ( TIDP ) of TB  := ( others => (null,null) );
     A : TPLB ;
     ID: TIDP ;
   begin
     loop
       select
        accept RECEPTION ( P: POUR_GARDIEN.TPP ) do
          ID := POUR_GARDIEN.DESTINATAIRE (P);
          A  := new TLB '( BP=>P , BS => null ) ;
          if BOITE ( ID ).L = null then
             BOITE ( ID ).F := A ;                  --liste vide
            else
             BOITE ( ID ).L.all.BS := A ;           --mise au bout
          end if ;
          BOITE ( ID ).L := A ;                     -- nouveau dernier
        end ;
       or
        accept EMETTRE ( D: TIDP ; P: out POUR_GARDIEN.TPP ) do
          A := BOITE ( D ).F ;                      -- tete de liste
          if A = null then                          -- liste vide
               P := POUR_GARDIEN.AUCUN;
            else
              P := A.all.BP ;
              BOITE ( D ).F := A.all.BS ;           --enlever de la tete;
              if BOITE ( D ).F =null then
                 BOITE ( D ).L := null ;            --c'etait le dernier;
              end if ;
           end if;
        end ;
       end select ;
     end loop ;
   end GARDIEN ;
end ;
```

```
ackage FACTEUR is   end ;

ith POUR_FACTEUR,GARDIEN ;
ackage body FACTEUR is
  PAQ : array( TIDP ) of POUR_FACTEUR.TCP:=(others => POUR_FACTEUR.AUCUNE);
  PR  : POUR_FACTEUR.TPP ;
  CP  : POUR_FACTEUR.TCP ;
  L   : POUR_FACTEUR.TLETTRE ;
  IDF : constant TIDP := ...          ; -- son identification

   loop
      loop
         GARDIEN.RECEVOIR  ( IDF , PR ) ;
         exit when PR = POUR_FACTEUR.AUCUN;
         POUR_FACTEUR.OUVRIR ( PR , CP )
         while POUR_FACTEUR.NONVIDE( CP ) loop
            POUR_FACTEUR.EXTRAIRE ( CP , L ) ;
            POUR_FACTEUR.AJOUTER ( PAQ(POUR_FACTEUR.DESTINATAIRE(L)),L);
         end loop ;
      end loop ;
      for ID in TIDP loop
         if POUR_FACTEUR.NONVIDE (PAQ(ID)) then
            POUR_FACTEUR.FERMER (PAQ(ID) , ID,PR ) ;
            GARDIEN.ENVOYER  ( PR ) ;
            PAQ (ID) := POUR_FACTEUR.AUCUNE ;
         end if ;
      end loop ;
   end loop;
nd ;

ith POUR_PRISONNIER , GARDIEN ;

ackage PRISONNIER is    end ;
ackage body PRISONNIER is
  P  : POUR_PRISONNIER.TPP ;
  CP : POUR_PRISONNIER.TCP := POUR_PRISONNIER.AUCUNE ;
  L  : POUR_PRISONNIER.TLETTRE ;
  CL : POUR_PRISONNIER.TCL ;
  S  : TIDP ;
  IDP : constant TIDP := ...          ; -- son identification
begin
   loop
      for ID in TIDP loop
         CL.MESSAGE := "M" ;  -- ecrire message
         CL.SIGNATURE := IDP ;-- signer
         POUR_PRISONNIER.FERMER ( CL , ID , L ) ;
         POUR_PRISONNIER.AJOUTER ( CP , L ) ;
      end loop ;
      POUR_PRISONNIER.FERMER( CP , IDF , P ) ;
      GARDIEN.ENVOYER ( P ) ;
      loop
         GARDIEN.RECEVOIR ( IDP , P ) ;
         exit when P = POUR_PRISONNIER.AUCUN;
         POUR_PRISONNIER.OUVRIR ( P , CP ) ;
         while POUR_PRISONNIER.NONVIDE ( CP ) loop
            POUR_PRISONNIER.EXTRAIRE ( CP , L ) ;
            POUR_PRISONNIER.OUVRIR  ( L , CL ) ;
            M := CL.MESSAGE  ;        -- lire message
            S := CL.SIGNATURE ;       -- signature
         end loop  ;
      end loop ;
  end loop ;
```

ANNEXE B

DEUXIÈME SOLUTION

```
package CONTENU_LETTRE is
    type TCL is record
        MESSAGE:string  ;
        SIGNATURE:TIDP  ;
    end ;
end ;


with CONTENU_LETTRE;
package LETTRES is
    type  TLETTRE(DEST:TIDP) is limited private ;
    procedure FERMER  (CL:CONTENU_LETTRE.TCL;D:TIDP;L:out TLETTRE);
    procedure OUVRIR  (L:TLETTRE;CL:out CONTENU_LETTRE.TCL);
   type TCP is private;
    AUCUNE:constant TCP;
    function NONVIDE(CP:TCP) return BOOLEAN ;
    procedure EXTRAIRE(CP:in out TCP;L:out TLETTRE) ;
    procedure AJOUTER (CP:in out TCP;L:    TLETTRE) ;
 private
    type TCP is access TLETTRE ;
    type TLETTRE (DEST : TIDP) is record

       CONTENU : CONTENU_LETTRE.TCL ;
        SUIVANTE: TCP ;
    end ;
    AUCUNE : constant TCP:= null ;
end ;

package body LETTRES is
    procedure FERMER (CL:CONTENU_LETTRE.TCL ; D:TIDP ; L:out TLETTRE);
       begin
         L:=(DEST => D ; CONTENU => CL ; SUIVANTE =>null );
       end ;
    procedure OUVRIR (L:TLETTRE ; CL :out CONTENU_LETTRE.TCL ) ;
       begin
         CL := L.CONTENU ;
       end ;
      function NONVIDE (CP : TCP )return BOOLEAN ;
       begin
         return CP =null ;
       end ;
    procedure EXTRAIRE (CP :in out TCP ;L: out TLETTRE) ;
       CPL : TCP := CP ;
       begin
         CP:=CP.all.SUIVANTE ;
         L :=CPL.all
         L.SUIVANTE :=null ;
       end ;
    procedure AJOUTER (CP:in out TCP ; L:TLETTRE ) ;
       CPL : TCP := new TLETTRE '(L) ;
       begin
         CPL.all.SUIVANTE := CP ;
         CP := CPL ;
       end ;
end ;
```

```
with LETTRES ;
package PAQUET is
   type TP (DEST:TIDP) is limited private ;
   type TPP is access TP ;
   procedure FERMER(CP:in out LETTRES.TCP;D:TIDP;P:out TPP) ;
   procedure OUVRIR(P:in out TPP;CP:out LETTRES.TCP) ;
 private
   type TP (DEST : TIDP) is record
      CONTENU: LETTRES.TCP ;
   end ;
end ;

package body PAQUET is
   procedure FERMER (CP:in out LETTRES.TCP ; D:TIDP ; P:out TPP ) ;
     begin
       P:= new TP' (DEST =>D , CONTENU => CP ) ;
       CP:= LETTRES.AUCUNE ;
     end ;
   procedure OUVRIR ( P:in out TPP ; CP : out LETTRES.TCP ) ;
     begin
       CP:= P.all.CONTENU ;
       P := null ;
     end;
end ;

with PAQUET   ;
package GARDIEN is
   procedure ENVOYER ( P : PAQUET.TPP ) ;
   procedure RECEVOIR ( D :TIDP ; P : out PAQUET.TPP ) ;
end ;
```

```
package body GARDIEN is
  task GARDIEN is
      entry RECEPTION ( P : PAQUET.TPP ) ;
      entry EMETTRE   ( D : TIDP ; P : out PAQUET.TPP ) ;
  end ;
  procedure ENVOYER(P:PAQUET.TPP) renames GARDIEN.RECEPTION ;
  procedure RECEVOIR(D:TIDP;P:out PAQUET.TPP)renames GARDIEN.EMETTRE;
  task body GARDIEN is
     type TLB ;
     type TPLB is access TLB ;
     type TLB  is record
          BP : PAQUET.TPP ;
          BS : TPLB ;
     end ;
     type TB is record
          F,L : TPLB ;
     end ;
     BOITE : array ( TIDP ) of TB  := ( others => (null,null) );
     A : TPLB ;
     ID: TIDP ;
   begin
     loop
       select
        accept RECEPTION ( P: PAQUET.TPP ) do
          ID := P.all.DEST ;
          A  := new TLB '( BP=>P , BS => null ) ;
          if BOITE ( ID ).L = null then
             BOITE ( ID ).F := A ;                    --liste vide
           else
             BOITE ( ID ).L.all.BS := A ;             --mise au bout
          end if ;
          BOITE ( ID ).L := A ;                       -- nouveau dernier
        end ;
       or
        accept EMETTRE ( D: TIDP ; P: out PAQUET.TPP ) do
          A := BOITE (  D ).F ;                       -- tete de liste
          if A = null then                            -- liste vide
              P := null ;
            else
              P := A.all.BP ;
              BOITE (  D ).F := A.all.BS ;             --enlever de la tete;
              if BOITE (  D ).F =null then
                 BOITE (  D ).L := null ;              --c'etait le dernier;
              end if ;
           end if;
         end ;
       end select ;
     end loop
   end GARDIEN ;
end ;
```

```
package FACTEUR is

end ;

with PAQUET , LETTRES ,GARDIEN ;
package body FACTEUR is
   PAQ : array ( TIDP ) of LETTRES.TCP := ( others => LETTRES.AUCUNE );
   PR  : PAQUET.TPP ;
   CP  : LETTRES.TCP ;
   L   : LETTRES.TLETTRE ;
   IDF : constant TIDP := ...          ; -- son identification


    loop
        loop
            GARDIEN.RECEVOIR  ( IDF , PR ) ;
            exit when PR = null ;
            PAQUET.OUVRIR ( PR , CP )
            while LETTRES.NONVIDE( CP ) loop
                            LETTRES.EXTRAIRE ( CP , L ) ;
                            LETTRES.AJOUTER ( PAQ(L.DEST),L) ;
                        end loop ;
        end loop ;
        for ID in TIDP loop
              if LETTRES.NONVIDE (PAQ(ID)) then
                 PAQUET.FERMER (PAQ(ID) , ID,PR ) ;
                 GARDIEN.ENVOYER  ( PR ) ;
                 PAQ (ID) := LETTRES.AUCUNE ;
              end if ;
              end loop ;
    end loop;
end ;
```

```
with PAQUET , LETTRES , CONTENU_LETTRE , GARDIEN ;

package PRISONNIER is

end ;

package body PRISONNIER is
   P   : PAQUET.TPP ;
   CP  : LETTRES.TCP := LETTRES.AUCUNE ;
   L   : LETTRES.TLETTRE ;
   CL  : CONTENU_LETTRE.TCL ;
   S   : TIDP ;
   IDP : constant TIDP := ...          ; -- son identification
 begin
    loop
       for ID in TIDP loop
                CL.MESSAGE := "M" ;  -- ecrire message
                CL.SIGNATURE := IDP ;-- signer
                LETTRES.FERMER ( CL , ID , L ) ;
                LETTRES.AJOUTER ( CP , L ) ;
             end loop ;
        PAQUET.FERMER( CP , IDF , P ) ;
        GARDIEN.ENVOYER ( P ) ;
        loop
           GARDIEN.RECEVOIR ( IDP , P ) ;
           exit when P = null ;
           PAQUET.OUVRIR ( P , CP ) ;
           while LETTRES.NONVIDE ( CP ) loop
                     LETTRES.EXTRAIRE ( CP , L ) ;
                     LETTRES.OUVRIR  ( L , CL ) ;
                     M := CL.MESSAGE  ;        -- lire message
                     S := CL.SIGNATURE ;       -- signature
                   end loop  ;
        end loop ;
   end loop ;
end ;
```

Remarque.

L'absence d'un compilateur ADA (prévu pour 1983) ne remet pas en cause les solutions proposées qui sont, et nous l'avons vérifié à plusieurs reprises, syntaxiquement et sémantiquement correctes. Cependant, nous regrettons de ne pas pouvoir les tester sur un système équipé d'un tel compilateur et de l'environnement de programmation également prévu.

CONCLUSION

Après avoir présenté le concept de protection dans un système informatique et certains problèmes types, puis, l'apport de différents langages existants, nous avons étudié les solutions que le langage ADA apporte à certains de ces problèmes.

Ce travail s'inscrit dans le cadre du projet OMPHALE, et s'inspire de travaux de recherches coordonnés tant pour le logiciel que pour le matériel ; nous avons montré d'une part, que, si le langage ADA ne résoud pas tous les problèmes de protection (en particulier il ne résoud pas complètement celui de la sélectivité d'accès aux objets, il se comporte plutôt mieux que les autres langages, et il pourrait contribuer à l'implémentation du système OMPHALE.

D'autre part, en étudiant la correspondance ADA-OMPHALE, nous abons mis en évidence l'incompatibilité entre la définition d'OMPHALE, qui imposait que les opérations soient attachés à un type unique, et la possibilité offerte par ADA d'avoir plusieurs types dans le même paquetage, les procédures d'un tel paquetage pouvant manipuler les représentation des objets de ces différents types. Nous avons proposé une modification d'OMPHALE, pour résoudre ce problème en restant cohérant avec l'ensemble du système.

Bien entendu, tous les problèmes ne sont pas complètement résolus et d'autre peuvent avoir une certaine influence sur la protection et par conséquent sur la fiabilité du système, en particulier ceux de la communication et du traitement d'exceptions :

Communication protection.

Dans ADA, l'entité communicante est la tâche.
La communication entre deux tâches s'appuie sur un mécanisme de RENDEZ-VOUS. L'échange des informations, éventuellement, typés (paramètres ou messages) se fait lors de rendez-vous. Nous pensons, d'une part, qu'il serait intéressant d'étudier la compatibilité de ce mécanisme avec ceux d'OMPHALE (ACTIVATION, REACTIVATION etc...) dans un environnement d'activités coopérantes.
D'autre part, il y a un certain rapport entre les objets échangés et leur protection. Il nous semble important de préciser ces aspects.

Traitement d'exceptions.

Le langage ADA, dans le but de permettre une programmation fiable a mis l'accent sur un mécanisme de traitement d'exception. Là aussi, il nous paraît nécessaire d'étudier l'apport d'un tel mécanisme dans le cadre du projet OMPHALE.

BIBLIOGRAPHIE

[CAR 82] CARREZ C.
Portée et visibilité [AME 82],
Chapitre 7, à paraître Dunod ed., à paraître 1982.

[CRO 79] CROCUS
Systemes d'exploitation des ordinateurs,
Dunod ed, seconde édition, 1975.

-

[DAHL 67] DAHL et al.,
Common Base Language,
Norvegian Computing Center, Publication S22, octobre 1970.

[DEL 79] DELATTRE E.
Contribution à la répartition d'un système à structure de domaines sur un réseau local de micro-ordinateurs faiblement couplés,
Université de Lille 1, Décembre 1973.

-

[ENG 74] ENGLAND D.M.
Capability concept mechanisms and structure in system 250,
International Workshop on Protection in Operating Systems,
Août 1974.

-

[ADA 79] ICHBIAH J.D., et al.
Preliminary ADA Reference manual,
SIGPLAN Notices, vol. 14 n° 6A - Juin 1979.

[ADA 80] [ICH 80] ICHBIAH J.D., et al.
Reference Manual for the programming language,
Juillet 1980.

[ADA 82] ANSI
Standard Ada Reference Manual,
à paraître, 1982.

[AME 82] LE VERRAND Dominique - Groupe ADA de l'AFCET -
Le langage Ada : manuel d'évaluation,
Dunod ed., à paraître, Paris 1982.

[AMB 77] AMBLER A.L., HOCH C.G.,
A study of Protection in Programming languages,
in Proc. of ACM conf. on Lnaguage Design for Reliable Software, Raleigh, Mar 1977, pp 25-40.

[AND 79] ANDREWS A.
Experience with a capability based fault-tolerant system,
2ème cours avancé, la sûreté de fonctionnement, Toulouse, (septembre 1979).

-

[CAR 79] CARREZ C.
La protection dans un système et son expression dans les langages de programmation,
Université des Sciences et Techniques de Lille 1, nov. 1979.

[FAB 74] FABRY R.S.
Capability-based addressing,
Communications of the ACM, vol. 17, n° 7, pp 403-412,
Juillet 1974.

-

[GEI 82] GEIB J.M.
Etude de l'iAPX 432
Mémoire de DEA - Lille 1 - Juillet 1982.

-

[IBM 78] IBM System/38 - BERSTIS V.
Security and Protection of Data in the IBM system 38
IEEE Soft. Eng. - 1980.

[IBM 78] IBM system/38
Technical developments,
Décembre 1979.

DURNIAK A.
System/38 shows how IBM aims to keep up with the times,
Electronics, pp 102-113 - Mars 1979.

[ICH 74] ICHBIAH J.D., et al.
The system implementation langage LIS
Décembre 1974.

[RAT 81] RATTNER J., LATTIN W.W.
Ada determines architecture of 32-bit microprocessor,
Electronics, pp 119-126, Février 1981.

[INT 81] INTEL
Introduction to the iAPX 432 architecture,
1981.

[JONES 76] JONES A.K., LISKOV B.H.,
A language extension for controlling access to shared data,
IEEE Trans., Soft. Eng., oct. 76.

-

[KIR 78] KIEBURTZ R.B., SILBERSCHATZ A.,
Capability managers
IEEE Transactions on Software Engineering, vol SE 4,
n° 6, nov 1978, 467-477.

-

[LAN 78] LANCIAUX D.
Mécanismes de structuration des systèmes à capacités,
Thèse, Université de Lille, octobre 1978.

[LIN 74] LINDEN T.A.
Different goals for protection,
International workshop on protection in operating systems
Août 1974.

[LIN 76] LINDEN T.A.
Operating system structures to support security and reliable software,
Computing surveys, vol. 8, n° 4, pp 409-445, Décembre 1976.

[LIS 75] LISKOV B.
An introduction to CLU. New directions in algorithmic languages,
ed. S.A. Schumann, IRIA, 1975.

[LIS 77] LISKOV B., SNYDER A., ATKINSON R., SCHAFFERT C.,
Abstraction mécanisme in clu
Com. ACM, vol 20, 8 Aug. 1977.

[LIS 79] LISKOV B.
Introduction to CLU - Modular program construction.
2ème cours avancé, *La sûreté de Fonctionnement,* Toulouse, septembre 1979.

-

[PAL 76] PALME J.
New Feature for Module P protection in SIMULA,
SIGPLAN notices, 11, 5 - 1976.

[PAR 72] PARNAS D.L.
A technique for Software Module Specification,
Com. ACM - Mai 1972.

[PAR 72] PARNAS D.L.
On the criteria to be used in decomposing systems into modules,
Communications of the ACM, vol 15, n° 12, décembre 1972, pp 1053-1058.

[PAR 75] PARNAS D.L., HANDZEL G.
More on Specification Technique for Software Modules,
Technical Report, Technische Hochschule, Darmstadt, West Germany, Feb. 1975.

[PAR 76] PARNAS D.L.
Some hypotheses about the "uses" hierarchy for operating systems,
Technical Report, T.H. Darmstadt, Fachbereich Informatik, 1976.

-

[*RED 74*] REDELL D.D., FABRY R.S.
Selective revocation of capabilities,
International Workshop on Protection in Operating System, août 1974.

-

[*SIL 81*] SILBERSCHATZ A., STEPOWAX S.L.
An extension of the language b Based Access-Control Mechanism of Jones and Liskov,
SIGPLAN Notices, vol. 16, n° 5, May 1981.

-

[*WUL 74*] WULF W.A. and al.
Hydra : the Kernel of a Multiprocessor Operating System,
CACM, vol. 17, n° 6, June 1974.

[*WUL 75*] WULF W.A., LONDON R.L., SHAW M.,
Abstration and verification in Alphard in New Directions in Algorithmic Languages,
S.A. Schuman ed., IRIA, 1975.

-

[*ZEI 81*] ZEIGLER S., ALLEGRE.M., JOHNSON R., MORRIS J.
Ada for INTEL 432 Microcomputer,
Computer, June 1981.