UNIVERSITE DE NANCY I
U. E. R. DE MATHEMATIQUES

définition et interprétation des modules de commande dans le projet civa

thèse

pour l'obtention
du doctorat de spécialité
informatique
(mathématiques appliquées)

soutenue le 4 mars 1975
par
daniel viault

jury :

M. C. PAIR Président
M. J. C. DERNIAME Examinateurs
Mme C. ROLLAND
M. S. KRAKOWIAK

UNIVERSITE DE NANCY I
U. E. R. DE MATHEMATIQUES

définition et interprétation des modules de commande dans le projet civa

thèse

pour l'obtention
du doctorat de spécialité
informatique
(mathématiques appliquées)

soutenue le 4 mars 1975
par
daniel viault

jury :

M. C. PAIR Président
M. J. C. DERNIAME Examinateurs
Mme C. ROLLAND
M. S. KRAKOWIAK

Je remercie vivement Monsieur le Professeur PAIR, Directeur de l'Institut Universitaire de Calcul Automatique, pour la bienveillance qu'il n'a cessé de me témoigner et pour l'honneur qu'il me fait en présidant ce Jury.

Je tiens à exprimer toute ma gratitude à Madame ROLLAND qui à accepté de participer au Jury.

Je remercie vivement Monsieur KRAKOWIAK pour l'honneur qu'il me fait en participant au Jury.

Que Monsieur DERNIAME trouve, dans ce travail, l'expression de ma profonde et amicale reconnaissance pour l'attention, les précieux conseils et les orientations fructueuses qu'il m'a prodigués dans la direction de mes recherches.

J'adresse mes remerciements aux membres de l'équipe CIVA, spécialement à Madame BRETHES, pour leur collaboration efficace.

Enfin, je remercie mon épouse, Madame ZANNAD, Madame DRIQUERT et Monsieur DEBERDT pour la qualité et la rapidité de leur travail dans la réalisation matérielle de cette thèse.

SOMMAIRE

TABLE DES MATIERES

INTRODUCTION

Le but du projet CIVA est de donner à l'utilisateur d'un système informatique les moyens nécessaires à la réalisation d'un travail en utilisant une formulation unique au cours des différentes phases que sont la conception, l'analyse organique, la programmation, la mise au point et l'exploitation des programmes.
Pour atteindre ce but, plusieurs actions furent menées :

- Définir un langage unique permettant d'exprimer les actions entreprises au cours de l'analyse, de la programmation et de l'exploitation (DERNIAME [8]).
- Etablir un système de compilation permettant de traduire ce nouveau langage (DUCLOY [9] , DENDIEN [7] , FIEGEL [II]).
- Proposer aux futurs utilisateurs une méthode d'analyse qui les oriente dans l'utilisation de ce système (HERTSCHUH [I4] , CHABRIER [5]).

En plus de la participation, comme chacun d'entre nous, à la définition de l'ensemble du projet, notre rôle particulier a été d'étudier les actions à entreprendre pour la mise en oeuvre d'une chaine de traitement et de réaliser les outils proposés. Ceci nous a amené à créer les modules de commande qui regroupent les instructions permettant d'exprimer quelles sont les actions à entreprendre en vue de l'exploitation des chaines de traitement.

On trouvera donc ici une première partie de présentation des modules de commande dans le projet CIVA. L'idée d'avoir un langage unique pour exprimer les actions à entreprendre n'est pas nouvelle. Elle à déjà été developpée dans (GURSKI [26]) et se base sur la remarque suivante : les programmeurs ont accepté depuis longtemps le fait que langage de programmation et langage d'exploitation étaient disjoints et jamais que les deux pouvaient se regrouper. Mais cette acceptation n'est pas naturelle : pour se parler et se comprendre deux personnes utilisent généralement le même langage quel que soit le sujet de leur conversation. Alors pourquoi un programmeur devrait-il avoir à utiliser deux (ou même plusieurs) langages différents pour exprimer à l'ordinateur tout ce dont il a besoin de faire ?

Il est possible de trouver deux réponses à cette question :

- Les progrès réalisés dans le domaine des langages de programmation depuis une quinzaine d'années ne se sont généralement pas reflétés dans le domaine des langages d'exploitation. Ces derniers pourtant d'usage fréquent, sont restés, pour la plupart, primitifs et mal adaptés aux besoins actuels (STEPHENSON [28]).

Cette situation est née certainement du fait que les interprètes des langages de commande constituent une partie intégrante des systèmes d'exploitation. Du fait d'un certain besoin de stabilité éprouvé par les utilisateurs d'un système informatique, les possibilités de mise en oeuvre de nouveaux langages de commande ont été restreintes et les personnes qui ressentaient un intérêt à leur changement, voire même à leur évolution, ont rarement pu appliquer leurs études. Le développement des systèmes à usager unique fonctionnant en mémoire virtuelle (du type GEMAU) aurait, cependant, présenté l'occasion de modifier librement une ou plusieurs parties d'un système d'exploitation et , par là-même, de tenter des expériences dans ce domaine.

- Une deuxième réponse pourrait être la suivante : aucun des langages actuels ne permet au programmeur d'exprimer tout ce dont il a besoin de dire à l'ordinateur. Cette dernière thèse a été developper dans (NEBUT [36]) qui considère que, pour mener à bien un projet informatique il est nécessaire d'utiliser au moins trois langages différents : un langage d'interaction avec la machine du type assembleur dont le but est de procurer l'utilisation totale des possibilités du matériel, un langage de production évolué dont le but est de permettre une programmation lisible et rapide et enfin un langage intermédiaire pour optimiser les parties privilégiées du projet et pour assurer la communication avec les deux autres langages.

A ces trois langages nécessaires à la réalisation du projet, il est généralement utile d'ajouter des "paralangages" qui permettent d'assurer la communication avec le système d'exploitation ou avec des services annexes tels que l'aide à la mise au point, un éditeur de texte,.....

Cette solution, qui a été utilisée pour la conception et la réalisation d'un système d'exploitation, présente toutefois l'inconvénient majeur d'obliger le programmeur à être "polyglotte".

La deuxième partie de cet exposé présente la réalisation des interfaces nécessaires entre le langage de commande que nous avons défini et le système d'exploitation de l'ordinateur (ici, nous utiliserons le système SIRIS 7 du CII IO070) qui possède son propre langage d'exploitation. Au cours de cette réalisation, nous avons été amené à régler les problèmes de la gestion de la mémoire centrale de façon à être compatible avec SIRIS 7 à écrire un générateur de cartes de commande pour pouvoir exprimer les actions à entreprendre et gérer les fichiers des utilisateurs, enfin à définir les liaisons qui devaient exister entre les différents modules du système CIVA pour permettre l'exploitation du système global CIVA-SIRIS 7.

1 ère PARTIE

DEFINITION
DES
MODULES DE COMMANDE
DANS
LE PROJET CIVA

1

NOTIONS DEVELOPPEES

DANS

LE PROJET CIVA

Ce chapitre a pour but de familiariser le lecteur avec les différents constituants de CIVA. Nous nous contenterons de rappeler brièvement ces éléments et leurs caractéristiques essentielles. Le lecteur pourra se reporter à (DERNIAME [8]) pour les définitions correspondantes.

I.I. LES UNITES DE NOMENCLATURE ET LEURS RELATIONS

(DERNIAME [8] : § I.I., I.2., 9.5.)

Comme dans tout système de programmation, les utilisateurs de notre projet peuvent définir des objets désignés par des identificateurs. Ces objets peuvent être de trois types différents :

- les objets élémentaires qui sont de type primitif (entier, réel,....)
- les objets structurés qui sont des compositions d'objets élémentaires,
- les objets de type file d'objets élémentaires ou structurés.

Le projet CIVA est un système de programmation modulaire : les objets déclarés par l'utilisateur sont regroupés dans des unités de nomenclature. Ces unités de nomenclature, ou plus simplement unités, sont de deux types différents : les classes ou les modules. Illustrons par des exemples leur utilisation.

- Soit la classe C1 définie de la façon suivante :

```
classe C1 ;
S entier ; A entier ; B entier = 3 ;
D entier ;
finclasse ;
```

La classe C1 contient les déclarations de trois objets élémentaires de type variable entière désignés respectivement par S, A, et D et d'un objet élémentaire de type constante entière désigné par B.

- Soient deux modules M1 et M2 définis de la façon suivante :

```
module M1 ;
utilise C1 ;
S = A + B ;
finmod ;
```

```
Module M2 ;
utilise C1 ;
S = A - B ;
finmod ;
```

Le module M1 "utilise" la classe C1 : tous les identificateurs déclarés dans C1 sont utilisables dans M1.

Le module M2 "utilise" également C1. Il y a donc partage de la description des identificateurs de C1. Mais les objets désignés par S, A ou B dan M1 et M2 sont différents.

- Soit un troisième module défini par :

```
module M3 ;
utilise C1 ;
M1 ; M2 ;
finmod ;
```

Le module M3 "appelle" successivement les modules M1 et M2. M1 et M2 peuvent être considérés comme des sous-programmes fermés externes de M3. M3 utilisant également C1, c'est alors le même objet élémentaire qui est désigné par l'identificateur S dans M1 et dans M2. Il y a alors communication de valeur entre les deux modules M1 et M2. La durée de vie de l'objet désigné par S est égale à la durée d'activation du module M3.

Dans le projet CIVA, en plus des possibilités offertes par le langage de programmation, l'utilisateur peut employer un métalangage qui lui permet de générer du texte source dans ces unités (ce métalangage est défini dans (BENAMGHAR [3] et HARMANT [I3]). Il permet de déclarer des métavariables, d'utiliser des métainstructions ou des métafonctions.

Par exemple, si la classe C2 est déclarée comme suit :

```
classe C2 ;
S entier ;
$B = "D entier ; " ;
$C = "D réel ; " ;
$si  $A>0      $alors     $B ;
               $sinon     $C ; $fsi ;
finclasse ;
```

$B et $C sont des métavariables auxquelles sont affectées des chaînes de caractères représentant des éléments du langage de base.

Si la valeur de la métavariable $A est positive, alors le texte de la classe C2 est :

```
classe C2 ;
S entier ;
D entier ;
finclasse ;
```

Par contre si la valeur de la métavariable $A est négative ou nulle, le texte de la classe C2 est :

```
classe C2 ;
S entier ;
D réel ;
finclasse ;
```


L'utilisateur peut également utiliser le métalangage à l'aide des métamodules. Ces derniers permettent également de produire des textes de modules ou de classes. Ce sont des objets autonomes et l'intérêt essentiel de ces métamodules est de simplifier l'écriture des unités d'une application. De plus, il faut noter que ces métamodules permettent de rendre conditionnels les utilisations de classes et les appels de module.

```
métamod $MM ;
$si  $A>0  $alors utilise X ;
                 F(3) = F(2) + F(1) ;
           $sinon utilise Y ;
                 G(3) = G(2) * G(1) ; $fsi ;
$finmod ;

module SOMME ;          module MULTI ;
$A = 1 ;                $A = 0 ;
$$MM ;                  $$MM ;
finmod ;                finmod ;
```

Les textes de ces modules sont alors :

```
module SOMME ;          module MULTI ;
utilise X ;             utilise Y ;
F(3) = F(2) + F (1)     G(3) = G(2) * G(1) ;
finmod ;                finmod ;
```

Un objet autonome de type métamodule peut être considéré soit comme une entité indépendante (dans ce cas, il est déclaré comme une classe ou un module et peut être employé par n'importe quelle unité) soit comme une entité locale (dans ce cas, il est déclaré à l'intérieur d'une classe ou d'un module : la portée de son identificateur est limitée à l'unité qui contient sa déclaration).

```
classe C3 ;
F file (max = 3) entier ;
métamod $M ;
$si $A>0  $alors  F(3) = 2  ; $fsi ;
$finmod ;
finclasse ;
```

C3 contient la déclaration d'un objet F de type file et d'un objet autonome de type métamodule.

Soit le module M5 déclaré par :

module M5 ;
utilise C3 ;
$$M ;
finmod ;

Le module M5 utilise la classe C3 : il peut donc utiliser tous les identificateurs déclarés dans cette classe. En particulier, il peut contenir une demande de la génération du texte obtenue par le métamodule $M.

Remarque

Comme nous le verrons au chapitre 2, le système CIVA à besoin de connaître l'environnement complet de chaque unité x définie par l'utilisateur, c'est-à-dire l'ensemble des unités appartenant à $\hat{U}(x)$ et à $\hat{A}(x)$ où U désigne la relation "utilise" et A la relation "appelle".
Cet environnement est connu directement du système si le texte source de l'unité x est entièrement déterminé. Or nous venons de voir que l'utilisateur pouvait employer le métalangage et plus particulièrement les métamodules pour générer ce texte source.
Le système CIVA est donc obligé de prendre en considération ces objets autonomes et pour cette raison, nous assimilerons les métamodules à des unités de nomenclature. En plus des relations de bases du langage (relations "utilise " et"appelle"), l'appel d'un métamodule constitue une relation importante pour le système. Nous appellerons cette relation, la relation "$appelle".

I.2. NOTION D'APPLICATION (DERNIAME [8] : § I.3.)

La notion d'application est à rapprocher de la notion de service dans une entreprise. Une application renferme l'idée de travail attaché à un groupe d'hommes qui étudie le même sujet. Pour notre système, l'application est un niveau intermédiaire entre les objets CIVA et le système d'exploitation. A l'intérieur d'une application, l'utilisateur peut définir un certain nombre de traitements et prendre un certain nombre d'options implicites (valeurs de variables d'exploitation, constantes utilisables par tous les programmes ...).
A chaque application correspond une "bibliothèque"appelée classe d'application.
Cette classe particulière contient les déclarations des unités de nomenclature de l'application (classes ou modules), ainsi que les déclarations des paramètres implicites utilisés par l'application. Elle contient également les déclarations des objets autonomes de type métamodule qui sont utilisables par toutes les unités.

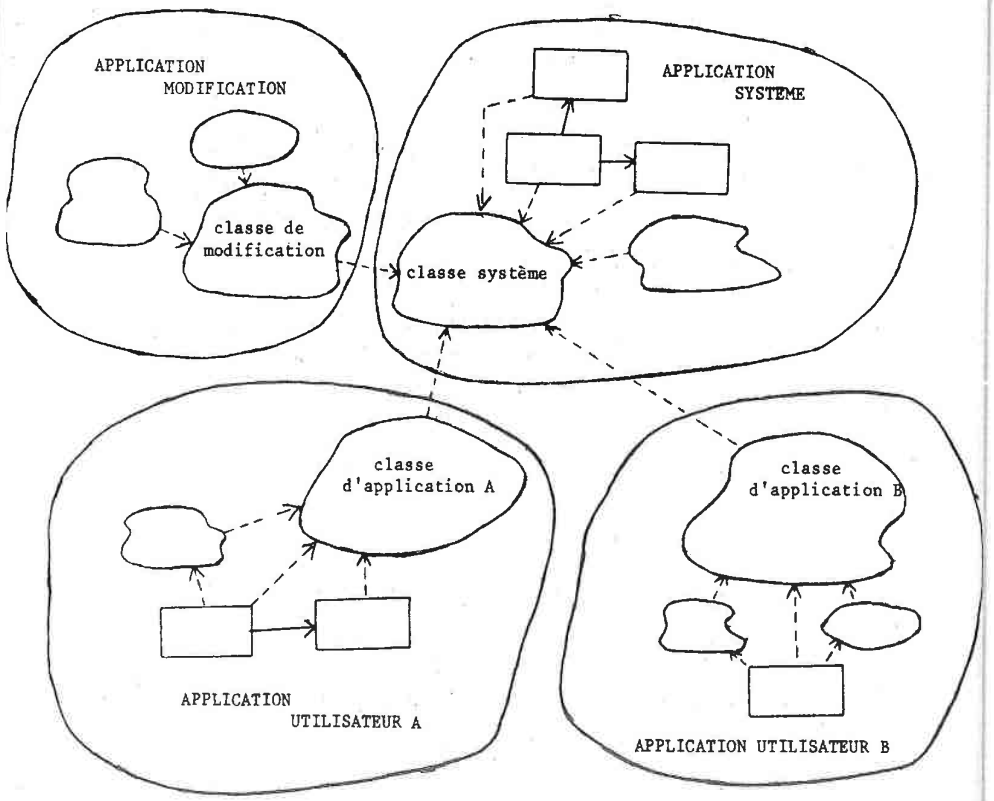
La classe d'application est utilisée implicitement par toutes les unités de nomenclature de l'application : les objets qui y sont déclarés sont donc permanents pour l'application.

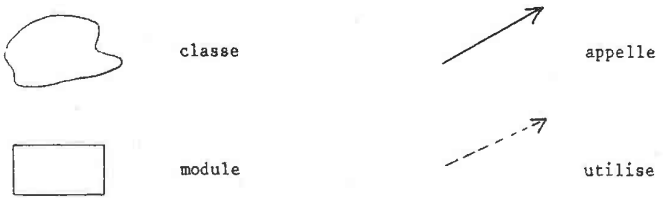
Une première application particulière est l'application système qui contient les modules propres au système CIVA (compilateur, éditeur de liens ...).
Cette application possède elle aussi une classe d'application appelée classe système qui est utilisée implicitement par toutes les autres classes d'application.
Cette classe contient :

- les déclarations des unités du système
- les déclarations de tous les paramètres du système (paramètres d'exploitation, indicateurs de contrôle d'exécution ...)
- les déclarations des procédures de gestion des applications qui sont communes à tous les utilisateurs (cf : chapitre 2)

Une deuxième application particulière est l'application modification.
Cette dernière réunit les classes qui contiennent les déclarations de certains paramètres et de certaines procédures dont l'accès par les utilisateurs est protégé. Le rôle de ces procédures est de permettre aux utilisateurs de notre projet de réaliser des opérations de mise à jour de leurs applications ou même de l'application système (cf : chapitre 2).
Par exemple, un utilisateur pourra supprimer des déclarations de la classe de l'application dans laquelle il travaille par l'appel d'une de ces procédures. Cette application, comme toutes les autres, possèdent une classe d'application appelée classe de modification qui utilise donc implicitement la classe système.

Il nous est possible de schématiser toutes les entités que nous venons de présenter dans la représentation suivante :





Dans la suite de cet exposé, nous utiliserons la terminologie suivante :

- soit E l'ensemble des unités d'une application :

$$E = C \cup M \cup MM$$

où C est l'ensemble des classes déclarées dans la classe d'application,
M est l'ensemble des modules déclarés dans la classe d'application,
MM est l'ensemble des métamodules déclarés dans cette même classe.

- Nous noterons également :
  U la relation "utilise"
  A la relation "appelle"
  $A la relation "$appelle"

I.3. LES RELATIONS EXTERNES A UNE APPLICATION (DERNIAME [8] : § I.II)

Les relations "utilise", "appelle" et "$appelle" que nous venons de rappeler ne lient entre elles que des unités définies dans une même application.
Ces relations permettent d'assurer une bonne protection des informations entre les différentes applications : un utilisateur d'une application A ne peut pas accéder aux informations d'une application B. Cette restriction peut être contraignante pour les utilisateurs qui peuvent être amenés à utiliser des informations ou des traitements déjà décrits dans une autre application.

Pour cette raison, dans le projet CIVA, nous avons autorisé un certain partage des informations à condition que le propriétaire d'une application en ait donné l'autorisation préalable d'une façon explicite. Ainsi, le propriétaire de l'application B, lorsqu'il créé des unités dans son application peut indiquer au système CIVA que ces nouvelles unités sont locales à son application ou bien qu'elles peuvent être utilisées dans une (ou plusieurs) autre(s) application(s) ou même dans toutes.

Exemple : le module M1 est déclaré dans la classe d'application A de la façon suivante :

```
module M1 ;
utilise C de B ;
S = E + F ;
finmod ;
```

le module M1 utilise une classe C qui est déclarée dans la classe de l'application B,

module M2 ;
appel M *de* B ;
utilise X ;
M *de* B ;
$MM *de* B ;
finmod ;

Le module M2 utilise la classe X de son application A. Il appelle le module M déclaré dans la classe de l'application B et le métamodule MM déclaré également dans cette classe.

Remarque générale sur ces relations externes

Un utilisateur d'une application A peut employer une unité Y déclarée dans la classe de l'application B en utilisant les relations externes que nous venons de présenter. L'emploi d'une de ces relations externes dans une unité X de l'application A est équivalent à une déclaration locale à cette unité X : l'identificateur "Y de B" ainsi désigné a une portée limitée à l'unité X. La durée de vie de l'unité externe Y dans l'application A est donc limitée à la durée d'activation de cette unité X.

I.4. NOTIONS DE MODULE ET DE CLASSE DE COMMANDE (DERNIAME [8] : § I.8.)

I.4.I. - Définition du module de commande

Lorsque l'utilisateur désire travailler dans son application, que ce soit demander l'exécution d'une ou plusieurs unités de traitement, ou bien réaliser des modifications dans le contenu de l'application, il écrit un module particulier appelé module de commande et il demande son exécution par le système CIVA.
Le module de commande correspond donc au paquet de cartes de commande auquel nous sommes habitués pour décrire les phases de mise au point ou d'exploitation d'un travail. Il regroupe les commandes formulées par l'utilisateur à destination du système CIVA. Le module de commande est donc un moyen de communication entre l'utilisateur d'une part et le système d'autre part. Son exécution correspond généralement à l'exploitation d'une chaîne de traitement.

I.4.2. - Contenu général du module de commande

Nous venons de présenter les modules de commande comme des unités particulières d'une application qui permettent l'exploitation des modules de l'utilisateur

Permettre l'exploitation des programmes ne signifie pas seulement offrir à l'utilisateur la possibilité de gérer ses programmes, d'en lancer l'exécution et de gérer ses fichiers. L'utilisateur doit également pouvoir faire des calculs, affecter des valeurs à des paramètres d'exploitation, agir sur l'exécution et l'enchaînement des programmes.
Ces possibilités sont parfois réalisées dans certains langages de commande : procédures cataloguées, utilisation de clés ...(cf langage de commande sous SIRIS 7 [22] , Operating System EXEC 8 [38]).
Comme un langage de programmation, le langage de commande doit donc contenir des instructions d'affectation à des variables de commande, des instructions conditionnelles, des instructions de saut .... Or tous ces outils sont déjà utilisés pour l'écriture des modules ordinaires. Il est donc très important que ces mêmes outils puissent être retrouvés et sous la même forme dans le module de commande. Ainsi nous aurons atteint le but que nous nous sommes fixé, à savoir avoir un langage unique qui permet de décrire des traitements que ce soit au cours de l'analyse, de la programmation ou de l'exploitation.

L'utilisateur désirant exploiter, modifier ou même créer une application particulière indique en tête du module de commande le nom de cette application.
Ensuite l'utilisateur peut soit créer une classe de commande (voir sa définition ci-après) soit utiliser une classe existant déjà : dans ce cas, il indique le nom de la classe de commande utilisée. Peuvent suivre, éventuellement, des demandes de modification de la classe d'application : introduction de nouvelles unités dans l'application, suppression d'unités existant déjà, mise à jour des textes des unités créés...Toutes ces demandes correspondent à des appels de procédures de modification.
Comme dans tout autre module, l'utilisateur peut également déclarer des identificateurs locaux au module de commande : dans ce cas, la durée de vie des objets ainsi créés sera égale à celle du module de commande.
Enfin, l'utilisateur peut utiliser des instructions CIVA dans ce module.
Ces instructions sont de type :

- instruction d'affectation,
- instruction conditionnelle simple
- des table de décision et de sélection...
- des appels de module : l'appel d'un module de l'application

dans le module de commande correspond à la demande d'éxécution d'une unité de traitement. Un module appelé dans un module de commande sera nommé module directeur.

I.4.3. - Définition de la classe de commande

Un module de commande contient donc des instructions CIVA avec quelques restrictions (pas d'instruction itérative "pour chaque", par exemple). Il utilise des objets locaux au module de commande, des objets communs à celui-ci et à toutes les unités de l'application (les objets définis dans la classe d'application). Il peut également utiliser des objets communs au module de commande et à quelques modules de l'application : ces objets sont alors décrits dans une classe particulière appelée classe de commande. La classe de commande permet alors d'échanger des valeurs entre le module de commande et certains autres modules de l'application (ceux qui utilisent cette classe). Elle sert, en quelque sorte, de "boite aux lettres" entre l'exploitation (exécution d'un module de commande) et la programmation (exécution d'une unité de traitement).
C'est à cause de ce rôle particulier que la classe de commande est distinguée des autres classes dans la nomenclature CIVA.

L'utilisation d'une classe de commande dans un module de commande est explicite. Pour pouvoir utiliser les déclarations d'une classe de commande, le module de commande doit contenir une instruction déclarative d'utilisation de classe de la forme :

UTILISE <Nom de la classe de commande> ;

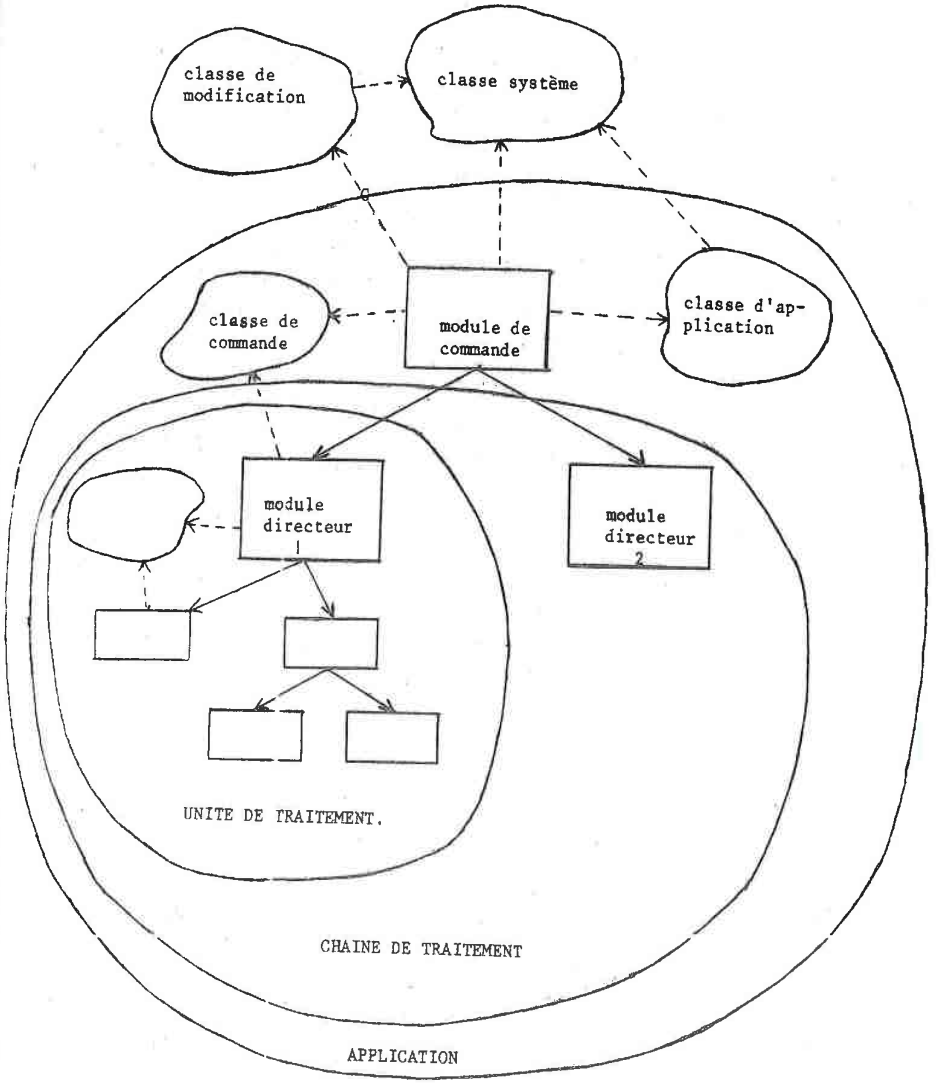
Cette instruction permet au programmeur d'utiliser tous les identificateurs déclarés dans la classe de commande.
Le module de commande ne peut utiliser explicitement que cette classe.

remarque

A part ce rôle distinct, la classe de commande n'a pas de propriété particulière. Comme les autres classes, elle peut elle-même utiliser d'autres classes qui définissent des objets utilisables dans le module de commande.

I.4.4. - Liaisons du module de commande avec les autres unités

On peut représenter les liens du module de commande avec les autres entités définies dans le projet CIVA par le schéma suivant :



I.4.5. - Les objets utilisés par le module de commande

Le module de commande peut utiliser des objets élémentaires ou structurés déclarés à trois endroits différents.

- Les objets peuvent être déclarés dans le module de commande lui-même. Dans ce cas, ils lui sont locaux et ils se déclarent comme ceux de n'importe quel autre module . (DERNIAME [8] : § 4.22.).

- Les objets peuvent être déclarés dans la classe d'application. Alors, ils sont permanents dans le système et peuvent être utilisés d'un module de commande à l'autre. La vie d'un objet de la classe d'application commence à l'instant même de sa création (cf la procédure CREER en 2.4.2.) et se termine lorsque l'utilisateur décide de le supprimer (cf la procédure SUPPRIMER en 2.5.4.I.).

- Enfin, les objets utilisés par le module de commande peuvent être déclarés dans une classe de commande. Cette classe de commande devra être utilisée explicitement par le module de commande.

exemple

contenu du module de commande :

```
                    ≷
                CREER ;
texte source de  classe com CCOM ;
la classe de     A entier ; B réel ;
commande         .....
                 finclasse ;
            UTILISE CCOM ;
```

CO l'appel de UTILISE entraine que les objets A et B sont utilisables dans le module de commande CO ;

```
            C complexe ;
```

CO le module de commande déclare un objet élémentaire de type complexe. Cet objet est local CO ;

```
            CREER (D décimal 99V9 ; ) ;
```

CO l'appel de cette procédure permet de déclarer l'objet D dans la classe d'application. D peut alors être utilisé dans n'importe quel module de commande CO ;

```
                    ≷
```


I.4.6. - Les instructions CIVA du module de commande

Un module de commande peut contenir les instructions CIVA suivantes :

- l'instruction d'affection (BENAMGHAR [3])

Cette instruction permet d'affecter une valeur à un objet utilisable dans un module de commande. Les règles concernant cette instruction sont les mêmes que celles applicables aux instructions d'affectation utilisées dans les autres modules. Toutefois, dans un module de commande on n'autorise pas le traitement des files et des ensembles.

- l'instruction SORTIR (DERNIAME [8] : § 6.4.)

Cette instruction permet de mettre fin à l'exécution du module de commande, lorsque cette fin ne correspond pas à la fin des instructions du module.

- l'instruction conditionnelle SI

Le module de commande peut également contenir l'instruction conditionnelle du langage CIVA : l'instruction SI. Comme dans les autres modules, les instructions SI imbriquées sont autorisées.

- les boucles classiques (DERNIAME [8] : § 6.5.)

L'instruction décrivant une boucle classique est l'instruction POUR. Elle permet à l'utilisateur de décrire un calcul itératif dans son module de commande. Ce calcul itératif ne peut correspondre au traitement global d'une file ou d'un ensemble, car ces types d'objet ne sont pas admis à ce niveau. Comme dans les autres modules, les instructions POUR imbriquées sont autorisées.

- les tables de décision et de sélection (AUBRY [1] , DERNIAME [8] Chapitre 5).

Dans le langage CIVA, ont été définies les tables de décision et les tables de sélection, celles-ci étant une écriture simplifiée des premières. Dans un module de commande, ces tables pourront être également utilisées sous une forme simplifiée : par exemple, on n'autorise pas les valeurs +V ou +F dans une entrée de condition. Dans une entrée d'action on ne peut pas avoir à la fois un numéro d'ordre et une instruction CIVA....

Ces simplifications sont été faites pour la raison simple que le module de commande est interprété par le module initial (voir sa définition en I.3.) alors que les autres modules sont compilés. Le module initial, à la différence du compilateur, ne fait aucune optimisation de la traduction de la table.

Exemple d'utilisation

1) Composition conditionnelle d'une chaîne de traitement

contenu du module de commande

```
UTILISE CLASCOMMANDE ;
```

CO la classe de commande contient les définitions des conditions CI, C2 et C3 qui sont utilisées dans le module de commande CO ;

module d'initialisation de la chaine ;

CO l'appel de ce premier module directeur permet de positionner les booléens représentant les conditions C1, C2 et C3 suivant certaines conditions CO ;

```
CONDITIONS    SINON
C1|V|F|V ;
C2|V|F|V ;
C3|V|-|F ;

ACTIONS
UT1 ; |1|-|-|1 ;
UT2 ; |3|2|*|2 ;
UT3 ; |2|1|*|3 ;
FINTABLE ;
FINMOD ;
```

Si les trois conditions C1, C2 et C3 sont vraies, la chaîne de traitement représentée par le module de commande est formée des unités de traitement UT1, UT3 et UT2 dans cet ordre.
Si C1 et C2 sont faux, la chaine de traitement est formée des unités de traitement UT3 et UT2.
Si C1 et C2 sont vrais et C3 faux alors la chaine de traitement est formée des unités UT2 et UT3. L'ordre d'exécution des unités de traitement est indifférent.
Enfin, dans tous les autres cas, la chaine de traitement est formée de UT1, UT2 et UT3 dans cet ordre.

2) contrôle de l'exécution d'une unité de traitement

contenu du module de commande :

```
APPLICATION X ;
unité de traitement A ;
```


CO l'appel de ce module entraine l'exécution de l'unité de traitement A Cette unité de traitement utilise la classe système dans laquelle sont déclarés les booléens DBENT qui est positionné à vrai en cas de débordement entier ; DBFLT qui est positionné à vrai en cas de débordement flottant, IHLLECT qui représente le débordement de file pour une file en position d'émetteur CO ;

```
SELECTION
DBENT et non IHLLECT | DBFLT et non IHLLECT | IHLLECT ;
     A1              |       A2             |  A3 ;
SINON A4 ;
```

```
A1 = module impression 1 ; sortir ;
A2 = module impression 2 ; sortir ;
A3 = module impression 3 ; sortir ;
A4 =
     suite de l'exécution de la chaine de traitement
     FINMOD ;
```

L'utilisateur lance l'exécution de l'unité de traitement A qu'il est en train de tester.
Si au cours de cette exécution, le système détecte un débordement de file, le booléen IHLLECT est positionné à vrai. Le contrôle est alors rendu au système CIVA qui interprète la table de sélection et exécute l'action A3. Il lance alors l'exécution du module d'impression 3 et l'interprétation du module de commande est terminée.
Par contre, si au cours de l'exécution de l'unité de traitement A, le système détecte un débordement entier, alors c'est le module d'impression 1 qui est exécuté. Dans le cas d'un débordement flottant, le module d'impression 2 est exécuté.
Si ces trois anomalies ne se produisent pas, l'interprète exécute l'action A4 : il continue l'interprétation du module de commande, c'est-à-dire l'exécution de la chaine de traitement.

I.5. NOTION DE MODULE INITIAL (DERNIAME [8] : § I.9.)

Le module de commande permet donc à l'utilisateur d'exploiter une chaine de traitement composée d'une ou plusieurs unités de traitement.

I.5.I. - Définition du module initial

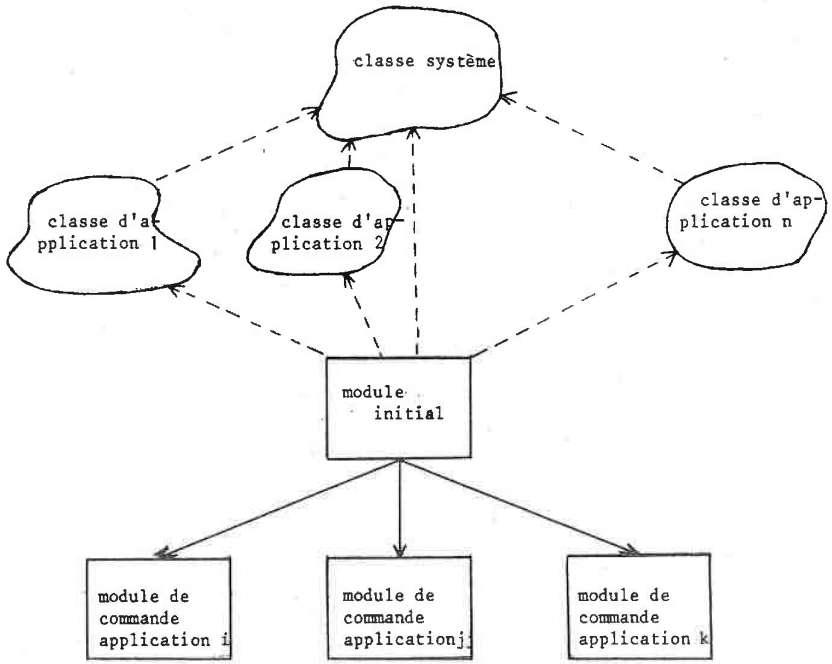
Alors que les unités d'une application, qui représentent des traitements répétitifs, sont compilées, le module de commande représente par contre des traitements qui varient d'une exploitation à l'autre.
Ce module sera donc interprété par le système CIVA qui, lui, est permanent et que nous appellerons module initial ou interprète des modules de commande noté M_o.
C'est donc à ce module initial que sont soumis les travaux à exécuter, représentés par les modules de commande. Ce module initial constitue donc un niveau intermédiaire, permanent, entre les applications CIVA et le système d'exploitation.
Afin d'avoir accès à tous les objets permanents du système, le module initial utilise la classe système. Il utilise également les classes des différentes applications afin de pouvoir les mettre à jour suivant les demandes formulées par l'utilisateur.

I.5.2. - Rôle du module initial

Le module initial a pour rôle essentiel de créer un module de commande, de l'interpréter (c'est-à-dire de le traduire et de l'exécuter instruction par instruction), et de le détruire.
La première instruction que doit rencontrer le module initial dans un module de commande est un appel de la procédure APPLICATION (cf en 2.4.I.) qui lui indique le nom de l'application à laquelle appartient le module de commande. Grâce aux procédures de modification ou du système, le module initial va pouvoir mettre à jour la classe application. Il peut également affecter des valeurs aux paramètres d'exploitation déclarés dans la classe système.

I.5.3. - Représentation des liaisons du module initial

On peut également représenter les liaisons du module initial avec les autres entités définies dans le projet CIVA par le schéma suivant :



2

LE MODULE DE COMMANDE

ET

LA GESTION DES APPLICATIONS

Les rappels du chapitre I sur les notions devéloppées dans le projet CIVA nous permettent de constater que l'application peut être considérée comme la plus grande entité connue de notre système.

A chaque application est associée une classe d'application unique qui représente cette application. Dans ce chapitre nous allons voir quelles sont les liaisons de cette classe d'application avec le module de commande et comment il est possible au module initial d'assurer automatiquement une partie de la gestion des applications.

Dans un premier temps, nous regarderons en quoi consiste la gestion des applications dans un système classique. Puis, nous définirons les bases de la gestion des applications mise en place dans le système CIVA.

2.I. LA GESTION DES APPLICATIONS DANS UN SYSTEME CLASSIQUE

La gestion de toutes les applications est la pièce maîtresse des activités opérationnelles d'un centre de traitement de l'information. Nous allons donc analyser, dans un premier point, les composantes de cette gestion. Ensuite, nous exposerons comment cette gestion est réalisée actuellement dans la plupart des centres de traitement de l'information.

2.I.I. - Les composants de la gestion des applications (BURNIAT [39])

La gestion des applications dépend de trois grands moyens :

- la mise en oeuvre des programmes élaborés par l'utilisateur,
- le fonctionnement des diverses unités de l'ordinateur, c'est-à-dire du matériel,
- l'intervention des programmes du constructeur parmi lesquels le superviseur.

Le déroulement des programmes, leur bonne exécution, la façon de préparer celle-ci, tout cela est étroitement lié à divers facteurs matériels tels que :

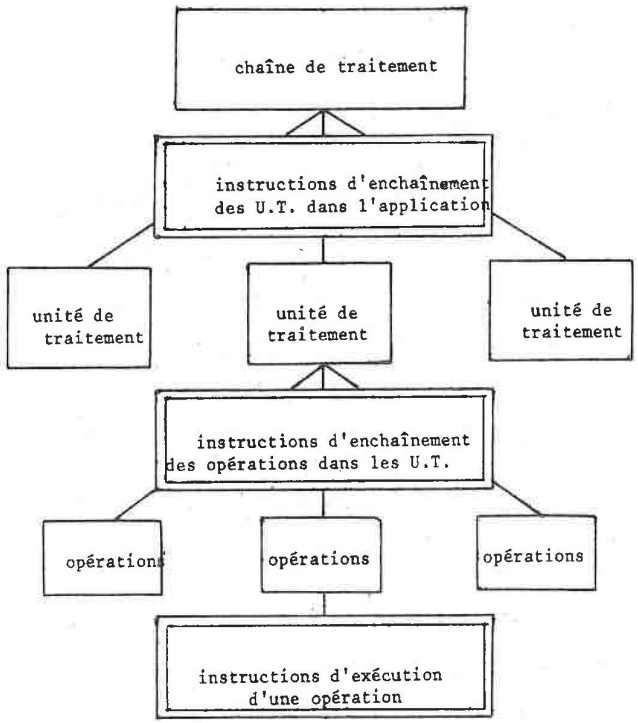
- la connaissance des applications elles-mêmes, c'est-à-dire des différentes phases de leur réalisation, de leur délai d'achèvement et des volumes à traiter,
- les variations dans le flux des informations,
- la possibilité de travailler en simultanéité (monoprogrammation ou multiprogrammation) et la possibilité de répartitions des activités en systèmes ON-LINE et OFF-LINE.

2.I.2. - Présentation du système classique de gestion des applications

Une application, au sens général du terme, peut être considérée comme un ensemble fini de sorties, une sortie étant une collection de données fournie par le système de traitement de l'information (HERSCHUH [I4] :§5.2). L'application est composée de la réunion de sous-ensembles appelés chaînes de traitement. Chaque chaîne de traitement a pour rôle de fournir des sorties ayant même périodicité. Une chaîne peut elle-même être découpée en subdivisions appelées les unités de traitement. Une unité de traitement est un ensemble qui se forme dans une application selon des critères variables et notamment selon l'objectif poursuivi. L'unité de traitement peut elle-même se diviser en opérations élémentaires, chaque opération étant une transformation d'une ou de plusieurs notions d'entrée pour obtenir une notion différente, le résultat de l'opération.

2.I.2.I. Les instructions d'exploitation

L'énumération des différents facteurs de la gestion des applications que nous avons vu au paragraphe précédent justifie l'utilisation d'instructions d'exploitation connues à la fois par les utilisateurs et par le service d'exploitation.
En effet l'exécution d'un programme de l'utilisateur, les manipulations techniques (changement de support par exemple), le recours à des programmes de service et d'une manière générale l'exercice d'une activité quelconque réclame, de la part des utilisateurs des directives claires et précises pour l'exploitation de leurs programmes.
Le responsable de l'exploitation est amené à fournir à l'ensemble des utilisateurs des consignes pour l'exploitation de tous leurs travaux : par exemple chaque utilisateur doit indiquer le planning de l'enchaînement de ses unités de traitement dans le cadre de l'exécution d'une application. Le but de ces consignes d'exploitation est donc de définir sans ambiguïté les différentes tâches à accomplir afin de simplifier l'enchaînement et l'exécution des travaux. Ces consignes sont fournies par l'utilisateur à différents niveaux comme l'indique le schéma ci-dessous :



A partir des instructions d'enchaînement fournies par les utilisateurs au niveau de l'application et au niveau de chaque unité de traitement, le service d'exploitation peut alors organiser les traitements sur l'ordinateur pour la journée, ou bien pour la semaine, afin de répartir la charge des travaux. Pour cela, le responsable d'exploitation regroupe les opérations à effectuer dans la journée et crée un document appelé feuille de route qui indique aux opérateurs les différentes opérations qu'ils ont à réaliser et l'ordre dans lequel elles doivent être réalisées.

2.I.2.2. Les dossiers de l'utilisateur

Nous pouvons remarquer que ce système oblige l'utilisateur à conserver, pour chaque application, toute une documentation supplémentaire qui lui fournit les consignes pour l'exploitation de ses travaux. Cette documentation est regroupée dans les dossiers que nous allons analyser. Ces dossiers sont décrits dans (REIX [4I]).

- le dossier d'application

Ce dossier permet de renseigner l'utilisateur sur le fonctionnement général de son application. Il contient en particulier :

a) un document qui définit la matière traitée par l'application et son rôle par rapport aux autres applications du système d'information.

b) un document indiquant globalement le matériel nécessaire à l'exploitation de l'application (pour chaque fichier, par exemple, figurera le nom du (ou des) support utilisé).

c) la découpe de l'application en unités de traitement ainsi que les relations qui lient ces unités de traitement entre elles et leur enchaînement.

- le dossier d'unité de traitement

Ce dossier permet de regrouper toutes les instructions qui concernent l'exploitation d'une unité de traitement. Il contient donc :

a) une description générale de l'unité de traitement avec ses relations avec les autres unités de l'application.

b) la division de l'unité en opérations élémentaires.

c) pour chaque opération, les besoins en matériel nécessaires à l'exploitation de l'opération ainsi que l'état du programme dans le système (état source, compilé ou édité...).

Grâce à ces documents, l'utilisateur peut donc fournir tous les renseignements nécessaires au service exploitation.

2.I.3. - Critique de la méthode

L'avantage de cette méthode, qui est beaucoup utilisée, est de fournir au service d'exploitation un planning complet des opérations à effectuer et de fixer les relations existant entre ce service et tous les utilisateurs du centre de traitement de l'information. Mais on peut relever deux inconvénients importants.

D'une part, l'utilisateur est obligé de gérer toute une documentation, qui peut être très importante, pour pouvoir exploiter ses applications. De plus, en cas de modification, adjonction ou suppression d'unités de traitement, l'utilisateur est amené à mettre continuellement à jour cette documentation, ces opérations pouvant même l'amener à refaire complètement ses dossiers. Cette méthode est donc assez lourde à employer mais son utilisation a été répandue car elle correspond à la seule possibilité offerte aux utilisateurs par des systèmes qui n'assurent pas la gestion automatique des unités formant une application et qui ne leur offrent aucun service documentaire.

Le second inconvénient de la méthode peut être décelé d'autre part: en effet, pour que cette méthode soit rentable, il faut que toutes les exécutions de travaux soient connues à l'avance, prédéterminées de façon statique afin que le service d'exploitation puisse constituer un planning de fonctionnement. Or il s'avère dans la réalité que certaines exécutions sont conditionnelles et dépendent des résultats des opérations précédentes. La charge du système peut donc varier considérablement par rapport aux prévisions : à certains instants, elle sera très faible car un travail prévu ne pourra être effectué, et à d'autres moments, elle sera trop forte car certains travaux non prévus à l'avance devront être exécutés.

2.2. LA GESTION DES APPLICATIONS DANS LE PROJET CIVA

Dans notre système, les unités formant une application sont liées entre elles par des relations définies dans le langage : les relations "utilise" et "appelle". Si le système peut relever ces relations au cours de la création des unités dans l'application, alors une bonne partie de la gestion des applications peut être automatisée. C'est sur cette possibilité que s'appuie le système CIVA pour gérer les applications, en partie à la place de l'utilisateur. Il faut ajouter que cette gestion n'est complète que si elle est facilitée par l'emploi d'un système documentaire. Ce système automatique de documentation sera défini par ailleurs.(BARTHELEMY [2] , PHILIPPE [2]).

En partant de la critique exposée précédemment, nous avons mis en place une gestion des applications dans le projet CIVA en conservant les avantages que pouvait tirer l'utilisateur de la méthode classique et en essayant de réduire les inconvénients de celle-ci.

2.2.I. - Les relations de l'utilisateur avec le service exploitation

Dans la méthode classique, ces relations sont matérialisées par l'utilisation des instructions d'exploitation. Ces instructions, qui doivent être suffisamment claires et précises, sont nécessaires pour fixer une fois pour toutes les relations entre les utilisateurs d'une part et le service d'exploitation d'autre part et elles constituent un avantage certain.
Dans notre système, nous pouvons penser que ces instructions continueront d'être utilisées et nous n'en parlerons pas d'avantage ici, le but que nous nous sommes fixé étant d'apporter une aide maximale à l'utilisateur quant à ses problèmes de maintenance de programmes ou de documentation.

2.2.2. - L'automatisation des dossiers de l'utilisateur

L'utilisateur a besoin d'une documentation pour pouvoir gérer ses applications. Il est naturel de penser à automatiser la production de tels dossiers.

- Par exemple, le dossier d'application contient un document définissant la matière traitée par l'application. Ce document peut être entré dans l'ordinateur une fois pour toutes, sous une forme précise, et être géré par le système automatique de documentation. Ce dossier contient également les indications sur le matériel necéssaire à l'exploitation de l'application. Or lorsqu'un module appartenant à l'application a été traité par le système de compilation, par exemple, il est possible de savoir quelle est la place nécessaire à son implantation en mémoire centrale. De plus, si l'utilisateur indique au système quels sont les fichiers physiques utilisés, c'est-à-dire les supports de l'information manipulée par l'application, le système peut alors stocker ces renseignements et les communiquer à l'utilisateur lorsqu'il le désire.

- Reprenons l'analyse du contenu du dossier d'unité de traitement. Celui-ci contient la description des relations qui lient l'unité avec les autres unités de l'application. Or nous avons vu au chapitre I que ces relations que nous avons appelées "utilise", "appelle", et "$appelle" font partie du langage. Le système peut donc les relever dans les textes sources des programmes et les communiquer à l'utilisateur lorsque celui-ci le souhaite. Un deuxième document contient l'étude du découpage de l'unité de traitement : l'étude de ce découpage, pour le système CIVA, revient à l'étude du graphe formé par les relations liant les unités d'une part et les unités de l'application d'autre part.
Le système CIVA, aidé du système documentaire, peut donc automatiser les dossiers de l'utilisateur.

2.2.3. - L'intégration des commandes du système

Pour exploiter des traitements, l'utilisateur a besoin d'outils permettant :

- de déclarer des variables dites d'exploitation (par exemple, temps maximum d'exécution, variables de contrôle du déroulement des exécutions des programmes ...)
- d'exprimer des actions à réaliser sur ces variables (par exemple, calcul, affectation, instructions conditionnelles...)
- de lancer l'exécution de certaines fonctions d'exploitation bien déterminées (par exemple, compilation d'un programme, lancement d'une exécution, listage du contenu d'un fichier....).

Pour décrire les actions à entreprendre par un système classique, l'utilisateur emploie un langage d'exploitation différent du langage de programmation. Ce langage dépend étroitement du système considéré et une changement de ce dernier oblige l'utilisateur à réexaminer les commandes nécessaires à l'exploitation de ses chaînes de traitement.
De plus dans un tel système bilangage (ou même multilangage), les informations d'exploitation sont décrites de façon différente des informations utilisées dans les traitements et généralement il n'y a pas d'interaction possible par partage d'informations entre la commande du système et l'exécution des programmes.

Le but que nous avons poursuivi en concevant les modules de commande était de réaliser en partie l'intégration des commandes du système. Nous avons vu au chapitre I que le module de commande était un module particulier d'une application. Comme tout module, il contient des instructions et des déclarations d'objets d'exploitation.

- La description des objets d'exploitation est en tout point identique à celle des objets utilisés dans les modules de traitement. Nous avons donc un système unique de description des informations, que celles-ci soient utilisées pour l'exploitation ou pour la programmation. De même les instructions qui opèrent sur ces objets d'exploitation sont celles du langage de base.

- De plus, grâce à la classe de commande et à la classe d'application, le système CIVA autorise un certain passage des informations entre un module de commande et les unités de traitement : il y a donc possibilités d'influencer les commandes du système à partir des programmes.

- Il faut toutefois noter que l'utilisateur, pour exploiter ces programmes, a besoin de fonctions particulières d'exploitation. Les outils qui implémentent ces fonctions ne sont accessibles que dans un module de commande et c'est pourquoi ce dernier contient obligatoirement des instructions particulières. Mais, dans notre système, les commandes de ces fonctions sont simplifiées. Par exemple, lorsqu'un utilisateur désire lancer l'exécution d'une unité de traitement, il indique seulement le nom du module directeur et le système CIVA gère automatiquement les transitions d'état des unités : il lance les opérations de compilation et de reliure et ceci de façon transparente à l'utilisateur. Néanmoins, pour celui-ci, les problèmes de contrôle restent bien qu'ils soient également simplifiés : l'utilisateur n'a à se soucier que de la présence des différentes unités formant l'application, c'est-à-dire de l'accessibilité du texte source de ces unités au module initial.

2.2.4. - Critique de la méthode

L'avantage de cette gestion en partie automatisée est important pour l'utilisateur : celui-ci n'est plus obligé de mettre à jour une documentation importante à chaque fois qu'il apporte des modifications à ses applications, tous les renseignements utiles pouvant être fournis par le système. De plus, le lancement de l'exécution simplifié et la gestion des transitions d'état automatisée permettent de lancer des unités de traitement de façon dynamique. En effet, l'utilisateur peut manipuler des variables soit du système, soit propres à ces traitements, faire des calculs sur ces variables et lancer des exécutions si certaines conditions sont remplies.

On peut toutefois noter que cette gestion automatisée peut se révéler moins souple pour le service d'exploitation qui ne peut donc pas planifier à l'avance les traitements à effectuer, puisque ceux-ci peuvent être conditionnels. Mais la gestion automatisée des traitements n'empêche pas l'utilisateur de prévoir le déroulement des opérations et de fournir au responsable de l'exploitation une certaine prévision des opérations effectuées.

De plus, le système d'exploitation peut être muni d'un analyseur de charge qui permet de régulariser le fonctionnement global de l'installation.

Dans la suite de ce chapitre, nous allons analyser les différentes procédures du système CIVA que l'utilisateur peut appeler à partir d'un module de commande. Ces procédures ne sont pas les seuls services mis à sa disposition pour la gestion des applications. En effet, un système documentaire doit accompagner chaque application de l'utilisateur. L'utilisation de ce système sera décrite dans (BARTHELEMY [2] , PHILIPPE [2]).

2.3. LES MECANISMES DE PROTECTION DES OBJETS DEFINIS DANS LE PROJET CIVA

On désigne par protection le contrôle du bon emploi de l'information et, d'une manière plus générale, des ressources dans un système informatique (FERRIE [32]). Un système de protection n'a pas comme rôle d'empêcher la production des erreurs, ou des malveillances, mais seulement leurs incidences sur les objets protégés.

La difficulté d'assurer une protection efficace tient aux multiples objets à protéger, ou tout au moins à surveiller, et aux nombreux accès aux objets définis, à savoir :
- les objets élémentaires, structurés ou de type file,
- les unités de nomenclature,
- les applications,
- les fichiers.

Pour chacune de ces catégories, il est donc nécessaire de définir des mécanismes de protection. Nous étudierons donc les mécanismes mis en place pour les applications. Puis nous nous intéresserons aux mécanismes de protection des objets primitifs et des unités de nomenclature. Le système de protection des fichiers fera l'objet d'un chapitre spécial (cf le chapitre 3).

2.3.I. - Les mécanismes de protection des applications

Le système de gestion des applications que nous avons voulu mettre en place dans le projet CIVA doit refléter la réalité fonctionnelle qui existe dans tout service informatique.

2.3.I.I. L'organisation fonctionnelle d'un service informatique

Si nous analysons la structure fonctionnelle d'un service informatique, nous pouvons constater qu'il existe deux catégories de personnel :

- Une première catégorie a pour rôle essentiel d'assurer la maintenance du système d'exploitation. Généralement cette équipe est dirigée par un responsable qui prend toutes les décisions de modification importante du système et qui en assure la pleine responsabilité.

- D'autre part, il existe des personnes travaillant sur des applications mises en place dans l'entreprise. Ces personnes sont groupées par équipes,une équipe ayant pour rôle de créer, de tester, d'exploiter et de maintenir une application particulière. Chaque équipe possède un dirigeant qui assure la responsabilité du fonctionnement de l'application.

Du fait de cette distinction fonctionnelle du personnel informatique, tout système automatisé de gestion des applications doit pouvoir refléter ces différents niveaux de responsabilités et de séparation des pouvoirs. Les mécanismes de protection des applications mis en place dans notre projet ont donc été créés en vue de représenter cette réalité fonctionnelle.

2.3.I.2. Les différentes phases de la vie d'une application

Au cours de sa vie, toute application passe par différentes étapes :

- La première action menée dans le cadre d'une application est la création : les personnels de l'équipe attachée à l'application réalisent l'analyse des traitements et introduisent les unités dans le système en vue de réaliser des tests avant de passer à l'exploitation courante.

- Lorsque toutes les unités définissant l'application ont été créées, testées et sont donc opérationnelles, celle-ci entre dans une deuxième phase qui est son exploitation : à périodes fixes et prédéterminées, une personne se charge du lancement de l'exécution des différentes unités de traitement et contrôle le bon fonctionnement général de cette exploitation.

- Toute application doit être évolutive pour s'adapter aux besoins de la clientèle informatique. Il est donc nécessaire de temps en temps, de faire passer l'application dans une phase intermédiaire qui est la maintenance. Nous désignons sous le mot "maintenance" l'ensemble des travaux qui sont effectués sur une application alors qu'elle était dans sa phase d'exploitation. Il s'agit soit d'entretenir les programmes, soit de corriger les erreurs, soit de les adapter en vue de nouvelles applications, ou de les améliorer (SAVARY [42]).

Or la modification d'un programme ou d'une définition d'une information peut entraîner des changements importants dans l'ensemble des unités de l'application. Par exemple, dans une application de facturation, le changement d'une constante représentant la valeur d'un taux de T.V.A. nécessite la recompilation de toutes les unités qui peuvent accéder à cette constante.

Généralement, c'est donc au dirigeant de l'équipe attachée à l'application de prendre les décisions de modification et d'assurer la responsabilité de la maintenance générale des programmes.

Enfin on peut remarquer que ces différentes phases existent également pour l'application système : la création correspond à ce que l'on appelle généralement la génération du système. L'exploitation du système est réalisée par l'équipe système. La maintenance est assurée par le constructeur avec contrôle du responsable de l'équipe système.

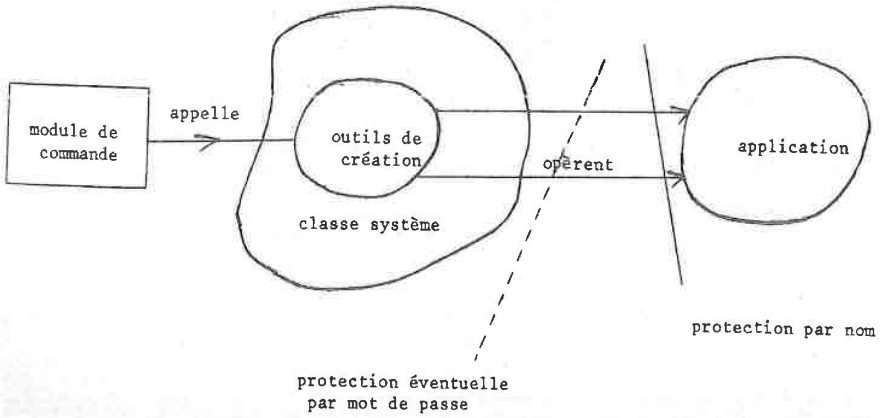
2.3.I.3. Conséquences sur l'aspect fonctionnel des modules de commande

Suite aux remarques précédentes, il est aisé de constater qu'il peut exister trois types de modules de commande d'un point de vue fonctionnel :

- Un premier type de module de commande permet de réaliser les opérations de création. Les demandes de ces opérations correspondent à l'appel de procédures accessibles à tous les membres de l'équipe attachée à l'application. Ces outils sont communs à toutes les applications et il n'est alors pas nécessaire de mettre en place une protection particulière pour eux : ces outils seront donc définis dans la classe système qui est utilisée implicitement par toutes les classes d'application.

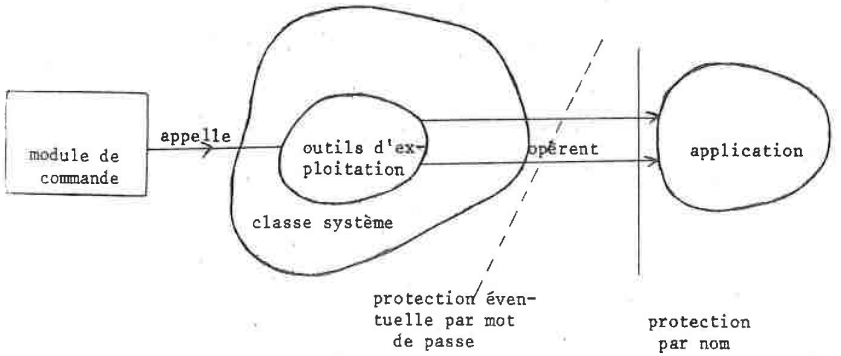
Toutefois l'exécution de telles procédures entraîne des créations de déclarations dans la classe de l'application. Or, si nous voulons représenter la séparation des pouvoirs des personnels du service informatique, il faut que cette classe d'application ne puisse être accessible qu'aux membres de l'équipe chargée de l'étude de l'application.

Pour cette raison, l'accès à une application est protégé par deux mécanismes de protection. Le premier mécanisme assure une protection par nom : seuls les utilisateurs connaissant le nom de l'application peuvent opérer dans celle-ci. Comme nous l'avons vu au chapitre 1, cette protection est une protection d'accès aux unités définies dans une application. Les relations "utilise", "appelle" ou "s'appelle" ne permettent de relier entre elles que des unités d'une même application. Tout accès venant de l'extérieur doit être préalablement autorisé. Ce mécanisme est permanent : il est toujours mis en place par le système (cf la procédure application en 2.4.I.). Si cette protection ne semble pas suffisante, le responsable de l'application peut demander au système de mettre en place un deuxième mécanisme qui assure une protection par mot de passe : au moment de la création de l'application, l'utilisateur déclare éventuellement un mot de passe qui doit être rappelé avant chaque exploitation de l'application. La mise en place de ce mécanisme est facultative et c'est le responsable de l'application qui décide si son application doit être protégée ou non par ce système de mot de passe.



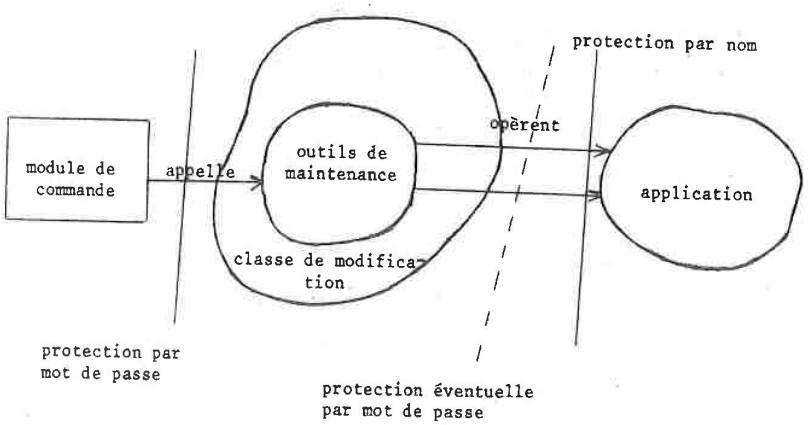
- Un deuxième type fonctionnel de module de commande permet d'assurer l'exploitation de l'application. Une demande d'exploitation peut être un appel de module directeur d'une unité de traitement ou un appel de procédure d'exploitation. Toute personne travaillant dans une équipe chargée du contrôle du fonctionnement d'une application doit pouvoir utiliser ces outils. Ceux-ci ne font donc pas l'objet d'une protection particulière et sont définis dans la classe système.

Toutefois nous pouvons remarquer, si l'application fait l'objet d'une protection par mot de passe mise en place au moment de sa création, toute demande d'exploitation doit être précédée de l'indication du mot de passe assurant cette protection.



- Un dernier type fonctionnel de module de commande permet d'exprimer les demandes de maintenance de l'application. Ces demandes correspondent à des appels de procédures de mise à jour de la classe de l'application. Or nous avons vu que certaines modifications pouvaient avoir des conséquences importantes sur les traitements définis dans l'application. Il est donc souhaitable que certains outils mis à la disposition du utilisateurs pour maintenir une application ne soient pas accessibles à tous ces utilisateurs. Seuls les responsables des applications doivent avoir la possibilité de les utiliser.

Pour cette raison, certains outils seront déclarés dans la classe système et donc accessibles à tout utilisateur, d'autres outils seront déclarés dans la classe de l'application modification. Cette application possède toujours un mécanisme de protection par mot de passe, mis en place au moment de sa création, c'est-à-dire de la génération du système. De ce fait, pour pouvoir accéder à un outil de maintenance, il est nécessaire d'indiquer le mot de passe de l'application modification.
De plus ces objets opèrent sur l'application concernée par la mise à jour. Si celle-ci est protégée par le deuxième mécanisme de protection, il faut également indiquer le mot de passe de cette dernière pour pouvoir réaliser les opérations de maintenance.



On peut remarquer que la protection mise en place pour les applications des utilisateurs est tout aussi valable pour l'application système : tout programmeur ne doit pas pouvoir modifier une partie du compilateur ou même de l'interprète des modules de commande.
Au moment de sa création, l'application système pourra également être protégée éventuellement par un mot de passe. Pour pouvoir opérer une modification dans les unités du système, il est alors nécessaire de connaître le mot de passe de l'application modification et le mot de passe de l'application système.

Toutefois, il faut noter que la classe système doit pouvoir être accéder par toute personne : la classe système est donc la seule unité du système CIVA qui n'est pas protégée par la protection éventuelle de l'application système. Tout utilisateur peut y créer des objets, en détruire d'autres ....

2.3.2. - La protection des objets et des unités

Comme dans tout système de programmation, la protection des objets élémentaires, structurés ou de type file définis dans le langage CIVA est une protection d'accès : un objet est protégé de l'extérieur s'il n'y a pas de fonction d'accès pour l'atteindre.
Les règles de portée des identificateurs que nous avons rappelées au chapitre I assurent la plus simple des protections : un objet local à un module ne peut être accédé par un autre module. Un objet déclaré dans une classe ne peut être accédé que par une unité qui utilise cette classe.

Les mécanismes de protection mis en place pour les unités d'une application se basent sur les types de relation entre ces unités et sur la portée des identificateurs.
L'identificateur d'un métamodule déclaré dans une classe a une portée limitée à cette classe : il ne peut donc pas être accédé par une autre unité qui n'utilise pas cette classe.
Généralement les unités sont déclarées dans la classe d'application : la portée de leurs identificateurs est donc limitée à l'application. Tout accès venant de l'extérieur est inopérant du fait qu'il n'y a plus de fonction d'accès à l'unité.
Toutefois nous avons vu en I.3. qu'il était possible d'enfreindre explicitement cette règle de protection. Seul le responsable d'une application peut décider, au moment de la création d'une unité, de la partageabilité de l'unité entre les autres applications présentes dans le système.

2.5. LES OUTILS DE CREATION

Pour permettre la création d'applications ou d'unités dans une application, l'utilisateur dispose d'outils particuliers qui se présentent sous la forme de procédures définies dans la classe système. Pour utiliser ces procédures, il suffit d'en écrire un appel dans le module de commande qui utilise implicitement cette classe système.

Celle-ci, comme nous venons de le voir, ne fait l'objet d'aucune protection particulière.
Nous allons définir les procédures de cette classe utilisables dans un module de commande en précisant leur rôle.

2.4.I. - La procédure APPLICATION

Son appel constitue la première instruction d'un module de commande et est obligatoire. Cette procédure permet au module de commande d'indiquer à quelle application il appartient et donc de fixer au système la classe d'application utilisée.

- Pour créer une nouvelle application, cet appel s'écrit :

APPLICATION (<nom de l'application> , NOUVELLE
[, (PASSE, <mot de passe>)]) ;

Le nom d'une application est une chaîne d'au plus IO caractères alphanumériques.
L'interprète, à la rencontre de cet appel, doit vérifier que le nom de l'application déclaré par l'utilisateur ne désigne aucune application existant déjà dans le système. S'il trouve un nom identique, l'interprète émet un message d'erreur à l'utilisateur et l'interprétation du module de commande est terminée. Par contre, s'il ne trouve pas de nom identique, l'interprète créé une nouvelle application et il donne comme valeur à la variable APPLICATION COURANTE de la classe système, le nom de l'application.
L'utilisateur a la possibilité d'indiquer un mot de passe à la création d'une application, mot de passe qui servira à sa protection. En effet, pour toutes les utilisations ultérieures, tout programmeur devra indiquer ce mot de passe au système CIVA. Un mot de passe est une chaîne d'au plus 8 caractères alphanumériques.

- Pour travailler dans une application déjà existante, l'appel de la procédure APPLICATION s'écrit :

APPLICATION (<nom de l'application>
[, (PASSE, <mot de passe>)]) ;

Dans ce cas, l'interprète vérifie que le nom indiqué désigne bien une application existante. Si c'est le cas, il vérifie alors que l'utilisateur a bien indiqué le mot de passe correct si une protection avait été demandée au moment de la création de l'application.

Il donne alors le nom de l'application comme valeur à la variable APPLICATION COURANTE.
Si le nom ne désigne aucune application ou si le mot de passe est incorrect, l'interprète envoie un message d'erreur à l'utilisateur et l'interprétation du module de commande est terminée.

2.4.2. - L'adjonction de déclarations dans la classe d'application

Une fois que l'utilisateur a indiqué dans quelle application il voulait travailler, le module de commande peut utiliser la classe d'application. Comme ce module utilise également la classe système, l'utilisateur va pouvoir, en appelant une procédure de cette classe, créer des déclarations dans la classe d'application. Ces créations vont être réalisées par l'apple de la procédure CREER.

2.4.2.I. Adjonction de déclarations d'unités

L'utilisateur peut tout d'abord créer des unités dans son application, que ce soit des classes, des modules ou des métamodules. Chaque unité est repérée par un identificateur qui est le nom de cette unité. Si une unité de même nom existe déjà dans l'application, l'adjonction n'est pas réalisée par l'interprète et un message d'erreur est envoyé : en effet, si l'utilisateur veut remplacer le texte source d'une unité déjà créée, il peut soit supprimer cette unité et la recréer avec le nouveau texte (voir la procédure SUPPRIMER en 2.5.4.I.), soit changer le texte de son unité (voir la procédure CHANGER en 2.5.3.I.I.). L'appel de la procédure CREER peut s'écrire de trois façons différentes, en ce qui concerne la création d'unités.

- CREER ;
  ≈ suite des textes sources des différentes unités à créer.

Par cet appel, toutes les unités dont le texte source suit l'appel de la procédure sont ajoutées à la classe de l'application et elles ne pourront pas être utilisées ou appelées par des unités appartenant à d'autres applications. Ces unités seront donc protégées des utilisations éventuelles demandées par des personnes travaillant sur d'autres réalisations.

La suite d'unités à créer s'arrête à la première ligne du module de commande qui n'est pas une déclaration de classe, de module ou de métamodule.

- CREER (COMMUNS) ;
  suite des textes sources des différentes unités à créer.

Par cet appel, l'utilisateur décide que ses unités sont utilisables par tous les utilisateurs, quelle que soit l'application sur laquelle ils travaillent. Donc toute unité pourra employer les unités ainsi créés (en utilisant les relations internes ou externes définies dans le langage).

- CREER (<liste de noms d'applications>) ;
  suite des textes sources des unités à créer.

Dans ce cas, l'utilisateur décide que ses unités ne sont utilisables que par des unités appartenant à des applications bien définies, dont les noms figurent dans la liste indiquée. Evidemment, elles pourront être employées par toute unité de l'application courante dans laquelle travaille actuellement l'utilisateur.
Dans la liste des noms d'application, chaque nom est séparé du suivant par une virgule.

Exemple

```
APPLICATION PERSONNEL ;
CREER (PAIE, FACTURE) ;
module ALPHA ; utilise classex ;
pour chaque x de liste faire
    A = A + 1 fpc ; finmod ;
```

L'unité ALPHA est créée dans l'application PERSONNEL. Elle ne pourra être employée que par les unités déclarées dans les applications PERSONNEL, PAIE et FACTURE.

Remarques

1) Si, au cours des utilisations ultérieures de son application, l'utilisateur se rend compte qu'il n'a pas donné la bonne protection à une ou plusieurs unités, il ne peut pas modifier directement cette protection. S'il désire changer la protection d'une ou plusieurs unités, la seule possibilité est de détruire la ou les unités correspondantes de l'application et de les créer à nouveau avec, cette fois-ci, la bonne protection. Cela l'oblige donc à avoir à sa disposition le texte source de cette ou de ces unités. Le système CIVA ne fournit aucune procédure permettant de modifier directement la protection d'une unité, par soucis d'efficacité.

2) Si, au cours de l'opération de création d'unités, le module initial rencontre une nouvelle unité désignée par un identificateur déjà présent dans la classe d'application, il émet un message d'erreur. L'unité est ignorée et le module initial lance l'opération de création de l'unité suivante. L'unité ainsi ignorée pourra être ajoutée à l'application par un nouvel appel de la procédure CREER dans un autre module de commande.

2.4.2.2. Adjonction de déclarations d'objets élémentaires ou structurés

La classe d'application contient également des déclarations d'objets primitifs. C'est également dans le module de commande que l'utilisateur va pouvoir créer de tels objets dans la classe d'application. Ces objets ainsi créés sont permanents dans le système, c'est-à-dire que leur durée de vie est supérieure en général à celle d'un module de commande.
La création d'un objet dans la classe d'application se fait par l'appel de la procédure CREER qui s'écrit :

CREER (<liste de déclarations>) ;

Chaque déclaration est séparée de la suivante par un point virgule qui est le séparateur habituel des instructions CIVA.
Si, au cours de l'analyse syntaxique d'une déclaration, le module initial relève une erreur, il émet un message à destination de l'utilisateur.

La déclaration est ignorée et l'interprétation du module de de commande se poursuit par l'analyse de la déclaration suivante :

Exemple

CREER (I entier ; J file (max = 10) car ;
A réel = 3.14 ;) ;

Cet appel crée 3 objets élémentaires dans la classe d'application. A partir de cet appel, ces objets sont utilisables dans tous les modules et dans toutes les classes de l'application et donc, en particulier, dans le module de commande.

2.4.3. - Adjonction d'une unité définie dans une autre application

Nous avons vu en 2.4.2.I. que, grâce à la procédure CREER, l'utilisateur pouvait ajouter dans son application, des unités nouvelles dont il vient d'écrire le texte source. Les unités ainsi ajoutées constituent des éléments permanents de l'application, car elles sont déclarées dans la classe d'application.
Nous avons également vu en I.3 que, grâce à l'utilisation des relations externes, l'utilisateur pouvait également ajouter des unités externes, déclarées dans d'autres applications, en écrivant dans le texte source d'une de ses unités, une des déclarations suivantes :

UTILISE <nom de classe> DE <nom d'application> ;
APPEL <nom de module> DE <nom d'application> ;
ou $ <nom de métamodule> DE <nom d'application> ;

Or, nous avons remarqué que l'unité externe ainsi employée avait une durée de vie au plus égale à celle de l'unité qui l'emploie. L'identificateur de l'unité externe est local à l'unité qui contient son occurence. Une telle utilisation ne correspond donc qu'à une exécution particulière d'un module de commande.

L'utilisateur peut également souhaiter que les unités externes qu'il emploie deviennent des objets permanents de son application. Dans ce cas, il écrira dans un module de commande, un appel de la procédure AJOUTER définie dans la classe système. Cette procédure a un rôle analogue à celui de la procédure CREER. Elle permet d'ajouter une déclaration dans la classe de l'application courante, c'est-à-dire de créer un objet permanent.

La différence entre ces deux procédures réside dans le fait que l'unité ainsi créée doit avoir été préalablement déclarée dans une autre application. De plus le propriétaire de cette autre application a dû donner son autorisation préalable.
L'appel de la procédure AJOUTER s'écrit :

AJOUTER (<nom d'unité 1>, <nom d'application>, { SEUL | ENVIRONNEMENT } [, <nom d'unité 2>]) ;

<nom d'unité 1> désigne une certaine unité v qui doit être déclarée dans l'application B de nom <nom d'application>.
Le module initial vérifie que l'unité v est effectivement déclarée dans l'application B et que le propriétaire de B a donné l'autorisation de l'utilisation de v. Si c'est le cas, le module initial prend une copie de v dans B et l'ajoute dans l'application courante A. Si ce n'est pas le cas, M_o émet un message d'erreur et interprète l'instruction suivante du module de commande.

Une adjonction pose toujours un problème d'homonymie. Ce problème doit être résolu par l'utilisateur : c'est-à-dire qui si une unité de nom <nom d'unité 1> figure déjà dans l'application en cours,il est nécessaire que l'utilisateur change le nom de l'unité v. Pour cela, il utilise le quatrième paramètre de la procédure AJOUTER. L'unité v désignée par <nom d'unité 1> dans l'application B, sera alors désignée par <nom d'unité 2> dans l'application courante A. Si l'utilisateur ne modifie pas le nom de l'unité v, un message d'erreur lui est communiqué.
Une fois l'adjonction réalisée, l'utilisateur pourra se servir de l'unité v comme de toute autre unité créée dans l'application courante. Cette utilisation se fera par l'intermédiaire de l'identificateur <nom d'unité 1> si aucune modification de nom n'a été nécessaire, ou par l'intermédiaire de <nom d'unité 2> si une modification du nom a été réalisée. L'unité v est alors déclarée dans la classe de l'application et est un objet permanent pour cette application.

L'unité v "utilise" , "appelle" ou"$appelle" certainement d'autres unités de l'application B dans laquelle elle a été déclarée.

L'adjonction de v dans A nécessite donc, en général, l'adjonction de toutes les autres unités de B, faisant partie de l'environnement de v, c'est-à-dire les unités appartenant à :

$\widehat{U \cup A \cup \$A}$ (v)

Si l'utilisateur emploie l'option SEUL, l'unité v, et seulement elle, sera ajoutée à l'application A.

Par contre, si l'utilisateur emploie l'option ENVIRONNEMENT, le module initial ajoute à A l'unité v et l'environnement de v précédemment défini. Pour chaque unité w de cet environnement se pose également le problème de l'homonymie : l'unité w n'est ajoutée à A que s'il n'existe pas déjà une unité de même nom déclarée dans la classe de l'application A. Si une telle unité existe, l'adjonction de w ne peut être réalisée et le module initial communique à l'utilisateur le nom de w. Cette unité pourra alors être ajoutée à A ultérieurement par un appel de la procédure AJOUTER avec modification de nom.

Remarque

D'un point de vue pratique, il faut enfin remarquer que l'utilisation de la procédure AJOUTER avec l'option ENVIRONNEMENT ne peut porter que sur une unité qui est déjà à l'état "compilé" dans son application origine afin que le module initial puisse connaître tout son environnement.

5. LES OUTILS DE MAINTENANCE D'UNE APPLICATION

Les outils de maintenance sont des services annexes introduits par le système CIVA pour renseigner et guider l'utilisateur dans l'élaboration de son application.

Un système de traduction doit toujours mettre à la disposition des programmeurs un ensemble de services leur permettant d'atteindre rapidement les objectifs qu'ils se sont fixés dans leurs programmes.

D'après (NEBUT [36]), ces services peuvent se situer à deux niveaux :

- Lorsqu'un programme est en cours de réalisation, le système doit offrir au programmeur les outils d'aide à la mise au point.
- Lorsqu'un programme est correct, l'utilisateur doit disposer d'un ensemble de services lui permettant d'étudier le comportement de ce programme dans l'ensemble de l'application.
- A ces deux niveaux, il paraît souhaitable d'ajouter un troisième niveau de services : ce dernier regroupe les services permettant d'assurer la maintenance des applications d'un service de traitement de l'information.

En effet, toute application doit évoluer dans le temps et il est nécessaire qu'un système automatique de gestion d'applications offre aux utilisateurs des outils de mise à jour des programmes.

2.5.I. - Les outils d'aide à la mise au point

Lorsqu'une nouvelle application est en cours de réalisation, le programmeur doit tester la validité du texte source de chaque unité créée. Dans la plupart des mises au point de nouveaux programmes, on peut constater qu'il existe deux types caractéristiques d'erreurs qui peuvent être commises :

- des erreurs dans la logique même du programme,
- des erreurs de syntaxe dues à une mauvaise utilisation du langage de programmation.

Le premier type d'erreurs ne peut être détecté qu'au cours de l'exécution du programme. Cette détection peut être faite par le système lui-même en relevant, par exemple, une anomalie dans le déroulement de l'exécution. Elle peut également être faite par le programmeur, par suite d'une production de résultats erronés.

Afin de faciliter la tâche du programmeur dans la recherche des erreurs de ce type, le système de traduction doit avoir la possibilité de mettre en place des aides à la mise au point qui fonctionnent au moment de l'exécution. Dans notre système, le compilateur CIVA est prévu sous deux versions différentes : le mode "exploitation" peut être utilisé lorsqu'un programme a fait ses preuves, le texte objet étant alors optimisé. Le mode "mise au point" permet au programmeur de demander la mise en place des outils d'aides à la mise au point. Ces outils ont été décrits dans (DUCLOY [9]) et nous pouvons les rappeler rapidement : ce sont les mécanismes de trace d'une variable, de chaînage des appels aux différents modules à partir du module directeur, d'impression à des instants déterminés de l'état de la machine...

En plus de ces services classiques, le système de traduction doit détecter rapidement les erreurs syntaxiques commises dans le texte d'une unité. Dans l'état actuel de notre système, l'utilisateur, pour pouvoir tester la syntaxe d'une unité, est amené à demander la création de l'unité, demander également le lancement de la compilation en mode "mise au point", pour s'apercevoir enfin que le texte de l'unité est erroné et que celle-ci doit être modifiée partiellement ou même globalement. Cette "procédure" est donc assez lourde à mettre en oeuvre et la rapidité du système est mise en cause.

Si l'utilisateur désire éviter toutes ces opérations coûteuses, il peut alors faire appel à la procédure ANALYSER de la classe système. L'appel de cette procédure s'écrit :

ANALYSER ;
≋ texte source de l'unité ou des unités à analyser.

La demande d'analyse syntaxique du texte d'une unité est différente de l'opération de création. La procédure ANALYSER n'entraîne aucune création de déclaration dans la classe de l'application. L'unité ne peut donc pas être considérée comme un objet permanent de cette application et, lorsqu'elle aura été mise au point syntaxiquement, cette unité devra faire l'objet d'une demande de création.
A l'appel de la procédure ANALYSER, le module initial lance l'exécution du compilateur CIVA. Celui-ci vérifie la portée des noms des objets ainsi que leur type et la syntaxe des instructions. Cette opération terminée, le module initial reprend l'interprétation du module de commande.
On peut remarquer que cette procédure ANALYSER étant déclarée dans la classe système ne fait l'objet d'aucune protection particulière de la part du système CIVA.

2.5.2. - Etude du comportement des programmes

Lorsqu'un programme est correct, le système de traduction doit mettre à la disposition d'un utilisateur un ensemble de services qui le renseignent sur le comportement du programme. Ces renseignements se présentent généralement sous la forme de mesures statistiques ou comptables: par exemple, nombre de fois qu'un module ou une certaine séquence d'instructions a été exécuté, nombre d'accès réalisés à un objet au cours d'une exécution, pourcentage d'utilisation des enregistrements d'un fichier en accès direct..
L'utilisateur ne doit pas être amené à programmer l'obtention de ces renseignements : l'introduction d'instructions supplémentaires dans le langage alourdit la tâche du compilateur et elle peut engendrer des erreurs extérieures au programme mesuré. Le compilateur doit donc introduire de façon systématique, s'il travaille en mode "exploitation", des instruments de mesure de fréquence, transparents à l'utilisateur.

Par exemple, le compilateur CIVA et plus particulièrement le traducteur de table de décisions implante des compteurs transparents à l'utilisateur qui permettent de mesurer les fréquences d'exécution des actions d'une table de décision ou de sélection. Ces renseignements seront exploités par le compilateur, lors d'une recompilation du programme, pour optimiser les temps d'exécution (AUBRY [1]).
Grâce au système automatique de documentation, l'utilisateur peut alors disposer des mesures réalisées :

- fréquence d'entrée dans un module,
- fréquence d'utilisation d'une donnée, ou d'un champ de données..

L'obtention de ces renseignements sera décrite dans (BARTHELEMY [2] et PHILIPPE [2]).

2.5.3. - EVOLUTION ET CONTROLE DE L'APPLICATION

Enfin, le système CIVA doit permettre à l'utilisateur de faire évoluer les programmes de son application et de controler cette évolution. Trois ensembles de services annexes sont mis à sa disposition dans ce but :

- l'éditeur de texte regroupe les services permettant de modifier le texte source d'une unité ;
- les aides à la modification des programmes permettent de connaître la répercussion d'une modification sur l'ensemble des modules de l'application. Cela évite donc d'introduire des erreurs de programmation en modifiant un programme déjà opérationnel. Ces services seront décrits dans BARTHELEMY [2] et PHILIPPE [2]);
- les services d'obtention de renseignements généraux sur l'application.

2.5.3.I. L'éditeur de texte du projet CIVA

L'éditeur de texte mis à la disposition des utilisateurs du système CIVA permet de réaliser trois types de modification du texte source d'une unité : modification globale de tout le texte de l'unité, modification partielle, instruction par instruction, ou encore modification du nom de l'unité.

Cet éditeur de texte se présente sous la forme d'un ensemble de procédures déclarées dans la classe de modification.

Comme nous l'avons vu en 2.3.I.2., toute modification du texte source d'une unité peut avoir des conséquences graves sur l'ensemble de l'application. C'est pour cette raison que les procédures d'édition de texte font l'objet d'une protection particulière de la part du système CIVA, étant déclarées dans la classe de modification.
Pour pouvoir utiliser ces procédures dans un module de commande, il faut faire la déclaration préalable du mot de passe de l'application modification. Cette déclaration consiste à affecter à la variable système MODIFICATION la chaîne de caractères représentant le mot de passe. Ce mot de passe est fixé une fois pour toutes au moment de la génération du système par les responsables des applications. Si, au cours de l'interprétation du module de commande, le module initial rencontre un appel d'une procédure d'édition de texte alors que la variable MODIFICATION n'a pas été affectée ou a reçu une valeur incorrecte, l'interprète du module de commande émet un message d'erreur et continue son interprétation.

2.5.3.I.I. Modification globale du texte source

Au cours de ses travaux, l'utilisateur peut être amené à changer globalement tout le texte source d'une unité de son application. Pour faire cette opération, il pourrait supprimer son unité de l'application et la recréer ensuite avec le nouveau texte (cf les procédures CREER en 2.4.2. , SUPPRIMER en 2.5.4.I.). Il est donc obligé d'appeler deux procédures différentes.
Afin de rendre l'opération de changement de texte source plus rapide et plus simple pour l'utilisateur, nous avons créé la procédure CHANGER dans la classe de modification. L'appel de cette procédure s'écrit de la façon suivante :

CHANGER (<nom d'unité>) ;
≋ nouveau texte source

L'unité désignée par l'identificateur a dû être précédemment créée dans l'application. Si ce n'est pas le cas, le module initial émet un message d'erreur et passe à l'instruction suivante dans le module de commande.
On peut remarquer également que, si la procédure CHANGER facilite la tâche de l'utilisateur pour réaliser une modification globale, elle est également utile à l'implémenteur qui n'a pas à effectuer toutes les opérations nécessaires pour une suppression et une adjonction de déclaration d'unités.

2.5.3.I.2. Modification partielle

Dans le module de commande, l'utilisateur peut également demander au système de faire des corrections dans le texte source d'une unité de son application. Pour cela il peut utiliser un jeu de procédures de la classe de modification. Ces procédures de modification de texte permettent de construire un nouveau texte source à partir d'un texte existant déjà, en conservant le même nom à l'unité. Ces procédures sont décrites dans (PAYAFAR [I7]). Nous n'avons fait que reprendre ces procédures pour les adapter à l'environnement du module initial et en particulier à la présentation du texte source de l'unité.
Toute demande de modification partielle commence par l'instruction :

CORRIGER (<nom d'unité>) ;

L'appel de cette procédure permet au système de donner une valeur à la variable TEXTE-A-MODIFIER définie dans la classe système. Aucune modification de texte n'est effectuée.

L'appel de CORRIGER est suivi par un ou plusieurs appels d'une des trois procédures suivantes :

- INSERER (<localisation> , <nombre n>) ;
  ≋ n enregistrements

La procédure INSERER permet d'ajouter des instructions CIVA au texte source de l'unité désignée par la variable TEXTE-A-MODIFIER. Les instructions ajoutées sont celles qui figurent dans les n lignes qui suivent l'appel de la procédure. L'insertion est réalisée après la ligne de l'ancien texte repérée par la localisation.

- OTER (<localisation> , <nombre n>) ;

La procédure OTER permet de supprimer des instructions du texte source de l'unité désignée par la variable TEXTE-A-MODIFIER. Les instructions supprimées sont celles qui figurent dans les n lignes de l'ancien texte source, à partir de la ligne repérée par la localisation.

- MODIFIER (<localisation> , <nombre n>) ;
  ≋ n enregistrements

La procédure MODIFIER permet de remplacer une ou plusieurs instructions de l'ancien texte source de l'unité désignée par TEXTE-A-MODIFIER par une ou plusieurs nouvelles instructions. Ces nouvelles instructions doivent suivre immédiatement l'appel de MODIFIER, dans les n lignes suivantes du module de commande. La modification a lieu à partir de ligne repérée par la localisation.

Pour déterminer l'endroit où doit être faite la mise à jour de l'ancien texte source, nous avons prévu trois types de localisation :

- la localisation absolue qui représente le numéro de l'enregistrement le premier à traiter avec OTER et MODIFIER ou juste après lequel on ajoute des instructions avec INSERER.

- la localisation relative qui situe l'emplacement actuellement traité dans le texte source (l'utilisation de la localisation relative est matérialisée par le caractère "astérisque").

- la localisation par identité en écrivant tout simplement l'ancien texte source.

La correction du texte source de l'unité désignée par TEXTE-A-MODIFIER se termine lorsque le module initial rencontre dans le module de commande une instruction autre qu'un appel d'une de ces trois procédures.

Lorsque le module initial interprète les appels de ces procédures, il peut détecter plusieurs erreurs :

- si l'identificateur indiqué dans l'appel de CORRIGER ne désigne pas une unité dans l'application, le module initial émet un message d'erreur. Aucune modification de texte n'est effectuée. L'interprètation du module de commande se poursuit à la première instruction qui n'est pas un appel d'une des trois procédures de modification.

- si la localisation indiquée dans une procédure de modification de texte est incorrecte (par exemple, le système ne retrouve pas l'instruction indiquée dans le cas d'une localisation par identité), le module initial émet un message d'erreur et arrête les opérations de modification de texte. L'interprétation du module de commande se poursuit à la première instruction qui n'est pas un appel d'une des trois procédures citées.

Exemple d'utilisation

numéro d'enregistrement	ancien texte source
1	module prime ;
2	utilise structure ; nbpc entier ;
3	nbpc = coef (code, nb, assurance) ;
4	num de résultat = num de assuré ;
5	pour i de 1 à nb faire
6	impression ; fp ;
7	finmod ;

contenu du module de commande :

```
≋ début du module de commande
    CORRIGER (PRIME) ;
CO La variable texte-à-modifier reçoit la valeur : PRIME CO ;
    INSERER (1, 2) ;
        utilise type ;
        NB réel ;
CO L'appel de INSERER permet d'ajouter 2 enregistrements après l'enregis-
trement numéro 1 de l'ancien texte source CO ;
    OTER ('num de résultat = num de l'assuré'; , 1) ;
CO L'appel de OTER permet de supprimer l'enregistrement de l'ancien texte
source qui contient l'instruction indiquée. Après cet appel, la localisa-
tion relative pointe vers l'enregistrement suivant (c'est-à-dire l'enre-
gistrement 5) CO ;
    MODIFIER (*, 1) ;
        pour j de 1 à m faire
CO L'appel de MODIFIER remplace l'enregistrement 5 par celui indiqué CO ;
≋ suite du module de commande
```

Après ces corrections, le nouveau texte source sera le suivant :

numéro d'enregistrement	nouveau texte source
1	Module prime ;
2	utilise type ;
3	nb réel ;
4	utilise structure ; nbpc entier ;
5	nbpc = coef (code, nb, assurance) ;
6	pour j de 1 à m faire
7	impression ; fp ;
8	finmod ;

2.5.3.I.3. Changement du nom d'une unité

Dans un module de commande, l'utilisateur peut également changer le nom d'une unité déjà créée dans son application.

Cette possibilité peut être très utile si, au cours de l'analyse d'une application importante, l'utilisateur s'aperçoit que deux unités ont le même nom et que l'une d'entre elles est déjà créée. Il peut alors en changer le nom. Cette opération peut aussi être effectuée dans le cas où l'utilisateur veut ajouter des unités d'autres applications sans leur changer le nom et que ces noms figurent déjà dans sa propre application (cf la procédure AJOUTER en 2.4.3.).

Le changement de nom se fait par l'appel de la procédure REMPLACER de la classe de modification. Cet appel s'écrit :

REMPLACER (⟨nom 1 d'unité⟩,
⟨nom 2 d'unité⟩
[, REFERENCE]) ;

Après l'appel de cette procédure, ⟨nom 2 d'unité⟩ désigne l'unité qui, avant l'appel, était désignée par ⟨nom 1 d'unité⟩ .

Cette unité a du être précédemment créée dans l'application. Si ce n'est pas le cas, le module initial émet un message d'erreur et passe à l'instruction suivante. Si l'utilisateur emploie l'option "REFERENCE", le module initial lui communique la liste des identificateurs de l'environnement, à savoir la liste des noms des unités appartenant à $(\widehat{U \cup A \cup \$A})$ (v) puis la liste des noms appartenant à $(\widehat{U \cup A \cup \$A})^{-1}$ (v), où v est l'unité désignée par <nom 2 d'unité>.

2.5.3.2. Obtention de renseignements sur les unités d'une application

L'utilisateur peut avoir besoin de renseignements sur les unités qui lui appartiennent pour pouvoir suivre l'évolution de son application. Afin de satisfaire ce besoin, la classe système met à sa disposition deux procédures : LISTER et SOURCE. Toutefois, nous sommes conscients que ces procédures ne fournissent pas tous les renseignements qui pourraient être utiles à l'utilisateur. D'autres services lui sont offerts grâce au service de documentation décrit dans (BARTHELEMY [2]) et dans (PHILIPPE [2]). Ces procédures étant déclarées dans la classe système ne sont pas protégées : tout module de commande peut les appeler.

2.5.3.2.I. Obtention de renseignements généraux

Lorsqu'une application est importante et que plusieurs personnes travaillent à sa réalisation et à son exploitation ,au bout d'un certain temps, il s'avère difficile de connaître très exactement quelles sont les unités qui font partie de l'application et quelles sont les relations qui les lient entre elles. De plus, si les unités sont à l'état "compilé", l'utilisateur peut avoir besoin de connaître la place qu'elles occuperont en mémoire (cf l'automatisation du dossier d'application en 2.2.2.). Tous ces renseignements peuvent être obtenus par l'appel de la procédure LISTER de la classe système. L'appel de cette procédure s'écrit en règle générale de la façon suivante :

LISTER [(<liste d'options>)] [, (<liste de noms d'unités>)] ;

- premier type d'appel

LISTER ;

Dans ce cas, l'utilisateur demande au module initial la liste des noms des unités qui sont créées dans son application.

- deuxième type d'appel

LISTER (<liste d'options>) ;

Dans la liste d'options, chaque option est séparée de la suivante par une virgule. Dans ce cas, l'utilisateur demande au module initial la liste des noms des unités qui sont créées dans l'application, ainsi que certains renseignements concernant chaque unité.

L'effet de la procédure LISTER est le suivant selon les options :
TYPE : donne le type de l'unité (classe, module, métamodule).
ETAT : donne son état actuel dans l'application (source, compilé en mode mise au point, compilé en mode exploitation, édité en mode mise au point, édité en mode exploitation).
ERREUR : fournit le degré maximum d'erreur que le compilateur a trouvé après traduction de l'unité (évidemment, ce renseignement n'est indiqué que si l'unité n'est pas à l'état source).
CONSTANTE : donne la taille de la zone des constantes nécessaires à l'implantation de l'unité. Ce renseignement ne peut être obtenu qu'après compilation.
VARIABLE : donne la taille de la zone réservée aux variables de l'unité. Ce renseignement ne peut être obtenu lui aussi qu'après compilation de l'unité.
METAMOD : fournit la liste des noms des métamodules qui sont appelés par l'unité.
CLASSE : fournit la liste des noms des classes utilisées par l'unité.
MODULE : fournit la liste des noms des modules appelés par l'unité (si l'unité n'a jamais été compilée, cette liste contient les noms des modules et des procédures sans paramètre appelés par l'unité).

UNITE : cette option regroupe les trois options précédentes. Elle fournit la liste des noms des modules et métamodules appelés ainsi que les noms des classes utilisées.
PROTECTION : donne la protection inter-application qui a été affectée à l'unité au moment de sa création dans l'application.

- troisième type d'appel

LISTER (<liste d'options>), (<liste de noms d'unités>) ;

Dans la liste des noms d'unité, chaque nom est séparé du suivant par une virgule.
Si l'utilisateur emploie ce type d'appel, il désire connaître des renseignements concernant seulement quelques unités créées dans son application, celles dont il a indiqué les noms.
Les options sont alors les mêmes que celles du deuxième type d'appel.

- quatrième type d'appel

LISTER (<liste d'options>), (CLASSE APPLICATION) ;

Par cet appel, l'utilisateur peut obtenir des renseignements sur le contenu de la classe d'application. Ces renseignements sont de deux types suivant l'option choisie :
IDENTIFICATEUR : donne la liste des identificateurs d'objets primitifs déclarés dans la classe d'application ainsi que le type de ces objets.
FICHIER : donne la liste des noms des fichiers physiques déclarés dans la classe d'application ainsi que tous les renseignements concernant ces fichiers : localisation, protection ...(cf : la définition d'un fichier physique en 3.4.).

Remarques

1) Si une option est inconnue du module initial, celui-ci émet un message d'erreur et passe à l'instruction suivante du module de commande. L'appel de LISTER est alors ignoré.

2) Si, dans un appel du troisième type, le module initial rencontre un identificateur inconnu, il émet un message d'erreur et passe à l'identificateur suivant dans la liste des noms d'unités.

Exemples d'utilisation

contenu du module de commande

LISTER ;

CO cet appel permet d'obtenir la liste des unités créées dans l'application CO ;

LISTER (ETAT, UNITE, TYPE) ;

CO cet appel permet d'obtenir, pour chaque unité créée dans l'application, le type de l'unité, les noms des métamodules et des modules appelés, les noms des classes utilisées ainsi que l'état actuel de l'unité dans l'application CO ;

2.5.3.2.I. Obtention du texte source

Nous avons vu que, grâce à une procédure de la classe de modification, la procédure CORRIGER, l'utilisateur pouvait modifier le texte source de ses unités (cf en 2.5.3.I.2.). Après avoir fait plusieurs modifications, l'utilisateur peut avoir besoin de connaître le texte source de ces unités qu'il a modifiées, tel qu'il se trouve présent dans le système.
Pour cela, il appelera la procédure SOURCE de la classe système.
L'appel de cette procédure s'écrit :

SOURCE (<liste de noms d'unité>) ;

Dans la liste de noms d'unité, chaque nom est séparé du suivant par une virgule.
Si le module initial rencontre un nom inconnu, il émet un message d'erreur et passe à l'identificateur suivant.

2.5.4. - Les opérations de suppression

Après avoir vu les outils permettant de créer des déclarations et de modifier les unités, il nous reste à présenter les outils de suppression de déclarations qui sont également des outils de maintenance.

D'après la structure des constituants de notre projet, on peut remarquer qu'il existe trois types de suppression :

- suppression d'un objet élémentaire ou structuré déclaré dans la classe d'une application.
- suppression de la déclaration d'une unité de nomenclature dans la classe d'une application.
- suppression de la déclaration d'une classe d'application dans la classe système.

Ces opérations de suppression pouvant avoir des répercussions importantes sur l'ensemble du système, il est nécessaire que les outils qui les réalisent ne soient pas accessibles à tous les utilisateurs. Ils seront donc déclarés dans la classe de modification et seuls, les modules de commande ayant affecté le mot de passe de l'application modification à la variable système MODIFICATION pourront les utiliser.

2.5.4.I. Suppression de déclarations dans la classe d'application

Les suppressions de déclarations dans la classe de l'application courante sont réalisées par l'appel d'une procédure de la classe de modification, la procédure SUPPRIMER.

2.5.4.I.I. l'appel de la procédure SUPPRIMER

L'appel de cette procédure s'écrit :

SUPPRIMER (<liste d'éléments à supprimer>) ;

Dans la liste d'éléments à supprimer, chaque élément est séparé du suivant par une virgule. Chaque élément indique quelle est la déclaration de la classe de l'application courante à supprimer.

- Un élément peut représenter un objet primitif (élémentaire ou structuré)de la classed'application : dans ce cas, l'élément est une occurence de l'identificateur de l'objet à supprimer.
- Un élément peut également représenter une unité de l'application, que ce soit un module, une classe ou un métamodule. L'utilisateur indique alors le nom de l'unité.

Si une suppression d'unité est demandée, seule cette unité est détruite. Mais, afin de faciliter la tâche de l'utilisateur pour la mise à jour de l'application, le système peut lui indiquer quelles sont les unités appartenant à l'environnement de l'unité supprimée.

Pour cela, l'utilisateur indique l'option REFERENCE dans l'appel de la procédure.

L'utilisateur peut préciser cette option pour une ou plusieurs unités de la liste. L'interprète communique alors la liste des identificateurs des unités appartenant à $\widehat{(\dot{U} \cup A \cup \$A)}(v)$ puis à $\widehat{(\dot{U} \cup A \cup \$A)}^{-1}(v)$, où v désigne successivement les différentes unités supprimées sur lesquelles porte l'option REFERENCE.

<élément à supprimer> ::= <identificateur d'objet primitif>
| <nom d'une unité>
| (<liste des noms d'unité>, REFERENCE)

Exemple

SUPPRIMER (I, J, (M1, M2, REFERENCE), M3) ;

Cet appel permet de supprimer les déclarations :

- *de deux objets élémentaires I et J,*
- *de deux unités de l'application M1 et M2. Pour chacune de ces unités, l'interprète communique la liste des noms des unités de leur environnement.*
- *de l'unité M3.*

Remarque

Chaque identificateur de la liste des éléments à supprimer doit être connu du système. Si le module initial rencontre un identificateur inconnu, il émet un message d'erreur et passe à l'identificateur suivant.

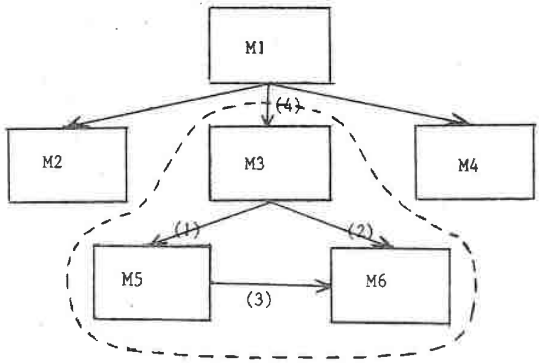
2.5.4.I.2. Le problème de la suppression d'une unité

Pour supprimer une unité de l'application, l'interprète des modules de commande a deux possibilités.

L'interprète peut faire des contrôles sur les suppressions demandées, c'est-à-dire vérifier que la ou les suppressions n'entraînent pas d'erreurs qui ne seraient détectées qu'à la compilation ou pendant la reliure, donc à un instant qui peut être lointain de celui de la suppression.

L'interprète peut également ne faire aucun contrôle sur les suppressions demandées. L'utilisateur est alors libre de supprimer n'importe quelle unité de son application et l'interprète ne vérifie pas les mises à jour réalisées par l'utilisateur sur les unités touchées par la suppression. Si l'opération de suppression entraîne des erreurs, celles-ci ne seront détectées que plus tard par une autre module du système.

Cette possibilité de contrôle de la part de l'interprète oblige l'utilisateur à respecter un certain ordre pour réaliser ces suppressions.
Par exemple, considérons le graphe suivant d'une unité de traitement :



Supposons que l'utilisateur veuille supprimer les modules M3, M5 et M6. Si des contrôles sont effectués, l'utilisateur est obligé de supprimer :

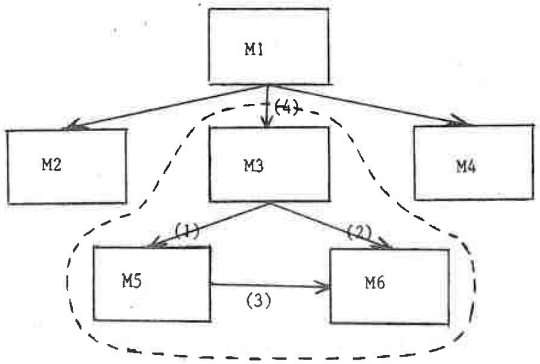
- dans M3, l'appel de M5 (1)
- dans M3, l'appel de M6 (2)
- dans M5, l'appel de M6 (3)
- dans M1, l'appel de M3 (4)
- les unités M3, M5 et M6.

L'opération est donc assez longue et trop contraignante pour l'utilisateur.
Par contre, si l'interprète du module de commande ne fait pas de contrôle, l'utilisateur supprime seulement :

- les unités M3, M5 et M6
- l'appel de M3 dans le module M1 (4).

L'interprète peut également ne faire aucun contrôle sur les suppressions demandées. L'utilisateur est alors libre de supprimer n'importe quelle unité de son application et l'interprète ne vérifie pas les mises à jour réalisées par l'utilisateur sur les unités touchées par la suppression. Si l'opération de suppression entraîne des erreurs, celles-ci ne seront détectées que plus tard par une autre module du système.

Cette possibilité de contrôle de la part de l'interprète oblige l'utilisateur à respecter un certain ordre pour réaliser ces suppressions.
Par exemple, considérons le graphe suivant d'une unité de traitement :



Supposons que l'utilisateur veuille supprimer les modules M3, M5 et M6. Si des contrôles sont effectués, l'utilisateur est obligé de supprimer :

- dans M3, l'appel de M5 (1)
- dans M3, l'appel de M6 (2)
- dans M5, l'appel de M6 (3)
- dans M1, l'appel de M3 (4)
- les unités M3, M5 et M6.

L'opération est donc assez longue et trop contraignante pour l'utilisateur.
Par contre, si l'interprète du module de commande ne fait pas de contrôle, l'utilisateur supprime seulement :

- les unités M3, M5 et M6
- l'appel de M3 dans le module M1 (4).

L'opération est donc plus courte.
La possibilité de contrôle a donc été abandonnée. L'interprète ne vérifie pas l'ordre des suppressions demandées.
La vérification de la validité de l'unité de traitement ainsi modifiée est donc à la charge de l'utilisateur.

2.5.4.2. Suppression d'une application du système

Lorsqu'une application n'a plus aucune utilité dans un centre de traitement de l'information, par suite de l'évolution des besoins des utilisateurs, il faut mettre fin à ses jours. Le fait de tuer une application périmée n'est pas une opération obligatoire en elle-même, mais elle est utile à l'implémenteur qui détruit alors tous les objets rattachés à cette application, ce qui permet d'alléger sa gestion des applications.
Pour pouvoir réaliser cette opération, les utilisateurs disposent d'une procédure particulière déclarée dans la classe des modification , la procédure SUPAPPLICATION. L'appel de cette procédure s'écrit :

SUPAPPLICATION (<nom de l'application> [,(PASSE, <mot de passe>)]) ;

Par mesure de protection, l'opération de suppression de la déclaration d'une application dans la classe système ne peut être réalisée que si l'utilisateur connaît le mot de passe de l'application à supprimer.
A la rencontre de cet appel, le module initial vérifie que le nom de l'application indiqué par l'utilisateur désigne bien une application existant dans le système et que le mot de passe est correct. Si ce n'est pas le cas, le module initial émet un message d'erreur et continue l'interprétation. Si les paramètres de l'appel de la procédure sont corrects, l'interprète supprime l'application du système CIVA : il détruit donc tous les objets déclarés dans la classe de l'application (objets élémentaires ou structurés, unités, fichiers permanents) ainsi que cette classe d'application.

2.6. LES OUTILS D'EXPLOITATION

2.6.I. - Présentation générale de l'exploitation d'une unité de traitement.

Les outils d'exploitation mis à la disposition des utilisateurs du système CIVA permettent simplement de réaliser le lancement de l'exécution des unités de traitement. Ces exécutions peuvent dépendre de valeurs affectées à des paramètres d'exploitation, ces paramètres étant accessibles à tout utilisateur. Ils seront donc définis dans la classe système.

Comme dans n'importe quel module, l'utilisateur peut appeler dans un module de commande d'autres modules de son application. L'appel d'un module M dans le module de commande correspond à une demande d'exécution d'une unité de traitement, les différents appels de modules contenus dans un module de commande constituant une chaîne de traitement.
Une unité de traitement de premier module M, ou de module directeur M, est constituée du sous-ensemble $\hat{U}(M)$ des classes et du sous-ensemble $\hat{A}(M)$ des modules de l'application, $\hat{U}$ et $\hat{A}$ désignant respectivement les fermetures transitives de la relation UTILISE et de la relation APPELLE.
Nous avons vu au chapitre 1 que, contraitement au module de commande, les unités d'une application étaient compilées. Donc, pour pouvoir être exécutée, une unité de traitement doit se présenter sous la forme d'un module de chargement produit par l'éditeur de liens, c'est-à-dire que le module directeur de cette unité doit être à l'état édité. Pour pouvoir éditer l'unité de traitement, il faut que le système possède la traduction de toutes les unités qu'elle contient : toutes les unités de $\hat{U}(M)$ et de $\hat{A}(M)$ doivent donc être à l'état compilé.
Le module initial est donc amené à gérer l'ensemble des unités d'une application. Il assure automatiquement tous les changements d'état nécessaires pour pouvoir lancer les exécutions.

2.6.2. - La demande d'exécution d'une unité de traitement

La demande de l'exécution est formulée par l'utilisateur dans le module de commande en indiquant simplement le nom du module directeur de l'unité de traitement. Comme nous le verrons en 5.5., cette demande correspond, pour le système CIVA, à la création d'un travail indépendant et à son adjonction dans le train des travaux qui sont entrés dans le système d'exploitation.

Pour pouvoir lancer l'exécution d'un travail, le système doit disposer de certains paramètres d'exploitation (par exemple la valeur affectée à un de ces paramètres correspond à la taille de mémoire désirée par l'utilisateur pour exécuter son programme). Ces paramètres sont déclarés dans la classe système et ils sont initialisés à des valeurs implicites au début de l'interprétation du module de commande par le module initial. Les modules de commande utilise la classe système et peut donc avoir accès à ces paramètres. L'utilisateur peut donc modifier les valeurs attribuées par le système à ces paramètres en utilisant des instructions d'affectation de la même façon que pour tout autre objet élémentaire accessible au module initial.

2.6.2.I. Les paramètres d'exploitation

Les paramètres d'exploitation accessibles à l'utilisateur pour la demande d'exécution d'une unité de traitement sont au nombre de cinq.

a) Un premier paramètre permet à l'utilisateur d'indiquer au système dans quel mode doit être compilé l'unité de traitement.
Nous pouvons rappeler que le système de compilation mis en place dans le projet CIVA possède deux modes de compilation (DUCLOY [9]):

- le mode"mise au point" permet de réaliser des contrôles sur le déroulement de l'exécution et fournit à l'utilisateur des renseignements utiles lui permettant de retrouver des erreurs dans la logique de ses programmes.
- le mode "exploitation" permet par contre au système de compilation d'optimiser la traduction des programmes, ceci afin de rendre minimum le temps d'exécution.

Ce paramètre est une variable de la classe système de type chaîne de caractères. Cette variable est définie de la façon suivante :

MODE-COMPIL file (max = 12) car ;

Les valeurs susceptibles d'être affectées à ce paramètre sont prédéfinies ;

"MISE AU POINT" pour obtenir des compilations en mode "mise au point"

"EXPLOITATION" pour obtenir des compilation en mode "exploitation"

Cette dernière valeur est la valeur implicite prise par le système à l'initialisation.

b) Dans un module de commande, l'utilisateur peut également indiquer au système quel est le type de travail qui correspond à l'unité de traitement. Dans le système CIVA, les unités de traitement sont classées en trois types :

- les unités dont le temps d'exécution est court (de l'ordre de 5 à IO mn maximum),
- les unités de traitement dites"scientifiques", c'est-à-dire dont le temps d'unité centrale est très supérieur au temps d'attente. Parmi cette classe d'unité de traitement on distingue les unités peu encombrantes (dont la taille n'exède pas 20K mots) et les unités encombrantes de taille plus importante.
- le troisième type d'unités de traitement regroupe les unités dites "de gestion", c'est-à-dire dont le temps d'unité centrale est de l'ordre de grandeur du temps d'attente :

$$\frac{\text{temps U.C.}}{\text{temps d'attente}} \leqslant 1$$

Dans cette classe d'unités de traitement, nous ferons la même distinction entre les unités peu encombrantes (dont la taille est inférieure à 20K mots) et les unités plus encombrantes.
Le paramètre d'exploitation correspondant est un objet de la classe système de type chaîne de caractères. Cet objet est défini par :

TYPETRAV file (2) car ;

Les valeurs que peut donner l'utilisateur à ce paramètre sont les suivantes :

'TC' pour les travaux "courts",
'S1' pour les travaux "scientifiques" peu encombrants,
'S2' pour les travaux "scientifiques" encombrants,
'G1' pour les travaux "de gestion" peu encombrants,
'G2' pour les travaux "de gestion" encombrants.

La valeur implicite donnée à ce paramètre par le module initial est 'TC'.

Dans le tableau suivant, nous avons indiqué, pour chaque type d'unité de traitement quelle est la taille mémoire maximum donnée au programme et quel est le temps d'exécution qui lui est accordé (A titre d'exemple nous avons fait le rapprochement avec les classes de travaux du système CII SIRIS 7).

TYPETRAV	TAILLE MEMOIRE MAXIMUM	TEMPS MAXIMUM	CLASSE SIRIS 7
'TC'	20K mots	IO mn	T ou C
'SI'	20K mots	900 mn	D
'S2'	50K mots (*)	900 mn	E
'G1'	25K mots	10 mn	B
'G2'	50K mots (*)	900 mn	A

(*) = 50K mots est la taille maximum de mémoire mise à notre disposition actuellement. Cette valeur est donc appelée à être modifiée par la suite.

c) Le troisième paramètre d'exploitation relatif au lancement d'exécution d'une unité de traitement est XNBPAGE qui indique au système le nombre de pages maximum que l'unité de traitement est susceptible de fournir sur l'imprimante.
Ce paramètre est défini dans la classe système de la façon suivante :

XNBPAGE entier ;

La valeur implicite est 30.

d) Du fait que la demande d'éxécution d'une unité de traitement est considérée par le système comme le lancement d'un travail indépendant, il peut être intéressant pour l'utilisateur de continuer l'interprétation du module de commande sans attendre la fin de l'exécution de l'unité de traitement. Cette possibilité de continuer l'interprétation permet un gain de temps assez important, l'interprétation du module de commande et l'éxécution de l'unité de traitement se déroulant alors collatéralement. Toutefois,pour pouvoir bénéficier de ce service, il ne faut pas que l'unité de traitement modifie des valeurs affectées à des objets déclarés dans une classe commune à l'unité et au module de commande : ces modifications n'auraient alors aucune conséquence sur la suite de l'exécution du module de commande, n'étant pas prises en considération par le module initial.
Le rôle du quatrième paramètre d'exploitation est de permettre à l'utilisateur de faire part de son choix au système. Ce paramètre est un booléen déclaré dans la classe système, de nom ATTENTE. Si l'utilisateur désire que l'interprétation du module de commande continue aussitôt après avoir lancé l'exécution de l'unité de traitement, il positionnera ATTENTE à la valeur "faux". Par contre, si le module initial doit attendre la fin de l'exécution de l'unité de traitement avant de reprendre l'interprétation du module de commande, l'utilisateur affectera à ATTENTE la valeur "vrai". C'est cette dernière valeur qui est la valeur par défaut prise par le système.

e) Nous avons vu au chapitre 1 qu'un module de commande correspondait à la demande d'exécution d'une chaîne de traitement. En général, cette chaîne de traitement est composée de plusieurs unités de traitement et les demandes d'exécution de ces unités se succèdent dans le module de commande. Si l'exécution d'une de ces unités se termine mal, le système doit savoir s'il doit continuer l'interprétation du module de commande, c'est-à-dire lancer les exécutions des unités suivantes dans la chaîne,ou non.
Le rôle du dernier paramètre d'exploitation est d'indiquer au système le choix de l'utilisateur quant à l'enchaînement des exécutions des unités de traitement.

Ce paramètre, de nom ENCHAINEMENT, est un objet élémentaire de type booléen, déclaré dans la classe système.

- Si l'utilisateur affecte à ENCHAINEMENT la valeur "vrai", lorsque l'exécution d'une unité de traitement se termine mal (à cause soit d'une erreur, soit d'une anomalie), le module initial reprend le contrôle de l'unité centrale et continue l'interprétation du module de commande. La suite de la chaîne de traitement est alors exécutée, l'unité de traitement érronée pouvant de nouveau être exécutée dans un autre module de commande, après correction des erreurs.

- Par contre, si l'utilisateur affecte la valeur "faux" au booléen ENCHAINEMENT, en cas d'erreur dans l'exécution d'une unité de traitement, le module initial arrête l'interprétation du module de commande : il réalise alors les opérations de sauvegarde du système. L'utilisateur pourra donc reprendre, après correction des erreurs, l'exploitation de la chaîne dans un autre module de commande à l'endroit où celle-ci s'était arrêtée.

2.6.2.2. Exemple de lancement d'exécution

Supposons que l'utilisateur désire exécuter une chaîne de traitement formées de trois unités de traitement :

- la première unité de traitement est de type travail court.
- la deuxième unité est de type scientifique peu encombrante.
- la troisième, en cours de mise au point, est de type gestion encombrante.

Dans le module de commande, on peut trouver :

```
ATTENTE = faux ;
ENCHAINEMENT = vrai ;
UNITRAIT-1 ;
CO c'est la demande d'exécution de la première unité de traitement.
MODECOMPIL indique le mode "exploitation", TYPETRAV indique un
travail court CO ;
TYPETRAV = 'S1' ;
XNBPAGE = 10 ;
UNITRAIT-2 ;
MODE COMPIL = "MISE AU POINT " ;
XNBPAGE = 50 ;
TYPETRAV = 'G2' ;
UNITRAIT-3 ;
finmod ;
```

2.6.3. - Demandes explicites de compilation ou de reliure

Nous venons de voir que le module initial assurait la gestion des unités d'une application et qu'il assurait automatiquement tous les changements d'état nécessaires, lançant les opérations de compilation et de reliure. Toutefois il peut être souhaitable, pour l'utilisateur, de demander explicitement la compilation ou l'édition d'un ou de plusieurs modules. Cette possibilité lui est nécessaire dans le cas où il veut utiliser une unité dans une autre application (voir les relations externes au chapitre 1). En effet, nous avons signalé que, dans ce cas, l'unité externe doit être à l'état compilé.
Cette possibilité peut lui être également utile pour faire la mise au point de ses programmes et relever les erreurs de programmation.

2.6.3.I. Demande explicite de compilation

La demande explicite de compilation peut être formulée par l'utilisateur pour un module ou pour une classe (ou même plusieurs). Cette demande s'écrit dans le module de commande de la façon suivante :

COMPILER ({ SEUL / ENVIRONNEMENT } , { MISE AU POINT / EXPLOITATION } , <liste des unités à compiler>) ;

Cet appel permet de demander la compilation des unités désignées dans la liste.
L'utilisateur indique également s'il veut que la (ou les) unité soit compilée seule (option SEUL) ou avec son environnement (option ENVIRONNEMENT), c'est-à-dire avec les unités de $\hat{A}(x)$ et de $\hat{U}(x)$.

2.6.3.2. Demande explicite de reliure

La demande explicite de reliure peut être formulée par l'utilisateur pour un ou plusieurs modules de son application. Cette demande s'écrit :

RELIER (<taille mémoire>, <liste des unités à relier>) ;

Cet appel permet de demander la reliure de chacune des unités indiquées dans la liste.
L'utilisateur indique quelle est la taille maximum en mémoire qu'il réserve à l'implantation de chacune de ses unités. En effet, dans une reliure implicite, cette taille est connue du système par le paramètre d'exploitation TYPETRAV (voir § 5.2.). Pour une demande explicite, cette taille doit être indiquée.

2.6.4. - Le contrôle de l'exécution et le retour dans le module de commande

Une fois les unités formant l'unité de traitement compilées et éditées, le module initial lance l'exécution. Cette exécution peut se dérouler normalement mais il peut également arriver des incidents plus ou moins graves qui la perturbe. Le système est donc amené à contrôler l'exécution des programmes utilisateurs. Ce contrôle de l'exécution a été décrit dans (DUCLOY [9] chapitre 6). Nous ne faisons, ici, qu'en rappeler brièvement le fonctionnement.
Le contrôle opéré par le système est rattaché à la notion d'évènements : la réalisation d'un évènement est caractérisé par le changement d'état d'un objet survenant à un instant non défini et souvent non prévisible.
On peut distinguer 3 types d'évènements :

a) Une anomalie est un évènement détecté par le matériel qui, le plus souvent, ne constitue pas un danger pour l'exécution de l'unité de traitement. A chaque anomalie possible, le système associe dans la classe système, une variable contrôlable ANOM(i) qui prend la valeur 'vrai' lorsque l'anomalie "i" est détectée.

Parmis les anomalies détectées par le système, on peut citer :

- le débordement entier,
- le débordement flottant,
- l'erreur arithmétique décimale...

b) Une erreur est un évènement détecté soit par le matériel soit par le système. Elle entraîne généralement l'arrêt de l'unité de traitement. Certaines erreurs peuvent être récupérées par le système : à chacune d'elles est associée une variable contrôlable comme pour les anomalies. Toutefois, il existe des erreurs qui ne permettent pas de reprendre le contrôle de l'ordinateur. Pour celles-ci, une seule variable controlable est associée : la variable ERREUR.

c) D'autres évènements sont également décrits par des variables système : ce sont les évènements qui sont liés aux compteurs de temps ou à l'intervention de l'opérateur.
Pour chaque évènement ainsi décrit, le système associe à la variable contrôlable une séquence standard de récupération. Mais dans son unité de traitement, l'utilisateur peut définir une autre séquence de recupération grâce à l'instruction QUAND.

Lorsque l'exécution de l'unité de traitement est terminée, le module initial peut continuer l'interprétation du module de commande (si le paramètre d'exploitation ATTENTE avait la valeur vrai). Il indique alors à l'utilisateur les renseignements suivants :

- le temps nécessaire à l'exécution de l'unité de traitement
- le nombre de pages de mémoire que prend l'unité de traitement

pour son exécution.
Ces renseignements peuvent être particulièrement utiles à l'utilisateur pour savoir dans quel type de travail il pourra alors ranger l'unité de traitement pour les exploitations futures.

3

LE MODULE DE COMMANDE ET LE SYSTEME DE GESTION DE FICHIERS

3.I. ROLE DU SYSTEME DE GESTION DE FICHIERS DANS LE PROJET CIVA

3.I.I. - Présentation générale

Les fichiers , dans un système informatique de taille importante, peuvent être nombreux et très variés. Il faut donc trouver un moyen de les ordonner. Tous les fichiers sont rangés dans les mémoires secondaires et, au cours des traitements, peuvent être amenés à se déplacer dans une éventuelle hiérarchie de mémoire. Le but de l'organisation que nous allons décrire est de permettre l'utilisation de fichiers pendant l'éxécution des unités de traitement.

Tout ordinateur possède son système d'exploitation qui peut être activé par l'utilisation des instructions de commande. Une partie du système assure la gestion des fichiers des utilisateurs (par exemple, le SGF du système SIRIS 7, l'IOCS du système IBM 360). Le but de la gestion des fichiers dans le système CIVA est donc double :

- jouer un rôle d'intermédiaire entre l'utilisateur et le système d'exploitation pour les déclarations et les traitements de fichiers et si possible en permettant à l'utilisateur d'employer le même langage que celui utilisé pour la programmation. Ce rôle est donc simplificateur.
- assurer à l'utilisateur des services complémentaires qui n'existent pas dans les systèmes d'exploitation classiques.

Le système global composé du système d'exploitation d'une part et du système CIVA d'autre part doit donc assurer l'organisation des fichiers à l'insu de l'utilisateur. Tout ce que ce dernier doit faire, c'est de créer des fichiers, d'en supprimer d'autres,de changer des caractéristiques, de traiter un ou plusieurs fichiers. Le fonctionnement du système de gestion de fichiers doit lui être transparent.

3.I.2. - Définitions préliminaires

Pour pouvoir présenter le système de gestion de fichiers de CIVA, il nous faut donner 2 définitions pour introduire la notion même de fichier. Nous dirons donc que :

- un fichier logique est une collection, généralement structurée, de données sans précision de support. Il représente uniquement la valeur informationnelle.
- un fichier physique est un support bien précis, conditionné pour recevoir les informations logiques.

Par exemple, une entreprise peut avoir un certain "fichier" PERSONNEL. L'ensemble des renseignements qu'il constitue représente le fichier logique. Ces informations peuvent être mémorisées à la fois sur cartes, sur bande magnétique et sur disque. Ces trois formes constituent trois fichiers physiques du même fichier logique.

Dans le système de gestion de fichiers que nous allons définir, nous nous appuyons sur cette distinction. Nous allons donc étudier comment sont définis en CIVA, les fichiers logiques d'une part, les fichiers physiques d'autre part. Nous indiquerons ensuite les opérations réalisables sur ces deux entités.

3.I.3. - Les fonctions à exprimer

Dans un système de gestion de fichiers, on est toujours amené à résoudre le problème de pouvoir désigner, créer, supprimer et manipuler des fichiers sur un ordinateur particulier muni de son système d'exploitation (SGF de SIRIS 7 [31] , OS de IBM 360, EXEC 8 de UNIVAC 1100 [38] , ESOPE [34] [35]) ou même sur un réseau d'ordinateurs (les études déjà réalisées sur les réseaux ont été présentées par DU MASLE [25], DE CALUWE [21] ....)

Certaines manipulations font l'objet de fonctions concernant un "traitement" global du fichier. Ces fonctions sont généralement appelées "utilitaires" dans les systèmes classiques et permettent :

- la copie d'un fichier
- la concaténation de plusieurs fichiers en un seul
- l'impression du contenu d'un fichier
- le tri d'un fichier

En fait, la plupart de ces fonctions se ramènent à des copies d'une façon ou d'une autre, si on raisonne sans tenir compte des types d'unités qui y interviennent. Nous exploiterons cette remarque dans notre système. Elle a d'ailleurs été déjà exploitée dans le système MTS (FLANIGAN [33]).

D'autres manipulations concernent les traitements des enregistrements du fichier. Elles font l'objet de fonctions, généralement appelées "procédures système" et permettent :

- d'avoir accès à un enregistrement donné du fichier,
- de traiter un fichier enregistrement par enregistrement : positionnement en début de fichier, passage d'un enregistrement au suivant, positionnement en fin de fichier...

Il faut remarquer que la plupart des systèmes d'exploitation sont des systèmes multilangages. Or le problème des accès aux informations est un problème commun aux différents langages. De plus, il peut être utile de permettre des échanges de fichiers entre programmes écrits dans des langages différents. Pour ces raisons, les fonctions d'accès aux enregistrements sont réalisées au niveau du système. Mais quel que soit le langage de programmation utilisé, le programmeur doit pouvoir exprimer dans son programme la demande d'accès aux informations :ce besoin est donc satisfait par un "interface" entre le langage de programmation et le système, cet interface contenant des instructions spéciales appelées instructions d'entrées-sorties.

Ces instructions n'ont de sens que du fait du décalage entre la demande d'accès aux informations exprimée dans un programme et la fonction d'accès elle-même réalisée dans le système.

Le projet CIVA se place dans l'optique d'un système monolangage qui veut aborder tous les problèmes de mise en oeuvre d'un travail. Dans cette optique aucune instruction d'entrée-sortie n'est nécessaire et nous verrons qu'aucune instruction particulière n'a eu besoin d'être ajoutée au langage pour traiter les enregistrements d'un fichier.

3.2. ROLE DU MODULE DE COMMANDE DANS LE SYSTEME DE GESTION DE FICHIERS

3.2.I. - Principe général de l'utilisation des fichiers

Dans la définition du système de gestion de fichiers mis en place dans le projet CIVA, notre but principal a été d'établir une certaine indépendance entre d'une part la description des traitements (définis dans les unités formant une unité de traitement) et d'autre part leur exécution (lancée par l'éxécution d'un module de commande).

Lorsque l'utilisateur écrit ses programmes, la question du support de l'information ne l'intéresse guère : la seule chose qu'il peut savoir c'est que tel objet peut être en mémoire centrale, tel autre doit être en mémoire secondaire. Ce choix de l'implantation d'un objet dépend de deux critères : la durée de vie de l'objet et sa taille. Un objet de taille importante ou de durée de vie supérieure à celle d'une unité de traitement devra être implanté sur mémoire secondaire. Or, nous venons de voir qu'un fichier logique était une collection d'informations sans précision de support. Ce seront donc les fichiers logiques qui seront déclarés et traités dans les unités autres que le module de commande.

Ce n'est qu'au moment de l'exploitation des programmes que l'utilisateur va se poser la question de savoir sur quelle unité périphérique il va installer ses fichiers et c'est donc dans le module de commande qu'il fera part au système de ce choix. Aussi, le module de commande doit permettre à l'utilisateur de définir les supports sur lesquels il veut travailler, c'est-à-dire les fichiers physiques.

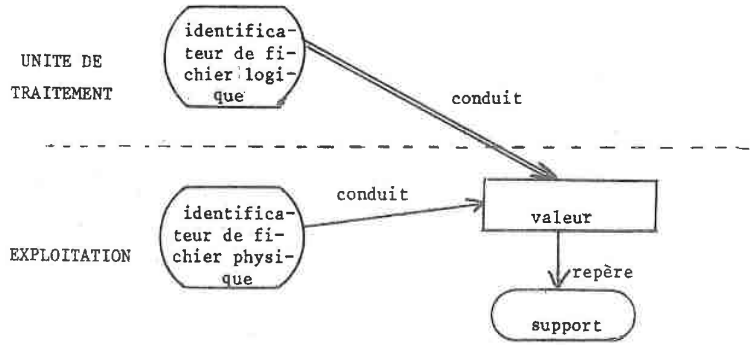
Ce module doit également lui permettre d'établir la liaison entre un fichier logique traité dans les modules de son application et un fichier physique déclaré dans le module de commande.

3.2.2. - La liaison entre fichier physique et fichier logique

Cette liaison est établie lorsque l'utilisateur, dans son module de commande, écrit un appel de la procédure LIAISON définie dans la classe du système . Cet appel s'écrit :

```
LIAISON ( <identificateur de fichier logique>,
          <identificateur de fichier physique>,
          <utilisation de fichier>
          [, PERMANENTE] ) ;
```

L'appel de cette procédure permet donc au système CIVA d'établir le lien entre d'une part un ensemble d'opérations définies sur un fichier physique dans un programme et d'autre part les informations enregistrées sur un support. Pour représenter cette liaison nous utiliserons la terminologie présentée dans (VERJUS [49])



Une fois cette liaison établie, l'utilisateur peut alors demander l'éxécution d'une unité de traitement qui contient les opérations définies sur le fichier logique.

Comme dans les systèmes d'exploitation classiques il nous a semblé bon de définir une protection des informations contre des opérations non souhaitées par l'utilisateur, opérations définies dans les programmes. Le fichier physique doit être protégé à l'intérieur même de l'application. Cette protection est assurée par la déclaration du type d'utilisation qui est fait du fichier au cours d'une certaine exploitation.

<utilisation du fichier> ::= <type d'utilisation> | (<liste de type d'utilisation>)

<type d'utilisation> ::= LECTURE | ECRITURE | MISE A JOUR

Le fait d'indiquer un ou plusieurs types d'utilisation d'un fichier permet au système de n'établir la liaison entre l'identificateur du fichier logique et la valeur repérant le support que pour les utilisations indiquées. Si au cours de l'exploitation d'une unité de traitement, le système est sollicité pour réaliser une opération qui ne correspond pas à un type d'utilisation déclaré dans l'appel de la procédure LIAISON, il n'y a pas de fonction d'accès au support : le module initial émet un message d'erreur et l'exploitation de l'unité de traitement est abandonnée.

Dans certains traitements, un fichier logique peut désigner toujours le même support, c'est-à-dire le même fichier physique d'une exploitation à l'autre.
Alors que dans les systèmes d'exploitation classiques, il est toujours nécessaire de rappeler la liaison entre le fichier logique et le support, dans le système CIVA, l'utilisateur a la possibilité de déclarer que cette liaison est permanente en indiquant le paramètre PERMANENTE dans l'appel. Dans ce cas, la liaison est conservée par le système d'une exploitation à l'autre et l'utilisateur n'est pas obligé de la rappeler. La liaison ainsi établie ne sera détruite par le système CIVA que lorsque l'utilisateur fera un nouvel appel à la procédure LIAISON avec pour paramètre le fichier logique concerné, ou bien lorsque le fichier logique sera détruit.

Exemple :

LIAISON (F1, FICHPHY, LECTURE) ;

Par cet appel, l'utilisateur indique au système que l'identificateur F1 du fichier logique conduit à la même valeur que l'identificateur FICHPHY du fichier physique. La fonction "conduit" de F1 à cette valeur n'existe que pour les opérations de lecture du fichier logique. Dans l'unité de traitement, toutes les opérations de lecture de F1 s'effectueront sur le support repéré par la valeur. Toute autre opération sur F1 n'aura aucune action sur le support.

3.3. DEFINITION ET UTILISATION D'UN FICHIER LOGIQUE

Un fichier logique est un ensemble d'informations sans précision de support. Ces informations peuvent être traitées dans n'importe quel module d'une application, autre que le module de commande, le rôle fondamental de celui-ci n'étant pas la manipulation de fichier logique.

3.3.I. - Classification des fichiers logiques

Il existe deux types de fichiers logiques (CHABRIER [5]) :

- un fichier logique est dit interne au système si ce fichier est le résultat d'une exécution d'un module CIVA, que ce soit un module d'acquisition ou non, excepté les modules d'édition.
- un fichier logique est dit externe au système CIVA dans tous les autres cas. Il contient des informations brutes qui ne peuvent être traitées directement par un module CIVA, sinon par un module d'acquisition.

3.3.2. - La déclaration d'un fichier logique

3.3.2.I. Fichier logique externe

La déclaration d'un fichier logique externe est réalisée par l'instruction "% FICHIER" qui figure dans un métamodule d'acquisition.
Nous ne parlerons pas d'avantage de ces fichiers ici, leurs déclarations et leurs traitements ayant été déjà présentés par ailleurs (CHABRIER [5]).

3.3.2.2. Fichier logique interne

a) un fichier logique est une file,

Un fichier logique est une collection d'informations : or, dans le projet CIVA, un tel objet a déjà été défini sous sous le nom de file qui est un ensemble d'emplacements totalement ordonnés.

Il semblerait donc raisonnable de réunir ces deux entités en disant que les fichiers logiques sont des files de CIVA possédant certaines particularités. L'avantage de regrouper la notion de fichier logique et la notion de file est très grand : nous n'avons pas alors à introduire dans notre langage une nouvelle entité et un nouveau type de déclaration pour les fichiers logiques. De plus, en considérant que file et fichier logique ne font qu'un, toutes les instructions applicables aux files, le sont aux fichiers logiques. Nous n'avons donc pas à définir de nouvelles instructions dans le langage.

Un fichier logique est donc défini par l'utilisation de la déclaration : FILE.

Une file peut être considérée comme un ensemble d'objets élémentaires contenant un ensemble de valeurs et un ensemble d'emplacements, sans précision de support. Dans l'ensemble des files, nous distinguerons deux sous-ensembles :

- le sous-ensemble des files internes qui sont des objets créés par un module autre que le module initial ou le module de commande. Une file interne est donc un objet temporaire dont la durée de vie est au plus égale à celle du module qui le crée. Une file interne est implantée en mémoire centrale. Tous les éléments d'une telle file sont donc présents en mémoire à un instant donné.

- le sous-ensemble des files externes ou fichiers logiques qui sont des objets créés par le module initial (dans ce cas, ils sont permanents) ou par un module de commande (ils sont alors temporaires). La durée de vie des informations rangées dans un fichier logique peut donc être supérieure à une exécution d'une chaîne de traitement.

Pour cette raison, ces informations seront stockés sur des mémoires secondaires. Les éléments d'un fichier logique appelés enregistrements ne sont donc pas tous présents simultanément en mémoire centrale.

b) Différence entre file interne et fichier logique

Le but de cette réunion des files internes et des fichiers logiques sous la même identité est de réduire les différences qui existent entre ces deux notions à des différences d'implantation uniquement. Du fait de leur implantation différente, la même opération réalisée sur une file interne ou sur un fichier logique au niveau du langage évolué n'entraîne pas les mêmes opérations physiques au niveau de la machine. Par exemple, l'opération qui consiste à passer d'un élément d'une file interne à l'élément suivant correspond à un changement de la valeur du pointeur associé à la file, car tous les éléments sont présents en mémoire. Par contre cette même opération réalisée sur un fichier logique, utilisé par exemple en lecture, consiste à amener l'enregistrement suivant en mémoire centrale et à écraser la valeur affectée à l'enregistrement qui vient d'être traité.

Cette différence d'implantation peut être constaté dans la plupart des langages de programmation dans lesquels ont été définis des opérations sur les files internes (par exemple, positionnement au premier élément, passage à l'élément suivant ...) et d'autres opérations sur les fichiers logiques (lecture d'un enregistrement, écriture...), ces deux types d'opérations s'écrivant avec des ordres différents.

Par exemple, en FORTRAN, une file interne correspond à la notion de tableau et on écrira :

```
I = 1
T (I) = ..... (positionnement au premier élément)
I = I + 1
T (I) = ..... (passage à l'élément suivant)
```

Par contre les fichiers logiques sont traités avec des instructions d'entrées-sorties :

```
READ (100,200) .....(lecture de l'enregistrement suivant)
```

Dans le langage ALGOL W (CHION [43]), le programmeur dispose également d'instructions d'entrées-sorties pour les files externes :

```
SORTIR (A, B, C)      (écriture d'un enregistrement)
ENTRER (X, Y, Z)      (lecture d'un enregistrement)
```

Ces instructions sont différentes pour les files internes :

```
CHAINE (20) TABLEAU T
I = 1                 (positionnement d'un enregistrement)
T (I) = "ABCDE"
I = I + 1             (passage à l'élément suivant)
```

Mais pour l'utilisateur d'un langage de haut niveau, il est bon de n'avoir qu'un seul jeu d'instructions permettant de réaliser ces opérations que ce soit sur des files internes ou bien sur des fichiers logiques. En effet, pour lui, dans son programme, ce qui l'intéresse c'est de pouvoir accéder à la valeur correspondant à un élément de file, que celui-ci soit en mémoire centrale ou non. Il est donc souhaitable que ce soit le système qui génère les opérations nécessaires dans tous les cas de manière transparente à l'utilisateur.

c) <u>le fichier logique dans le projet CIVA</u>

Dans le projet CIVA, un fichier logique est donc une file : les opérations définies sur les files sont donc applicables à la fois aux files internes et aux fichiers logiques. Le programmeur a à sa disposition un jeu unique d'instructions pour les deux cas. Celui-ci se contente d'indiquer quel est l'élement de la file sur lequel il veut travailler et c'est le compilateur CIVA qui génère les ordres correspondants suivant le cas où cet élément est implanté en mémoire centrale ou bien en mémoire secondaire.

Nous voyons donc que le compilateur, pour traduire une opération demandée sur une file est amené à générer soit une opération de changement de la valeur du pointeur associé (cas d'une file interne) soit une opération d'entrée-sortie (cas d'un fichier logique). Il est donc nécessaire que dans le texte même du programme, ce compilateur puisse savoir à quel type de file il a à faire.

Le système doit connaître l'implantation de la file. Pour satisfaire ce besoin, deux possibilités nous sont offertes :

- Une demande d'opération sur une file quelle que soit son implantation se traduit par le compilateur par l'appel d'un module système. Dans son module de commande, avant de demander une exécution, le programmeur indique quelles sont les files déclarées qui correspondent à des fichiers logiques. A chaque file déclarée par l'utilisateur, le système associe un bit qui est positionné par le module initial lorsque la file correspondante est un fichier logique. Au cours de l'exécution, le module système appelé teste l'état du bit et il exécute soit une opération de changement de valeur d'un pointeur si le bit n'est pas positionné, soit une opération d'entrée-sortie si le bit est positionné.

- La deuxième possibilité est la suivante : le programmeur indique dans l'unité de traitement, avant de pouvoir utiliser une file, quelle doit être son implantation. A la rencontre d'une demande d'opération sur cette file, le compilateur peut alors générer soit une instruction de modification de pointeur si la file est interne, soit une instruction d'entrée-sortie si la file est un fichier logique.

L'avantage de la première solution réside dans le fait que le programmeur, lorsqu'il écrit des modules, ne se soucie pas du tout de l'implantation des files qu'il utilise. C'est seulement lorsqu'il demande une exécution qu'il se pose la question de savoir où doivent être implantés les éléments de la file.

Par contre, cette solution présente un inconvénient important : le système est alourdi par la gestion des files et les temps d'éxécution s'allongent d'autant.
La deuxième possibilité oblige le programmeur à se poser la question de l'implantation d'une file pendant l'écriture de ses programmes. Il doit savoir si une file doit être implantée en mémoire centrale ou en mémoire secondaire. Mais l'avantage de cette méthode est l'amélioration des temps d'éxécution, le système étant beaucoup plus performant.
Nous avons choisi la deuxième possibilité car la performance du système est un des buts que nous avons poursuivis. De plus, on peut facilement constater que pour l'utilisateur la question de l'implantation d'une file se pose bien avant la programmation (en général au cours de l'analyse) et l'inconvénient de la solution choisie est donc peu contraignant.

Afin de pouvoir reconnaître les files internes des fichiers logiques, nous avons introduit deux éléments nouveaux dans le langage.

- au moment de la déclaration d'une file, le compilateur doit reconnaître une file interne d'un fichier logique. Pour cette raison nous avons ajouté un attribut supplémentaire dans la déclaration : l'implantation. Cet attribut peut prendre deux valeurs INTERNE ou EXTERNE, la première valeur étant prise par défaut.

Exemple

```
Classe A ; utilise C ;
F file externe type ENREGISTREMENT ;
X file (max = 10) entier ;
Y file interne (35) booléen ;
fin classe ;
```

La classe A contient les déclarations :

- *d'un fichier logique F*
- *de 2 files internes X et Y.*

- de plus, lorsque le compilateur rencontre un fichier logique, il doit connaître le type d'utilisation réalisée sur ce fichier. Pour cette raison, dans un module, au moment de l'utilisation d'un fichier logique, le programmeur écrit un appel à une procédure particulière, la procédure UTILISATION définie dans la classe système. Cette procédure admet deux paramètres : le nom du fichier logique et le type d'utilisation. Son appel s'écrit :

```
<appel de la procédure UTILISATION> ::=
        UTILISATION ( <identificateur de fichier logique>,
                      <mode d'utilisation> ) ;
<mode d'utilisation> ::= LECTURE | ECRITURE | MISE A JOUR
```

Exemple

```
classe A ;
utilise C ;
F file externe type ENR ;
fin classe ;
module M ;
utilise A ;
   :
UTILISATION (F, LECTURE) ;
```

3.3.3. - L'application des opérations définies sur les files aux fichiers

Les opérations sur les files définies dans (DERNIAME-[8]:chapitre 7) sont donc utilisables que la file soit un fichier logique ou non. Illustrons ici leur utilisation dans le cas des fichiers logiques.

3.3.3.1. Les opérations globales

Celles-ci concernent un traitement global des files, c'est-à-dire qu'elles peuvent être exécutées sur le fichier dans sa totalité par opposition aux opérations qui ne concernent qu'un élément.
La première instruction globale est la concaténation qui permet de réunir deux ou plusieurs files en une seule.

La deuxième instruction globale est l'instruction d'affectation qui permet de transférer les valeurs contenues dans les emplacements d'une file dans les emplacements d'une autre file.

Exemples

1) F1 = F2

Cette instruction permet d'affecter une valeur au fichier logique F1. Chaque enregistrement de F1 reçoit comme valeur celle contenue dans l'élement correspondant de F2. F2 peut être une file interne ou un fichier logique.

2) F1 = F2 (1:15), F3, F2(16:.)

Dans ce cas, au fichier logique F1 est affectéela suite de valeurs composée des 15 premiers enregistrements de F2, de la totalité de F3 et des derniers enregistrements de F2.

Remarques

- dans les opérations globales, les fichiers logiques indiqués à gauche et à droite du signe "=" doivent désigner des fichiers physiques ayant même type d'enregistrement : les longueurs des enregistrements logiques et physiques doivent être égales.

- l'indication du mode d'utilisation du fichier est facultative, si ce fichier fait l'objet d'une opération globale :

+ le fichier logique dont le nom figure à gauche du signe "=" a implicitement un mode d'utilisation MISE A JOUR ou ECRITURE

+ le (ou les) fichier(s) logique(s) dont le (ou les) nom(s) figure(nt) à droite du signe "=" ont implicitement le mode d'utilisation LECTURE

F1 = F2 ;
est analogue à :
UTILISATION (F1, ECRITURE) ;
UTILISATION (F2, LECTURE) ;
F1 = F2 ;

En plus de ces deux instructions globales, il a été défini des instructions particulières aux ensembles, un ensemble au sens CIVA étant une file munie d'un préordre.

Exemple

Si B est un fichier logique composé d'enregistrements de type structure dont C, D, E, sont des champs,

B par C croissant, D, E décroissant ;

est une opération qui trie le fichier B selon les critères indiqués.

3.3.3.2. Les opérations sur un élément de file

D'autres opérations ont été définies sur les files. Ce sont les opérations permettant de manipuler un élément. Lorsque la file est un fichier logique, les opérations sur un élément portent donc sur des enregistrements. Un élément de file est repéré grâce à l'élément courant affecté à la file : c'est un identificateur qui "pointe" vers un objet de la file.
Nous pouvons rappeler brièvement les principales opérations.

3.3.3.2.I) instructions définies sur les éléments courants

- l'instruction "en" permet d'attribuer à l'élément courant le nom d'un emplacement de la file par l'intermédiaire d'un identificateur de variable indicée ou par rapport à l'emplacement actuellement pointé. Par exemple, si X est l'élément courant d'un fichier logique, l'écriture de : X en X + 1 ; permet de passer d'un enregistrement à l'enregistrement suivant dans le fichier logique.
- l'instruction "premier" a pour effet de faire pointer à l'élément courant le premier emplacement de sa file attachée.

Exemple

UTILISATION (F, LECTURE) ;
X de F (10, .) ; premier X ;
L'instruction "premier" permet de pointer vers le 10ème enregistrement de F. F étant utilisé en lecture, cette instruction amène donc le 10ème enregistrement en mémoire centrale.

- de même, l'instruction "dernier" permet de faire pointer à l'élément courant le dernier élément de la file associée. Si la file est un fichier logique, cette instruction peut être utile pour les rallongements du fichier.

Exemple

UTILISATION (F, MISE A JOUR) ;
X de F ;
dernier X ;
positionne sur le dernier enregistrement du fichier logique.
X en X + 1 ;
X = (DUPONT, JEAN, 3100) ;
permet de créer un nouvel enregistrement en fin de fichier.

- toutes les instructions que nous venons de voir concernent l'accès séquentiel dans un fichier logique. Toutefois, dans le projet CIVA, nous admettons également les fichiers d'organisation séquentielle indexée. D'autres instructions ont été définies pour permettre l'accès direct à un élément de file.
- Si le programmeur désire utiliser l'élément courant de la file, il peut réaliser un accès direct grâce à l'instruction SOIT.
  Sinon il est toujours possible d'utiliser la clé de l'élément.

Exemple

UTILISATION (F, LECTURE) ;
X de F ;
Soit X tel que A = 3 ;
permet de pointer vers le premier enregistrement du fichier logique F dont le champ A contient la valeur 3. A peut être considéré comme une clé de l'enregistrement, au sens habituel du terme.

Le programmeur aurait pu écrire :
F(A) = 3 ;
Le résultat aurait été identique.

3.3.3.2.2) instructions définies sur les éléments eux-mêmes.

- la première instruction est l'affectation. Cette instruction permet d'affecter une valeur à un élément de file.

Exemple

F *file externe type* A ;
type A *structure* (X *entier*, Y Réel, Z *file* (5) car) ;
UTILISATION (F, ECRITURE) ;
X de F ; premier X ;
X = (3, 5.2, "ABCDE") ;
CO : *cette instruction affecte une valeur au premier enregistrement de* F CO ;
X en X + 1 ;
X = (4, 3.1, "XYZWU") ;
CO : *l'instruction en permet d'écrire le premier enregistrement de F et fait pointer X vers le deuxième enregistrement* CO ;

- une autre instruction définie est l'instruction "supprimer".

Exemple

UTILISATION (F1, MISE A JOUR) ;
supprimer X tel que A>35 et A<42 dans F1 ;
Cette instruction permet de supprimer dans F1 tous les enregistrements tels que le champ A de X est compris entre 35 et 42.

3.3.3.3. Les calculs itératifs : l'instruction "pour chaque"

Une instruction permettant de définir des calculs itératifs sur les files est l'instruction " pour chaque ".

Exemple

On désire imprimer certains enregistrements d'un fichier logique remplissant les conditions suivantes :

- *A étant un champ à valeur entière de l'enregistrement, A ne doit pas être nul.*
- *A doit être inférieur à 100*

Pour cela il suffit d'écrire :

UTILISATION (F, LECTURE) ;
pour chaque X de F tel que A≠0 et A<100 faire
imprimer (X) ; fpc ;

On peut trouver un exemple général d'utilisation des fichiers logiques dans le projet CIVA dans (HERTSCHUH [14] - Annexe : analyse en CIVA d'une application "la paie du lait").

3.4. DEFINITION ET UTILISATION D'UN FICHIER PHYSIQUE

Un fichier physique représente d'une part un support bien déterminé et d'autre part un ensemble d'informations enregistré sur ce support. Comme nous l'avons vu en 3.2.I., les fichiers physiques sont déclarés dans un module de commande.

3.4.I. - Classification des fichiers physiques

Dans le projet CIVA, les fichiers physiques sont classés en deux catégories :

- un fichier physique est dit catalogué si tous les renseignements permettant de décrire et de retrouver ce fichier sont permanents dans le système. Un tel fichier est donc créer par le module initial qui est le seul élément permanent défini dans CIVA. Les fichiers catalogués sont donc, en général, des fichiers enregistrés sur support magnétique dont la durée de vie est, en principe, supérieure à celle d'un module de commande (Cette notion correspond sensiblement à la notion classique de fichier permanent catalogué). Mais l'utilisateur peut très bien cataloguer un fichier de type périphérique (carte ou imprimante) s'il désire que le système conserve les renseignements le concernant.

- un fichier physique est dit non-catalogué si les renseignements le concernant ne sont pas permanents dans le système. Un tel fichier est créer par le module de commande et sa durée de vie est donc égale à celle de ce module. Avant chaque utilisation de ce fichier, l'utilisateur est donc obligé de définir ses paramètres (cette notion correspond donc à la notion de fichier temporaire).

3.4.2. - La déclaration d'un fichier physique

Un fichier physique est entièrement défini par un objet déclaré dans un module de commande ou dans la classe d'application de type structurée fichier.
La déclaration d'objets de types implicites dans un langage d'exploitation a déjà été utilisée pour le langage du système OSL - 2 (ALSBERG [20])

- Si l'objet structuré de type fichier est déclaré dans un module de commande, alors il est local à celui-ci et est détruit à la fin de l'éxécution du module. Un tel objet ne peut donc repérer qu'un fichier physique non catalogué, c'est-à-dire de type temporaire, puisque les renseignements le concernant ne sont pas conservés par le système.

- Par contre, si l'objet structuré est déclaré dans la classe d'application, alors son nom ainsi que la valeur qui lui est affectée est conservé par le système d'une exploitation à l'autre, ce qui est le propre de la classe d'application. Dans ce cas, l'objet repère un fichier physique de type catalogué. L'utilisateur peut affecter à cet objet une valeur qui définit le fichier au moment de la création de celui-ci. Cette valeur reste constante pendant les exploitations suivantes et l'utilisateur n'est pas obligé de la répéter avant chaque utilisation du fichier.

Le type fichier est déclaré dans la classe système de la façon suivante :

```
type fichier structure (
        localisation    type    localisation,
        spécification   type    spécification,
        identification  type    identification
        protection      type    protection) ;
```

L'objet structuré de type fichier est déclaré par :

```
<identificateur> type fichier ;
```

L'identificateur est une chaîne de caractères qui indique le nom du fichier physique. Cette chaîne est affectée au fichier physique au moment de sa création et doit être rappelée chaque fois que l'on voudra accéder au fichier.

Exemples

1) FICHPHY *type fichier* ;
   Cette instruction déclare un fichier physique de nom FICHPHY. Ce fichier est non catalogué, la déclaration étant faite dans le module de commande.

2) CREER (FICH1 *type fichier* ;) ;
   L'appel de CREER permet de créer une déclaration d'objet dans la classe d'application. Cette déclaration concerne un fichier physique de nom FICH1. Ce fichier est catalogué.

Une structure de type fichier contient quatre champs, chacun de ces champs permettant de définir le fichier physique. Nous allons donc analyser chacun de ces champs.

3.4.3. - La localisation d'un fichier physique

Le premier type d'information permettant de définir un fichier physique est la localisation de ce fichier, c'est-à-dire la définition de son support. La localisation est définie en CIVA comme un objet d'un module de commande ou de la classe d'application de type structuré localisation.

Le type de cette structure est défini dans la classe système par :

```
type localisation structure (
        UNIT file (max = 5) car,
        NUM file type NUMUNIT) ;
type NUMUNIT file (max = 6) car ;
```

Le premier champ, UNIT, introduit le type d'unité sur laquelle est implanté le fichier physique (bande, disque, imprimante...).
Le deuxième champ, NUM, introduit le numéro de volume ou la liste des numéros de volumes (dans le cas d'un fichier multivolume) qui contiennent le fichier physique.

L'objet structuré de type localisation est déclaré par :

```
<identificateur> type localisation ;
```

Les valeurs susceptibles d'être affectées à une localisation de fichier physique ne sont pas quelconques. Le premier champ de la localisation ne peut recevoir que des valeurs prédéfinies qui sont les suivantes :

"BANDE"	indique que le fichier physique est enregistré sur bande magnétique.
"IMPR"	pour un fichier sur imprimante
"CARTE"	" " " sur lecteur de cartes
"DIMAS"	" " " sur disque amovible DIMAS
"RAD"	" " " sur disque inamovible RAD.

Exemples

1) LOC *type localisation* ;
   LOC = ("IMPR") ;
   LOC est un objet structuré du module de commande. La valeur de LOC représente une localisation de fichier physique sur imprimante.

2) CREER (LOC *type localisation* ;) ;
   LOC = ("BANDE", (BO1, BO2)) ;
   LOC, définie par la procédure CREER, est donc une variable de la classe d'application de l'application courante. C'est donc un objet permanent et accessible dans toute l'application. Sa valeur pourra donc être utilisée à nouveau dans n'importe quel autre module de commande.

3.4.4. - La spécification d'un fichier physique

Le deuxième type d'information permettant de définir un fichier physique est la spécification de ce fichier, c'est-à-dire la définition du type d'enregistrement du fichier sur son support. En CIVA, la spécification est également définie comme un objet d'un module de commande ou de la classe d'application de type structuré spécification.
Si l'objet est déclaré dans le module de commande, la valeur affectée à cet objet ne peut être conservée par le système. Par contre si l'objet appartient à la classe d'application, il est permanent et sa valeur peut être utilisée dans n'importe quel module de commande.

Le type spécification est défini dans la classe système par :

```
type spécification structure (
     ORG    file (2) car,
     ENR    structure   (FORMAT   file (2) car,
                         TENRL    entier,
                         TENRP    entier),
     TAILLE file (2) entier) ;
```

- le premier champ, ORG, permet d'introduire le type d'organisation du fichier. Nous pouvons signaler que, dans le langage CIVA, deux types d'organisation sont envisagés : l'organisation séquentielle et l'organisation séquentielle indexée.

- le deuxième champ, ENR, indique le type des enregistrements du fichier :
  - + FORMAT définit le format de l'enregistrement logique (longueur fixe, variable ou indéfinie)
  - + TENRL permet de définir la longueur de l'enregistrement logique
  - + TENRP permet de définir la longueur de l'enregistrement physique.

Remarque

Pour un enregistrement logique de longueur fixe, TENRL et TENRP correspondent aux longueurs effectives respectivement de l'enregistrement logique et physique. Pour un enregistrement logique de longueur variable ou indéfinie, TENRL et TENRP correspondent aux longueurs maximales.

- le troisième champ de la structure appelé TAILLE introduit la taille du fichier sur son support. Cette taille n'est utile que pour les fichiers implantés sur disque. La première valeur entière de la file définit la taille de la partie primaire du fichier (c'est-à-dire des blocs "données" du fichier). La deuxième valeur entière définit la taille de la partie secondaire du fichier (c'est-à-dire,pour un fichier séquentiel, la taille de l'espace supplémentaire à allouer en cas de débordement ; pour un fichier séquentiel indexé, la taille de l'espace à réserver aux blocs index et aus blocs de débordement).

Remarque

sur le CII 10070, les tailles indiquées seront exprimées en quanta (1 quantum = 8192 octets = 8 K octets).

L'objet structuré de type spécification est déclaré par :

<identificateur> type spécification ;

Les valeurs susceptibles d'être affectées à une spécification de fichier physique ne sont pas quelconques.

- le champ ORG de la structure ne peut recevoir que les valeurs prédéfinies suivantes :
  - "SE" pour un fichier d'organisation séquentielle
  - "SI" pour un fichier d'organisation séquentielle indexée.

- le champ FORMAT doit lui aussi des valeurs prédéfinies qui sont :
  - "LX" pour une longueur fixe
  - "LV" pour une longueur variable
  - "LI" pour une longueur indéfinie

Exemples

1)
```
SPE   type spécification ;
SPE = ("SE", ( "LX", 80, 800), (5, 3) ) ;
```

SPE est un objet structuré déclaré dans un module de commande donc local à celui-ci. La valeur affectée à SPE définit une spécification d'un fichier physique d'organisation séquentielle, d'enregistrement logique de longueur fixe de 80 octets, d'enregistrement physique de longueur 800 octets.

Ce fichier a une zone primaire de 5 quanta et une zone secondaire de 3 quanta.

2) CREER (S type spécification ;) ;
S = ("SI", ("LV", I32, I32)) ;

S, défini par un appel de la procédure CREER, est donc un objet de la classe d'application. S est donc permanent et accessible dans toute l'application. Sa valeur pourra donc être utilisée dans un autre module de commande. Cette valeur définit la spécification d'un fichier d'organisation séquentielle indexée, d'enregistrements logiques et physiques de longueur variable, inférieure ou égale à I32 octets.

3.4.5. - L'identification d'un fichier physique

L'identification complète d'un fichier physique est composée de deux types de renseignements :

- un renseignement obligatoire est le nom du fichier. Ce nom est une chaîne de caractères alphanumériques qui est affectée au fichier physique au moment de sa création et qui doit être rappelée avant chaque utilisation du fichier. Comme nous l'avons vu en 3.4.2, le nom d'un fichier est l'identificateur de l'objet élémentaire de type structuré fichier qui déclare le fichier physique.

- le deuxième type de renseignements regroupe l'identification complémentaire qui est facultative. Celle-ci regroupe trois éléments :
  + le nom de l'application spécifie l'identité de l'application propriétaire du fichier. Si un utilisateur d'une application veut utiliser un fichier défini dans une autre application, il devra indiquer le nom de cette application pour que le système puisse retrouver le fichier et faire les contrôles de protection.
  + le numéro de génération est une extension du nom du fichier. Lorsqu'il est précisé, il permet de distinguer plusieurs générations différentes d'un fichier de nom donné. Si un fichier est conservé sous plusieurs générations, son nom représente alors un nom de groupe généalogique.

  + le numéro de version est attribué par le propriétaire du fichier au moment de sa création. S'il est spécifié, c'est forcément à la suite du numéro de génération et il permet de demander à ce que le contrôle d'identification porte non seulement sur le nom et le numéro de génération mais également sur le numéro de version. Cela permet de s'assurer que l'on accède bien à une version spécifiée d'une génération donnée.

L'identification complémentaire d'un fichier physique est définie en CIVA comme un objet d'un module de commande ou de la classe d'application de type structuré identification.
Le type identification est défini dans la classe système de la façon suivante :

```
type  identification  structure (
        nom application file (max = 10) car,
        num génération entier,
        num-version entier) ;
```

Un objet structuré de type identification est déclaré par :

```
<identificateur> type identification ;
```

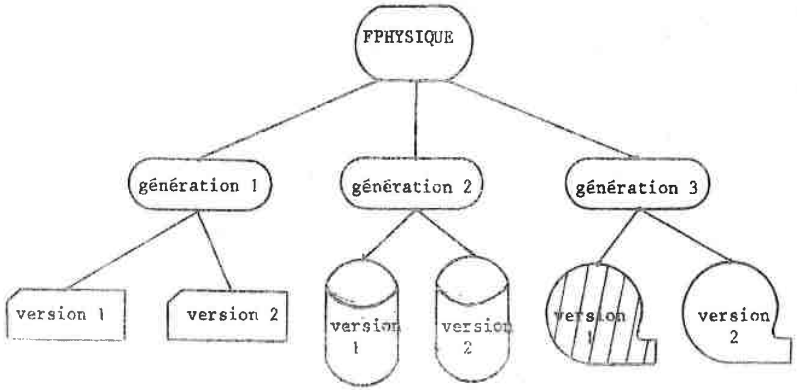
Les valeurs susceptibles d'être affectées à une variable de type identification sont entièrement libres.

Exemples

1)
```
FICHPHY  type  fichier ;
   I     type  identification ;
   I   =  ("PERSONNEL") ;
FICHPHY = (I) ;
```

La variable d'identification I est déclaré dans le module de commande. Elle est donc locale à celui-ci. Cette variable permet d'identifier un fichier physique déclaré dans une autre application de nom PERSONNEL. FICHPHY est l'identificateur d'un objet du module de commande de type fichier. FICHPHY est donc le nom d'un fichier physique. Ce fichier est déclaré dans l'application PERSONNEL.

2) *Soit un groupe généalogique de nom FPHYSIQUE possédant 3 générations (par exemple : grand-père, père et fils), chaque génération étant présente sous deux versions différentes. On peut représenter ce groupe de la façon suivante :*



Si on veut désigner le fichier hachuré, on écrira :

```
F PHYSIQUE type fichier ;
ID   type  identification ;
ID = (,3, 1) ;
FPHYSIQUE = (ID) ;
```

ou plus simplement :

```
FPHYSIQUE  type  fichier ;
FPHYSIQUE = ( (, 3 , 1) );
```

3.4.6. - La protection d'un fichier physique

Le dernier type d'information utile à la définition d'un fichier physique est sa protection. Dans le projet CIVA, nous avons défini deux niveaux de protection d'un fichier.

3.4.6.I. La protection du fichier à l'intérieur de l'application

Un fichier physique doit être protégé à l'intérieur même de l'application qui l'utilise. Comme nous l'avons vu en 3.2.2, cette protection est assurée au niveau de la connexion d'un fichier physique avec le fichier logique qui le désigne.

Lorsque l'utilisateur écrit un appel de la procédure LIAISON, le système établit la liaison entre les opérations d'entrée-sortie écrites dans les programmes et le fichier physique décrit dans le module de commande ou la classe d'application. Cette liaison n'est établie que pour certains types d'utilisation. Si au cours de l'exécution, une utilisation non voulue du fichier physique est demandée, le système émet un message d'erreur et l'exécution est arrêtée.

3.4.6.2. La protection du fichier vis-à-vis des autres applications

Nous venons de voir dans la présentation de l'identification d'un fichier physique en 3.4.5 qu'il était obligatoire d'ajouter le nom de l'application propriétaire lorsque l'on veut utiliser un fichier déclaré dans une autre application afin que le système puisse faire les contrôles de protection. Cette protection inter-application du fichier physique est définie par le propriétaire au moment de la création du fichier. Elle indique au système la liste des applications autorisées à lire le fichier et la liste des applications autorisées à écrire dans le fichier.

On peut tout de suite remarquer que l'indication de la protection inter-application pour un fichier non catalogué n'a aucun sens : celui-ci a une durée de vie inférieure ou égale à celle du module de commande qui le crée et ne peut donc pas être utilisé de l'extérieur.
Pour un fichier catalogué, l'indication de protection inter-applition est facultative. Si elle ne figure pas dans la définition d'un fichier physique catalogué, le fichier est alors protégé contre tout accès venant de l'extérieur de l'application propriétaire.

Dans le projet CIVA, la protection inter-application d'un fichier est également définie par un objet de la classe d'application ou du module de commande de type structuré protection.

Le type de cette structure est déclaré dans la classe système par :

```
type protection structure (
     lecture file (max = 5) type nom-application,
     écriture file (max = 5) type nom-application) ;
type nom-application file (max = IO) caractère ;
```

L'objet structuré de type protection est alors déclaré par :

⟨identificateur⟩ type protection ;

La file "lecture" de l'objet de type protection peut recevoir comme valeur la liste des noms des applications autorisées à utiliser le fichier en lecture (la liste est limitée à 5 noms). Si le fichier est commun à toutes les applications en lecture, il suffit d'écrire la valeur prédéfinie :

"COMMUNS"

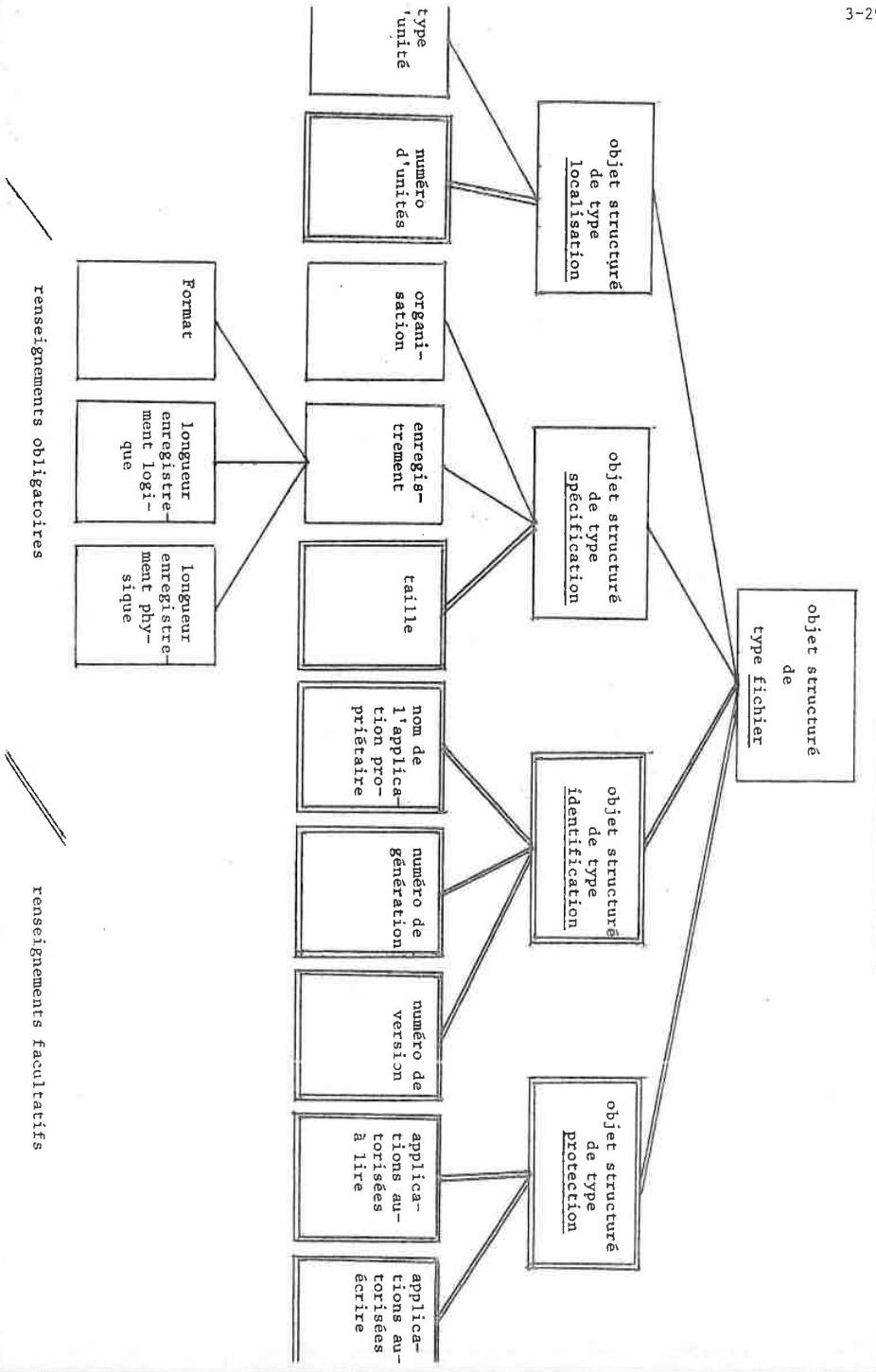
La file "écriture" peut recevoir comme valeur la liste des noms des applications autorisées à utiliser le fichier en écriture. DE même, si le fichier est commun à toutes les applications en écriture, il suffit d'écrire la valeur prédéfinie : "COMMUNS".

Exemple

```
PRO type protection ;
PRO = ( ("COMMUNS"), (PAIE, PERSONNEL) ) ;
```

PRO est l'identificateur d'un objet de type protection déclaré dans le module de commande. La valeur affectée à PRO définit la protection d'un fichier physique. Ce fichier est commun à toutes les applications en lecture. Seules, les applications PAIE, PERSONNEL et l'application courante sont autorisées à y écrire.

PRESENTATION GENERALE DE LA DEFINITION D'UN FICHIER PHYSIQUE



3.4.7. - Les opérations définies sur les fichiers physiques

Après avoir vu comment était défini un fichier physique dans le projet CIVA, nous allons présenter les différentes opérations qu'il est possible de réaliser sur ces fichiers dans le module de commande. Ces opérations sont de trois types : la création, la destruction et la copie d'un fichier physique dans un autre.

3.4.7.I. La destruction d'un fichier physique

La destruction d'un fichier physique peut être implicite ou explicite.

- La destruction du fichier est implicite au moment de la suppression de l'objet structuré de type fichier qui définit ce fichier.

Les fichiers non catalogués sont définis par des objets déclarés dans un module de commande donc qui lui sont locaux. A la fin de l'exécution du module de commande, le module initial détruit ces objets ainsi que tous les autres locaux. Les fichiers physiques non catalogués sont donc détruits à cet instant-là, puisqu'il n'existe plus de fonction d'accès pour eux.

Les fichiers catalogués sont définis par des objets déclarés dans la classe d'application. Ces objets sont permanents dans le système. Toutefois, comme n'importe quel autre objet déclaré dans cette classe, l'utilisateur a la possibilité de supprimer un objet structuré repérant un fichier par l'appel de la procédure SUPPRIMER (voir en 2.3.3). Le fait de supprimer un objet définissant un fichier entraîne la destruction de ce fichier également puisqu'il n'existe plus de fonction d'accès.

- Toutefois, il peut être intéressant pour l'utilisateur de détruire un fichier catalogué sans pour cela être obligé de détruire l'objet élémentaire dans la classe d'application. L'utilisateur peut désirer conserver les renseignements de ce fichier afin d'en recréer un nouveau, ayant absolument les mêmes caractéristiques que l'ancien. Pour pouvoir réaliser cette opération, nous avons défini une procédure particulière dans la classe de modification: la procédure DETRUIRE.

Etant définiedans la classe de modification, cette procédure fait l'objet d'une protection de la part du système. Seuls, les modules de commande ayant donné le mot de passe de l'application MODIFICATION peuvent explicitement détruire un fichier physique.

3.4.7.2. La création d'un fichier physique

L'opération de création d'un fichier physique est toujours implicite. Cette création peut avoir lieu dans deux cas :

- la création est implicite pour une opération de transfert de fichier physique réalisée par la procédure COPIER (voir ci-après en 3.4.7.3). A l'appel de cette procédure, le système copie un fichier physique émetteur dans un autre fichier physique récepteur. Ce dernier est donc créé à ce moment-là par le module initial.

- la création est également implicite lorsque l'utilisateur demande l'exécution d'une unité de traitement qui manipule un certain fichier logique dont le mode d'utilisation est l'écriture (voir en 3.2.2). Dans ce cas, l'exécution de l'unité de traitement entraîne la création du fichier physique qui est désigné par le fichier logique. Cette création est réalisée par le module initial si le fichier est catalogué, et par le module de commande s'il est non catalogué.

3.4.7.3. La copie d'un fichier physique

Les transferts de fichiers physiques d'un support à un autre sont des opérations fréquentes, surtout en informatique de gestion. Il est cependant regrettable de constater que, dans la plupart des systèmes d'exploitation actuels, il existe toute une "panoplie" d'instructions du langage de commande pour réaliser ces transferts. Nous appelerons donc copie d'un fichier l'opération qui consiste à transférer le contenu d'un fichier physique dans un autre fichier physique.

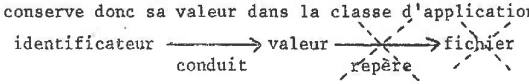
3.4.7.4. Les procédures de traitement de fichiers physiques

Les procédures réalisant les opérations définies sur les fichiers physiques sont au nombre de 2 :

- la procédure DETRUIRE de la classe de modification qui ne peut donc être appelée que par un module de commande habilité à utiliser l'application MODIFICATION. L'appel de cette procédure s'écrit :

DETRUIRE (<identificateur>) ;

L'identificateur est celui d'une variable de la classe de l'application courante, variable de type fichier. Cette variable définit un fichier catalogué et l'appel de DETRUIRE entraine la destruction du fichier mais pas celle de l'objet : la variable n'est pas détruite et conserve donc sa valeur dans la classe d'application.



- la procédure COPIER de la classe système réalise la copie d'un fichier. L'appel de cette procédure s'écrit :

COPIER (<identificateur 1 de fichier physique>,
<identificateur 2 de fichier physique>) ;

Cet appel réalise la copie du fichier physique émetteur repéré par l'identificateur 1 dans le fichier physique récepteur repéré par l'identificateur 2.

Exemples

```
1)  CREER (A type fichier ; B type fichier ; ) ;
    LOCA type localisation ; LOCA = ("RAD") ;
    LOC B type localisation ; LOC B = ("BANDE", (B01) ) ;
    CREER (SPE type spécification ; ) ;
    SPE = ("SE", ("LV", 80, 800) ) ;
    A =(LOC A, SPE, ("PERSONNEL") ) ;
    B = (LOC B, SPE) ;
    COPIER (A, B) ;
```

Cet appel permet de copier le fichier physique de nom A appartenant à l'application PERSONNEL dans le fichier physique de nom B de l'application courante.

Evidemment, cette copie n'est réalisée que si l'application courante est habilitée à lire le fichier A.

2) Dans le projet CIVA, il n'existe pas d'ordre spécial pour imprimer le contenu d'un fichier physique. En effet, l'opération d'impression d'un fichier physique est une opération de copie d'un fichier à imprimer dans un fichier physique localisé sur l'imprimante.
On écrira donc, par exemple ;

```
    A type fichier ;
    A = ( ("IMPR"), ("SE", ("LX", 132, I32) ) ) ;
    COPIER (FICHIER, A) ;
```

FICHIER est l'identificateur d'un objet structuré précédemment déclaré dans la classe d'application. Cet objet définit donc un fichier catalogué de nom FICHIER. Par l'appel de la procédure COPIER, le contenu de ce fichier est imprimé dans le fichier A.

Remarques

- les opérations de transfert de fichiers physiques non autorisées sont les suivantes :

+ le fichier émetteur est un paquet de cartes et le fichier récepteur est localisé sur l'imprimante. Cette opération est interdite car elle peut être réalisée directement en demandant au service d'exploitation l'impression d'un paquet de cartes (par exemple, à l'aide d'une tabulatrice).

+ le fichier émetteur est un paquet de cartes et le fichier récepteur est un fichier d'organisation séquentielle indexée localisé sur disque.

- dans certains cas, le fichier récepteur ne peut pas être utilisé directement. Ces cas correspondent à des opérations de sauvegarde. Par exemple, si le fichier émetteur est un fichier d'organisation séquentielle indexée et le fichier récepteur est localisé sur bande magnétique. Ce dernier ne peut être utilisé directement par des programmes CIVA. L'opération inverse de restauration du premier fichier doit être effectuée dans le module de commande.

3.5. EXEMPLE D'UTILISATION DU SYSTEME DE GESTION DE FICHIERS

Un utilisateur possède une chaîne de traitement composée de deux unités de traitement. La première, de module directeur CREATION, permet de créer un fichier d'organisation séquentielle. La deuxième unité de traitement, de module directeur EXPLOITATION utilise les renseignements regroupés dans ce fichier.
Le contenu du module de commande permettant de lancer cette exploitation peut être le suivant :

```
LOC type localisation ;
SPE type spécification ;
CREER (A type fichier ;) ;
LOC = ("DIMAS", (D01) ) ;
SPE = ("SE", ("LX", 80, 800), (5, 3) ) ;
A = (LOC, SPE,, ( ("COMMUNS"), (PAIE, PERSONNEL) ) ) ;
LIAISON (F1, A, ECRITURE) ;
CREATION ;
```

CO l'appel de la première unité de traitement lance l'exécution des ordres d'écriture dans le fichier logique F1. Ces opérations engendrent la création du fichier physique catalogué sous le nom de A. Ce fichier séquentiel est créé sur le DIMAS de numéro D01. Il est commun à toutes les applications en lecture. Seules, les applications PAIE, PERSONNEL et l'application courante peuvent y écrire CO ;

```
OUT type fichier ;
OUT = ( ("IMPR"), ("SE", ("LX, 80, 80) ) ) ;
COPIER (A, OUT) ;
```

CO l'appel de cette procédure permet à l'utilisateur d'obtenir le contenu des enregistrements du fichier A qui vient d'être créé CO ;

```
LIAISON (F2, A, LECTURE) ;
EXPLOITATION ;
```

CO la 2ème unité de traitement manipule le fichier logique F2. Ce fichier désigne le fichier physique qui vient d'être créé CO ; .

2ème PARTIE

INTERPRETATION DES MODULES DE COMMANDE DANS LE PROJET CIVA

Lorsque nous avons présenté d'une façon générale les modules de commande, nous avons dit que ces derniers étaient interprétés par le module initial noté M_o (cf Paragraphe I.5.). M_o peut être considéré comme le principal module du système, c'est-à-dire de l'application système du projet CIVA.

En fait le système CIVA a une taille relativement importante et nous avons pensé qu'il serait intéressant d'appliquer les principes mêmes de la programmation modulaire présentés dans notre projet à la conception et à la réalisation du système, à savoir description modulaire des informations et des traitements. Cette analyse nous a donc amené à réaliser un ensemble de modules qui représente le "module initial" vu par l'utilisateur et un ensemble de classes qui déclarent les informations de l'environnement de ce "module initial".

Toutefois, les modules et les classes de l'application système peuvent parfois utiliser des notions qui n'ont pas été introduites dans notre langage. Nous donnerons cependant leur description en langage CIVA, essentiellement pour pouvoir bien exprimer les relations qui lient ces unités entre elles.

C'est pour cette dernière raison que les programmes eux-mêmes sont écrits soit dans le langage METASYMBOL, langage symbolique du CII-10070, soit dans le langage $CIVA_o$ qui est un langage intermédiaire entre le langage symbolique et le langage CIVA (FIEGEL [12]).

De plus, nous ne donnerons pas dans cette seconde partie, l'analyse complète du "module initial" et de son environnement, certaines parties ayant été ou seront décrites par ailleurs (CRIDLIG [6] , BRETHES [4] , annexes). Nous nous limiterons donc à certains modules et à certaines classes qui nous ont parus les plus intéressants et qui montrent bien le fonctionnement général de l'interprète.

4

ENVIRONNEMENT DE L'INTERPRETE DES MODULES DE COMMANDE

Dans ce chapitre, nous allons présenter l'analyse de l'environnement du "module initial", c'est-à-dire l'analyse des différentes classes utilisées par le système CIVA ainsi que des objets qu'elles déclarent. L'étude de la représentation des informations constituant cet environnement nous a conduits à concevoir d'une part la représentation d'une classe d'application et d'autre part la représentation de la classe système.

De même que le "module initial" est représenté par un ensemble de modules de l'application système, les classes "d'application" et "système" seront également représentées par un ensemble de classes résultant de l'application même du principe de la programmation modulaire.

De plus, ce découpage nous semble nécessaire pour éviter la propagation des erreurs, qu'elles proviennent d'un utilisateur maladroit ou même d'un module du système. Ce besoin de protection peut également être ressenti d'une autre manière : en effet, bien que la "classe système" et la "classe d'application" soient utilisées implicitement par tout module écrit par un utilisateur, ces dernières doivent contenir des informations propres aux modules du système, ou des informations qui ne doivent pas être directement accessibles à l'utilisateur. Celui-ci, pour y accéder, devra écrire dans son module de commande des appels desprocédures que nous avons décrites dans les chapîtres précédents. De tels appels se traduiront par le lancement de l'exécution de modules du système qui ont le pouvoir d'accéder à ces objets.

4.I. LA REPRESENTATION D'UNE CLASSE D'APPLICATION

Nous avons vu au chapitre I qu'à chaque application créée dans le système CIVA est associée une "classe d'application" qui contient les définitions des paramètres communs à toutes les unités de l'application ainsi que les déclarations de ces unités (cf §I.2.).
Cette classe est implicitement utilisée par toutes les unités de l'application et en particulier par les modules de commande, dans lesquels l'utilisateur peut modifier soit des déclarations d'unités, soit des définitions de paramètres. Le "module initial" doit donc utiliser toutes les "classes d'application" pour pouvoir accéder à toutes ces variables ou unités.

Une "classe d'application" est représentée dans le système comme une bibliothèque formée de plusieurs fichiers ou partitions de fichiers partitionnés. La première indication fournie par un utilisateur dans le module de commande est le nom de l'application dans laquelle il veut travailler (cf: la procédure APPLICATION en 2.4.I.). Le "module initial M_o", une fois qu'il connaît l'application, doit gérer la classe de cette application et donc retrouver tous les renseignements contenus dans celle-ci.

4.I.I. - Définitions préalables

Ces définitions concernent la situation d'une unité CIVA par rapport au système :

- nous dirons qu'une unité est créée (ou interne au système) dans une application si cette unité a été l'objet d'un appel à la procédure CREER (cf en 2.4.2.1), c'est-à-dire si le système en possède le texte source.

- nous dirons qu'une unité est présente dans une application si le nom de cette unité a déjà été rencontrée par le "module initial", alors qu'il n'en possède pas encore le texte source.

En effet, lorsqu'on donne le texte source d'une unité qui "appelle ou"utilise" une autre unité qui n'a pas encore été créée, celle-ci devient"présente" sans être encore "créée".

- Dans tous les autres cas, nous dirons que l'unité est "absente" de l'application.

4.I.2. - Représentation des objets de la "classe d'application"

Dans un module de commande, l'utilisateur peut créer ou détruire des objets élémentaires ou structurés dans sa "classe d'application" (cf : la procédure CREER en 2.4.2.2 et la procédure SUPPRIMER en 2.5.4.1). Leur durée de vie peut donc être celle de l'application, ou plus petite à cause des créations et destructions explicites. Nous dirons encore que ce sont des objets permanents et le système CIVA doit donc les conserver d'une activation à l'autre.

Les objets élémentaires et structurés déclarés dans les classes d'application sont representés dans un fichier unique géré par M_o : le fichier des objets des classes d'application.

Ce fichier a une organisation partitionnée. Chaque partition de ce fichier contient la représentation des objets d'une "classe d'application". Le nom de la partition est le nom de l'application correspondante.

4.I.3. - Le fichier dictionnaire

A chaque unité créée ou présente dans une application, l'interprète du module de commande associe un numéro particulier : le numéro de matricule. Ce numéro de matricule définit l'unité de façon unique dans l'application et il suit l'unité tout au long de sa vie dans le système (il est utilisé par tous les modules du système, que ce soit le module initial, le compilateur ou le relieur.

Ces numéros de matricule ont été introduits dans notre système car il est plus rentable de gérer les unités d'une application en utilisant des numéros plutôt que les noms de ces unités. En effet, un numéro de matricule est alloué à une unité lorsque le système rencontre pour la première fois son nom et il suit cette unité pendant toute sa vie. Il ne change pas. Par contre, le nom d'une unité peut changer au cours de son existence (cf : la procédure REMPLACER en 2.5.3.I.3). De plus, un numéro peut servir d'indice par exemple et cette possibilité sera très souvent utilisée par le "module initial".

La correspondance entre le nom d'une unité et son matricule est représentée dans le fichier dictionnaire, défini pour chaque application. De plus pour chaque unité, le fichier dictionnaire contient un indicatif qui signale si l'unité est créée dans l'application ou seulement présente. Rappelons en effet qu'une unité peut être présente avant d'avoir été créée (cf § 4.I.I.).
D'un point de vue pratique, ce fichier a une organisation séquentielle. De plus, ayant une taille assez réduite, le fichier dictionnaire est implanté en mémoire centrale au moment de l'initialisation de la bibliothèque de l'application (cf § 5.I.2.). En effet, ce fichier étant très souvent consulté au cours de l'interprétation du module de commande, son implantation en mémoire (et plus particulièrement dans la zone dynamique locale au module initial) permet d'améliorer les performances globales du système.

4.I.4. - Le fichier descriptif

C'est le fichier descriptif des classes et des modules d'une application qui contient tous les renseignements nécessaires au système CIVA aux différents niveaux (module initial, compilateur, relieur) essentiellement, le type de l'unité (module, classe ou métamodule), sa protection, les classes qu'elle utilise et/ou les modules qu'elle appelle ....
Ce fichier a une organisation séquentielle indexée : chaque enregistrement peut être accédé directement en indiquant sa clé, celle-ci étant le numéro de matricule de l'unité décrite dans l'enregistrement.

```
classe fichier descriptif ;
type ADESC file (max = 10) caractère ;
DESC file structure (
          PROTDESC entier,
          NUMEROVERSION entier,
          NOMDESC type ADESC,
          MATDESC entier,
          ERRDESC entier,
          TYPEDESC code (module, classe, procédure, métamodule,,,,
                    classecom),
          ETATDESC code (source, exploitation, mise au point,
                    édité),
          CONSTDESC entier,
          VARDESC entier,
          MODDESC file (I28) bit,
          APPLDESC file (max = 5) type ADESC,
          NBMETA entier,
          METADESC file type MATDESC,
          NBCLAS entier,
          CLASDESC file type MATDESC,
          NBMOD entier,
          MODDESC file type MATDESC) ;
DESCRIPTIF de DESC ;
finclasse ;
```


D'un point de vue pratique, chaque enregistrement possède en tête un octet d'effacement. En effet, si l'utilisateur décide de supprimer une unité de son application (voir la procédure SUPPRIMER en 2.5.4.I), le module initial doit effacer l'enregistrement correspondant du fichier descriptif. Sous le système d'exploitation SIRIS 7 du IOO70, l'enregistrement d'un fichier doit posséder un octet spécial dit d'effacement si on veut pouvoir effacer des enregistrements de ce fichier (cf : système de gestion de fichiers SCF sous SIRIS 7 [3I]).

- PROTDESC indique la protection de l'unité qui a été demandée par son propriétaire au moment de la création (voir la procédure CREER en 2.4.2.I).
  PROTDESC = - 1 si l'unité est protégée
  0 si l'unité est commune à toutes les applications
  a si un certain nombre d'applications sont autorisées à employer l'unité (a = nombre d'applications autorisées $0 < a \leqslant 5$)
- NUMEROVERSION est un numéro qui permet d'identifier la version de l'unité dans l'application. Lors de la création de l'unité, ce numéro est initialisé à 1. Ensuite, à chaque fois que l'utilisateur fait une modification du texte de l'unité (modification globale ou partielle), ce numéro est incrémenté de 1. Ce renseignement est utilisé par le "module initial" pour les opérations d'inclusion d'unités externes, lorsque l'utilisateur demande l'exécution d'une unité de traitement (cf § 2.6.I).
- NOMDESC indique le nom de l'unité.
- MATDESC contient le numéro de matricule de l'unité. C'est la clé de l'enregistrement.
- ERRDESC contient le degré d'erreur détecté après compilation de l'unité (le degré d'erreur est 0 lorsqu'il n'y a aucune faute, de l'ordre de 3 pour des erreurs mineures apparemment récupérées par le compilateur, 7 lorsque l'exécution est hasardeuse mais possible, de l'ordre de 12 lorsqu'à priori il paraît inutile de passer à l'exécution.

- TYPEDESC est le code indiquant le type de l'unité.
- ETATDESC est le code qui indique l'état actuel de l'unité dans l'application.
- CONSTDESC contient la taille en octets de la zone réservée pour les constantes de l'unité, cette taille étant déterminée après compilation et servant au relieur (DENDIEN [7])
- VARDESC contient la taille en octets de la zone réservée pour les variables de l'unité. Cette taille est également déterminée par le compilateur et est utilisée par le relieur.
- MODDESC est une file de bits, jouant le rôle de booléens. Cette file est garnie par le compilateur. Elle indique au relieur quels sont les sous-programmes de la librairie CIVA qui sont appelés par l'unité. On peut raisonnablement supposer que la librairie contiendra au plus I28 sous-programmes.
  MODDESC (I) = vrai si le sous-programme I de la librairie est appelé par l'unité
  = faux sinon.
- APPLDESC est la liste des noms des applications autorisées à employer l'unité. Cette liste n'est créée qui si PROTDESC = a $>$ 0, c'est-à-dire que si le propriétaire de l'unité a indiqué, lors de la création, que son unité n'est partageable que par un nombre restreint d'applications.
- NBMETA est le nombre de métamodules appelés par l'unité. Ce nombre est déterminé par le module initial à la création de l'unité.
- METADESC est la liste des numéros de matricule des métamodules appelés par l'unité. Cette liste n'existe que si NBMETA $>$ 0.
- NBCLAS est le nombre de classes utilisées par l'unité. Ce nombre est aussi déterminé par le module initial à la création de l'unité.
- CLASDESC est la liste des numéros de matricule des classes utilisées. Cette liste n'existe que si NBCLAS $>$ 0.

- NBMOD est le nombre de modules appelés par l'unité. Ce nombre n'est déterminé que partiellement par le module initial. Il est exact lorsque l'unité a été compilée. En effet, nous verrons en annexes que le module initial ne différencie pas dans le texte source de l'unité un appel de module et un appel de procédure sans paramètre.
- MODDESC est la liste des matricules des modules appelés par l'unité. Cette liste n'est créée que si NBMOD $>$ 0.

4.1.5. - La représentation des textes sources des unités

4.I.5.I. Le fichier des textes sources

Lorsqu'un utilisateur désire créer une unité dans son application, le module initial doit rendre cette unité interne au système CIVA. En plus de la création d'un enregistrement dans le fichier descriptif et dans le fichier dictionnaire (ou bien du positionnement de l'indicatif de création si l'unité était déjà présente), l'interprète des modules de commande doit conserver le texte source de cette unité afin de pouvoir le soumettre ultérieurement au compilateur.

Les textes sources des différentes unités d'une application seront rangés dans un fichier propre à chaque application : le fichier des textes sources. Ce fichier a une organisation partitionnée. Chaque partition contient le texte source d'une unité de l'application. Le nom d'une partition est le numéro de matricule de l'unité. Ainsi le système peut donc accéder au texte source de cette unité, par l'intermédiaire du dictionnaire.

4.I.5.2. La forme du texte source

Le texte source d'une unité est rangé sous forme de chaîne compressée dans une partition du fichier des textes sources.
Nous n'avons par utilisé le programme de compression de SIRIS 7 (cf : normes de programmation SIRIS 7 [46]) car la forme compressée obtenue ne pouvait pas satisfaire tous nos besoins.

Pour nous, une chaîne compressée est une suite continue de caractères sans coupure logique. Physiquement, elle est découpée en articles de longueur égale à 1016 octets. La fin du texte source d'une unité est marquée par un enregistrement de longueur nulle. Le code des caractères formant la chaîne compressée est le code EBCDIC des caractères lus sur les cartes.

Cependant, la chaîne compressée comporte 3 différences essentielles avec le texte lu sur les cartes :

- Le texte source est dit compressé car les caractères blancs inutiles ont été supprimés. En effet, on sait que l'écriture des ordres CIVA a un format entièrement libre, la seule contrainte étant qu'un identificateur ne doit pas être contenu sur 2 cartes consécutives. Nous considérons que le séparateur d'instructions CIVA est le point virgule et que le séparateur de mots est le caractère blanc.

- Afin de permettre à l'utilisateur de pouvoir modifier le texte source de ses unités (cf : § 2.5.3.1.), nous avons ajouté au texte un compteur de cartes. Ce compteur est matérialisé par les caractères :

= = numéro d'ordre

Le numéro d'ordre est rangé sur 2 octets, ce qui permet à l'utilisateur d'avoir des unités dont le texte source a une taille considérable (65 533 cartes au plus).

- Enfin, afin de permettre à l'utilisateur de changer le nom d'une de ses unités (cf la procédure REMPLACER en 2.5.3.I.3), il est nécessaire que la taille réservée pour ce nom soit fixe et égale à 10 caractères. En effet le système de gestion de fichiers de SIRIS 7 ne permet pas les mises à jour d'enregistrements qui modifient leur longueur, pour les fichiers partitionnés. Donc lorsque le module de compression de texte rencontre le nom de l'unité, il réserve automatiquement une place d'une longueur égale à 10 octets, même si l'identificateur a une taille inférieure.

4.I.6. - Le fichier des textes compilés

Avant de lancer l'exécution d'une unité de traitement, le module initial doit vérifier que le module directeur est à l'état édité : tous les modules de l'unité de traitement ont du être compilés et édités ensemble. Si ce n'est pas le cas, le module initial doit lancer les compilations des unités qui sont à l'état source. Le compilateur prend alors pour donnée le texte source de chacune des unités à compiler et fournit pour résultat un module objet pour chaque unité compilée. Ces textes objets sont alors rangés dans un autre fichier partitionné déclaré dans la classe d'application : le fichier des textes compilés. Chaque partition contient le texte objet d'une unité et son nom est le numéro de matricule de cette unité. Le texte objet se présente sous la forme d'une chaîne de caractères. C'est le résultat du compilateur CIVA et la chaîne a la forme standard décrite dans ("Normes de programmation SIRIS 7" [46]).

4.I.7. - Le fichier des textes édités

Enfin, une fois que toutes les unités formant l'unité de traitement ont été compilées, le "module initial" demande au relieur de faire la préédition et l'édition de liens, c'est-à-dire la reliure. Le relieur admet pour donnée l'ensemble des textes objets et fournit comme résultat un module de chargement exécutable.

Ce module de chargement est également conservé par le système CIVA en vue d'exécutions ultérieures de l'unité de traitement. Tous les modules de chargement des différentes unités de traitement composant l'application sont conservés dans un fichier propre à chaque application : le fichier des textes édités.
Ce fichier a une organisation partitionnée. Chaque partition contient le module de chargement correspondant à une unité de traitement, le nom de cette partition étant le numéro de matricule du module directeur de l'unité de traitement. Le texte édité d'une unité de traitement se présente sous la forme d'une chaîne de caractères dont le format standard est décrit dans ("Normes de programmation SIRIS 7" [46]).

4.I.8. - Le fichier catalogue

Le fichier catalogue permet au système CIVA de gérer les fichiers logiques d'une application. Dans les systèmes d'exploitation classiques, avant chaque utilisation particulière d'un fichier, l'utilisateur doit établir le lien entre les noms des fichiers utilisés dans son programme (sous SIRIS 7, les noms des DCB) et le fichier réel situé sur un support par l'intermédiaire d'une carte de liaison (carte ! ASSIGN de SIRIS 7). Le nom du fichier réel n'est en général pas catalogué pour les fichiers sur supports externes (sous SIRIS 7, les fichiers sur bandes magnétiques ne sont pas catalogués, les fichiers sur disque pouvant l'être).

Dans le système CIVA, tous les fichiers peuvent être catalogués (cf : § 3.4.2.) quel que soit leur support. De plus, nous avons vu en 3.2.2 que l'utilisateur avait la possibilité de déclarer que la liaison qu'il désirait établir entre un fichier logique et son support était permanente, c'est-à-dire conservée par le système CIVA d'une exploitation à l'autre.
Le fichier catalogue est une "table" des fichiers logiques déclarés par l'utilisateur dans les programmes de son application . Ce fichier regroupe les renseignements utiles au système de gestion des fichiers de l'utilisateur :

- le nom du fichier logique,
- les modes d'utilisation du fichier logique, modes

déclarés par l'utilisateur dans ses programmes (cf : la procédure UTILISATION en 3.3.2.2.).
De plus, si le fichier logique a été l'objet d'un appel à la procédure LIAISON (cf en 3.2.2.), le module initial ajoutera les renseignements suivants :

- le nom du fichier physique en liaison avec le fichier logique,
- le type de liaison établie (temporaire ou permanente),
- les modes d'utilisation du fichier physique ainsi relié.

Ce fichier est donc exploité principalement par le compilateur CIVA (qui range les renseignements contenus dans l'appel de la procédure UTILISATION), par l'interprète des modules de commande (qui range les renseignements contenus dans l'appel de la procédure LIAISON) et enfin par le module de lancement (qui vérifie que tous les fichiers logiques de l'unité de traitement sont bien mis en relation avec un support avant l'exécution).

4.I.9. - La table des unités externes

Nous avons vu au chapitre I (§1-3) que, dans son application, l'utilisateur pouvait employer des unités d'autres applications, grâce aux relations externes. Ces unités ne sont pas ajoutées à l'application de façon définitive mais elles ne sont incluses que pour une demande d'exécution d'une unité de traitement.

Ceci est réalisé par le "module initial" qui dispose, par application, d'une table contenant, pour chaque unité de traitement, la liste des unités externes à inclure au moment de la demande d'exécution. Chaque élément de cette table matérialise une relation entre une unité de l'application et une unité d'une autre application.
D'un point de vue pratique, l'image de cette table est conservée d'une exploitation à l'autre dans une partition d'un fichier partitionné unique pour toutes les applications. Le nom de la partition contenant une image de table des unités externes est le nom de l'application concernée.
L'image de cette table est rangée en mémoire (dans la zone dynamique locale au "module initial") au moment de l'initialisation de la bibliothèque représentant la classe d'application.

4.I.IO.- Le fichier des sauvegardes

Avant de pouvoir rendre opérationnelle une chaîne de traitement d'une application, l'utilisateur a dû mettre au point les différentes unités de son application, regrouper ces unités pour former une chaîne et enfin tester le comportement global de la chaîne de traitement. Au cours de ces différentes phases, l'utilisateur est amené à utiliser les services de l'aide à la mise au point du projet CIVA. L'ensemble de ces services sont décrits dans (DUCLOY [9]).

Parmi tous les services décrits, on peut rappeler les métamodules :

$ sauvegarder

et $ restaurer

qui permettent à l'utilisateur de programmer un ou plusieurs points de reprise dans la chaîne de traitement, c'est-à-dire plus particulièrement de sauvegarder un certain nombre d'informations et de les récupérer ultérieurement.

Les appels de ces métamodules s'écrivent :

$ $ sauvegarder (<nom> , <liste de sortie>) ;

$ $ restaurer (<nom> , <liste d'entrée>) ;

<nom> est une variable ou une constante de type chaîne de caractères.
La liste de sortie (ou liste d'entrée) indique quelles sont les informations qui doivent être sauvegarder (ou restaurer).
Les informations désignées dans la liste vont alors constituer des enregistrements d'un fichier séquentiel indexé. Ce fichier est transparent pour l'utilisateur et en particulier tous les blocages d'enregistrements sont réalisés par des modules de service appelés par les métamodules.
Le fichier des sauvegardes est unique pour une application et il est permanent : il est donc déclaré dans la classe d'application.

4.I.II - Les matrices des graphes des relations

Dans le projet CIVA, nous avons défini entre les unités d'une application 3 types de relations : la relation "utilise", la relation "appelle" et la relation "$ appelle" (cf § I.I).
A tout instant, l'interprète du module de commande est obligé de connaître quelles sont les unités qui sont liées entre elles par ces relations. Par exemple, si l'utilisateur décide de supprimer une unité, nous avons vu (cf § 2.5.4.I.) que l'interprète pouvait lui communiquer la liste des unités qui sont en relation avec l'unité supprimée. Il est donc nécessaire de représenter les relations citées.

4.I.II.I. Définitions des graphes G1 et G2

Dans une application, le "module initial" connaît toutes les unités qui y sont déclarées.

- Soit E l'ensemble des unités déclarées dans la "classe d'application"

  $E = C \cup M \cup MM$

  avec, définis au niveau de l'application :

  C = ensemble des classes, y compris la (ou les) classe de commande

  M = ensemble des modules

  MM = ensemble des métamodules.

Remarque

Un métamodule peut être déclaré dans une classe autre que celle de l'application. Ici, nous ne nous intéresserons qu'aux métamodules déclarés dans la classe d'application, qui sont donc utilisables par toutes les unités d'application. Les autres métamodules peuvent ne pas être pris en compte par le "module initial" comme nous le verrons ultérieurement en 5.7.2.

- soit R la relation définie par :

  $R = U \cup A \cup \$A$

  où U désigne la relation "utilise"

  A désigne la relation "appelle"

  $A désigne la relation "$ appelle"

- soit $\hat{R}$ la fermeture transitive de la relation R.

- nous pouvons alors définir deux graphes :

  + le graphe G1 = (E, R) qui admet pour points les unités d'une application et pour arcs les liens créés par la relation R.
  + le graphe G2 = (E, $\hat{R}$) qui est le graphe de la fermeture transitive de la relation R.

Remarques

1) D'après les définitions des relations (cf : DERNIAME [8], DENDIEN [7]), nous n'acceptons pas de circuit, même réduit à une boucle, dans les graphes G1 et G2. Tout circuit sera détecté et considéré comme une erreur.

2) Les relations R et $\hat{R}$ sont des relations d'ordre partiel. Comme il n'y a pas de circuit, on peut déduire de $\hat{R}$ un ordre total en définissant un ordre arbitraire entre les éléments du graphe qui ne sont pas ordonnés. Le cheminement dans ces graphes sera donc défini.

4.I.II.2. Représentation des graphes en mémoire

Les graphes G1 et G2 sont représentés respectivement par les matrices de bits M1 et M2.
Les indices de ces matrices sont les numéros de matricule des unités de l'application.

Nous avons :

M1 (i, j) = 1 si l'unité de matricule i est en relation avec l'unité de matricule j par la relation R.
= 0 sinon.

M2 (i, j) = 1 si l'unité de matricule i est en relation avec l'unité de matricule j par la relation $\hat{R}$.
= 0 sinon.

On peut constater que :

- les bits positionnés à 1 sur une ligne i de la matrice M1 (ou M2) représente l'ensemble des successeurs de l'unité de matricule i dans le graphe G1 (ou G2).

- les bits positionnés à 1 sur une colonne j de la matrice M1 (ou M2) représente l'ensemble des prédecesseurs de l'unité de matricule j dans le graphe G1 (ou G2).

4.I.II.3. Les traitements réalisés sur les matrices

1) La matrice M1 est créée au début de l'interprétation du module de commande, à partir du fichier descriptif.

```
module création de M1 ;
I entier ;
type A file bit ;
M1 file type A ;
utilisation (DESC, lecture) ;
M1 (*, *) = 0 ;
si INDICATIF (APPLICATION COURANTE, 1) alors
CO cet indicatif a la valeur vrai si le fichier descriptif de
l'application courante est non vide CO ;
pour chaque DESCRIPTIF de DESC faire
        si iAj ou iUj ou i $A j alors M1 (i, j) = 1 ;
        fpc ;
        fsi ;
        finmod ;
```

2) La matrice M2 n'est pas présente en mémoire de façon permanente, pendant l'interprétation du module de commande. En effet, cettesolution évite des traitements assez lourds pour assurer la maintenance de cette matrice. Seule donc la matrice M1 est présente continuellement en mémoire et est modifiée au cours de l'interprétation.

Le "module initial" ne créé la matrice M2 que lorsqu'il veut connaître l'environnement d'une unité (c'est par exemple le cas dans l'opération de suppression d'une unité : le système communique à l'utilisateur les noms des unités appartenant à l'environnement de l'unité v supprimée c'est-à-dire à $\hat{R}$ (v) et à $\hat{R}^{-1}$(v))

Suite aux remarques que nous avons faites en 4.I.II.I, il n'est pas nécessaire de créer toute la matrice M2 en mémoire mais seulement une ligne et une colonne de cette matrice pour connaître l'environnement d'une unité. Cette ligne et cette colonne sont créées à partir de la matrice M1, et implantées en zone dynamique locale de la mémoire.

- construction d'une ligne X de la matrice M2

Etant donné un point X fixé, on désire déterminer l'ensemble :

$\hat{R}$ (X)

Cette détermination est obtenue de façon récursive par les constructions successives de :

R (X)
R^2(X)
R^3(X)
"
"
R^{n-1}(X)

$R(X) = R(X) \cup R^2(X) \ldots\ldots \cup R^{n-1}(X)$ car le graphe G1 est sans circuit.

L'algorithme est alors le suivant :

```
M2 (X,*) = M1 (X,*) ;
pour I de 1 à N - 1 faire
    pour Z de 1 à N faire
        pour Y de 1 à N faire
            si (M2 (X,Z) = 1 et M1 (Z,Y) = 1) alors
                        M2 (X,Y) = 1 ; fsi ;
        fp ;
    fp ;
fp ;
```

En remarquant que les lignes des matrices M1 et M2 sont représentées en mémoire par des suites de bits on peut diminuer le nombre d'opérations en obtenant une variante de l'algorithme précédent :

```
proc construction d'une ligne de M2 (X) ;
X entier ;
M2 (X,*) = M1 (X,*) ;
pour I de 1 à N-1 faire
      pour Z de 1 à N faire
           si M2 (X, Z) = 1 alors
           M2 (X,*) = M2 (X,*) ou M1 (Z,*) ; fsi ;
      fp ;
fp ;
finproc ;
```

Cet algorithme demande : N (N-1) tests
et N (N-1) opérations OU sur des suites débits.

- construction d'une colonne Y de la matrice M2

Le problème est analogue au précédent : on désire maintenant déterminer l'ensemble $\widehat{R}^{-1}(Y)$:

$$\widehat{R}^{-1}(Y) = R^{-1}(Y) \quad R^{-2}(Y) \ldots\ R^{-n+1}(Y)$$

L'alogorithme est alors le suivant :

```
proc construction d'une colonne de M2 (Y) :
Y entier ;
M2 (*,Y) = M1 (*,Y) ;
pour I de 1 à N - 1 faire
     pour Z de 1 à N faire
          si M2 (Z, Y) = 1 alors
          M2 (*,Y) = M2 (*,Y) ou M2 (*,Z) ; fsi ;
     fp ;
fp ;
finproc ;
```

3) Les autres traitement réalisés sont :

- l'adjonction d'un arc dans le graphe G1.
- la suppression d'un arc dans le graphe G1,
- l'adjonction d'une unité dans le graphe G1, connaissant le nom de l'unité et la liste de ses successeurs,

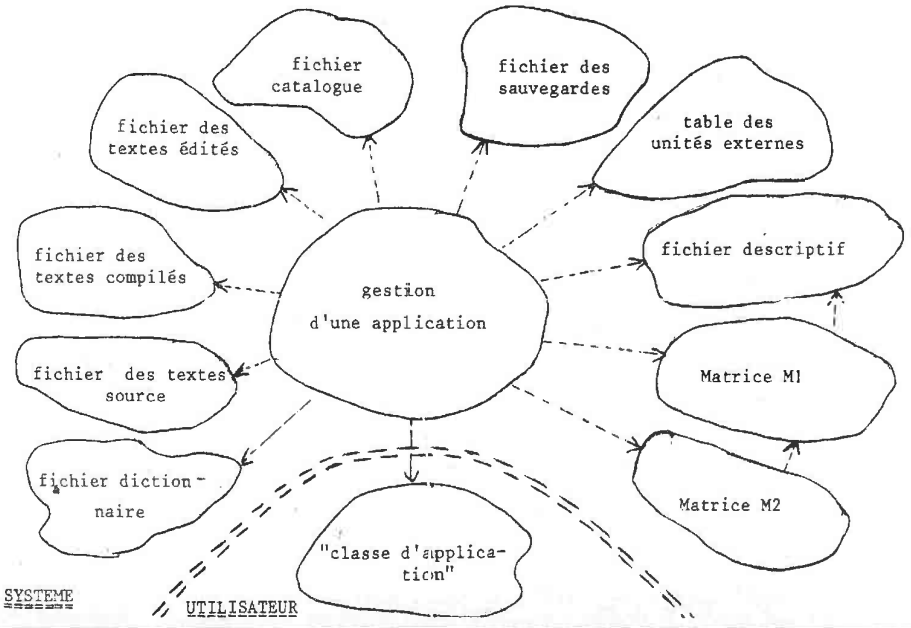
- la suppression d'une unité dans le graphe G1 (cette opération entraîne la mise à 0 de la ligne et de la colonne correspondant à l'unité dans la matrice M1)
- la modification des successeurs d'une unité dans G1
- la recherche des successeurs d'une unité de matricule donné dans le graphe G2
- la recherche des prédesseurs d'une unité de matricule donné dans le graphe G2

Les algorithmes de ces traitements étant triviaux, nous ne les avons pas indiqués ici.

4.I.I2. - Shémas généraux du système CIVA

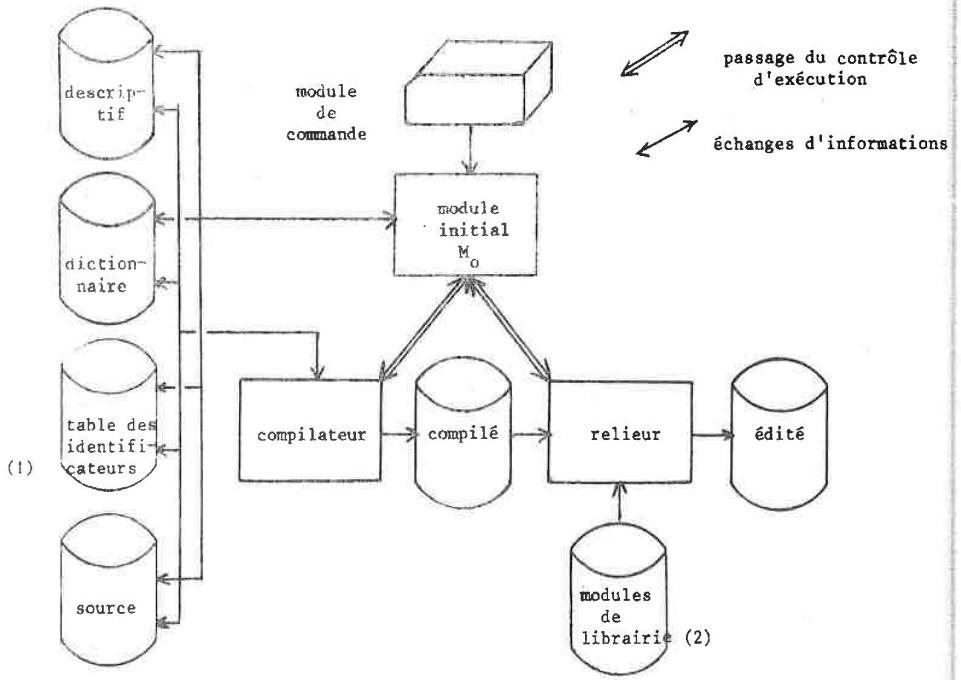
Après avoir décrit les différents éléments de la "classe d'application", éléments qui permettent au "module initial" de gérer les unités de l'utilisateur, nous allons montrer à l'aide de shémas comment s'articulent ces éléments entre eux.

4.I.I2.I. Représentation des classes de gestion d'une application



Seuls les objets définis dans la "classe d'application" sont accessibles à l'utilisateur (c'est lui qui les crée, les détruit...). Les autres objets ne sont accessibles qu'au système CIVA et sont utilisés par lui pour gérer l'application courante.

4.I.I2.2. Organisation générale du système CIVA

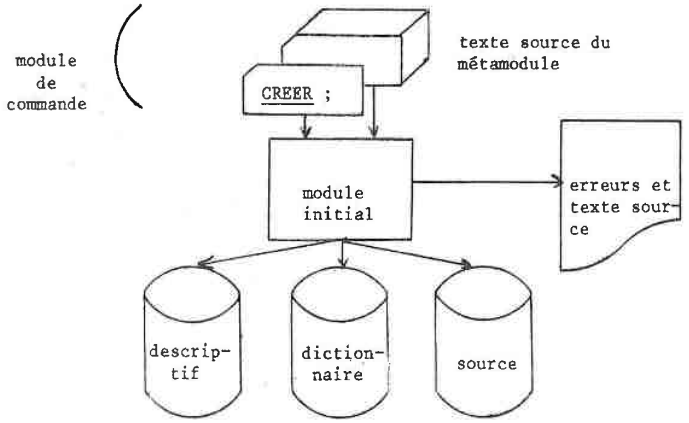


(1) Ce fichier n'est pas décrit ici. Sa description complète est donné dans (MANSUY).

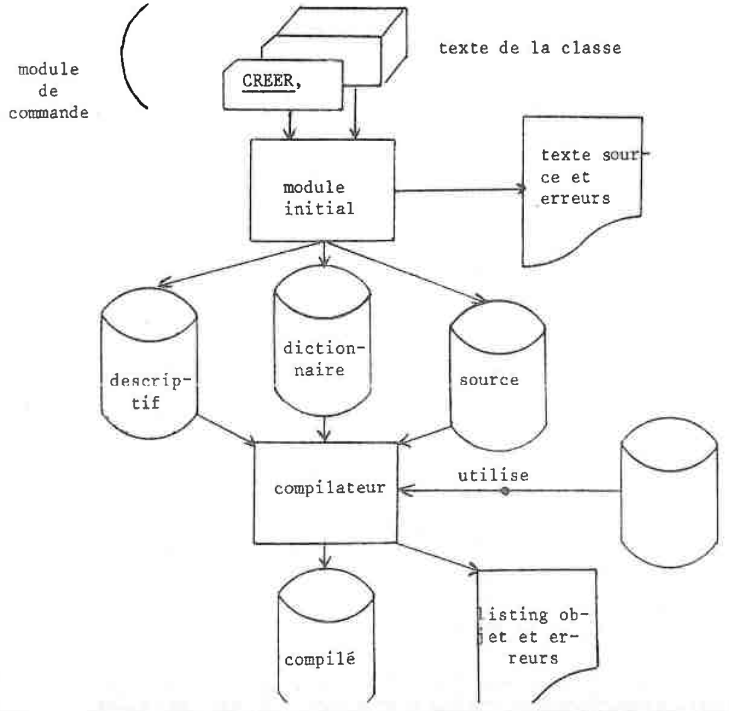
(2) Le fichier contenant les modules de librairie est décrit dans (FIEGEL).

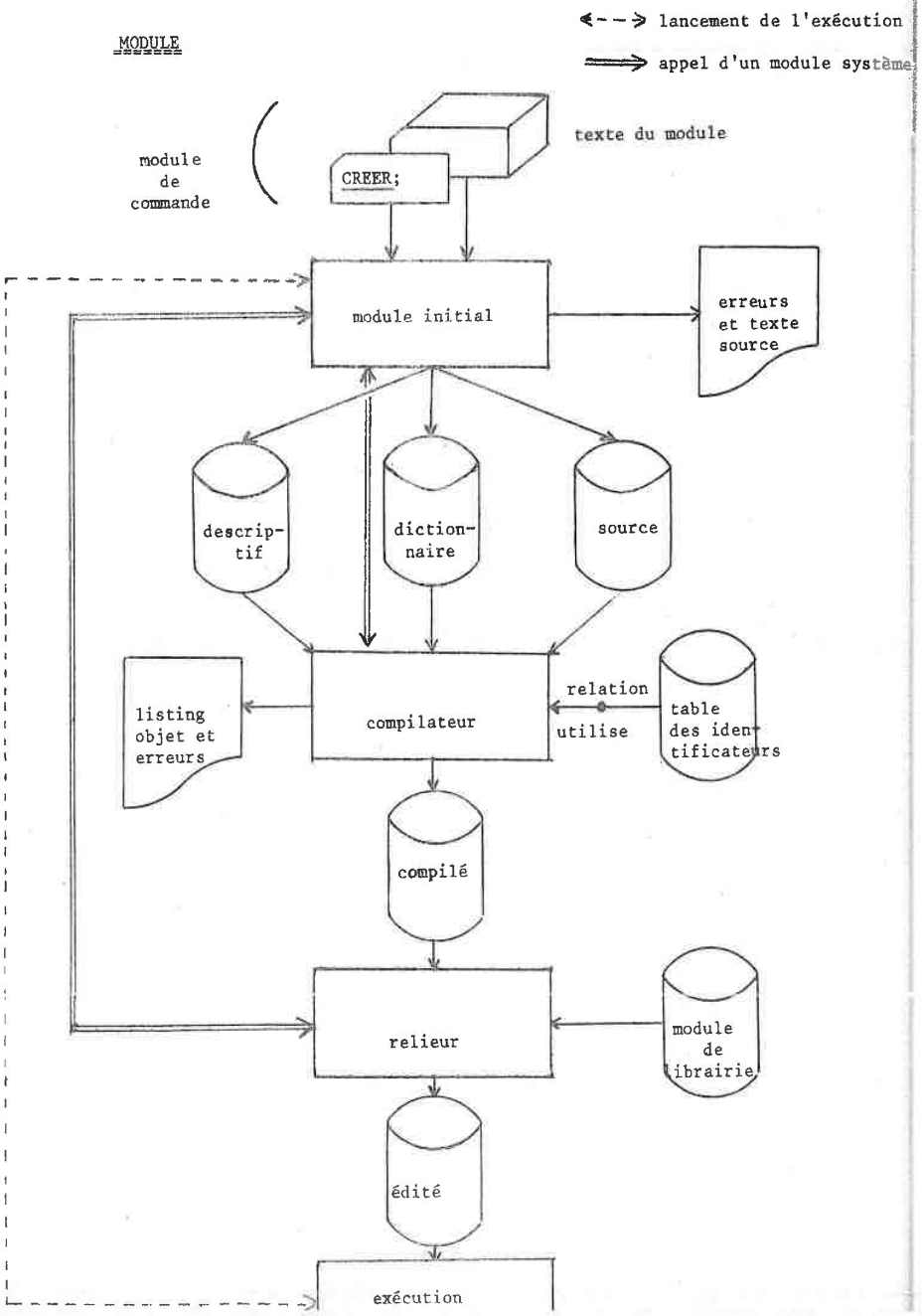
4.I.I2.3. Schéma d'l'entrée et du traitement d'une unité par le système CIVA

METAMODULE



CLASSE





4.2. LA REPRESENTATION DE LA CLASSE SYSTEME

L'environnement du "module initial" est composé d'une deuxième catagorie d'informations, celles déclarées dans la "classe système".
Cette classe contient tout d'abord des variables système quel'utilisateur peut modifier dans son module de commande.
De plus cette classe contient les descriptions des informations qui sont utilisées par tout le système CIVA. Ces informations ne sont pas accessibles à l'utilisateur.
Parmi ces informations, nous décrirons plus particulièrement :

- la table des applications,
- la table des identificateurs.

4.2.I. - La table des applications

A différents instants de son exécution, l'interprète du module de commande a besoin de savoir si une certaine application existe dans le système et il a besoin de renseignements sur les différentes applications.
Ces renseignements sont groupés dans une table gérée par le module initial : la table des applications.

Pour chaque application existante dans le système, l'interprète doit connaître les renseignements suivants :

- le nom de l'application;
- un mot de passe éventuel permettant d'assurer la protection de l'application ;
- l'état de chacun des fichiers système, utilisés pour la gestion de l'application (l'état permet d'indiquer si le fichier a été créé ou non). En effet, il se peut très bien que dans une application en cours de création, le fichier des textes compilés, par exemple, n'a pas encore été créé. Le traitement de ce fichier ne sera alors pas le même que si ce fichier existe déjà. A chacun de ces fichiers est donc associé un indicatif qui permet de connaître l'état du fichier.
- le nombre d'unités créées dans l'application ;
- le plus grand numéro de matricule accordé ;
- enfin, nous avons vu que le propriétaire d'une application particulière pouvait protéger ses unités contre des utilisations éventuelles demandées par des utilisateurs d'autres applications (voir la procédure CREER).

Afin de rendre plus rapide le système de protection, l'interprète peut connaître le nombre d'unités qui sont protégées dans l'application contre tout accès de l'extérieur.
En effet, si un utilisateur désire employer une unité d'une autre application, l'interprète vérifie tout de suite qu'il existe des unités non protégées dans cette application (c'est-à-dire que le nombre d'unités protégées contre tout accès est différent du nombre d'unités créées). Si ce n'est pas le cas, il émet un message d'erreur à l'utilisateur, sans avoir besoin de rechercher l'unité demandée dans l'application et de faire des contrôles.

4.2.2. - La table des identificateurs

Dans un module de commande, le "module initial" peut rencontrer des identificateurs qui désignent des objets élémentaires ou structurés. Ces objets ont été déclarés soit dans la classe système, soit dans la classe de modification, soit dans la classe d'application, soit dans la classe de commande ou bien encore dans le module de commande.
Avant de pouvoir faire des calculs sur les valeurs affectées à ces objets, il faut que le"module initial" puisse retrouver les emplacements de ces objets en mémoire centrale. C'est le but de la table des identificateurs.

4.2.2.I. Organisation de la table des identificateurs

Lorsque le module initial rencontre un identificateur d'objet élémentaire, il doit pouvoir accéder à la valeur contenue dans l'objet. Le rôle de la table des identificateurs est de faciliter sa recherche.

Cette table est organisée suivant le principe du hashing : on dispose de I6 classes d'équivalence représentées chacune par un élément qui pointe vers une partie secondaire. Chaque identificateur est représenté, en partie secondaire, par un élément composé de trois parties :

- la première partie contient soit un pointeur vers l'identificateur suivant de la classe d'équivalence, soit 0 si l'identificateur est le dernier de la classe.

- la deuxième partie contient le descripteur de l'objet désigné par l'identificateur. Le rôle de ce descripteur a déjà été défini dans (PAYAFAR [I7]). Nous n'avons fait que reprendre sa structure pour l'adapter à nos besoins.

- enfin, la dernière partie contient la chaîne de caractères représentant l'identificateur.
(La description complète de cette table est donnée en annexe).

4.2.2.2. L'implantation des objets en mémoire

Le module initial doit manipuler les objets élémentaires auxquels peut accéder le module de commande. Ces objets sont implantés dans la zone dynamique de la mémoire pour les raisons indiquées ci-après.

- Les objets déclarés dans le module de commande sont locaux à celui-ci. Ils ne peuvent pas être utilisés dans les autres modules du système CIVA ou dans les unités de traitement de l'utilisateur. De plus à priori, le module initial ne connaît pas la taille et le nombre de ces objets. Ils seront donc implantés dans la zone dynamique locale de la mémoire. Cette zone sur le CII 10070 a pour caractéristique essentielle d'être sauvegardée lors de la demande d'exécution d'un programme (cf la procédure M:LINK définie dans [44]). Les objets qu'elle contient n'existent donc que pendant l'interprétation du module de commande.
A la rencontre de la déclaration d'un objet dans le module de commande, le module initial demande au système SIRIS 7 de lui allouer de la place en zone dynamique locale (cf la procédure M:GP définie dans [44]). Celui-ci retourne au module initial l'adresse du premier mot disponible ; cette adresse est alors rangée dans le descripteur de la table des identificateurs (zone "pointeur"

- Les objets déclarés dans la classe système ou dans la classe de modification sont en nombre connu du système et leur taille est parfaitement déterminée. Mais ces objets peuvent être utilisés par les autres modules du système et par les unités de de traitement de l'utilisateur (seulement pour la classe système).

C'est pourquoi ces objets seront également implantés en zone dynamique et pour qu'ils soient partageables, en zone dynamique commune.
A l'initialisation du système, le module initial demande de la place au système SIRIS 7 pour ranger tous les objets déclarés dans ces deux classes : l'adresse du premier mot réservé à la classe système est rangé dans DEBSYS, l'adresse du premier mot de la classe de modification est rangé dans DEBMOD. Grâce à la zone "localisation" du descripteur il est facile de retrouver la valeur affectée à un tel objet (il suffit d'ajouter le déplacement à l'adresse de la classe correspondante).

- Les objets déclarés dans la classe d'application sont utilisables par tous les modules,qu'ils appartiennent au système ou à l'utilisateur : ils seront donc implantés en zone dynamique commune. Toutefois, la gestion de ces objets est différente pour la raison suivante : comme tous les objets déclarés dans le module de commande, le module initial n'en connaît pas, a priori, le nombre et la taille, l'utilisateur pouvant à tout moment créer des objets dans cette classe. Il serait donc nécessaire d'avoir la même gestion que celle des objets locaux au module de commande. Mais, il faut penser que ces objets de la classe d'application peuvent également être utilisés dans les unités de traitement. Il faut donc que le compilateur dispose d'un système de gestion de la classe d'application identique à celui des classes système et modification. C'est pour cette raison qu'il y a deux types de descripteurs d'objet de la classe d'application.
A l'initialisation du système, tous les objets de cette classe sont implantés les uns à la suite des autres dans une zone contigüe de la mémoire. Le module d'initialisation demande la place nécessaire en zone dynamique commune. L'adresse du premier mot affecté à la classe d'application est rangé dans le paramètre DEBAPPL. Tous les objets sont représentés par un descripteur de type 2 qui indique le déplacement de l'objet par rapport au début de la zone.

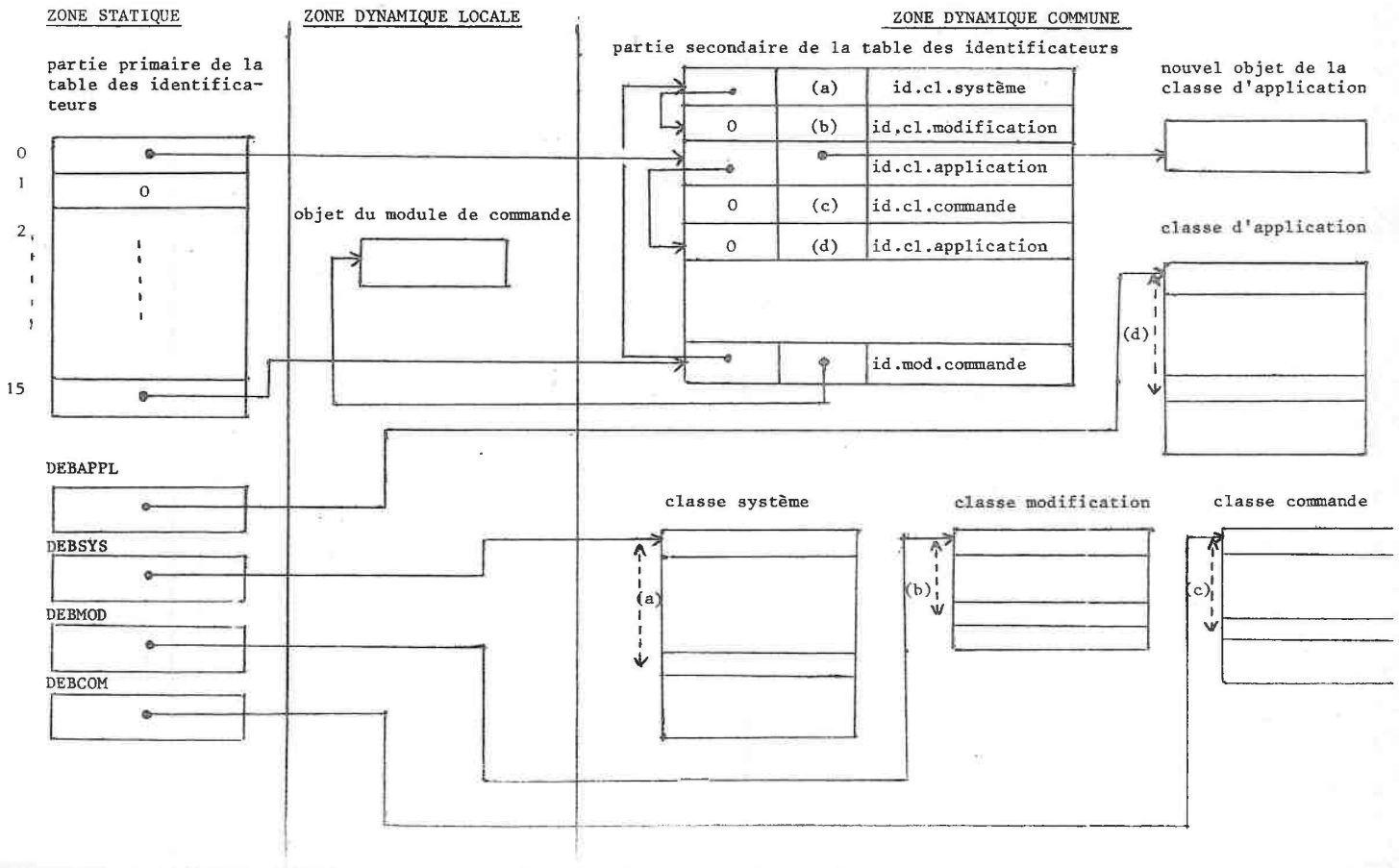
Lorsque l'interprète du module de commande rencontre la déclaration d'un objet dans la classe d'application par l'appel de la procédure CREER (cf § 2.4.2.2.) il crée un élément dans la table des identificateurs, associe à l'identificateur créé un descripteur de type 1. Il demande alors au système d'exploitation de la place pour implanter l'objet en zone dynamique qui lui retourne l'adresse du premier mot disponible dans cette zone. Il la range alors dans la zone "pointeur" du descripteur de la table des identificateurs.
Lorsque le module initial lance l'exécution du compilateur, ou d'une unité de traitement, ou bien lorsqu'il termine l'interprétation du module de commande, il réorganise alors la classe d'application en ajoutant les nouveaux objets qui viennent d'être déclarés à la suite de ceux qui figuraient déjà dans la classe. Tous les descripteurs de type 1 sont alors remplacés par des descripteurs de type 2.
Lorsque, enfin, l'utilisateur décide de supprimer un objet de cette classe, aucun tassage de la zone correspondante n'est effectuée afin d'éviter au système de recompiler toutes les unités de l'application.

- Les objets déclarés dans la classe de commande sont connus en nombre et en taille par le système, une fois cette classe compilée. Ces objets peuvent être utilisés soit par le module de commande soit par une unité de traitement qui utilise cette classe. L'implantation et la gestion de ces objets seront donc identiques à celle des objets de la classe système et de la classe de modification, c'est-à-dire en zone dynamique commune. L'adresse du premier mot réservé à cette classe est rangé dans DEBCOM.

5

FONCTIONNEMENT GENERAL DE L'INTERPRETE DES MODULES DE COMMANDE

SCHEMA DE LA GESTION DES IDENTIFICATEURS



Le "module initial" se compose de trois grandes parties que nous allons analyser dans ce chapitre :

- l'initialisation du système (cf§5.1),
- l'interprétation du module de commande (cf§5.2),
- les opérations de terminaison (cf§5.3).

Le "module initial" est déclaré dans l'application système. Cette dernière regroupe tous les programmes du système de programmation CIVA. Nous présenterons donc dans ce chapitre les liaisons de l'interprète des modules de commande avec ces autres programmes (cf§5.4).

De plus le principal rôle du "module initial" est d'interpréter les modules de commande. Parmi toutes les instructions qu'il peut rencontrer dans ce dernier, une est particulièrement intéressante à analyser : c'est la demande d'exécution d'une unité de traitement. Nous étudierons donc dans ce chapitre les liaisons du module initial avec les unités de traitements (cf : § (5, 5), la gestion des transitions d'état (cf : § 5.6) ainsi que le lancement des compilations (cf : § 5.7).

L'INITIALISATION DU SYSTEME

Lorsqu'un module de commande lui est soumis, le module initial doit faire les opérations d'initialisation du système. Ces opérations sont réalisées en deux temps : l'initialisation de la classe système et de la classe de modification, puis l'initialisation de la bibliothèque représentant la classe de l'application.

```
module initialisation du système ;
initialisation des classes système et modification ;
initialisation de la bibliothèque ;
finmod ;
```

5.I.I. - L'initialisation des classes système et modification

Nous avons vu que, dans la classe système, étaient déclarés des paramètres d'exploitation (voir la demande d'éxécution d'une unité de traitement en 2.6.2). Certains de ces paramètres doivent être initialisés à des valeurs implicites par le module d'initialisation du système. Dans cette classe, sont également définies les variables contrôlables qui permettent de réaliser le contrôle de l'exécution des unités de traitement. Ces variables doivent également initialisées au début de l'exécution du module initial (voir le contrôle de l'exécution en 2.6.4.). Nous venons de voir que cette classe contenait aussi les déclarations de la table des applications et de la table des identificateurs. Entre deux exploitations, les représentations de ces tables sont conservées par le système CIVA sur le disque système, dans le fichier des sauvegardes de l'application système. Enfin, la classe système contient les déclarations des autres modules du système CIVA : le système de compilation (codifieur et générateur) et le système de reliure (prééditeur et éditeur de liens). Le module initial va être amené à lancer l'exécution de ces modules : il faut donc que les paramètres d'exploitation du système servent de communication entre ces différents modules soient initialisés.

La classe de modification contient, elle, les déclaration des procédures de modification qui ont été définies au chapitre 2. Certains paramètres de ces procédures doivent également être initialisés au début de l'interprétation du module de commande.

Nous avons dit (§ 4.2.2.2.) que des objets élémentaires déclarés dans la classe système et dans la classe de modification étaient implantés en zone dynamique commune. Le module d'initialisation de ces classes doit donc demander au système d'exploitation de lui réserver de la place dans cette zone pour pouvoir implanter ces objets. En retour, le système d'exploitation renvoit au module d'initialisation l'adresse du premier emplacement réservé à la classe de modification.

5.I.2. - L'initialisation de la classe application

La première ligne d'un module de commande contient un appel de la procédure APPLICATION (cf : § 2.4.I.). Cette procédure permet d'indiquer au système dans quelle application le module de commande va travailler. Une fois cet appel réalisé, le module de commande peut utiliser la classe d'application. L'appel de la procédure APPLICATION a pour effet de donner une valeur à la variable de la classe système définie par :

```
APPLICATION COURANTE file (max = 10) car ;
```

L'interprète, à la rencontre de l'appel de la procédure, doit rechercher dans la table des applications le nom de l'application. Ensuite, il contrôle la syntaxe de l'appel et réalise les opérations d'initialisation de la bibliothèque représentant la classe de l'application qui sont :

- déterminer les noms des fichiers système déclarés dans la classe d'application et permettre au module initial d'utiliser ces fichiers ;
- regarder si la classe d'application contient des déclarations d'objets élémentaires ou structurés et si c'est le cas, implanter ces objets en mémoire et leur affecter les valeurs qu'ils possédaient à la fin de la dernière exploitation ;
- enfin regarder si la classe d'application contient des éléments dans la table des unités externes et si c'est le cas, implanter cette table en mémoire centrale.

L'INTERPRETATION DES INSTRUCTIONS DU MODULE DE COMMANDE

Une fois les opérations d'initialisation réalisées, le module initial M_o interprète les instructions du module de commande. Ces instructions sont de deux types :

- les instructions du langage CIVA,
- les appels des procédures de la classe système et de la classe de modification.

L'interprètation des instructions du langage CIVA est réalisée de façon très classique : nous ne nous étendrons donc pas sur cette partie. Signalons simplement que l'interprétation de l'instruction SI se fait à l'aide d'une pile qui autorise donc les instructions SI imbriquées. L'interprétation de l'instruction POUR est également réalisée à l'aide d'une pile pour permettre l'utilisation de POUR imbriqués. L'interprétation des tables de décision et de sélection utilise la méthode de la création d'un masque. Il existe deux phases dans le traitement. Au cours de la première phase, l'interprète créé un vecteur booléen (appelé masque) qui regroupe les résultats des tests des différentes conditions. Dans un deuxième temps, l'interprète détermine toutes les règles (dans le cas d'une table de décision) ou la la première règle simplement (pour une table de sélection) qui correspond à là valeur du masque. L'interprète exécute alors l'interprétation des actions qui correspondent à ces règles. Si aucune règle n'a été trouvée, l'interprète exécute alors les actions correspondant à la règle "sinon".

L'interprétation des appels des procédures de la classe système et de la classe de modification est analyséeen annexes C et D.

LA FIN DE L'INTERPRETATION DU MODULE DE COMMANDE

Lorsque le module initial rencontre, au cours de son interprétation, la fin du module de commande (matérialisée par une instruction "FINMOD ;") ou bien l'instruction "SORTIR;", il doit alors réaliser les opérations de sauvegarde du système. Comme toute application, l'application système possède un fichier des sauvegardes. Le module initial va donc tirer une photographie des objets permanents et ranger celle-ci dans le fichier des sauvegardes.

- les objets déclarés dans les classes système et modification ne peuvent subir aucune modification quant à leur nom, leur taille et leur structure (pas d'adjonction ni de suppression d'identificateur).

La partie de la table des symboles qui contient les représentations de ces identificateurs n'a donc pas lieu d'être sauvegardée à la fin de chaque exploitation. Il suffit que cette partie figure une fois pour toutes dans le fichier des sauvegardes. De plus ces classes sont toujours initialisées par l'interprète du module de commande au cours de l'opération d'initialisation du système : les valeurs affectées aux objets qui sont déclarés dans ces classes n'ont donc pas à être sauvegardées à la fin de l'exploitation de l'application.

- Par contre, la classe d'application, peut, elle, subir des modifications au cours de l'interprétation du module de commande (modification de structure par l'adjonction ou la suppression de déclaration, modification des valeurs affectées aux objets déclarés dans cette classe) : le module initial va donc ranger l'image des objets élémentaires de cette classe. Il sauvegarde également la table des unités externes ainsi que le fichier dictionnaire qui ont été mis en mémoire afin d'améliorer le rendement global du système.

- Enfin, le module initial doit détruire les objets déclarés dans le module de commande : ceux-ci sont détruits automatiquement car ils sont implantés dans la zone dynamique locale au module initial. Lorsque l'exécution de ce dernier est terminée, cette zone est détruite.
De plus, si,parmi ces objets, figurent des déclarations de fichiers, alors le module initial est obligé de détruire les fichiers ainsi déclarés (cf : le système de gestion de fichiers au chapitre 3).

5.4. LIAISONS DE L'INTERPRETE DU MODULE DE COMMANDE AVEC LES AUTRES MODULES DU SYSTEME CIVA

Lorsque l'utilisateur désire lancer l'exécution d'une unité de traitement, le système CIVA doit avant tout vérifier que le module directeur est à l'état édité. Si ce n'est pas le cas, l'interprète du module de commande doit pouvoir lancer l'exécution des autres modules du système CIVA (compilateur, relieur). Pour cela, il doit charger ces programmes en mémoire, leur fournir des paramètres et récupérer le contrôle de l'unité centrale, une fois ces programmes exécutés.

Le moniteur SIRIS 7 du CII 10070 permet à un programme appelant (ici, l'interprète du module de commande) de demander l'alimentation en mémoire et l'exécution d'un programme appelé, avec préservation du programme appelant dans un fichier temporaire du disque système.

Cette opération est effectuée par l'appel de la procédure du système M:LINK. L'espace mémoire occupé par le programme appelant et l'espace mémoire qui lui est alloué dans la zone dynamique locale sont libérés et préservés sur le disque système. Le programme appelé est alors chargé en mémoire et lancé à son adresse de départ. La zone dynamique commune n'étant pas affectée par ces opérations, peut être utilisée pour transmettre des informations entre programme appelant et programme appelé. De plus, la plupart des registres ne sont pas modifiés par ces opérations et peuvent donc eux aussi être utilisés pour le passage des paramètres.

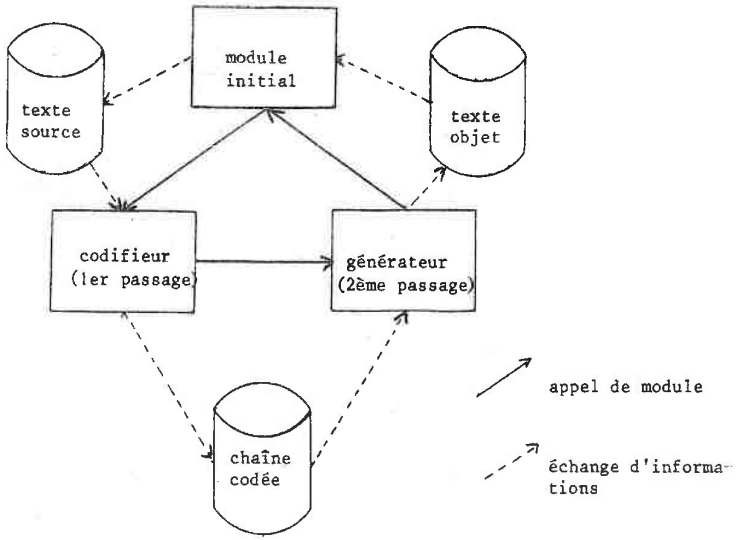
Lorsque le programme appelé a été exécuté, celui-ci rend le contrôle au programme appelant par un appel de la procédure SIRIS 7 appelée M:RETURN. Le programme appelant est alors rechargé en mémoire, et son exécution reprend à l'endroit où elle avait été arrêtée (c'est-à-dire, juste après l'appel de la procédure M:LINK). Là encore, le programme appelant peut récupérer les valeurs des paramètres soit dans les registres non affectés par l'opération, soit en zone dynamique commune. C'est donc cette opération qui sera utilisée par le module initial pour passer le contrôle soit au compilateur soit au relieur.

5.4.I. - Liaisons du module initial avec le compilateur

Si une ou plusieurs unités doivent être compilées, l'interprète du module de commande doit lancer l'exécution du compilateur CIVA. La compilation s'effectue en deux temps :

- Au cours d'un premier passage, le compilateur transforme le texte source d'une unité de nomenclature CIVA en une chaîne codée.
- Lors du 2ème passage, le compilateur CIVA reprend cette chaîne codée et la transforme en un texte objet qui contient tous les ordres machine correspondant à l'unité compilée. C'est la génération proprement dite. Ce texte objet est alors rangé dans la partition du fichier des textes compilés (cf : § 4.I6, DUCLOY [9]).

On peut schématiser ces liaisons de la façon suivante :



De plus, le compilateur CIVA est conçu de façon à pouvoir réaliser des compilations "en rafale", c'est-à-dire, qu'il peut compiler plusieurs unités en un seul appel, ces unités étant compilées les unes après les autres (cette possibilité permet un gain de temps appréciable car le compilateur garde toujours le contrôle de l'unité centrale).

5.4.I.I. Liaisons avec le codifieur

L'interprète du module de commande doit fournir au codifieur des renseignements de trois types :

- des renseignements lui permettant de trouver les données sur lesquelles le codifieur doit travailler. Ce sont donc essentiellement les noms des fichiers système de l'application en cours de traitement.
- le codifieur doit pouvoir traiter plusieurs unités en un seul appel. Le module initial lui fournira donc la liste des numéros de matricule des unités à compiler et dans l'ordre où elles doivent être compilées ainsi que le mode de compilation désiré par l'utilisateur pour chaque unité. Cette liste figurera en zone dynamique commune.

- pour pouvoir compiler les unités, le codifieur à besoin d'avoir accès aux variables de la classe système et de la classe d'application. L'interprète doit donc lui fournir également la table des symboles dans la zone dynamique commune.

5.4.I.2. Exécution du codifieur

Par leur matricule fourni par le module initial, le codifieur a accès aux textes à traiter. Il génère alors une chaîne codée dans la zone dynamique commune et met à jour l'enregistrement du fichier descriptif correspondant à cette unité (par exemple, modifier l'état, mettre à jour la liste des numéros de matricule des modules appelés .....).

Enfin le codifieur repère dans le texte source les déclarations et les utilisations de fichiers logiques, grâce à l'appel de la procédure UTILISATION. Pour chaque fichier logique déclaré, le codifieur créé un enregistrement dans le fichier catalogue . Il range le nom du fichier logique, ainsi que les différents types d'utilisation demandés par l'utilisateur. A chaque type d'utilisation, il affecte pour le fichier une étiquette logique, ce qui permettra au générateur de générer une table DCB dans le texte objet d'une part et au module initial de générer les cartes de commande d'autre part.

Lorsque le codifieur a terminé son travail pour les différentes unités indiquées, il doit passer le contrôle au générateur.

5.4.I.3 Exécution du générateur

Pour lancer le générateur, le codifieur utilise lui aussi la procédure M:LINK. Cette procédure peut en effet être utilisée à plusieurs niveaux.

Le générateur récupère la chaine codée qui se trouve en zone dynamique commune, génère le texte objet et détruit la chaine codée. Le code objet est alors rangé dans le fichier des textes compilés (cf § 4.I.6.) dans une partition ayant pour clé le matricule de l'unité.

De plus, le générateur met à jour chaque enregistrement du fichier descriptif (en particulier, la taille de la zone des constantes, taille de la zone des variables, le niveau d'erreur détecté après compilation).

5.4.I.4. Retour dans le module initial

Par un premier appel de la procédure M:RETURN, le générateur rend le contrôle au codifieur. L'environnement de celui-ci est alors restauré en mémoire. Le codifieur fait à son tour un appel de la procédure M:RETURN qui rend le contrôle à l'interprète du module de commande. L'environnement de l'interprète est restauré en mémoire. Le fichier temporaire qui avait servi à le préserver sur le disque système est détruit. L'exécution de l'interprète se poursuit.

5.4.2. - Liaisons du module initial avec le relieur

Avant de lancer l'exécution de l'unité de traitement, l'interprète doit vérifier que le module directeur, ainsi que tous les modules appelés, sont "reliés" entre eux. Si ce n'est pas le cas, l'interprète doit lancer l'exécution du relieur CIVA.

Le relieur travaille en deux temps (cf DENDIEN [7] ; FIEGEL [11]) :

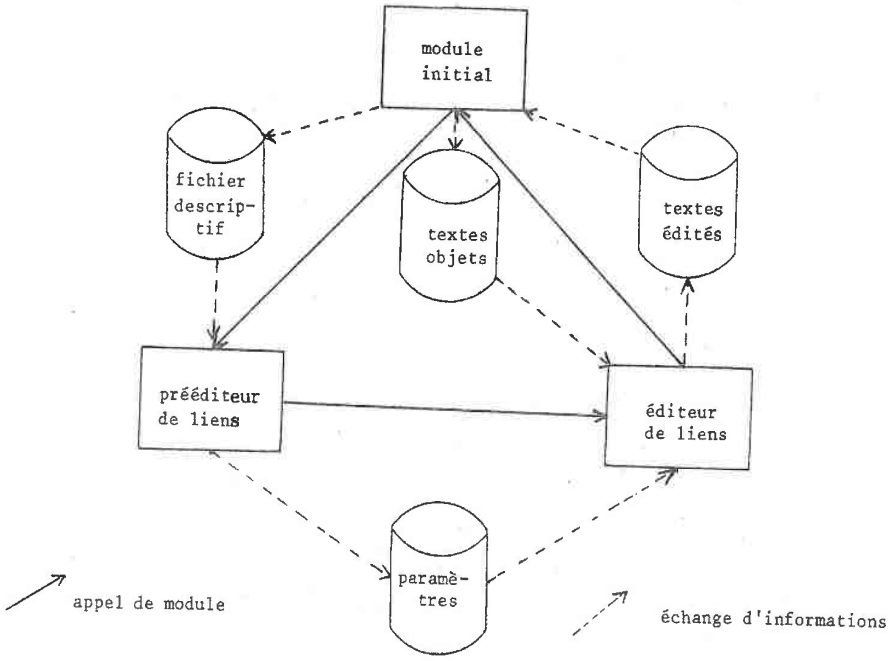
- dans un premier temps, a lieu la préédition de liens. Cette opération a pour but de déterminer quelles sont toutes les unités à relier et quel est le type de recouvrement qu'il va être nécessaire de faire, connaissant la taille de la zone mémoire disponible et la taille de chaque unité (cette dernière étant fournie par le fichier descriptif). On peut rappeler que, dans le système CIVA, trois types de recouvrement sont prévus :

  + recouvrement des zones de variables uniquement.
  + recouvrement des zones de variables et des zones de constantes des modules seuls,
  + recouvrement des zones de variables et des zones de constantes des modules et des classes.

Le choix d'un recouvrement est réalisé par le prééditeur en fonction de la taille de l'espace mémoire disponible (cette taille est connue du système grâce au paramètre TYPETRAV). Le premier type de recouvrement est rapide à l'exécution , le deuxième est déjà moins performant. Par contre, le traitement dans le cas du troisième type de recouvrement est encore plus lourd et il ne doit être entrepris q'en cas de nécessité absolue.

- Dans un deuxième temps a lieu l'édition de liens. L'éditeur de liens est l'éditeur SIRIS 7. Il prend les codes objets des différentes unités à relier et édite ces unités. Le résultat de l'édition est un module de chargement.

En général, le relieur n'aura qu'un module de chargement à créer.



5.4.2.I. Renseignements à fournir au relieur

L'interprète du module de commande fournit au relieur des renseignements de trois types :

- les noms des fichiers système de l'application en cours de traitement qui permettent au relieur de retrouver les données à traiter.
- les valeurs des paramètres d'exploitation suivants :
  + PAGE qui indique le nombre de pages de mémoire que l'utilisateur désire se voir allouer pour l'exécution de son unité de traitement (cf § 2.6.2.I).
  + XADBIAS qui indique l'adresse du premier mot disponible pour l'exécution de l'unité de traitement (ce paramètre est une constante dont la valeur est fixée au moment de la génération du système).
- enfin la liste des unités à relier entre elles. Cette liste est composée des numéros de matricule des unités formant l'unité de traitement. Elle est implantée en zone dynamique commune.

5.4.2.2. L'exécution du relieur

Le lancement de l'exécution du prééditeur de liens est réalisé par l'appel de la procédure système M:LINK. Le prééditeur dispose de tous les renseignements nécessaires dans le fichier descriptif et il détermine le type de recouvrement.
A son tour, le prééditeur passe le contrôle à l'éditeur de liens par un appel de la procédure M:LINK. L'éditeur de liens fournit, comme résultat, un module de chargement qu'il range dans la partition du fichier contenant les textes édités, cette partition ayant pour clé le numéro de matricule du module directeur. Lorsque l'éditeur a terminé son travail, il rend le contrôle au prééditeur par un appel de la procédure système M:RETURN.

Celui-ci positionne trois paramètres d'exploitation :

- XNIVERR qui contient le niveau maximum d'erreur rencontré au cours de l'édition de liens.
- XINDEDIT qui indique à l'interprète du module de commande comment s'est déroulé la reliure.

Si XINDEDIT a la valeur vrai, il y a eu erreur au cours de l'édition. Le module de chargement n'est pas créé. Si XINDEDIT à la valeur faux, l'exécution de l'unité de traitement peut avoir lieu.

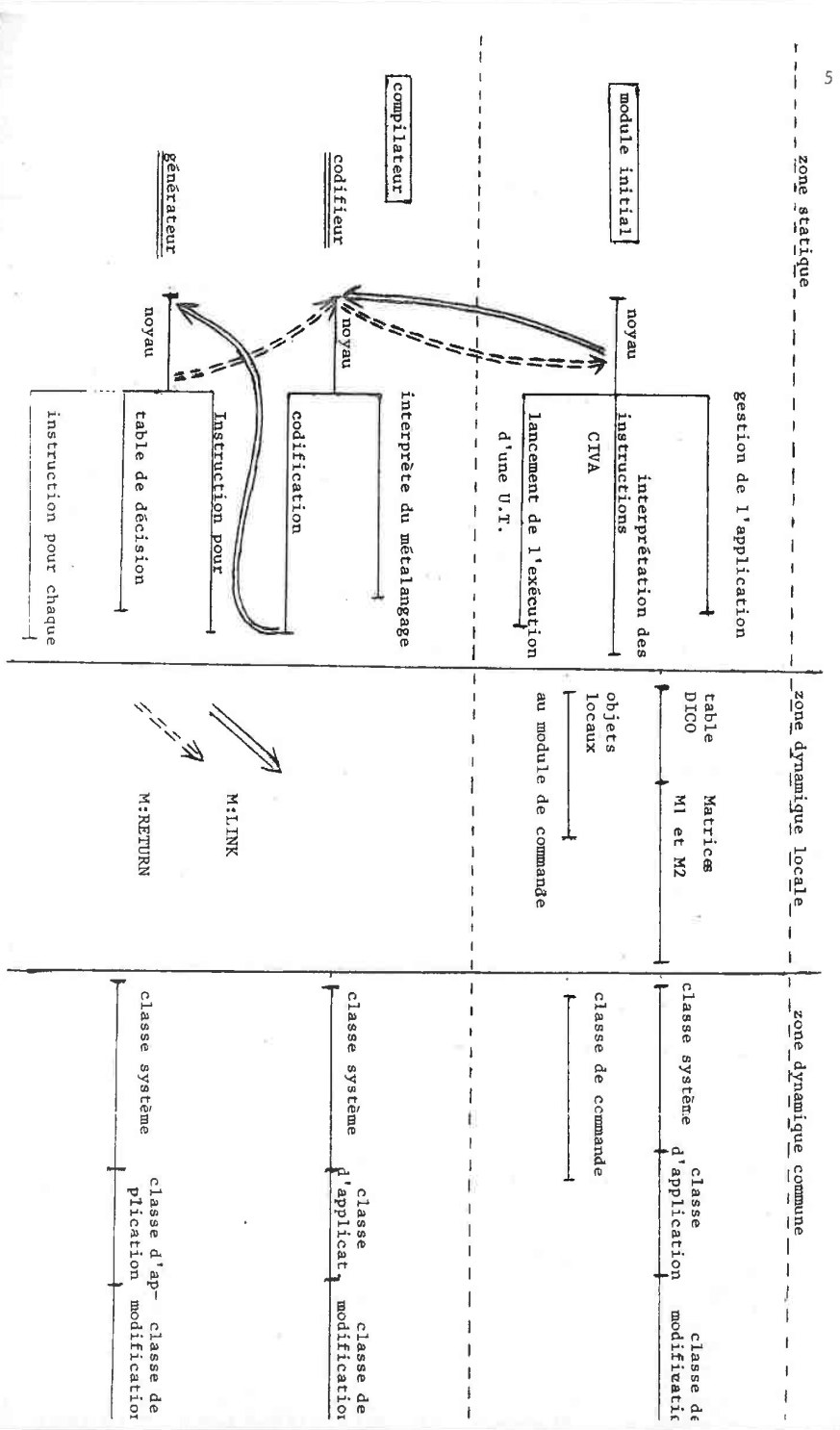
- Dans ce dernier cas, le module initial peut connaître le type de recouvrement qui a été effectué. Pour cela, le relieur positionne le paramètre XTYPREC.

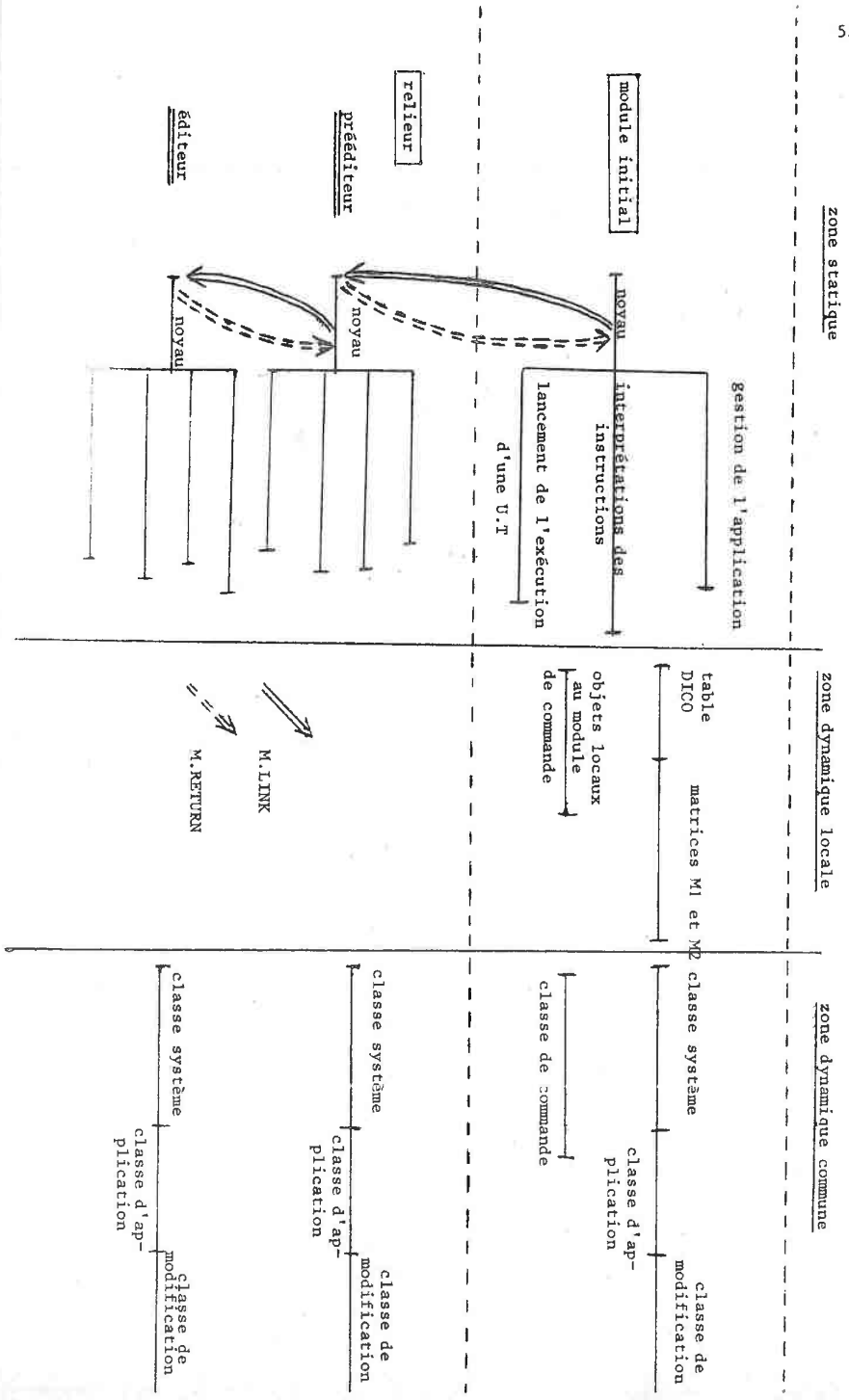
XTYPREC = 0 si la place mémoire est insuffisante pour lancer l'exécution de l'unité,
= 1 si le relieur a fait un recouvrement uniquement sur la zone des variables,
= 2 si le relieur a fait un recouvrement sur la zone des variables et sur la zone des instructions des modules.
= 3 si le relieur a fait un recouvrement sur :
+ la zone des variables,
+ la zone des constantes des modules,
+ la zone des constantes des classes.

5.4.2.3. Le retour dans le module initial

A son tour, le prééditeur rend le contrôle au module initial par l'appel de la procédure M.RETURN.

Après avoir été rechargé en mémoire, l'interprète du module de commande récupère les valeurs affectées aux paramètres d'exploitation et décide s'il lance l'exécution de l'unité de traitement (c'est-à-dire si XINDEDIT = faux) ou non.





5.5. LIAISONS DU MODULE INITIAL AVEC LES UNITES DE TRAITEMENT

Nous avons vu (en 2.4.) que, lorsque le module initial rencontre le nom d'un module dans le module de commande, il doit alors lancer l'exécution de l'unité de traitement. Pour lui, une unité de traitement est un module de chargement qu'il doit amener en mémoire et qu'il doit exécuter.

5.5.I. - Le principe du lancement de l'exécution

5.5.I.I. Les solutions possibles

Pour pouvoir réaliser cette opération, deux possibilités s'offraient à nous :

- La première possibilité était de réaliser la même opération pour une unité de traitement que pour un module du système, à savoir utiliser la procédure système M:LINK. Grâce à cette procédure, le système d'exploitation va chercher le module de chargement indiqué, l'amène en mémoire centrale et l'exécute. Le contrôle de l'unité centrale est rendu automatiquement au module initial à la fin de l'exécution.

- La deuxième possibilité est de considérer la demande d'exécution d'une unité de traitement comme la création d'un travail indépendant dans le système. Lorsque le module initial doit lancer l'exécution d'une unité de traitement, il va créer un travail en générant les cartes de commande nécessaires à la bonne exécution de l'unité, de la même façon que l'utilisateur aurait opéré dans un système classique de gestion d'application. Le module initial est donc amené à créer un fichier permanent sur disque qui contient les images des cartes de commande. Dans un deuxième temps, il lance l'exécution de ce travail en l'ajoutant au train des travaux qui sont introduits d'une manière classique dans le système d'exploitation. L'adjonction du travail au train se fait par l'appel d'une procédure du système SIRIS 7 : la procédure M:BATCH. Le travail ainsi généré est alors considéré par le système d'exploitation comme un travail classique et il est pris en compte après l'interprétation des cartes de commande. Parmis les étapes de ce travail figure la demande d'exécution de l'unité de traitement.

Le module de chargement est alors amené en mémoire et son exécution est lancée par le chargeur du système d'exploitation.

5.5.I.2 Critiques des solutions et choix

- Dans la première solution, la zone dynamique commune n'est pas concernée par l'opération. Donc, les classes système, modification et application étant implantées dans cette zone, l'unité de traitement peut avoir accès aux objets déclarés dans ces classes. L'inconvénient de cette méthode est que chaque fichier utilisateur déclaré et utilisé dans l'unité de traitement doit faire l'objet, au préalable, d'une définition pour le système d'exploitation dans une carte de commande ! ASSIGN. La définition des supports utilisés ne peut donc être que statique, la procédure M:ASSIGN permettant de faire des modifications dynamiques n'autorisant pas le changement de désignation de support. Or il est impossible au module initial de connaître, à priori, avant d'être exécuté, les localisations des fichiers physiques de l'utilisateur. Cette solution ne permet donc pas au module initial de remplir son rôle de système de gestion de fichiers.

- La deuxième solution présente l'avantage de pouvoir assurer à l'utilisateur la possibilité de manipuler des fichiers, les cartes de commande étant générées par le module initial. Par contre l'inconvénient de cette solution réside dans le fait que l'utilisateur, dans son unité de traitement, doit aussi avoir accès aux objets déclarés dans les classes communes au module de commande et à l'unité. Or si on considère une exécution d'unité de traitement comme un travail indépendant, il n'y a plus de liaisons entre les objets déclarés dans ces classes et les traitements réalisés.

D'après cette critique, nous voyons donc que la première solution doit être rejetée et que la deuxième, pour pouvoir être acceptée, doit être aménagée. Il est en effet nécessaire que l'unité de traitement puisse avoir accès aux objets des classe système ou d'application. L'unité de traitement doit donc réétablir les liaisons détruites au début de son exécution.

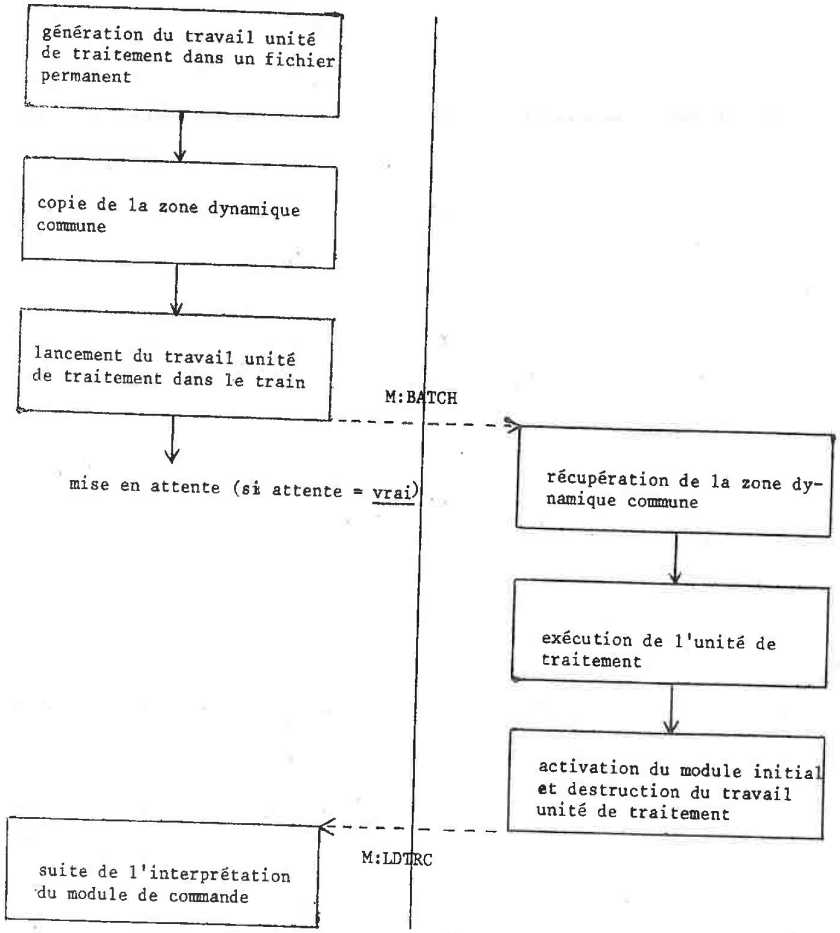
C'est pour cette raison que le module initial va tirer une copie de l'implantation des objets de la zone dynamique commune et ranger cette copie dans un fichier permanent.

Lorsque l'unité de traitement est lancée, son premier travail est de récupérer l'image conservée par le module initial et d'implanter les objets des classes système, application et commande dans la zone dynamique commune. Ainsi pendant l'exécution, l'unité de traitement peut avoir accès aux valeurs affectées à ces objets. Lorsque l'exécution est terminée, l'unité de traitement rend le contrôle au module initial qui va alors pouvoir continuer l'interprétation du module de commande (si le paramètre d'exploitation ATTENTE est positionné à vrai).

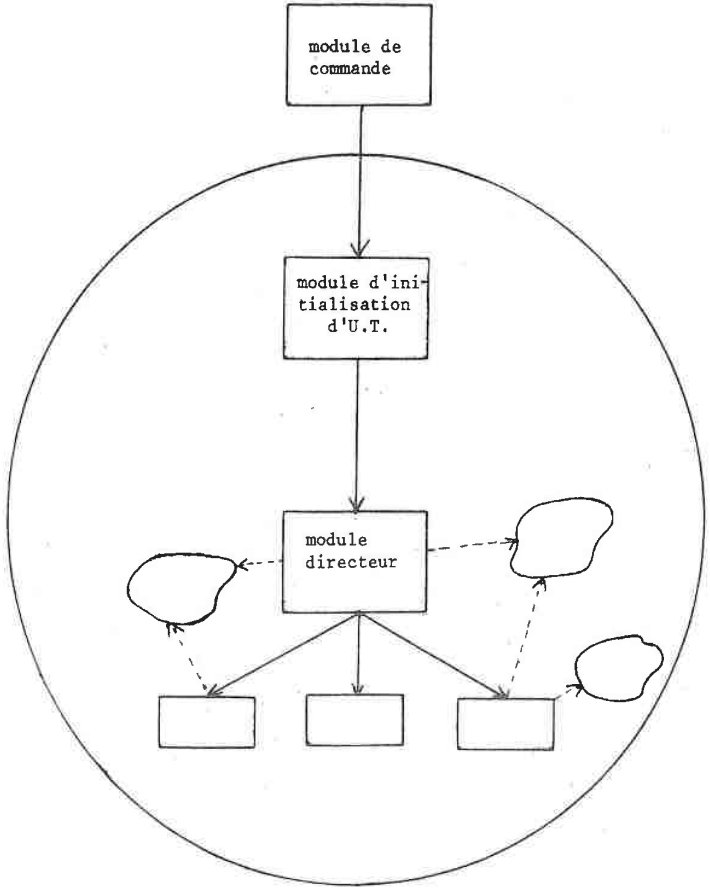
Le retour dans le module initial se fait par l'appel de la procédure système M:LDTRC qui a un rôle analogue à celui de la procédure M:LINK. La seule différence est que le programme appelant (ici l'unité de traitement) n'est pas conservé par le système d'exploitation. Cette destruction n'a aucune importance car le module initial conserve le module de chargement dans le fichier des textes édités. Le programme du module initial est alors rechargé en mémoire et son exécution est lancée. La zone dynamique commune n'étant pas affectée par cette opération, le module initial peut de nouveau avoir accès aux objets des classes communes.

5.5.2. - Représentation schématisée

On peut schématiser ces opérations de la façon suivante :

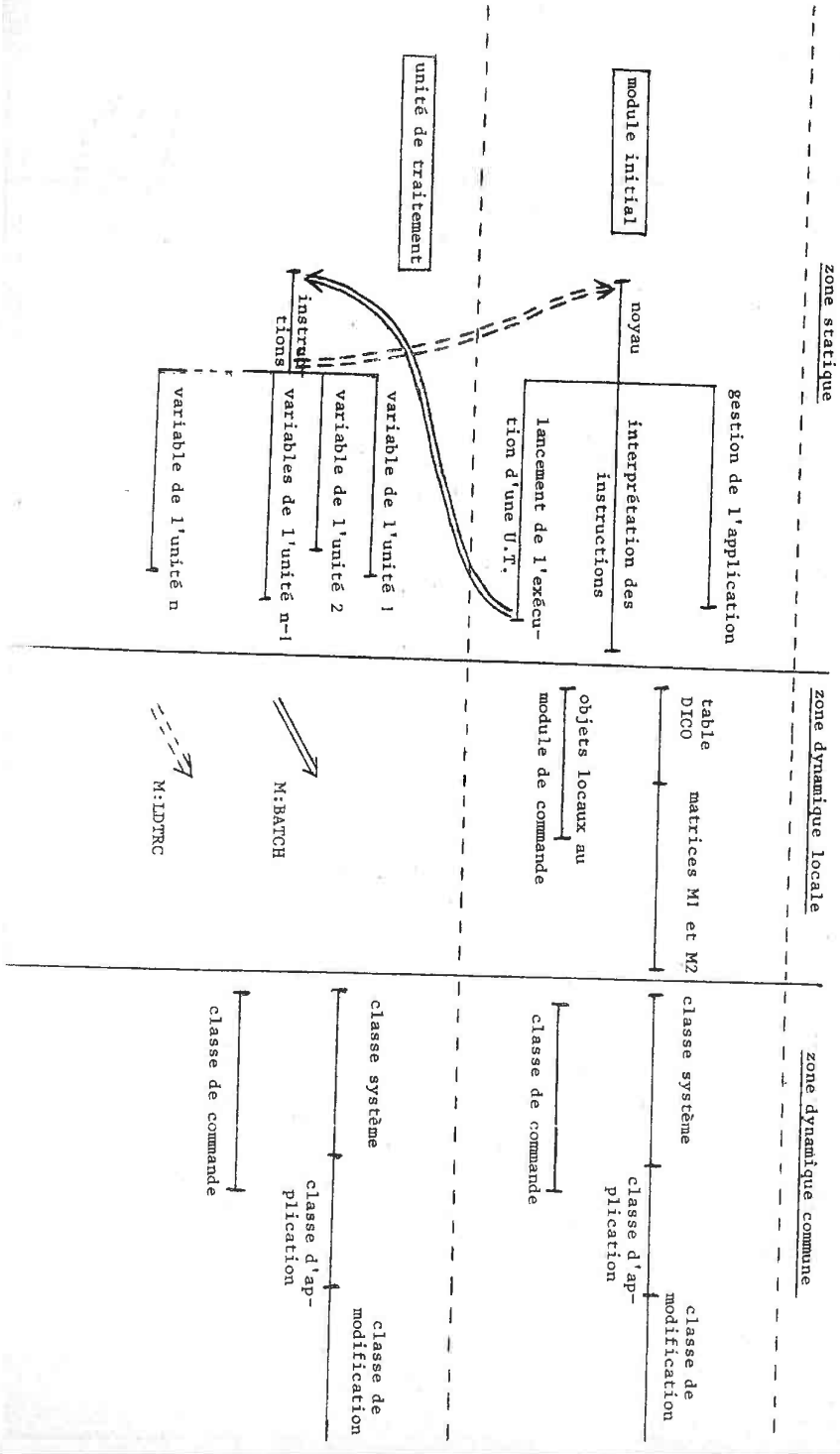


Le schéma général de fonctionnement du module de commande tel qu'il a été présenté au chapitre 1 doit donc être légèrement modifié par l'introduction d'un module supplémentaire à l'unité de traitement : le module d'initialisation d'unité de traitement. C'est lui qui est considéré comme "le programme principal" de l'unité de traitement. De toutes façons, la création de ce module était obligatoire cas tous les modules, qu'ils soient directeurs ou non, sont traités par le compilateur comme des sous-programmes fermés externes.



UNITE DE TRAITEMENT

L'existence du module d'initialisation d'unité de traitement est entièrement transparente à l'utilisateur.



5.6. LA GESTION DES TRANSITIONS D'ETAT DES UNITES

Nous avons vu en 2.6.2. que dans le système CIVA, l'utilisateur n'a pas à se préoccuper de la gestion de ses unités de programmes, il lui suffit de soumettre un texte source et d'en demander l'exécution : les différentes transformations du texte source en programme exécutable sont prises en charge par l'interprète des modules de commande. Dans ce paragraphe, nous allons analyser quels sont les états possibles d'une unité et comment l'interprète gère les transitions d'Etat.

5.6.I. - Les états possibles des unités

Nous avons distingué trois types d'unités dans une application : les classes, les modules et les métamodules.

5.6.I.I. Les classes

Une classe est un ensemble de déclarations (identificateurs, constantes, déclaration de types, procédures ....). L'ensemble des déclarations fait dans une classe peut être utilisé par un module.
On peut donc envisager deux états possibles d'une classe :

- l'état source : dans ce cas, le système CIVA ne connaît que le texte source de la classe, ce texte étant rangé dans le fichier système des textes sources de l'application.
- l'état compilé : la classe a été traitée par le compilateur CIVA. Celui-ci lui associe une table des identificateurs, ainsi que divers renseignements utiles pour les autres opérations (taille de la zone des constantes, taille de la zone des variables etc...). Le texte résultant de la compilation est rangé dans le fichier système des textes compilés de l'application.

Quelle que soit son état, une classe est représentée dans le système par un élément dans le dictionnaire et par un enregistrement dans le fichier descriptif.

5.6.I.2. Les modules

Un module correspond à la notion de sous-programme fermé telle qu'elle est définie dans les autres langages de programmation.
C'est une suite de déclarations et d'instructions du langage CIVA.

Un module peut exister sous trois états :

- l'état source : seul le texte source est connu du système CIVA.

- l'état compilé : le module a été traité par le compilateur CIVA. Celui-ci a donc généré un texte objet rangé dans le fichier système des textes compilés.

- l'état édité : l'état édité signifie que le module a ensuite été traité par le relieur. Ce module peut alors être considéré comme un programme principal dont toutes les références externes ont été satisfaites. Le texte résultant de la reliure est le module de chargement qui peut être exécuté. Ce module de chargement est rangé dans le fichier système des textes édités de l'application. Seul un module directeur peut être dans cet état. De même que pour les classes, les modules sont représentés dans le système par un élément dans le dictionnaire et par un enregistrement dans le fichier descriptif.

5.6.I.3. Les métamodules

Un métamodule peut être considéré comme un module dont le texte n'est généré que lors de la compilation de son appel, texte qui peut dépendre de paramètres connus à l'appel. Un métamodule permet de générer du texte source dans un module ou une classe. Il résulte qu'un métamodule ne peut exister dans le système qu'à l'état source.

5.6.2. - Relations entre les unités et conséquences

Au chapitre 1, nous avons rappelé les différentes relations qui peuvent lier les unités d'une application. Ces relations sont au nombre de trois.

5.6.2.I. La relation "utilise"

Soit x une unité et y une classe:
x $\cup$ y $\Longleftrightarrow$ tous les identificateurs déclarés dans y sont utilisables dans x.

Avant de pouvoir compiler x, il faut donc que y soit déjà à l'état compilé. Si y est modifié, quelles sont les conséquences sur x ?

- si x est un module, x est à recompiler. En effet l'utilisateur a très bien pu changer, par exemple, des types de déclaration dans y. Ces types doivent donc être changés dans x.

- si x est un métamodule, son état bien entendu ne change pas puisque x est toujours à l'état source.

- si x est une classe, x est à recompiler car toutes les déclarations de y sont utilisables dans x.

Donc nous pouvons constater que, quelque soit le type de l'unité x, si y est modifié, x passe à l'état source.

Remarque

On peut remarquer toutefois que la recompilation de x, dans le cas où x est une classe ou un module, n'est pas obligatoire.

Exemple

textes avant modification	*textes après modification*
classe y ; I *entier* ; J *réel* ; *finclasse* ;	*classe* y ; I *entier* ; K *entier* ; R *réel* ; *finclasse* ;
module x ; *utilise* y ; I = 1 ; *finmod* ;	*Module* x ; *utilise* y ; I = 1 ; *finmod* ;

On voit très bien que la modification apportée à la classe y n'entraîne pas de modifications sur le texte compilé de x. En effet, l'équivalence en METASYMBOL du résultat du compilateur est la suivante (DUCLOY [9]) :

```
classe y
        DEF     $y
$y      EQU     $
        RES     1   (I)
        RES     1   (J)
```

```
classe y
        DEF     $y
$y      EQU     $
        RES     1   (I)
        RES     1   (K)
        RES     1   (R)
```

```
module x
        REF     $y
        DEF     x
x       EQU     $
        LI,9    1
        STW,9   $y  (I=1)
```

```
module x
        REF     $y
        DEF     x
x       EQU     $
        LI,9    1
        STW,9   $y  (I=1)
```

Une étude plus fine pourrait donc essayer de déterminer automatiquement si oui ou non, l'unité x est à recompiler (l'interprète du module de commande pourrait, en particulier, s'appuyer sur les résultats fournis par le système automatique de documentation (cf ; BARTHELEMY [2] et PHILIPPE [2]).
Dans une première version, le module initial recompile systématiquement l'unité x, si x est un module ou une classe.

5.6.2.2. La relation "appelle"

Soient x un module ou un métamodule et y un module:

x A y $\Longleftrightarrow$ y est un sous-programme fermé externe de x.

- Si un module x appelle un autre module y, le compilateur génère alors dans le texte objet de x, l'équivalent METASYMBOL suivant (DUCLOY [9]) :

```
x   EQU     $
      "
      "
    REF     y
    BAL,15  y (instruction d'appel de sous-programme).
```


y devient alors une référence externe qui n'est satisfaite qu'au moment de l'édition de liens. Nous pouvons alors constater qu'une modification apportée au texte de y n'entraîne aucune modification dans le texte compilé de x.
Mais tout module x tel que x $\hat{A}$ y et dont l'état est édité doit revenir à l'état"compilé". Ce qui signifie en particulier que si z est une classe modifiée, tout module x à l'état "édité" et tel que x $\hat{A}\,\hat{U}$ z doit revenir à l'état "compilé".
En conséquence, si x est un module :

+ si x est à l'état source, x reste à l'état source,
+ si x est à l'état compilé, x reste à l'état compilé,
+ si x est à l'état édité, x passe à l'état compilé.

- si x est un métamodule, une modification du texte de y n'entraîne aucune modification dans x. Le métamodule est toujours à l'état source.

5.6.2.3. La relation "$appelle"

Soient x une unité quelconque et y un métamodule.

x $A y $\Longleftrightarrow$ y permet de générer du texte source dans x.

Donc, nous pouvons dire que, quel que soit le type et l'état de l'unité x, celle-ci doit passer à l'état source si y est modifié et éventuellement, les textes compilés et édités de x présents dans le système doivent être détruits.

5.6.3. - L'appel au module de gestion des transitions d'état

Après avoir présenté les différentes transitions d'état des unités, le problème qui se pose à nous est de savoir quand doit-on lancer l'exécution du module de gestion des transitions d'état. Ce problème admet deux solutions possibles :

- Le module initial lance la gestion des transitions d'état dès qu'il a repéré dans le texte du module de commande un appel à une procédure de modification de texte, que ce soit CHANGER (cf: 2.5.3.I.I.), CORRIGER (cf : 2.5.3.I.2.) ou REMPLACER (cf : 2.5.3.I.3).

- Le module initial, lorsqu'il rencontre un appel d'une telle procédure, se contente de conserver dans une liste le matricule de l'unité modifiée. C'est seulement avant de lancer l'exécution d'une unité de traitement ou avant de réaliser les opérations de terminaison du système que l'interprète lance l'exécution du module de gestion des transitions d'état.

Or, dans un module de commande, l'utilisateur peut opérer plusieurs modifications de la même unité (par exemple, changer son nom avec REMPLACER puis changer quelques instructions avec CORRIGER). Le choix de la première solution oblige le système CIVA à réaliser autant d'appel au module de gestion des transitions d'état qu'il y a de modifications réalisées. Par contre la deuxième solution ne réalise qu'un seul appel à ce module.
De plus, pour chaque unité y modifiée par l'utilisateur, le module initial doit appliquer les règles de transitions que nous avons donné précédemment à chacune des unités de :

$$\alpha = \hat{R}^{-1}(y)$$

Les unités de α sont les prédecesseurs de l'unité y dans le graphe G2 défini en 4.I.II. Ce graphe est représenté en mémoire par la matrice de bits M2. Or cette matrice n'est construite que lorsque le système désire l'utiliser. Le choix de la première solution nécessite donc que, pour chaque unité modifiée, le système CIVA construise cette matrice M2 et explore le graphe. Par contre, dans le cas de la deuxième solution, cette matrice M2 n'est construite qu'une seule fois pour toutes les unités qui ont subies une modification de texte depuis la dernière gestion de transition d'état.
A la suite de ces deux remarques, la deuxième solution s'avère la plus rapide et c'est cette dernière que nous avons adoptée dans notre réalisation.

Un problème similaire se présente également quant au choix de l'instant de la recompilation ou de la réédition des unités qui ont subi un changement d'état par suite d'une modification.

De même, ce problème admet deux solutions : ou bien la recompilation (ou la réédition) est réalisée dès que la transition de l'état de l'unité est connue, ou bien cette recompilation (ou réédition) n'est réalisée que lorsque l'utilisateur a fait une demande de'éxécution d'une unité de traitement contenant cette unité.
Pour la même raison que celle que nous venons d'indiquer, à savoir que l'utilisateur peut réaliser plusieurs modifications sur une même unité dans un module de commande, nous avons retenu la deuxième solution dans la réalisation de notre projet : si une unité était compilée et si le module de gestion des transitions d'état la fait passer à l'état"source"; le texte objet de cette unité est supprimé. Le nouveau texte ne sera obtenu que lorsque l'utilisateur aura fait une demande d'exécution d'une unité de traitement contenant cette unité (la recompilation de l'unité peut bien entendu avoir lieu avant, sur demande explicite de l'utilisateur dans son module de commande grâce à la procédure COMPILER présentée en 2.6.3.I.).

5.7. LE LANCEMENT DES COMPILATIONS DES UNITES

Lorsqu'il rencontre une demande d'exécution dans un module de commande, le module initial vérifie que le module directeur est crée dans l'application et qu'il y figure à l'état "édité".
Si le module directeur n'est pas crée dans l'application, c'est une erreur et le module initial émet un message d'erreur à destination de l'utilisateur. S'il est créé mais n'est pas à l'état "édité", il faut l'y mettre.
Or un module x ne peut passer à l'état "édité" que si tous les modules appartenant à $\hat{A}(x)$ et toutes les classes appartenant à $\hat{U}(x)$ sont créés dans l'application et y figurent à l'état"compilé". Si l'un d'entre eux ou l'une d'entre elles est absent, c'est une erreur et l'exécution ne peut avoir lieu. Si toutes les unités sont créées mais si au moins l'une d'entre elles n'est pas à l'état "compilé", le système CIVA doit d'abord lancer les compilations nécessaires. C'est cette opération que nous allons analyser maintenant.

5.7.I. - Détermination de l'ordre des compilations

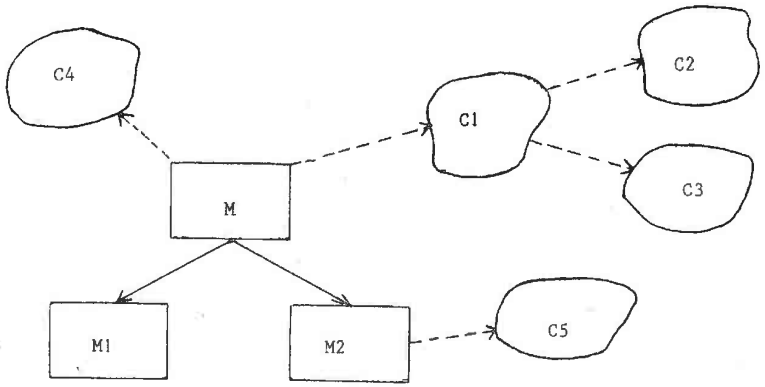
Pour pouvoir compiler une classe ou un module x, il faut que toutes les classes de $\hat{U}(x)$ soient créées et compilées de façon que le codifieur puisse inclure leur table des identificateurs dans celle de l'unité x. (DUCLOY [9]).

A la demande d'exécution d'une unité de traitement non éditée, le module initial doit donc déterminer un certain ordre de compilation des unités qui sont à l'état"source", avant même de pouvoir lancer leur compilation. Pour pouvoir déterminer cet ordre de compilation, on peut se baser sur plusieurs remarques :

- les compilations des modules sont indépendantes les unes des autres. Il n'y a donc pas d'ordre imposé pour compiler une série de modules.
- les classes utilisées par un module x doivent être compilées avant ce module et dans un ordre qui peut être obtenu grâce au graphe de la relation $\Uparrow^{-1}(x)$ où x est un module. Cette relation constitue un pré-ordre sur l'ensemble des classes : les classes équivalentes seront donc compilées dans un ordre arbitraire.

Exemple

Soit l'U.T. suivante, où M est le module directeur :



Supposons que toutes les unités sont à l'état "source".

- les modules M, M1, M2 peuvent être compilés dans un ordre arbitraires mais :

- avant de lancer la compilation de M, il faut compiler les classes C1, C2, C3 et C4.

+ C1 et C4 sont équivalentes;

+ C2 et C3 sont équivamentes mais leur compilation doit précéder celle de C1.

- avant de lancer la compilation de M2, il faut compiler C5. Un ordre de compilation peut donc être :

C2, C3, C1, C4, M, C5, M2, M1

5.7.2. - Modification apportée à la solution précédente

La solution qui vient d'être décrite est satisfaisante si nous connaissons toutes les relations liant les unités entre elles et que ces relations sont fixées. Par contre, cette solution doit être légèrement modifiée, à cause de la possibilité dans le langage d'utiliser des classes de façon conditionnelle.

En effet dans le texte d'un métamodule, l'utilisateur a la possibilité d'utiliser des classes de façon conditionnelle grâce à la métainstruction $ SI (Il peut également réaliser des appels conditionnels de modules).

Cette possibilité crée des contraintes importantes pour le module initial.

Exemple

```
classe x ;
entier I ; I = 1 ;
$ $ M ;
finclasse ;

métamod $M ;
$ SI  I = 1 alors utilise y sinon utilise z ;
finmod ;
```

Lorsque le codifieur traduit la classe x, il passe le contrôle à l'interprète du métalangage qui génère le texte source. Le métamodule $M peut entraîner une génération de "utilise x" ou de "utilise z" suivant les cas. A priori, le module initial ne sait pas quelle classe est utilisée par x, donc il ne sait pas que y (ou z) doit être compilé avant elle.

Une solution apportée afin de remédier à cet inconvénient peut être la suivante.

- Le module initial divise l'ensemble des métamodules en deux sous-ensembles :

  + le premier contient les métamodules déclarés dans la classe d'application (ce sous-ensemble correspond donc à l'ensemble MM défini précédemment).

+ le deuxième contient les métamodules déclarés dans les autres classes.

A la rencontre de la déclaration d'un métamodule dans la classe d'application, son nom est mis dans le fichier descriptif ainsi que les numéros de matricule des classes qui figurent dans des "utilise" de ce métamodule (utilise conditionnel ou non : dans l'exemple précédent on mettrait y et z).

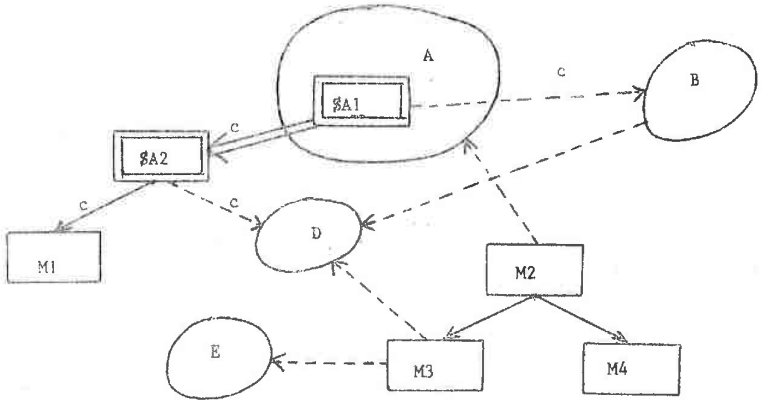
A la rencontre de la déclaration d'un métamodule dans une classe C autre que la classe d'application, le module initial ne crée pas d'enregistrement particulier pour le métamodule dans le fichier descriptif, car il n'est pas utilisable par toutes les unités de l'application. L'interprète traite donc le métamodule comme un objet élémentaire quelconque déclaré dans la classe.

- A la déclaration d'une classe ou d'un module dans la classe d'application, le module initial crée un enregistrement dans le fichier descriptif qui rassemble le nom de l'unité, les appels de métamodules, les utilisations de classes (conditionnelles ou non, déclarées dans un métamodule interne ou non), les appels de modules.

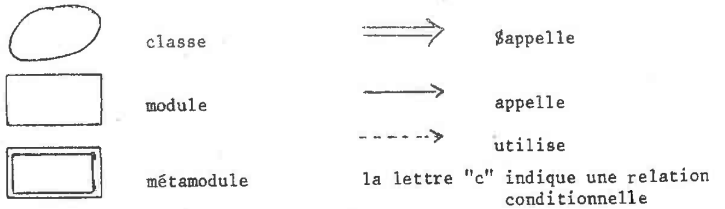
- A la demande d'exécution d'un module directeur, un certain nombre de classes et de modules ne sont pas encore compilés. Le module initial, avant de lancer la compilation d'un module examine à l'aide du graphe de la relation $(\widehat{U \cup \$A})^{-1}$, l'ensemble des classes à compiler et en déduit un ordre de compilation à respecter.(L'algorithme de cette ordre sera vu en 5.7.4.). Les classes sont alors compilées dans cet ordre. Si une classe est absente de l'application (elle peut n'avoir pas été créée), le compilateur passe à la classe suivante. Ce n'est pas forcément une erreur, la classe pouvant ne pas être utilisée effectivement. Si c'est une erreur, celle-ci sera découverte plus tard, au cours de la compilation d'un module par exemple, ou bien au cours de la reliure. Pendant l'opération de compilation et plus précisément pendant la première phase, le métatraitement est entrepris et il n'y a plus que des utilisations effectives de classe. Le fichier descriptif est alors mis à jour.

5.7.3. - Exemple de lancement de compilation

Prenons l'unité de traitement, de module directeur M2, composée des éléments suivants :



Légende



D'après le schéma précédent, l'unité de traitement contient deux métamodules :

$A1 est un métamodule interne à la classe A et il ne peut être appelé que par les unités utilisant la classe A.

$A2 est un métamodule déclaré dans la classe d'application (il a donc été créé par un appel de la procédure CREER) et il peut être appelé par toute unité de l'application.

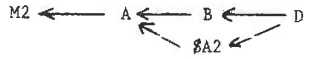
Le fichier descriptif contient alors les renseignements suivants :

nom de l'unité	type	état	métamodules appelés	classes utilisées	modules appelés
A	classe	source	$A2	B	
B	classe	source		D	
D	classe	source			
$A2	méta-module	source		D	M1
M1	module	source			
M2	module	source		A	M3, M4
M3	module	source		D, E	
M4	module	source			
E	classe	source			

Supposons que l'utilisateur demande l'exécution du module M2. Toutes les unités doivent donc être éditées. Pour cela il faut les compiler.

- compilation de M2

Avant de lancer la compilation de M2, il faut lancer les compilations des classes utilisées ou susceptibles d'être utilisées par M2. Ces classes appartiennent à $\widehat{(U \cup \$A)}$ (M2) et l'ordre de compilation peut être donné par la relation inverse.
On obtient le schéma suivant pour la relation inverse :



L'ordre de compilation sera : D, B, A, M2

- Compilation de M3

Le module initial doit lancer les compilations des classes de $\widehat{(U \cup \$A)}$ (M3).

L'ordre de compilation est donc D, E, M3.

- compilation de M4

M4 n'utilise pas de classe donc M4 peut être compilé.

- compilation de M1

M1 n'utilise pas de classe donc M1 peut être compilé.

L'ordre de compilation pour toutes les unités sera alors :

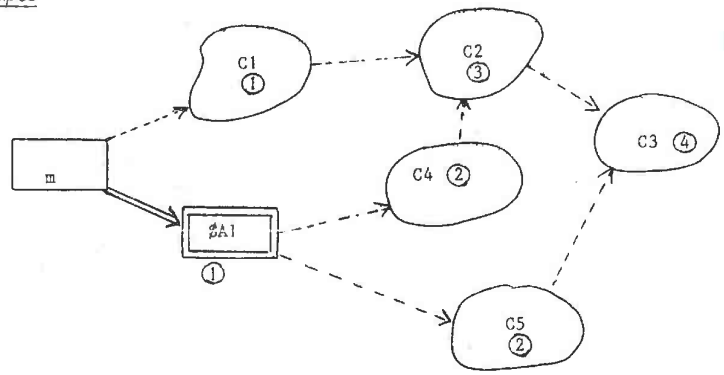
D, B, A, M2, E, M3, M4, M1.

Les opérations de compilation sont lancées et le module initial après avoir repris le contrôle, peut lancer l'opération de reliure avant l'exécution de l'unité de traitement.

5.7.4. - Algorithme de détermination de l'ordre de compilation des classes

Pour déterminer l'ordre de compilation des classes, le "module initial" pourrait affecter à chaque sommet du graphe $G = (\{m\} \cup C \cup MM, \widehat{U \cup \$A})$ où m est un module à compiler, un certain nombre qui représente la longueur du plus grand chemin du module à compiler à ce sommet.
Alors on peut remarquer que l'ordre des classes à compiler est obtenu à partir de ces longueurs : la première classe à compiler est celle qui a la plus grande longueur, la classe suivante est celle qui a une longueur immédiatement inférieure....Lorsque deux classes ont des longueurs égales, l'ordre de compilation est alors arbitraire.

Exemple



(n) = longueur du plus grand chemin du sommet au module.

L'ordre de compilation des classes est alors :

C3, C2, {C4, C5}, C1.

L'algorithme de détermination de l'ordre de compilation des classes se ramène donc à un algorithme de détermination de la longueur du plus grand chemin d'un point de graphe à un autre.
Dans (DERNIAME - PAIR [47]), plusieurs algorithmes sont proposés pour résoudre ce problème :

- l'algorithme de FORD ou de BELLEMAN-KALABA,
- la méthode de DEMOUCRON qui consiste à classer les points par "niveaux",
- un algorithme utilisant une pile annexe du graphe,

ces deux dernières méthodes permettant également de détecter les circuits.

La méthode de DEMOUCRON demande un nombre d'opérations proportionnel à n^2, où n est le nombre de points du graphe G, alors que l'emploi d'une pile demande un temps proportionnel au nombre m des arcs du graphes. C'est pour cette raison que nous avons choisi un algorithme utilisant une pile.

La pile annexe utilisée par l'algorithme est une suite finie $(a_o, a_1, \ldots a_n)$ sur l'ensemble des points du graphe, a_i étant un état de la pile.

- a_o = e (e est la suite vide)
- a_1 = m (1 est l'entrée du point représentant le module à compiler)
- pour $i \geqslant 2$:
  - + si i - 1 est une entrée de p :
    - a) s'il s'agit de la première entrée de p et si $(U \cup \$A)(p) \neq 0$, i est une entrée du premier élément de $(U \cup \$A)(p)$;
    - b) s'il s'agit d'une entrée de p autre que la première ou si $(U \cup \$A)(p) = \emptyset$, i est une sortie de p ;
  - + si i - 1 est une sortie de p et si a_{i-1}, n'est pas vide et admet pour sommet p' :
    - a) si p n'est pas le dernier point de $(U \cup \$A)(p')$ i est une entrée du point qui suit p dans $(U \cup \$A)(p')$;
    - b) si p est le dernier élément de $(U \cup \$A)(p')$, i est une sortie de p' ;
  - + si a_{i-1} est vide, a_{i-1} est le dernier état de la pile.

L'emploi de cette pile annexe permet alors de déterminer l'ordre des compilations : c'est l'ordre de sortie de la pile.

Exemple

Reprenons l'exemple précédent.
En appliquant l'algorithme, les états de la pile sont les suivants :

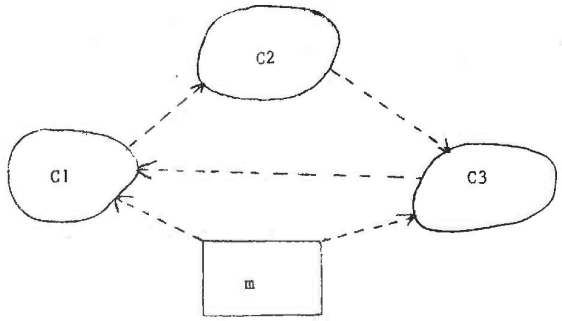
i	a_i
0	
1	m
2	m, C1
3	m, C1, C2
4	m, C1, C2, C3
5	m, C1, C2
6	m, C1
7	m
8	m, $A1
9	m, $A1, C4
10	m, $A1, C4, C2
11	m, $A1, C4
12	m, $A1
13	m, $A1, C5
14	m, $A1, C5, C3
15	m, $A1, C5
16	m, $A1
17	m
18	

L'ordre de compilation obtenu est alors :

C3 - C2 - C1 - C4 - C5

Cet algorithme permet de détecter la présence de circuits dans le graphe G. Si $a_i = (n_1, n_2, \ldots n_k)$ et si $n_m = n_p$ pour $1 \leqslant m \leqslant k$ et $1 \leqslant p \leqslant k$, alors le graphe possède un circuit.

Exemple



					C1						
				C3	C3	C3					
			C2	C2	C2	C2	C2				
		C1	C1	C1	C1	C1	C1	C1		C3	
m	m	m	m	m	m	m	m	m	m	m	m
a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	

L'état a_5 montre la présence du circuit C1 - C2 - C3 - C1.

CONCLUSION

Les modules de commande, tels que nous venons de les définir, permettent donc à un utilisateur d'exprimer les actions qu'il désire voir réaliser en vue de l'exploitation de ses traitements. La conception et l'implémentation des outils que nous avons proposés nous ont amenés à un certain nombre de réflexions relatives à l'exploitation des programmes et plus particulièrement aux définitions et aux utilisations des langages de commande.

On peut remarquer tout d'abord qu'il est à déplorer que la définition de langage d'exploitation de haut niveau ne soit pas un des principaux soucis des constructeurs de logiciels. L'application d'une telle idée donnerait naissance à des systèmes d'exploitation d'utilisation beaucoup plus simple et plus fonctionnelle que ceux que nous connaissons actuellement.

Par exemple, lorsqu'un utilisateur désire exécuter un programme qui se présente sous la forme d'un texte source, pourquoi doit-il rappeler à chaque exploitation qu'il est nécessaire de faire une compilation puis une édition de liens avant de passer à l'exécution proprement dite, les deux premières étapes étant obligatoires ?

Si l'on veut faire un pas de plus dans le domaine de l'aide à l'exploitation, il semble que la constitution d'un système de gestion automatique des applications des utilisateurs offrent des perspectives nouvelles. Il paraît en effet plus aisé et plus fiable d'assurer des opérations telles que gestion des transitions d'état des programmes, lancement des compilations, mise à jour des dossiers de l'utilisateur... de manière automatique plutôt que manuellement.

La deuxième remarque que nous ferons permet de ressituer notre travail par rapport aux études qui ont déjà été réalisées sur les langages d'exploitation de haut niveau.

Dans notre système, la notion de "travail" est remplacée par la notion de "module de commande" qui correspond généralement à la demande d'exécution d'une chaîne de traitement. Les objets manipulés par ce module ont des types définis dans le langage de base. Les instructions qu'il contient sont en tout point identiques à celles des modules de traitement. De plus, grâce aux classes de commande, d'application et système, le programmeur peut contrôler l'exécution de ces programmes et réaliser des exécutions conditionnelles de ses unités de traitement.

Enfin les outils de maintenance tels que aides à la mise au point, éditeur de texte ... utilisables dans un module de commande font de notre système un système de programmation intégré et sophistiqué.

La possibilité de cataloguer n'importe quél fichier physique qu'il soit implanté sur mémoires auxiliaires ou non, la possibilité de conserver d'une exploitation à l'autre la connexion entre un fichier logique et un fichier physique et enfin l'assimilation d'un fichier logique à un type d'objet déjà défini dans le langage de base sans introduction d'instructions supplémentaires de manipulation de fichiers font du système que nous avons proposé un système de gestion de fichiers intégré.

En ce qui concerne notre participation au sein de l'équipe CIVA, il nous paraît intéressant de développer plusieurs points afin d'améliorer l'utilisation du système proposé.

Les modules de commande donnant l'accès aux procédures de modification, des calculs pouvant y être exécutés, des compilations pouvant être demandées en mode mise au point, des informations pouvant être échangées entre les modules de commande et les unités de traitement ou même avec le système, le système CIVA constitue donc un système d'aide à la conception d'unités de traitement. Le système de documentation qui sera proposé dans (BARTHELEMY [2]), associé aux procédures de modifications en fait un système d'aide à la maintenance de ces unités.

L'ensemble de ces outils trouverait pleinement son intérêt dans un contexte conversationnel, ce qui présenterait l'avantage pour l'utilisateur de pouvoir contrôler immédiatement la validité et les résultats des demandes qu'il propose au système et d'agir en conséquence.

De plus, l'interprétation interactive du module de commande permettrait également d'assurer à l'utilisateur un accès direct aux données. La possibilité de réaliser les calculs au niveau du module de commande et d'accéder interactivement aux informations permettrait alors d'étendre l'utilisation du langage CIVA à la réalisation de banques de données. Cette extension du champ d'application de notre projet nécessitera certainement l'introduction dans le langage de base de nouveaux types de données plus sophistiqués pour pouvoir représenter les structures d'informations particulières manipulées par les systèmes de gestion de base de données.

Dans un avenir plus lointain, il serait également intéressant d'étudier la possibilité d'utilisation du langage CIVA dans un réseau d'ordinateurs. Nous pensons que grâce à la description modulaire des traitements et des informations, ce langage pourrait très bien être utilisé comme outil d'aide à la conception d'un système implanté sur un réseau : un ordinateur pourrait générer un ou plusieurs modules de commande à destination d'autres ordinateurs du réseau, ce ou ces modules représentant alors les demandes d'exploitation d'un site aux autres.

A N N E X E S

ANALYSE EN CIVA DU MODULE INITIAL ET DE SON ENVIRONNEMENT.

ANNEXE A : Les fichiers-systèmes d'une application.

ANNEXE B : Description des tables du système.

ANNEXE C : Les procédures de la classe système.

ANNEXE D : Les procédures de la classe de modification.

ANNEXE E : Description de quelques traitements réalisés par le module initial.

ANNEXE F : Exemples de modules de commande.

ANNEXE A : Les fichiers systèmes d'une application

Pour gérer les unités et les objets élémentaires déclarés dans une classe d'application, le module initial assure la gestion des fichiers systèmes d'une application (cf. 4.1.). Dans cette partie, nous présentons la définition de ces fichiers.

A.1) Le fichier dictionnaire :

```
classe fichier dictionnaire ;
NBDICØ entier ;
```

CØ NBDICØ contient le nombre d'unités figurant dans le fichier dictionnaire de l'application CØ ;

```
UNITE CREEE booléen ;
UNITE PRESENTE booléen ;
AJØUTE booléen ;
```

CØ ces objets sont utilisés par les procédures définies ci-après CØ ;

```
DICØ file externe structure (
     NØMDIC file (max = 10) caractère,
     NØM2DIC file (max = 10) caractère,
     MATDIC entier,
     TYPDIC code (module, classe, procédure, métamodule,,,, classe-com),
     INDIC booléen) ;
DICTIØNNAIRE de DICØ ;
```

CØ NØMDIC contient le nom de l'unité.
NØM2DIC est utilisé pour les procédures ou les métamodules qui ne sont pas déclarés dans la classe d'application mais dans une classe quelconque. Dans ce cas, NØM2DIC reçoit le nom de cette classe.
MATDIC contient le numéro de matricule de l'unité.
TYPDIC indique le type de l'unité.
INDIC est l'indicatif de création de l'unité dans l'application. Cet indicatif est positionné à "vrai" si l'unité a été créée dans l'application, à "faux" sinon CØ ;

```
procédure recherche dans dictionnaire (NØM, CLE) ;
NØM file (max = 10) car ; CLE entier ;
```

CØ cette procédure permet de chercher le numéro de matricule d'une unité ainsi que son rang, connaissant son nom. Elle positionne :

UNITE CREEE à vrai si l'unité est créée,

UNITE PRESENTE à vrai si l'unité figure dans le fichier CØ ;

```
UNITE CREEE = faux ;
UNITE PRESENTE = faux ;
soit  DICTIØNNAIRE telque NØMDIC = NØM ;
si DICTIØNNAIRE ≤ NBDICØ alors UNITE PRESENTE = vrai ;
                               CLE = MATDIC ;
                               UNITE CREEE = INDIC ; fsi ;
finproc ;
```

```
procédure recherche nom dans dictionnaire (NØM, CLE) ;
   NØM file (max = 10) car ; CLE entier ;
```

CØ cette procédure permet de chercher le nom d'une unité connaissant son numéro de matricule CØ ;

```
soit DICTIØNNAIRE  telque  MATDIC = CLE ;
si   DICTIØNNAIRE ≤ NBDICØ alors  UNITE PRESENTE = vrai ;
                                  NØM = NØMDIC ;
                                  UNITE CREEE = INDIC ;
                           sinon  UNITE PRESENTE = faux ; fsi ;
finproc ;
```

```
procédure ajouter à dictionnaire (NØM1, NØM2, TYPE, BØØL) ;
NØM1, NØM2 file (max = 10) car ; TYPE entier ; BØØL booléen ;
```

CØ cette procédure permet d'ajouter une unité au dictionnaire. NØM1 indique le nom de l'unité, NOM2 le nom de la classe dans laquelle est déclarée cette unité si ce n'est pas la classe d'application, TYPE est le type de l'unité, BOOL est un paramètre positionné à vrai si l'unité que l'on doit ajouter est en cours de création, à faux sinon CØ ;

```
CLE entier ;
Recherche dans dictionnaire (NØM1, CLE) ;
Si non UNITE PRESENTE alors
      si NBDICØ = 0 alors positionner indicatif (application courante, 'DICØ') ;
                                                                  fsi ;
```

CØ si aucun élément ne figurait encore dans le dictionnaire, la procédure positionne l'indicatif de création du fichier de l'application courante (cf. § 4, 2, 1) CØ ;


```
NBDICØ = NBDICØ + 1 ;
PGMAT  = PGMAT  + 1 ;
dernier DICTIØNNAIRE ; DICTIØNNAIRE en DICTIØNNAIRE + 1 ;
NØMDIC = NØM1 ;
NØM2DIC = NOM2 ;
MATDIC = PGMAT ;
INDIC  = BØØL ;
AJØUTE = vrai ;
sinon si non unité créée et BØØL alors
            INDIC = BØØL ; AJØUTE = vrai ;
      sinon AJØUTE = faux ; fsi ; fsi ;
fin proc ;
procédure supprimer de dictionnaire (NØM) ;
  NØM file (max = 10) car ;
```

CØ cette procédure permet de supprimer une unité du dictionnaire. Lorsque l'utilisateur supprime une unité, le module initial doit mettre l'indicatif de création à faux. De plus, si cette unité est une classe, il faut également enlever les procédures et métamodules déclarés dans cette classe CØ ;

```
  recherche dans dictionnaire (NØM, CLE) ;
si non unité présente alors sortir ; fsi ;
      INDIC = faux ;
si TYPEDIC = 'classe' alors
      pour chaque DICTIØNNAIRE de DICØ telque
                                    NØM2DIC = NØM faire
          INDIC = faux ; fsi ;
fin proc ;
fin classe ;
```

A.2) Le fichier descriptif :

La définition de ce fichier a été présentée par ailleurs (cf. 4.1.4.) pour des raisons de clarté et de compréhension de ce rapport.

A.3) Le fichier des textes sources :

```
classe fichier texte source ;
     FICHIER-SØURCE file externe structure (
               MATSØUR entier,
               TEXTESØURCE file caractère) ;
finclasse ;
```

A.4) Le fichier des textes compilés :

```
classe fichier texte compilé ;
     FICHIER-CØMPILE file externe structure (
               MATCØMP entier,
               TEXTECØMPILE file caractère) ;
fin classe ;
```

A.5) Le fichier des textes édités :

```
classe fichier texte édité ;
     FICHIER-EDITE file externe structure (
               MATEDITE entier,
               TEXTEEDITE file caractère) ;
fin classe ;
```

A.6) Le fichier catalogue :

```
classe fichier catalogue ;
FICHCATAL file externe structure (
      IDFICH file (max = 10) caractère,
      LISTETI file (3) file (max = 4) caractère,
      LISTUTILISATIØN file structure (
            NUMMAT entier,
            UTILISATIØN file (3) booléen),
      IDPHY file (max = 10) caractère,
      TYPELIAISØN booléen,
      LISTUTI file (3) booléen) ;
CATALØGUE de FICHCATAL ;
fin classe ;
```


Chaque enregistrement de ce fichier contient les renseignements concernant un fichier logique déclaré dans une unité de l'application :

- le nom du fichier logique (IDFICH).
- la liste des étiquettes logiques attribuées à ce fichier par le compilateur (il y a une étiquette logique par type d'utilisation possible = lecture, écriture ou mise à jour du fichier).
- la liste des unïtés qui manipulent le fichier logique.

Pour chaque unité, l'enregistrement contient le numéro de matricule de l'unité ainsi que 3 indicatifs, chaque indicatif caractérisant un mode d'utilisation du fichier dans cette unité : le premier indicatif reçoit la valeur "vrai" si le fichier est utilisé en lecture, la valeur "faux" sinon. Le deuxième indicatif correspond à l'écriture, le troisième au mode "mise à jour".
Tous ces renseignements sont fournis par le compilateur. Les renseignements suivants sont fournis par l'appel de la procédure LIAISØN et sont exploités par le générateur de cartes de commande.

- le nom du fichier physique qui est désigné par le fichier logique,
- le type de liaison établi par l'utilisateur (temporaire ou permanente),
- le type d'utilisation indiqué par l'utilisateur dans le module de commande.

A.7) Le fichier des sauvegardes :

```
classe fichier sauvegarde ;
FICHIER-SAUV file externe structure (
            NØMSAUV file (max = 10) car,
            TEXTESAUVE file caractère) ;
finclasse ;
```

ANNEXE B : Description des tables du système

Dans cette partie, nous présenterons les analyses en CIVA décrivant les tables définies dans la classe système :

- la table des applications
- la table des unités externes
- la table des identificateurs.

B.1) La table des applications :

Cette table regroupe les renseignements concernant les applications des utilisateurs connues du système CIVA. Son utilité a été présentée en 4.2.1.

```
classe table des applications ;
  NBAPPL entier ;
```

CØ NBAPPL contient le nombre d'applications présentes dans le système CØ ;

```
  XAPTAB file structure (
        NØM file (max = 10) caractère,
        PASSE file (max = 8) caractère,
        INDICATIF file (9) booléen,
        NBUNIT entier,
        PGMAT entier,
        NBPRØT entier) ;
  APPLICATION de XAPTAB ;
```

CØ INDICATIF contient la liste des indicatifs d'état des fichiers ou partitions de fichier affecté par le système à l'application. Chaque indicatif reçoit la valeur "vrai" si le fichier correspondant est créé "faux" sinon.

Les indicatifs sont les suivants :
INDICATIF (1) pour le fichier descriptif,
INDICATIF (2) pour le fichier des textes sources,
INDICATIF (3) pour le fichier des textes compilés,
INDICATIF (4) pour le fichier des textes édités,
INDICATIF (5) pour le fichier dictionnaire,
INDICATIF (6) pour le fichier catalogue,
INDICATIF (7) pour le fichier des sauvegardes.
INDICATIF (8) est l'indicatif de la présence de la partition contenant la représentation des objets élémentaires dans le fichier partitionné des classes d'applications.
INDICATIF (9) est l'indicatif de la présence de la partition contenant la

représentation de la table des unités externes dans le fichier partitionné correspondant CØ ;

```
IND booléen ;
```

CØ ce booléen est utilisé par les procédures qui suivent CØ ;

```
procédure ajouter l'application (NØMAPPLICATION, MØTPASSE) ;
NØMAPPLICATION file (max = 10) car ;
MØTPASSE file (max = 8) car ;
```

CØ cette procédure a pour but de créer un nouvel élément correspondant à une application nouvelle dans la table CØ ;

```
soit APPLICATIØN telque NØM = NØMAPPLICATIØN ;
si APPLICATIØN ≤ NBAPPL alors
     imprimer ('application déjà existante') ;
     IND = faux ; sortir ; fsi ;
IND = vrai ;
NBAPPL = NBAPPL + 1 ;
dernier APPLICATIØN ;
APPLICATIØN en APPLICATIØN + 1 ;
NØM = NØMAPPLICATIØN ;
PASSE = MØTPASSE ;
INDICATIF (*) = faux ;
NBUNIT = NBPRØT = PGMAT = 0 ;
finproc ;
procédure supprimer l'application (NØMAPPLICATIØN, MØTPASSE, BØØL) ;
NØMAPPLICATIØN file (max = 10) car ;
MØTPASSE file (max = 8) car ; BØØL booléen ;
```

CØ cette procédure a pour but d'enlever l'élément représentant une certaine application à supprimer de la table CØ ;

```
Soit APPLICATIØN telque NØM = NØMAPPLICATIØN ;
si APPLICATIØN > NBAPPL alors imprimer ('application inexistante') ;
                         BØØL = faux ;
                         sinon si PASSE ≠ MØTPASSE alors
                               imprimer ('mot de passe incorrect') ;
                         BØØL = faux ;
```



```
                         sinon supprimer APPLICATIØN ;
                         BØØL = vrai ;
                         fsi ;
fsi ;
finproc ;

procédure positionner l'indicatif (NØMAPPLICATIØN, FICHIER) ;
NØMAPPLICATIØN file (max = 10) car ;
FICHIER file (max = 4) car ;
```

CØ cette procédure permet de positionner à la valeur vrai l'indicatif d'un fichier d'une application CØ ;

```
I entier ;
Soit APPLICATION telque NØM = NØMAPPLICATIØN ;
si APPLICATIØN > NBAPPL alors imprimer ('APPLICATIØN INCØNNUE') ;
                              IND = faux ;
sinon IND = vrai ;
      sélection sinon
```

FICHIER =	'DESC'	'SØUR'	'CØMP'	'EDIT'	'DICØ'	'CATA'	'SAUVE'	'APPL'	'EXTE'
I =	1	2	3	4	5	6	7	8	9

```
fin table ;
Si I = 0 alors imprimer ('nom de fichier inconnu') ;
              IND = faux ;
              sortir ; fsi ;
INDICATIF (I) = vrai ; fsi ;
finproc ;
procédure recherche de l'application (NØMAPPLICATIØN, BØØL) ;
NØMAPPLICATIØN file (max = 10) car ;
BØØL booléen ;
```

CØ cette procédure permet de chercher une application de nom donné dans la table des applications. Si l'application n'existe pas, elle positionne BØØL à faux. Si l'application existe, le booléen BØØL est positionné à vrai et l'élément courant de la table des applications pointe vers l'élément correspondant à l'application cherchée CØ ;

```
Soit APPLICATIØN telque NØM = NØMAPPLICATIØN ;
si APPLICATIØN > NBAPPL alors BØØL = faux ;
                       sinon BØØL = vrai ;
fsi ;
finproc ;
finclasse ;
```

B.2) La table des unités externes :

Cette table contient, pour chaque unité de traitement, la liste des unités externes à inclure au moment de la demande d'exécution. Elle a été présentée en 4.1.9.

```
classe table des unités externes ;
TABEXT file structure (
      NUMMATINT entier,
      NUMMATEXT1 entier,
      NUMMATEXT2 entier,
      NØMUNIEXT file (max = 10) caractère,
      NØMAPPEXT file (max = 10) caractère,
      TYPEEXT entier,
      INCLUEXT file file (2) entier) ;
EXTERIEUR de TABEXT ;
PREMIEREXT entier ;
finclasse ;
```

Chaque élément de la table représente une relation externe :

- NUMMATINT est le numéro de matricule de l'unité interne à l'application courante.
- NUMMATEXT1 est le numéro de matricule de l'unité externe, numéro qui lui a été attribué dans l'application courante.
- NUMMATEXT2 est le numéro de matricule de l'unité externe, numéro qui lui a été attribué dans son application origine.
- NØMUNIEXT est le nom de l'unité externe.
- NØMAPPEXT est le nom de l'application origine de l'unité externe.
- TYPEEXT est le type de l'unité externe.
- INCLUEXT permet au système de savoir quelles sont les versions de l'unité externe qui ont été incluses dans chacune des unités de traitement qui l'utilise. Chaque élément de cette liste est formé du couple

composé d'une part du numéro de matricule du module directeur de l'unité de traitement, d'autre part du numéro de version de l'unité qui a été incluse.

Remarque :
D'un point de vue pratique, l'image de cette table est conservée dans une partition d'un fichier partitionné de la classe système.

B.3) La table des identificateurs :

B.3.1) Organisation de la table :

Cette table permet au module initial d'accéder à un objet élémentaire accessible par le module de commande (cf. 4.2.2.).
La table des identificateurs est organisée suivant le principe du hashing : on dispose de 16 classes d'équivalence. Le numéro de la classe d'équivalence à laquelle appartient un identificateur est obtenu à partir de l'identificateur lui-même de la façon suivante :

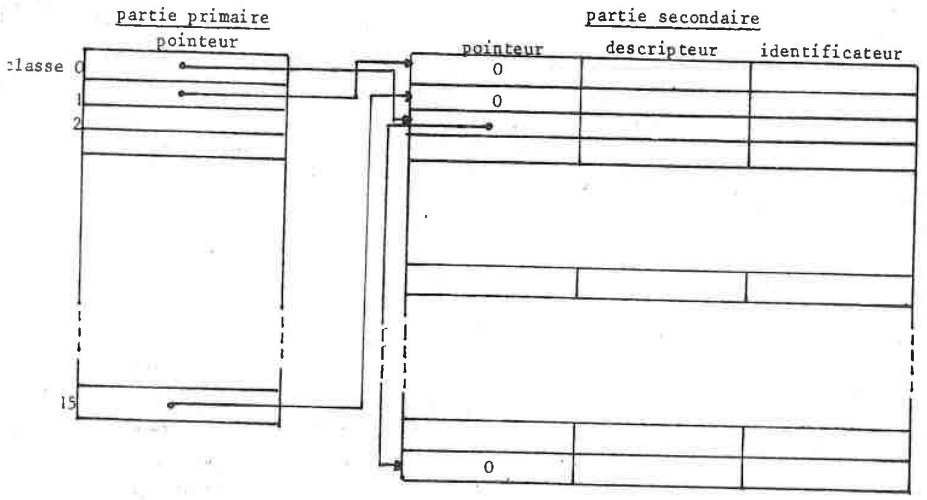
soit x la représentation en hexadécimal de la 1ère lettre de l'identificateur, y la représentation de la 2ème lettre (y = 0 si l'identificateur n'est composé que d'une lettre). Le numéro de la classe d'équivalence est alors le 2ème chiffre de la suite de caractères hexadécimaux obtenus en additionnant 16 * X et y.

Exemple :
soit l'identificateur IDENT.
I représenté en hexadécimal donne 'C9'.
D représenté en hexadécimal donne 'C4'.

```
  C 9 0
+ C 4
-------
  D 5 4
```

L'identificateur IDENT appartient à la classe d'équivalence numéro 5.

La table des identificateurs est composée de 2 parties comme l'indique le schéma suivant :



Chaque classe d'équivalence est représentée par un élément dans la partie primaire. Cet élément contient soit un pointeur qui indique l'adresse du premier identificateur appartenant à la classe d'équivalence, soit 0 si aucun identificateur de la classe d'équivalence n'a encore été déclaré.
Les identificateurs d'une même classe sont chaînés entre eux.
Le deuxième élément de la partie secondaire contient le descripteur de l'objet désigné par l'identificateur. Ce descripteur a une forme variable suivant l'endroit où est déclaré l'identificateur qu'il décrit.

Objet de la classe système

octet	octet	demi-mot
type	0	localisation

Objet de la classe de modification

type	- 3	localisation

Objet de la classe d'application

type 1

type	1 ou 2	localisation

type 2

type	1 ou 2	pointeur

Objet du module de commande

type	- 4	pointeur

Objet de la classe de commande

type	ou 3 4	localisation

Objet d'une unité quelconque de l'application

type	numéro	localisation

. Le type est un nombre entier qui indique le type de l'objet désigné par l'identificateur.

- $-9 \leq$ type ≤ -1 pour les objets de type élémentaire
- type = 0 si l'objet est une variable conditionnée
- type = 1 si c'est une structure de type identification
- type = 2 si c'est une structure de type localisation
- type = 3 si c'est une structure de type spécification
- type = 4 si c'est une structure de type protection
- type = 5 si c'est une structure de type fichier.

. Le numéro permet de savoir dans quelle classe ou dans quel module l'objet élémentaire a été déclaré et si cet objet est une constante ou une variable.
Ce numéro reçoit les valeurs $2n + 1$ si l'objet est une constante
$2n + 2$ si l'objet est une variable.

B.3.2) La classe "table des identificateurs" :

Cette classe déclare la table des identificateurs. Nous n'en donnons pas ici le contenu dans le langage CIVA pour la raison simple que cette table est formée d'octets, de demi-mots et contient des pointeurs. Toutes ces entités ne sont pas définies dans le langage CIVA.

Cette classe contient également un certain nombre de procédures utiles au module initial dont nous pouvons indiquer les principales :

- La procédure "Ajouter élément simple" permet d'ajouter à la table des identificateurs, un identificateur de variable ou de constante de type simple. Les paramètres de cette procédure sont les suivants :
  - une chaîne de caractères contenant l'identificateur
  - un entier TYPE indiquant le type de la variable (ou de la constante)
  - un entier LIEU qui indique le numéro de la classe ou du module dans lequel est déclaré l'objet.

  Cette procédure positionne un booléen AJØUTE à vrai si l'adjonction de l'identificateur à la table est réalisée, à faux sinon.

- La procédure "Ajouter file" permet d'ajouter l'identificateur d'une variable de type file à la table des identificateurs. Les paramètres de cette procédure sont analogues à ceux de la procédure précédente.

- La procédure "Ajouter structure" permet d'ajouter l'identificateur d'une variable de type structure.

- La procédure "Supprimer identificateur" permet de supprimer un identificateur de la table et par là-même de détruire l'objet désigné. Le résultat de cette procédure est le positionnement d'un booléen SUPPRIME à la valeur vrai si l'identificateur est supprimé, à faux sinon.

- La procédure "Rechercher identificateur" permet de rechercher un identificateur dans la table. Si l'identificateur est absent, le booléen IDENTIFICATEUR PRESENT reçoit la valeur faux. Sinon, le booléen IDENTIFICATEUR PRESENT reçoit la valeur vrai et ADRESSE contient d'adresse de l'emplacement affecté à l'objet désigné par l'identificateur.

ANNEXE C : Les procédures de la classe système

Les procédures déclarées dans cette classe ne font l'objet d'aucune protection de la part du système : tout module de commande peut les appeler (cf. 2.3.).

C.1) La procédure APPLICATIØN :

L'appel de cette procédure est la première instruction d'un module de commande. Il a pour rôle d'initialiser la classe de l'application dans laquelle travaille l'utilisateur (cf. 2.4.1.).

C.1.1.) Le texte de la procédure :

```
procédure APPLICATIØN ;
PASSAPPL file (max = 8) car ;
APPLICATIØN PRESENTE booléen ;
INDAPPL entier ;
TAILLE entier ;
APPLICATIØN CØURANTE = AF (1) ;
recherche de l'application (APPLICATIØN CØURANTE, APPLICATIØN PRESENTE),
décision
conditions
```

					ELSE
NUM (AF) = 1	V	-	-	-	
NUM (AF) = 2	-	V	V	-	
NUM (AF) = 3	-	-	-	V	
AF (2) = 'NØUVELLE'	-	-	V	V	
AF (2, 1) = 'PASSE'	-	V	-	-	
AF (3, 1) = 'PASSE'	-	-	-	V	
APPLICATIØN PRESENTE	V	V	F	F	
MØTPASSE ≠ ' ' ;	F	V	-	-	
MØTPASSE = AF (2, 2)	-	V	-	-	
A1	*	*			
A2			*	*	
A3					*

fin table ;

CØ l'action A1 correspond à la situation où l'utilisateur désire travailler dans une application qui existe déjà dans le système CIVA. Il faut donc réaliser les opérations d'initialisation de la classe d'application CØ ;

```
A1 : déterminerles noms des fichiers système de l'application ;
     assignation des fichiers système de l'application ;

     si INDICATIF (5) alors
          mettre le fichier dictionnaire en zone dynamique locale ;
     sinon NBDICØ = 0 ; fsi ;
     si INDICATIF (1) alors
          créer la matrice M1 en zone dynamique locale ;
     sinon M1 (*, *) = 0 ; fsi ;
     si INDICATIF (8) alors
          ajouter les identificateurs des objets élémentaires à la table ;
          TAILLE = taille de la classe d'application ;
          demander de la place en zone dynamique commune (DEBAPPL, TAILLE) ;
          initialiser les objets de la classe d'application ; fsi ;
     si INDICATIF (9) alors
          créer la table des unités externes en zone dynamique locale ;
          fsi ;
     sortir ;
```

CØ l'action A2 correspond à la situation où l'utilisateur désire créer une nouvelle application dans le système CIVA. Dans ce cas, la procédure doit créer un élément dans la table des applications CØ ;

```
A2 : si NUM (AF) = 2 alors PASSAPPL = '  ' ;
                     sinon PASSAPPL = AF (3, 2) ; fsi ;
     ajouter l'application (APPLICATIØN CØURANTE, PASSAPPL) ;
     déterminer les noms des fichiers système de l'application ;
     assignation des fichiers système de l'application ;
     NBDICØ = 0 ;
     M1 (*, *) = 0 ;
     imprimer ('application', APPLICATIØN CØURANTE, 'créée') ;
     sortir ;
```


CØ enfin l'action A3 correspond à une erreur de la part de l'utilisateur dans l'appel de la procédure. Dans ce cas, la procédure envoie un message d'erreur à l'utilisateur. Comme cette erreur porte sur la détermination de l'application dans laquelle l'utilisateur veut travailler, le module initial ne peut continuer son travail. Il arrête alors l'interprétation CØ ;

```
A3 : imprimer ('erreur dans l'appel de la procédure application') ;
     lire les lignes jusqu'à la fin du module de commande ;

     finproc ;
```

C.1.2.) Détermination des noms des fichiers système de l'application :

Chaque application connue du système CIVA possède ses propres fichiers qui assurent la gestion des unités de l'application. La première action que doit mener le système, une fois le nom de l'application connu, est la détermination des noms de ces fichiers.

```
Classe classe des fichiers ;
identificateur du fichier descriptif file (max = 16) car ;
identificateur du fichier source file     (max = 16) car ;
identificateur du fichier compilé file    (max = 16) car ;
identificateur du fichier édité  file     (max = 16) car ;
identificateur du fichier dictionnaire file (max = 16) car ;
identificateur du fichier catalogue file (max = 16) car ;
identificateur du fichier des sauvegardes file (max = 16) car ;

finclasse ;

module déterminer les noms des fichiers système de l'application ;
utilise classe des fichiers ;

identificateur du fichier descriptif = application courante .'-DESC' ;
      "             "    source      =          "          .'-SØUR' ;
      "             "    compilé     =          "          .'-CØMP' ;
      "             "    édité       =          "          .'-SAUV' ;
      "             "    dictionnaire =         "          .'-DICØ' ;
      "             "    catalogue   =          "          .'-CATA' ;
      "             "    des sauvegardes =      "          .'-SAUV' ;
finmod ;
```

C.1.3.) Assignation des fichiers système de l'application :

Une fois les noms des fichiers système déterminés, l'interprète des modules de commande doit indiquer en plus au système d'exploitation de l'ordinateur le type d'utilisation de ces fichiers, à savoir création, lecture ou mise à jour du fichier

```
module assignation des fichiers système de l'application ;
    si INDICATIF (1) alors assigner fichier descriptif en mise à jour ;
                     sinon assigner fichier descriptif en création ;
                                                              fsi ;
finmod ;
```

Sous le système SIRIS7, les renseignements concernant un fichier sont groupés dans une table du programme de l'utilisateur : la table DBC (Data Control Block). Normalement tous ces renseignements sont fournis par l'utilisateur avant l'exécution du programme par l'emploi d'une carte de commande ! ASSIGN (cf. langage de commande sous SIRIS7/SIRIS8 [22]).

Toutefois, il est possible à l'utilisateur de préciser certains paramètres d'une façon dynamique, pendant l'exécution du programme. Pour cela, il utilise la procédure système M : ASSIGN (cf. procédures système sous SIRIS7 [44]). Ces renseignements sont alors mis dans la table DCB par le moniteur. La procédure M : ASSIGN permet donc d'établir dynamiquement la liaison entre l'étiquette logique indiquée par l'utilisateur dans une carte ! ASSIGN qui repère la table DCB et une définition de fichier réel. C'est cette possibilité qui est utilisée par le module initial. La phrase CIVA "assigner fichier..." se traduit en METASYMBØL par un appel de la procédure M : ASSIGN.

C.2) Analyse de la procédure CREER :

La procédure CREER sert à créer des unités dans l'application ou à créer des déclarations d'identificateurs dans la classe de l'application (cf. 2.4.2.). L'interprète doit avant tout reconnaître quel est le type d'appel à la procédure CREER qui est demandé par l'utilisateur.
Si aucun paramètre ne figure, il s'agit de la création d'unités protégées. Si le paramètre CØMMUNS est spécifié, il s'agit de la création d'unités non protégées. Si une liste de paramètres est donnée, alors la procédure analyse les séparateurs et le contenu du premier élément de la liste. En effet, si l'utilisateur déclare des identificateurs dans la classe d'application, les séparateurs sont des points virgules et chaque élément de la liste est formé d'au moins 2 mots CIVA (le nom de l'identificateur suivi du type de la variable qu'il désigne). En cas de non-conformité, la procédure émet un message d'erreur.

```
procédure CREER ;
si NUM (AF) = 0 alors protection = - 1 ;
                      lire la ligne suivante ;
                      création d'unités ;
sinon si AF (1, 1) = 'CØMMUNS' alors protection = 0 ;
                                    lire la ligne suivante ;
                                    création d'unités ;
     sinon si AF (1,1) ne contient qu'un seul mot et le séparateur est une
              virgule
          alors protection = NUM (AF(1)) ;
          pour I de 1 à protection faire
          APPL (I) = AF (1, I) ; fp ;
          lire la ligne suivante ;
          création d'unités ;
          sinon création d'identificateurs ; fsi ;
          fsi ;
     fsi ;
finproc ;
```

C.2.1.) La création d'unité dans l'application :

C.2.1.1.) La syntaxe des unités

```
⟨ module ⟩ : : = module ⟨ nom du module ⟩ ;
                ⟨ corps du module ⟩  finmod ;
```

Le corps du module est une suite de déclarations et d'instructions séparés par des points virgules.

< classe > :: = classe < nom de la classe > ;
< corps de la classe > finclasse ;

Le corps de la classe est une suite de déclarations séparées par des points virgules.

< métamodule > :: = ⧫MØD < nom du métamodule >
[(< liste des paramètres formels >)] ;
< corps du métamodule > ⧫FINMØD ;

Le corps du métamodule est une suite d'instructions ou de métainstructions séparées par des points virgules. Le nom du métamodule commence par le caractère '⧫'.

< classe de commande > :: = CLASSECØM < nom de la classe > ;
< corps de la classe > FINCLASSE ;

Les identificateurs déclarés dans la classe de commande sont utilisables dans un module de commande ou dans un module de traitement (cf. la procédure UTILISE en 1.4. et la table des identificateurs en 4.2.2.). Pour cette raison, la classe de commande fait l'objet d'une déclaration particulière.

C.2.1.2.) Le module de création

Si l'utilisateur a écrit un appel de création d'unités, pour chaque unité x à créer, la procédure doit :

- vérifier que l'unité x n'existe pas déjà dans l'application, c'est-à-dire qu'aucune unité de même nom ne figure déjà dans le fichier dictionnaire avec l'indicatif de création positionné à vrai.

- demander un numéro de matricule pour l'unité x si celle-ci n'en a pas déjà un. En effet, il se peut très bien que l'unité x soit déjà présente dans l'application avec un numéro de matricule sans pour cela avoir été créée. Il suffit que, lors du traitement de la création d'une unité y, cette unité y appelle ou utilise l'unité x. Dans ce cas, l'enregistrement du fichier descriptif représentant y contient les numéros de matricules des unités appelées ou utilisées. Il a donc fallu que l'interprète donne à x un numéro de matricule et fasse entrer x dans le dictionnaire.

- compresser le texte source et le ranger dans une partition du fichier des textes sources.

- enfin, créer un enregistrement dans le fichier descriptif pour représenter l'unité dans l'application.

```
module création d'unités ;
TYPE = 0 ;
sélection
conditions
```

premier mot de la ligne =	'MØDULE'	'CLASSE'	'⧫MØD'	'CLASSECØM'
actions TYPE =	1	2	4	8

```
        fintable ;
si TYPE = 0 alors imprimer ('TYPE UNITE INCØNNU') ;
                  lire le texte source jusqu'à finmod, finclasse ou ⧫finmod ;
sinon XCRENØM = nom de l'unité ; BØØL = vrai ;
ajouter à dictionnaire (XCRENØM,,TYPE,BØØL) ;
```

CØ l'appel de cette procédure permet d'ajouter l'unité au fichier dictionnaire CØ ;

```
si non AJØUTE alors imprimer ('UNITE DEJA EXISTANTE DØNC IGNØREE') ;
            lire le texte source jusqu'à finmod, finclasse ou ⧫finmod ;
sinon compresser le texte source ;
      soit APPLICATIØN telque NØM = APPLICATIØN CØURANTE ;
      NBUNIT = NBUNIT + 1 ;
      si PRØTECTIØN = - 1 alors NBPRØT = NBPRØT + 1 ; fsi ;
      si INDICATIF (1) alors utilisation (DESC, mise à jour) ;
                              dernier DESCRIPTIF ;
                              DESCRIPTIF en DESCRIPTIF + 1 ;
                        sinon utilisation (DESC, écriture) ;
                              premier DESCRIPTIF ;
                              INDICATIF (1) = vrai ; fsi ;
```

CØ si le fichier descriptif a déjà été créé, le module de création l'utilise en rallongement. Sinon, le module créé le premier enregistrement dans le fichier et il positionne l'indicatif de création CØ ;

```
PRØTDESC = PRØTECTIØN ; NØMDESC = XCRENØM ; MATDESC = CLE ;
TYPEDESC = TYPE ; ETAT DESC = 'source' ; APPLDESC = APPL ;
            traiter les unités externes ;
            traiter les métamodules appelés ;
            traiter les classes utilisées ;
            traiter les modules ou procédures appelés ;
            créer l'unité dans la matrice MI ;
fsi ;
fsi ; lire la ligne suivante ;
si premier mot de la ligne = 'MØDULE' ou CLASSE' ou $MØD' ou 'CLASSECØM'
alors création d'unités ;
finmod ;
```

C.2.1.3.) La compression du texte source

Le module "compresser le texte source" a pour but :

a) de lire les enregistrements contenant le texte source d'une unité en cours de création. La fin du texte source d'une unité est matérialisée par les déclarations :

FINMØD ; pour un module
FINCLASSE ; pour une classe
et $FINMØD ; pour un métamodule.

b) de compresser le texte source et de ranger la chaîne de caractères ainsi obtenue dans une partition du fichier système contenant les textes des unités de l'application.

c) de relever dans le texte source les déclarations de métamodules ou de procédures internes à l'unité créée si celle-ci est une classe. Pour chacune d'elles, il crée un enregistrement dans le fichier dictionnaire.

d) de relever également les appels de métamodules, les utilisations de classes et les appels de modules ou de procédures sans paramètre.

. L'appel d'un métamodule est écrit de la façon suivante :

$ < nom du métamodule > [(< liste des paramètres >)] ;

. L'utilisation d'une classe est écrit :

UTILISE < liste des classes utilisées > ;

. Enfin, l'appel d'un module est une instruction CIVA qui n'est composée que d'un seul mot : le nom du module appelé. L'inconvénient est que l'appel d'une procédure sans paramètre est également une instruction CIVA composée d'un seul mot : le nom de la procédure. Donc, à ce niveau, l'interprète ne cherche pas à savoir si, dans le texte source, il rencontre un appel de module ou un appel de procédure. C'est le compilateur qui pourra faire la différence plus tard, connaissant tout l'environnement des unités.

e) de fournir trois listes au module de création d'unités :

La première liste LISTMETA contient les noms des métamodules appelés, internes à l'application.
La deuxième liste LISTCLAS contient les noms des classes utilisées, internes à l'application.
La troisième liste LISTMØDPRØC contient les noms des modules ou des procédures sans paramètre appelées.

f) les unités appelées ou utilisées externes à l'application sont traitées d'une façon différente. Les renseignements concernant chaque unité externe sont rangés dans la table des unités externes à l'application.
A ce niveau, l'interprète ne fait que relever les "références" à ces unités et il leur donne un numéro de matricule, ceci afin de compléter l'enregistrement du fichier descriptif de l'unité en cours de création. Afin de différencier dans le fichier dictionnaire les unités internes des unités externes, ces dernières auront des numéros de matricule supérieurs ou égaux à 30.000.

Toutes les unités externes à l'application ne sont ajoutées à celle-ci que d'une façon temporaire, lorsque l'utilisateur demande le lancement de l'exécution d'une unité de traitement. Il est préférable de traiter globalement toutes les unités externes demandées en une seule fois : ceci permet un gain de temps appréciable pour les contrôles de protection dans les autres applications. Le module de compression fournit une liste LISTEXT contenant les noms des unités externes employées par l'unité en cours de création.

exemple de fonctionnement du module de compression :

soit à créer l'unité suivante :

```
classe C ;
utilise A, B ;
utilise X de PERSØNNEL ;
procédure PRØC
    ¦
fin proc ;
$mod $A
    ¦
$finmod ;
fin classe ;
```

Le résultat du module de compression lorsqu'il traite cette unité est le suivant :

```
LISTMETA = vide ;
LISTCLAS = (A, B) ;
LISTMØDPRØC = vide ;
LISTEXT = ((X, PERSØNNEL, CLASSE)) ;
```

Le contenu du fichier dictionnaire sera le suivant :

Nom de l'unité	Nom de la classe	Numéro de matricule	Type	Indicatif de création
C		1	classe	vrai
A		2	classe	faux
B		3	classe	faux
X		30.000	classe	faux
PROC	C	4	proc.	vrai
$A	C	5	métamod	vrai

C.2.1.4.) Le traitement des unités appelées ou utilisées

Nous venons de voir que le module de compression de texte relevait les noms des unités appelées ou utilisées par l'unité de traitement. Celles-ci doivent être incluses au fichier dictionnaire.

- Les unités externes à l'application :

Les liaisons entre l'unité en cours de création et les unités externes de l'application sont conservées dans la table des unités externes.

Pour chaque unité externe appelée ou utilisée, le module de création doit ajouter un élément dans cette table. Il repère également l'indice de la première unité externe ajoutée : cet indice est rangé dans PREMIEREXT afin de pouvoir ajouter les numéros de matricule correspondant dans l'enregistrement du fichier descriptif.

```
classe unité externe ;
LISTEXT file structure (
          LIST1 file (max = 10) car,
          LIST2 file (max = 10) car,
          LIST3 code (module, classe,, métamodule)) ;
finclasse ;
module traiter les unités externes ;
utilise table des unités externes, unité externe ;
si taille actuelle (LISTEXT) = 0 alors sortir ; fsi ;
PREMIEREXT = taille actuelle (TABEXT) + 1 ;
```

CØ le module conserve l'indice du premier élément ajouté à la table CØ ;

```
dernier EXTERIEUR ;
pour chaque élément de LISTEXT faire
        EXTERIEUR en EXTERIEUR + 1 ;
        NUMMATINT = CLE ;
```

CØ on range le numéro de l'unité en cours de création CØ ;

```
        NUMMATEXT1 = 30.000 + taille actuelle (TABEXT) ;
```

CØ on range le numéro de matricule de l'unité externe CØ ;

```
        NØMUNIEXT = LIST1 ;
```

CØ LIST1 contient le nom de l'unité externe CØ ;
NØMAPPEXT = LIST2 ;

CØ LIST2 contient le nom de l'application origine CØ ;
TYPEEXT = LIST3 ;

CØ LIST3 contient le nom de l'application origine CØ ;
ajouter externe à dictionnaire (NØMUNIEXT,,TYPEEXT,faux,
NUMMATEXT1) ;

fpc ;
LISTEXT = vide ;
finmod ;

- Les unités internes à l'application :

Le module de compression fournit 3 listes contenant les noms des modules (ou procédures), des classes et des métamodules employés dans l'unité en cours de création.
Le fichier descriptif contenant, pour chaque unité créée, la liste des numéros de matricule des unités employées, internes ou externes, celles-ci doivent donc figurer dans le fichier dictionnaire.

module traiter les métamodules appelés ;
utilise liste métamodules internes, table des unités externes ;
TYPE entier = 4 ; BOOL booléen = faux ; I entier ;
NBMETA = taille actuelle (LISTMETA) ;
si NBMETA ≠ 0 alors I = 0 ;
pour chaque ELEMENT de LISTMETA faire I = I + 1 ;
ajouter à dictionnaire (ELEMENT,, TYPE, BØØL) ;
METADESC (I) = CLE ; fpc ; fsi ;

CØ cette première partie a pour rôle de donner à chaque métamodule appelé un numéro de matricule et de ranger ce numéro dans l'enregistrement du fichier descriptif CØ ;

pour chaque EXTERIEUR de TABEXT telque EXTERIEUR > PREMIEREXT faire
si TYPEEXT = 4 alors
NBMETA = NBMETA + 1 ;
METADESC (NBMETA) = NUMMATEXT1 ; fsi ;

fpc ;

CØ cette 2ème partie a pour rôle d'ajouter les numéros de matricule des métamodules externes, appelés par l'unité en cours de création, à l'enregistrement du fichier descriptif CØ ;

finmod ;

Les textes des modules "traiter les classes utilisées" et "traiter les modules ou procédures appelés" sont analogues à celui du module qui vient d'être décrit.

C.2.2.) La création d'objets élémentaires dans la classe d'application :

module création d'identificateurs ;
utilise table des symboles ;
UNITE entier = 3 ;

CØ ce paramètre indique que les identificateurs créés appartiennent à la classe d'application CØ ;
CREATIØN IDENTIFICATEUR = vrai ;

CØ ce paramètre indique au système que des identificateurs sont créés dans la classe d'application. Comme nous l'avons vu, ces identificateurs vont recevoir des descripteurs de type 1 et le système sera donc amené à réorganiser la classe d'application CØ ;

pour chaque déclaration de liste de déclarations faire
NØM = premier mot de déclaration ;
recherche dans la table des symboles (NØM) ;
si identificateur présent alors imprimer ('identificateur présent dans la table') ; sortir ; fsi ;
TYPE = deuxième mot de déclaration ;
si TYPE = 'type' alors NØMTYPE = troisième mot de déclaration ;
recherche dans la table des symboles (NØMTYPE) ;
si non identificateur présent alors
imprimer ('type inconnu') ; sortir ; fsi ;
TYPE = type de NØMTYPE ; fsi ;

si TYPE = 'entier' ou 'réel' ou 'décimal' ou 'booléen' ou 'reeldp' ou
'complexe' ou 'complexedp' alors
ajouter élément simple à la table des symboles (NØM, TYPE, UNITE) ;
sinon si TYPE = 'file' alors adjonction d'une file ;
sinon adjonction d'une structure ; fsi ;
fsi ; fpc ;
finmod ;

Le module "adjonction d'une file" permet d'ajouter le nom de la file à la table des symboles et de réserver de la place en zone dynamique commune. Le module "adjonction d'une structure" est analogue au module "création d'identificateurs", une structure étant considérée comme une suite de déclaration d'objets de type élémentaire ou structuré.

C.3) La procédure AJØUTER.

La première action que mène la procédure AJØUTER (cf. 2.4.3.) est la vérification de l'appel. Ensuite, elle vérifie que l'application à laquelle appartient l'unité externe à ajouter existe bien dans le système et qu'elle possède des unités non protégées.

procédure AJØUTER ;
XAJØNØM1 = AF (1)
XAJØAPPL = AF (2)
recherche de l'application (XAJØAPPL) ;
décisions
conditions

	APPLICATIØN PRESENTE	F	V	V	V	V	V
	NBPRØT = NBUNIT	-	V	F	F	F	F
	NUM (AF) = 4 (l'utilisateur demande un changement de nom)	-	-	V	V	F	F
	option ENVIRØNNEMENT présente	-	-	V	F	V	F
actions	imprimer ('nom d'application inconnue') ;	1					
	imprimer ('unité protégée') ;		1				
	sortir ;	2	2				
	ALE =			vrai;	vrai;	faux;	faux;
	ENV =			vrai;	faux;	vrai;	faux;
	XAJØNØM2 = AF (4) ;			1	1		
	recherche dans dictionnaire (XAJØNØM1, CLE) ;					1	1
	recherche dans dictionnaire (XAJØNØM2, CLE) ;			2	2		

fintable ;

si UNITE CREEE alors
imprimer ('unité déjà créée dans l'application courante') ;
sortir ; fsi ;
déterminer les noms des fichiers système de l'autre application ; assignation des fichiers système de l'autre application ; ajouter la première unité à l'application ;

CØ l'appel de ce module permet d'ajouter l'unité demandée à l'application courante. Si l'adjonction est faite, le booléen ADJONCTION est positionné à vrai, sinon il est positionné à faux. De plus, ce module positionne le booléen EMPLØI à vrai si l'unité ajoutée utilise ou appelle d'autres unités de son application, à faux sinon CØ ;

décision
conditions

	ADJØNCTIØN	F	V	V	V	V
	ENV	-	F	F	V	V
	EMPLØI	-	F	V	F	V
actions	sortir ;	x	x		x	
	A1			x		
	A2					x

fintable ;

CØ l'action A1 correspond au cas où l'utilisateur ajoute une unité externe qui possède un certain environnement et ne demande pas l'adjonction de cet environnement. La procédure prévient alors l'utilisateur CØ ;

```
A1 : imprimer ('attention = l'unité ajoutée emploie d'autres unités') ;
     pour chaque unité employée faire
              ajouter à dictionnaire (UNITE EMPLØYEE,, TYPE, faux) ; fpc ;
     sortir ;
```

CØ l'action A2 correspond au cas où l'unité ajoutée possède un certain environnement qui doit être ajouté aussi à l'application courante. La procédure, après avoir initialisé les opérations qui traitent l'environnement, lance l'adjonction de chaque unité CØ ;

```
A2 : initialiser l'environnement ;
     pour chaque unité de LISTAJOU tantque AJØUT ≠ FLIST
     faire ajouter une unité externe à l'application ;
     fpc ;
finproc ;
```


C.3.1.) L'adjonction de la première unité à l'application :

Pour réaliser l'adjonction de la première unité, le module initial doit tout d'abord vérifier que cette unité est bien créée dans son application origine et que l'autorisation préalable a été accordée par le propriétaire de cette autre application. Si ce n'est pas le cas, l'adjonction ne peut avoir lieu.

L'opération d'adjonction consiste tout simplement :

- à donner à l'unité ajoutée un nouveau numéro de matricule
- à ajouter le nom de l'unité et son matricule dans le fichier dictionnaire de l'application courante.
- à ajouter les partitions des fichiers contenant les textes source, compilé et édité (pour la dernière, seulement si elle existe).
- enfin à créer un enregistrement représentant cette unité dans le fichier descriptif.

```
module ajouter la première unité à l'application ;
recherche dans le dictionnaire de l'autre application (XAJØNØM1, CLE2) ;
```

CØ l'appel de cette procédure permet de connaître le numéro de matricule CLE2 correspondant à l'unité dans son application origine CØ ;

```
si non UNITE CREEE alors imprimer ('unité absente') ;
                         ADJØNCTIØN = faux ;
                         sortir ; fsi ;
utilisation (DESCRIPTIF - AUTRE - APPLICATIØN, lecture) ;
```

CØ la description du fichier descriptif de l'autre application est analogue à celle du fichier de l'application courante. Afin de faciliter la clarté de cet exposé, tous les identificateurs du fichier descriptif de l'autre application se termine par un 2 CØ ;

```
soit DESCRIPTIF2 telque MATDESC2 = CLE2 ;
vérifier la protection ;
```

CØ l'appel de ce module permet de vérifier la protection de l'unité. Si l'unité est protégée, le booléen AUTORISATION est positionné à faux, sinon à vrai CØ ;

si non AUTØRISATIØN alors imprimer ('l'unité', XAJØNØM1, 'est protégée dans
son application') ;
ADJØNCTIØN = faux ; sortir ; fsi ;
ADJØNCTIØN = vrai ;

CØ l'unité à ajouter doit recevoir un nouveau numéro de matricule dans l'application courante CØ ;

BØØL booléen = vrai ;
si ALE alors ajouter à dictionnaire (XAJØNØM2,, TYPEDESC2, BØØL) ;
sinon ajouter à dictionnaire (XAJØNØM1,, TYPEDESC2, BØØL) ;
fsi ;

CØ cette opération permet de ranger le nouveau nom de l'unité et son nouveau numéro de matricule dans le dictionnaire CØ ;

décision
conditions

actions

ETATDESC2 = 'édité'	V	-	-
ETATDESC2 = 'compilé'	-	V	-
ETATDESC2 = 'source'	-	-	V
recopier la partition (CLE2) du fichier des textes édités de l'autre application dans la partition (CLE) du fichier de l'application courante ;	*		
" " compilés " "	*	*	
" " source " "	*	*	*

fintable ;
utilisation (DESC, mise à jour) ;
dernier DESCRIPTIF ;

DESCRIPTIF en DESCRIPTIF + 1 ;
NØMDESC = si ALE alors XAJØNØM2 sinon XAJØNØM1 fsi ;
MATDESC = CLE ;
TYPEDESC = TYPEDESC2 ;
ETATDESC = ETATDESC2 ;
CØNSTDESC = CØNSTDESC2 ;
VARDESC = VARDESC2 ;
MØDDESC = MØDESC2 ;
NBMETA = NBMETA2 ;
pour I de 1 à NBMETA faire ;
METADESC (I) = PGMAT + I ; fp ;
PGMAT = PGMAT + NBMETA ;
NBCLAS = NBCLAS2 ;
pour I de 1 à NBCLAS faire
CLASDESC (I) = PGMAT + I ; fp ;
PGMAT = PGMAT + NBCLAS ;
NBMØD = NBMØD2 ;
pour I de 1 à NBMØD faire
MØDDESC (I) = PGMAT + I ; fp ;
PGMAT = PGMAT + NBMØD ;

CØ après avoir créé l'enregistrement dans le fichier descriptif, le module positionne le booléen EMPLOI à vrai si l'unité ajoutée utilise d'autres unités CØ ;

EMPLØI = (NBMETA ≠ 0) ou (NBCLAS ≠ 0) ou (NBMOD ≠ 0) ;
finmod ;

C.3.2.) Problème posé par l'adjonction des unités de l'environnement :

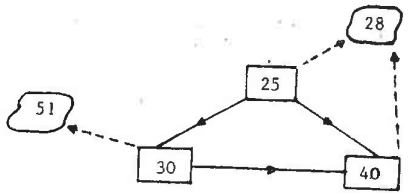
Si l'unité ajoutée à l'application courante possède un certain environnement, et si l'utilisateur demande que cet environnement soit également incorporé à l'application, le module initial doit lancer et contrôler les opérations d'adjonction pour toutes les unités de cet environnement. Il est donc amené à parcourir le graphe ayant pour points, toutes ces unités et pour arcs, les relations qui lient ces unités dans leur application origine.

L'exploration du graphe des relations.

La détermination de l'algorithme d'exploration du graphe est subordonnée à deux contraintes :

- Lorsque le module qui traite l'environnement parcourt le graphe, il peut arriver à un point qu'il a déjà traité au cours de son exploration. Dans ce cas, il ne doit pas réaliser une deuxième fois l'adjonction de l'unité à laquelle il vient d'accéder. Le module est donc obligé de conserver en mémoire la liste des unités qu'il a déjà incorporé à l'application.

Exemple :



Le module ajoute donc l'unité 25 à l'application courante A et détermine l'environnement immédiat de 25 qui est composé de 30, 40 et 28. Ensuite, il ajoute l'unité 30 et détermine l'environnement de cette unité qui est composé de 40 et 51. Il doit donc ajouter 40 et 51. Lorsqu'il revient à l'environnement de l'unité 25, il ne doit pas ajouter de nouveau l'unité 40 qui vient d'être incorporée.

- Lorsque le module qui traite l'environnement ajoute une unité à l'application courante, il doit donner à cette unité un nouveau numéro de matricule qui la définit de façon unique dans sa nouvelle application. De plus, il doit créer un enregistrement décrivant cette unité dans le fichier descriptif de l'application. Or, nous avons vu (en 4.1.4.) que cet enregistrement contenait les numéros de matricule des unités qui sont soit utilisées soit appelées par l'unité ajoutée. Il faut donc qu'à chacune de ces unités soit affecté un nouveau numéro de matricule dans l'application courante. Ces nouveaux numéros sont donnés par ordre croissant. Ensuite, le module doit ajouter chacune de ces unités à la nouvelle application mais cette adjonction doit se faire dans un ordre

bien établi. En effet, pour chaque unité ajoutée est créé un enregistrement dans le fichier descriptif. Ce fichier ayant une organisation séquentielle indexée, les enregistrements doivent être ajoutés dans l'ordre croissant des clés, c'est-à-dire des numéros de matricule.

L'algorithme d'exploration du graphe.

Pour les deux raisons que nous venons de citer, nous avons utilisé une liste particulière, la liste LISTAJØU, pour explorer le graphe. Cette liste contient les renseignements nécessaires concernant les unités qui ont déjà été ajoutées dans l'application et les renseignements concernant les unités qui doivent encore être ajoutées. Le principe de gestion de cette liste est le suivant :

- chaque unité qui doit être ajoutée à l'application courante est représentée par un élément de la liste contenant deux renseignements :
  . son ancien numéro de matricule dans son application origine ;
  . son nouveau numéro de matricule dans l'application courante.

- la liste possède deux éléments courants :
  . l'indice AJØUT permet de connaître, à un instant donné, quelle est l'unité qui est en cours d'adjonction dans l'application courante.
  . l'indice FLIST qui indique le premier élément libre de la liste.

- les unités sont ajoutées à l'application courante dans l'ordre où elles figurent dans la liste. L'adjonction d'une unité v de l'environnement consiste à :
  . ranger, à partir de LISTAJØU (FLIST), les numéros de matricule de chaque unité appelée ou utilisée par v, à condition que ces numéros ne figurent pas déjà dans la liste ;
  . ajouter l'unité v à l'application ;
  . ajouter 1 à l'indice AJØUT.

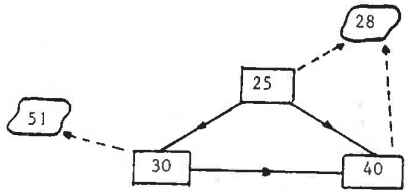
- cette liste est initialisée au départ de la façon suivante :
  . on range les numéros de matricule des unités employées par l'unité désignée dans l'appel de la procédure AJØUTER.
  . on initialise les éléments courants AJØUT et FLIST.

- la fin du traitement de l'environnement a lieu lorsque toutes les unités figurant dans la liste ont été ajoutées, c'est-à-dire lorsque

AJØUT = FLIST

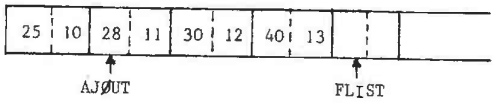
Exemple de fonctionnement :

Reprenons l'exemple cité précédemment :



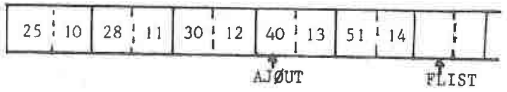
initialisation de la liste

Supposons que le premier matricule disponible dans l'application courante soit le numéro 10. L'opération d'initialisation permet de garnir le début de la liste.

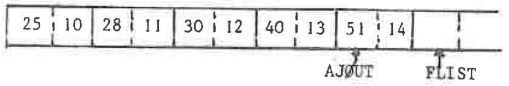


adjonction de l'unité 28

A la fin de l'opération, la liste aura la forme suivante :

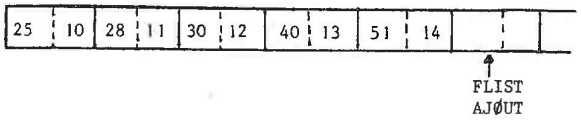


adjonction de l'unité 40



L'unité 28 qui est utilisée par l'unité 40 n'est pas ajoutée à la liste car son numéro y figure déjà.

adjonction de l'unité 51



C'est la fin du traitement : toutes les unités de l'environnement ont été ajoutées à l'application courante.

C.3.3.) L'adjonction des unités de l'environnement :

```
module initialiser l'environnement ;
   utilise définition de LISTAJØU ;
       LISTAJØU (1, 1) = CLE2 ;
       LISTAJØU (1, 2) = CLE ;
```

CØ rangement de l'ancien et du nouveau numéro de matricule de l'unité désignée dans la procédure AJØUTER CØ ;

```
AJØUT = 2 ;
FLIST = 2 ;
pour I de 1 à NBMETA faire
       LISTAJØU (FLIST, 1) = METADESC2 (I) ;
       LISTAJØU (FLIST, 2) = METADESC  (I) ;
       FLIST = FLIST + 1 ; fp ;
pour I de 1 à NBCLAS faire
       LISTAJØU (FLIST, 1) = CLASDESC2 (I) ;
       LISTAJØU (FLIST, 2) = CLASDESC  (I) ;
       FLIST = FLIST + 1 ; fp ;
pour I de 1 à NBMOD faire
       LISTAJØU (FLIST, 1) = MØDDESC2  (I) ;
       LISTAJØU (FLIST, 2) = MØDDESC   (I) ;
       FLIST = FLIST + 1 ; fp ;
finmod ;
```

Le texte du module "ajouter une unité extérieure à l'application" est analogue au module précédemment décrit "ajouter la première unité à l'application". Nous avons seulement inséré dans ce module les instructions permettant de gérer la liste LISTAJØU.

C.4) La procédure LISTER.

Cette procédure permet à l'utilisateur d'obtenir certains renseignements sur son application (cf. 2,5.3.2.1.), c'est-à-dire sur :

- les unités déclarées dans l'application
- les objets élémentaires ou structurés de la classe d'application
- les fichiers permanents de l'application.

```
procédure LISTER ;
ØPTIØN file (10) booléen ;
ØPTIØN (*) = faux ;
    sélection
    conditions
```

						ELSE
	NUM (AF) =	0	1	2	2	
	AF(2) = 'CLASSE APPLICATIØN'	-	-	F	V	
Actions	A1	*				
	A2		1	1		
	A3		2			
	A4			2		
	A5				*	
	imprimer ('ERREUR DANS L'APPEL') ; sortir ;					*

```
fintable ;
```

CØ l'action A1 correspond au cas où l'utilisateur ne veut connaître que la liste des noms des différentes unités de l'application CØ ;

```
A1 : pour chaque DICTIØNNAIRE de DICØ telque INDIC = vrai
        faire imprimer (NØMDIC) ; fpc ;
     sortir ;
```


CØ l'action A2 correspond au cas où l'utilisateur désire avoir des renseignements sur des unités de son application. La procédure doit alors reconnaître les options indiquées par l'utilisateur dans l'appel CØ

```
A2 : pour I de 1 à NUM (AF(1)) faire
     sélection
```

AF (1,I)=	'type'	'état'	'erreur'	'constante'	'variable'
	option(1)= vrai ;	option(2)= vrai ;	option (3)= vrai ;	option(4) = vrai ;	option(5) = vrai ;

'protection'	'unité'	'métamod'	'module'	'classe'	sinon
option(10)= vrai ;	option(6)= vrai ;	option(7)= vrai ;	option (8)= vrai ;	option(9) = vrai ;	imprimer 'option", I, 'fausse') ;

```
fintable ; fp ;
```

CØ l'action A3 correspond au cas où l'utilisateur veut connaître les renseignements demandés pour toutes les unités de l'application CØ ;

```
A3 : utilisation (DESC, lecture) ;
     pour chaque DESCRIPTIF de DESC faire
          imprimer renseignements ; fpc ; sortir ;
```

CØ l'action A4 correspond au cas où l'utilisateur ne veut connaître les renseignements demandés que pour certaines unités de l'application, celles dont les noms figurent en paramètre CØ ;

```
A4 : utilisation (DESC, lecture) ;
     pour I de 1 à NUM (AF(2)) faire
          recherche dans dictionnaire (AF(2,I), CLE) ;
si non unité créée alors imprimer ('unité', I, 'absente') ;
sinon soit DESCRIPTIF telque MATDESC = CLE ;
      imprimer renseignements ;
fsi ; fp ;
```

CØ l'action A5 correspondant au cas où l'utilisateur désire connaître des renseignements sur les objets déclarés dans la classe d'application : objets élémentaires ou objets descriptifs de fichier physique permanents CØ ;

```
pour I de 1 à 2 faire
sélection
```

AF (1, I) =	'IDENTIFICATEUR'	'FICHIER'
	ØPTIØN (1) = vrai ;	ØPTIØN (2) = vrai ;

```
fintable ; fp ;
pour chaque IDENTIFICATEUR de TABLE DES IDENTIFICATEURS
     telque NUMERØ de DESCRIPTIF = 1 ou 2 ou - 1 ou - 2
faire si TYPE ⩽ - 1 et ØPTIØN (1)
                alors imprimer (IDENTIFICATEUR) ; fsi ;
     si TYPE    = 5 et ØPTIØN (2)
                alors imprimer (IDENTIFICATEUR) ; fsi ;
fpc ;
finproc ;
module imprimer renseignements ;
```

CØ ce module permet de faire l'édition des renseignements demandés par l'utilisateur CØ ;

```
     CHAINTYPE file (max = 18) caractère ;
     CHAINETAT file (max = 21) caractère ;
imprimer (NØMDESC, MATDESC) ;
si ØPTIØN (1) alors
sélection
```

TYPEDESC =	1	2	4	8
CHAINTYPE =	'MODULE'	'classe'	'métamodule'	'classe de commande'

```
fintable ;
```



```
   imprimer (CHAINTYPE) ; fsi ;
si ØPTIØN (2) alors
   sélection
```

ETATDESC	1	2	3	4
CHAINETAT =	'source'	'compilé exploitation'	'compilé mise au point'	'édité'

```
   fintable ;
   imprimer (CHAINETAT ) ; fsi ;
si ØPTIØN (3) et ETATDESC ⩾ 2 alors
   imprimer ('degré d'erreur', ERRDESC) ; fsi ;
si ØPTIØN (4) et ETATDESC ⩾ 2 alors
   imprimer ('taille zone constante', CONSTDESC) ; fsi ;
si ØPTIØN (5) et ETATDESC ⩾ 2 alors
   imprimer ('taille zone variable', VARDESC) ; fsi ;
si ØPTIØN (6) ou ØPTIØN (7) alors
   pour chaque métamodule de METADESC faire
   recherche nom dans dictionnaire (NØM, métamodule) ;
   imprimer (NØM) ; fpc ; fsi ;
si ØPTIØN (6) ou ØPTIØN (8) alors
   pour chaque classe de CLASDESC faire
   recherche nom dans dictionnaire (NØM, classe) ;
   imprimer (NØM) ; fpc ; fsi ;
si ØPTIØN (6) ou ØPTIØN (9) alors
   pour chaque module de MØDDESC faire
   recherche nom dans dictionnaire (NØM, module) ;
   imprimer (NØM) ; fpc ; fsi ;
si ØPTIØN (10) alors
   décision
```

PRØTDESC =	- 1	0	sinon
	A1	A2	A3

```
   fintable ;
```

```
A1 : imprimer ('unité protégée contre tout accès') ;
A2 : imprimer ('unité commune à toutes les applications') ;
     sortir ;
A3 : imprimer ('applications autorisées à utiliser l'unité', APPLDESC) ;
     finmod ;
```

C.5) La procédure SOURCE.

Dans cette procédure, l'interprète vérifie que chaque unité est effectivement créée dans l'application. Si c'est le cas, il commande l'impression du texte source qui figure dans le fichier des textes sources de l'application (cf. 2.5.3.2.2.).

```
procédure SØURCE ;
pour I de 1 à NUM (AF) faire
     recherche dans dictionnaire (AF(I), CLE) ;
     si non unité créée alors
          imprimer ('identificateur', I, 'inconnu') ;
     sinon imprimer le texte source figurant dans la partition de clé CLE du
          fichier des textes sources ;
     fsi ; fp ;
finproc ;
```

L'impression du texte source est réalisée de la façon suivante :

- Le module lit les enregistrements de la partition du fichier des textes sources jusqu'à la rencontre d'un enregistrement de longueur nulle qui indique la fin de la partition.

- Pour chaque enregistrement lu, le module imprime le texte à raison d'une ligne par image de carte. Chaque ligne contient donc la chaîne de caractères comprise entre 2 compteurs d'enregistrement. De cette façon, le texte source se présente de la même façon qu'il avait été entré dans le système.

C.6) La procédure ANALYSER.

La procédure ANALYSER (cf. 2.5.1.) est différente de la procédure CREER du fait que l'unité qui en est le paramètre n'est pas rendue interne au système : son texte source n'est pas rangé dans le fichier système des textes sources, elle n'est pas représentée dans les fichiers dictionnaire et descriptif.


```
procédure ANALYSER ;
     compresser le texte source temporaire ;
     liste d'entrée (1, 1) = 2 ;
CØ la procédure indique au compilateur que l'unité n'est pas à compiler mais
   seulement à analyser syntaxiquement CØ ;
     lancer compilation ;
finproc ;
```

Le module "compresser le texte source temporaire" est analogue au module "compresser le texte source" décrit dans la procédure CREER (en 2.1.1.3.). Toutefois, dans ce module, on ne relève pas les unités appelées ou utilisées et celles-ci ne sont pas rangées dans le fichier dictionnaire. De même, on ne relève pas les déclarations de procédures ou de métamodules. Le fichier dictionnaire est inchangé après l'appel de ce module.

Le module "lancer compilation" est analysé dans l'étude de la demande d'exécution d'une unité de traitement (cf. annexe E.6).
Le fait de positionner le paramètre "liste d'entrée" à 2, indique au compilateur que le texte source est rangé dans le fichier temporaire des textes sources et que l'on demande deulement l'analyse syntaxique de ce texte.

C.7) La procédure UTILISE.

Lorsque le module initial rencontre l'appel de la procédure UTILISE dans le module de commande, il doit rendre accessible tous les objets déclarés dans la classe de commande indiquée (cf. 1.4.3.).
Si la classe de commande est à l'état source, il doit lancer sa compilation. Le résultat de la compilation est le rangement dans la partition des textes compilés de la table des identificateurs de la classe de commande et de la suite des valeurs des constantes de cette classe. La procédure doit donc inclure tous les identificateurs à la table des symboles, demander de la place en zone dynamique commune, ranger dans cette zone les valeurs des constantes de la classe de commande.

```
procédure UTILISE ;
NØM file (max = 10) caractère ;
CLE, RANG entier ;
NØM = AF ;
recherche dans dictionnaire (NØM, CLE)
si non unité créée alors imprimer (identificateur inconnu) ;
                         sortir ; fsi ;
si TYPEDIC ≠ 'classecommande' alors
     imprimer ('l'identificateur ne désigne pas une classe de commande') ;
                sortir ; fsi ;
  utilisation (DESC, lecture) ;
soit DESCRIPTIF telque MATDESC = CLE ;
si état = 'source' alors lancer compilation ; fsi ;
inclure les identificateurs de la partition (CLE) du fichier des textes
         compilés ;
demander de la place en zone dynamique commune (DEBLØM) ;
initialiser les constantes de la partition (CLE) des textes compilés ;

finproc ;
```

C.8) La procédure CØPIER.

La procédure CØPIER permet de faire la copie d'un fichier physique dans un autre fichier physique (cf. 3.4.7.3.)

```
procédure CØPIER ;
NØM1, NØM2 file (max = 10) caractère ;
NØM1 = AF(1) ;
NØM2 = AF(2) ;
recherche dans la TDS (NØM1) ;
si non identificateur présent alors imprimer ('identificateur 1 inconnu') ;
                                  sortir ; fsi ;
si type (NØM1) ≠ 'fichier' alors imprimer ('l'identificateur 1 ne désigne pas
                un fichier') ; sortir ; fsi ;
rechercher la valeur dans la TDS (NOM1) ;
si NØMAPPL (NØM1) ≠ ' ' alors vérifier la protection du fichier ('lecture') ;
                         si non autorisé alors
                         imprimer (utilisation interdite en lecture') ;
                         sortir ; fsi ; fsi ;
recherche dans la TDS (NØM2) ;
```



```
si non identificateur présent alors imprimer ('identificateur 2 inconnu') ;
                                  sortir ; fsi ;
si type (NØM2) ≠ 'fichier' alors imprimer ('l'identificateur 2 ne désigne pas
                                  un fichier') ; sortir ; fsi ;
   rechercher la valeur dans la TDS (NØM2) ;
si NØMAPPL (NØM2) ≠ ' ' alors imprimer ('le fichier récepteur doit appartenir
                                  à l'application') ; sortir ; fsi ;
si Organisation (NØM2) = 'indexée' et unité (NØM2) ≠ 'DIMAS' ou 'RAD'
   alors imprimer ('organisation incompatible avec le type d'unité') ;
        sortir ; fsi ;
   Décision
   Conditions
```

E

Organisation (NØM1) = 'séquentielle'	V	V	V	V	V	V	V	V	V	V	V	V	F	F	F	F	
Unité (NØM1) = 'DIMAS' ou 'RAD'	V	V	V	V	-	-	-	-	-	-	-	-	-	-	-	-	
Unité (NØM1) = 'BANDE'	-	-	-	-	V	V	V	V	-	-	-	-	-	-	-	-	
Unité (NØM1) = 'CARTE'	-	-	-	-	-	-	-	-	V	V	V	V	-	-	-	-	
Organisation (NØM2) = 'séquentielle'	V	V	V	F	V	V	V	F	V	V	V	F	V	V	V	F	
Unité (NØM2) = 'DIMAS' ou 'RAD'	V	-	-	-	V	-	-	-	V	-	-	-	V	-	-	-	
Unité (NØM2) = 'BANDE'	-	V	-	-	-	V	-	-	-	V	-	-	-	V	-	-	
Unité (NØM2) = 'IMPR'	-	-	V	-	-	-	V	-	-	-	V	-	-	-	V	-	
Actions																	
Générer carte ASSIGN ('EI','ØLD',NØM1) ;	1	1	1	1	1	1	1	1	1	1			1	1	1	1	
Générer carte ASSIGN ('EØ','NEW',NØM2) ;	2	2	2	2	2	2	2	2	2	2			2	2	2	2	
Générer FMGE CØPY ;	3	3			3	3										3	
Générer FMGE LIST ;			3				3								3		
Générer FMGE MØVE,IN ;									3	3							
Générer FMGE SAVE ;													3	3			
Générer FMGE RESTØRE ;				3				3									
Imprimer ('fichiers incompatibles avec une opération de transfert') ;											*	*				*	
Imprimer ('attention = le fichier récepteur ne peut être réutilisé directement. L'opération inverse doit être effectuée') ;													*	*			

fin table ;

fin proc ;

C.9) La procédure LIAISØN.

L'appel de cette procédure permet d'établir la liaison entre un fichier logique d'une part et un fichier physique d'autre part (cf. 3.2.2.). Pour l'interprète, cet appel se traduit par la mise à jour du fichier catalogue en vue de la génération d'une carte de commande par le générateur.

```
procédure LIAISØN ;
LØG file (max = 10) car ;
PHY file (max = 10) car ;
UTIL file (max = 3) file (max = 9) car ;
LØG = AF (1) ;
PHY = AF (2) ;
UTIL = AF (3) ;
recherche dans la TDS (PHY) ;
si non identificateur présent alors imprimer ('Identificateur 2 inconnu') ;
                                   sortir ; fsi ;
si type (PHY) ≠ 'fichier' alors imprimer ('l'identificateur 2 ne désigne
                                   pas un fichier') ; sortir ; fsi ;
   rechercher la valeur dans la TDS (PHY) ;
si NOMAPPL (PHY) ≠ ' ' alors
     pour I de 1 à taille actuelle (UTIL) faire
           vérifier la protection (UTIL(I)) ;
           si non autorisé alors imprimer ('utilisation interdite
                 en', UTIL (I)) ;
```

CØ l'utilisateur désire manipuler dans la chaîne de traitement un fichier d'une autre application. La procédure vérifie que, pour chaque type d'utilisation indiqué dans l'appel, le propriétaire a donné son autorisation CØ ;

```
                 sortir ; fsi ;
     fp ;
  recherche dans fichier catalogue (LØG) ;
  si non présent alors
     si INDICATIF (APPLICATIØN CØURANTE, 6)
     alors UTILISATIØN (FICHCATAL, 'MISE A JØUR') ;
           dernier CATALØGUE ; CATALØGUE en CATALØGUE + 1 ;
     sinon INDICATIF (APPLICATIØN CØURANTE, 6) = vrai ;
```

```
        UTILISATION (FICHCATAL, 'ECRITURE') ;
        premier CATALOGUE ; fsi ;
        IDFICH = LOG ;
    sinon UTILISATION (FICHCATAL, 'MISE A JOUR') ;
          soit CATALOGUE telque IDFICH = LOG ; fsi ;
          IDPHY = PHY ; LISTUTI (*) = faux ;
    pour I de 1 à taille actuelle (UTIL) faire
        sélection
```

UTIL (I) =	'lecture'	'écriture'	'mise à jour'	E
	LISTUTI (1) = vrai ;	LISTUTI (2) = vrai ;	LISTUTI (3) = vrai ;	A1

```
        sortir ;
A1      imprimer ('type d'utilisation incorrect') ;
        finproc ;
```


ANNEXE D : Les procédures de la classe de modification

Contrairement aux procédures de la classe système, les procédures de la classe de modification sont protégées par le système CIVA. L'accès à une telle procédure ne peut être effectif que si l'utilisateur a indiqué, dans son module de commande, le mot de passe de l'application MODIFICATION (cf. 2.3.1.3.).

D.1) Analyse de la procédure SUPPRIMER.

Dans cette procédure (cf. 2.5.4.1.), l'interprète doit savoir si c'est une unité ou un objet élémentaire que l'on veut supprimer.
Le système fera donc tout d'abord une recherche dans le dictionnaire. Si il trouve l'identificateur, c'est donc une unité qui doit être supprimée. Si il ne trouve pas l'identificateur, le système fait une recherche dans la table des identificateurs. Si il le trouve, c'est bien un objet élémentaire à supprimer, sinon c'est une erreur : dans ce cas, le module initial émet un message d'erreur et passe à l'identificateur suivant.

```
procédure SUPPRIMER ;
XSUPNOM file (max = 10) caractère ;
NBIDEN entier ; REF booléen ; REF1 booléen ; I entier ;
NBIDEN = NUM (AF(1)) ;
si NUM (AF) = 2 alors REF = vrai ; sinon REF = faux ; fsi ;
Pour I de 1 à NBIDEN faire
décision
conditions
```

E

	REF	F	-	V	
	NUM (AF (1,I)) = 2 et AF (1, I, 2) = 'REFERENCE'	V	-	V	
	NUM (AF (1,I)) = 1	-	V	-	
Actions	REF1 = vrai ;	1		1	
	REF1 = faux ;		1		
	XSUPNØM = AF (1,I,1) ;	2		2	
	XSUPNØM = AF (1,I) ;		2		
	Recherche dans dictionnaire (XSUPNØM, CLE)	3	3	3	
	imprimer ('IDENTIFICATEUR', I, 'INCONNU') ;				1
	passer à l'élément suivant ;				2

fintable ;
décision
conditions

	unité présente	V	V	F
	unité créée	V	F	-
Actions	A1	*		
	A2		*	
	A3			*

fintable ;
fp ;

CØ l'action A1 correspond au cas où l'interprète a trouvé l'unité dans le fichier dictionnaire, cette unité étant créée dans l'application. L'interprète doit donc supprimer tous les éléments représentant cette unité dans la bibliothèque de l'application CØ ;

A1 : utilisation (DESC, mise à jour) ;
soit DESCRIPTIF telque MATDESC = CLE ;
Décision
Conditions

	ETATDESC = édité	V	-	-
	ETATDESC = compilé	-	V	-
	ETATDESC = source	-	-	V
Actions	supprimer partition (CLE) dans le fichier des textes édités ;	*		
	supprimer partition (CLE) dans le fichier des textes compilés ;	*	*	
	supprimer partition (CLE) dans le fichier des textes sources ;	*	*	*

fintable ;

pour chaque EXTERIEUR de TABEXT telque NUMMATINT = CLE
faire supprimer EXTERIEUR ; fpc ;

CØ la procédure supprime les éléments de la table des unités externes représentant des liaisons de l'unité avec des unités externes CØ ;

supprimer de dictionnaire (XSUPNØM) ;
supprimer DESCRIPTIF ;
si REF ou REF1 alors
  déterminer les successeurs dans M2 (CLE) ;

CØ l'appel de cette procédure a pour but de fournir la liste LIST1 des numéros de matricule des unités qui succèdent à l'unité supprimée dans le graphe G2 CØ ;

imprimer ('liste des successeurs de', XSUPNØM) ;
pour chaque MAT de LIST1 faire
  recherche nom dans dictionnaire (NØM, MAT) ;

```
    imprimer (NØM) ; fpc ;
  déterminer les prédecesseurs dans M2 (CLE) ;
    imprimer ('liste des prédecesseurs de ', XSUPNØM) ;
pour chaque MAT de LIST2 faire
  recherche nom dans dictionnaire (NØM, MAT) ;
    imprimer (NØM) ; fpc ;
fsi ;

  supprimer dans la matrice M1 (CLE) ;

    CØ l'appel de cette procédure a pour but de rendre isolé le point représentant
       l'unité supprimée dans le graphe G1 CØ ;

    soit APPLICATION telque NØM = APPLICATIØN CØURANTE ; NBUNIT = NBUNIT - 1 ;

    CØ NBUNIT contient le nombre d'unités créées dans l'application CØ ;

    si protection = - 1 alors NBPRØT = NBPRØT - 1 ;
       sortir ;

    CØ l'action A2 correspond au cas où l'interprète a trouvé l'unité dans le
       fichier dictionnaire mais cette unité n'est pas encore créée dans l'appli-
       cation. L'interprète envoie donc un message à l'utilisateur CØ ;

  A2 : imprimer ('unité non créée') ;
       si REF ou REF1 alors
          déterminer les prédecesseurs dans M2 (CLE) ;
          imprimer ('liste des prédecesseurs de ', XSUPNØM) ;
  pour chaque MAT de LIST2 faire
       recherche nom dans dictionnaire (NØM, MAT) ;
          imprimer (NØM) ; fpc ;

    CØ l'unité n'étant pas créée, le système ne peut connaître les successeurs de
       cette unité dans le graphe G2 CØ ;

       sortir ;

    CØ l'action A3 correspond au cas où l'interprète n'a pas trouvé l'unité dans
       le fichier dictionnaire. Il recherche donc dans la table des symboles CØ ;
```



```
  A3 : recherche dans la table des symboles (XSUPNØM) ;
       si identificateur présent alors
          supprimer de la table des symboles (XSUPNØM) ;
       si non SUPPRIME alors imprimer ('l'identificateur n'appartient pas
                                        à la classe d'application') ; fsi ;
       sinon imprimer ('identificateur', I, 'inconnu') ; fsi ;
       finproc ;
```

D.2) Les procédures de modification de texte :

D.2.1.) La procédure CHANGER (cf. 2.5.3.1.1.) :

Pour le module initial, l'opération de modification globale du texte source d'une unité est sensiblement analogue à une opération de suppression suivie d'une opération de création. La différence essentielle réside dans le fait que le numéro de matricule étant conservé par l'unité, le fichier dictionnaire est inchangé et le fichier descriptif est simplement mis à jour.

```
procédure CHANGER ;
XCHANØM file (max = 10) caractère ;
XCHANØM = AF ;
recherche dans dictionnaire (XCHANØM, CLE) ;
si non unité créée alors
   imprimer ('unité non créée dans l'application') ;
   lire le texte source jusqu'à fin-mod, fin-classe ou Ø-fin-mod ;
   sortir ; fsi ;
utilisation (DESC, mise à jour) ;
soit DESCRIPTIF telque MATDESC = CLE ;
Décision
Conditions
```

	ETATDESC = 'édité'	V	-	-
	ETATDESC = 'compilé'	-	V	-
	ETATDESC = 'source'	-	-	V
Actions	supprimer partition (CLE) dans le fichier des textes édités ;	*		
	supprimer partition (CLE) dans le fichier des textes compilés ;	*	*	
	supprimer partition (CLE) dans le fichier des textes sources ;	*	*	*
	ETATDESC = 'source' ;	*	*	
	création d'une unité ;	*	*	*

fintable ;

pour chaque EXTERIEUR de TABEXT telque NUMMATINT = CLE
faire supprimer EXTERIEUR ;

CØ la table des unités externes est mise à jour CØ ;

NUMERØVERSIØN = NUMERØVERSIØN + 1 ;

CØ si l'unité est employée dans une autre application, c'est la dernière version qui doit être incluse à l'unité de traitement CØ ;

PREMØDIF = PREMØDIF + 1 ;
LISTMØDIF (PREMØDIF) = CLE ;

CØ le numéro de matricule de l'unité modifiée est rangé dans la liste des unités qui ont subi une modification, ceci afin de permettre au système de gérer les transitions d'états CØ ;

finproc ;

Le module "création d'une unité" est semblable au module "création d'unités" décrit dans la procédure CREER (cf. annexe C.2).
Toutefois, ils diffèrent par les points suivants :
- le module "création d'une unité" n'est pas réentrant puisqu'il n'a à traiter que le texte source d'une seule unité.
- l'unité existant déjà dans l'application, il n'est pas nécessaire d'ajouter ses caractéristiques dans le fichier dictionnaire et il suffit de mettre à jour l'enregistrement du fichier descriptif.

D.2.2.) Les procédures CØRRIGER, INSERER, ØTER, MØDIFIER (cf. 2.5.3.1.2.)

Les textes de ces procédures ayant déjà été indiqués par ailleurs (voir PAYAFAR (17)), nous ne les rappellerons pas ici.

D.2.3.) La procédure REMPLACER (cf. 2.5.3.1.3.)

procédure REMPLACER ;
REMP1 file (max = 10) caractère ;
REMP2 file (max = 10) caractère ;
REMP1 = AF (1) ;
REMP2 = AF (2) ;
CLE entier ; RANG entier ; REF booléen ;
recherche dans dictionnaire (REMP2, CLE) ;

CØ la procédure REMPLACER vérifie que le nouvel identificateur ne désigne pas déjà une unité créée dans l'application CØ ;

décision
conditions

	NUM (AF) = 3 et AF (3) = REFERENCE	V	F	-
	unité créée	F	F	V
actions	REF = vrai ;	*		
	REF = faux ;		*	
	Recherche dans dictionnaire (REMP1, CLE)	*	*	
imprimer	('IDENTIFICATEUR DEJA PRESENT DANS L'APPLICATIØN') ;			*
	sortir ;			*

fintable ;

CØ la procédure contrôle que l'unité qui doit changer de nom est bien créée dans l'application CØ ;

si non UNITE CREEE alors imprimer ('unité inconnue') ;
sortir ; fsi ;

NØMDIC = REMP2 ;
utilisation (DESC, mise à jour) ;
soit DESCRIPTIF telque MATDESC = CLE ;
NØMDESC = REMP2 ;
décision
conditions

	ETATDESC = 'édité'	V	-	-
	ETATDESC = 'compilé'	-	V	-
	ETATDESC = 'source'	-	-	V
Actions	supprimer partition (CLE) dans le fichier des textes édités ;	*		
	supprimer partition (CLE) dans le fichier des textes compilés ;	*	*	
	modifier le nom de l'unité dans le fichier des textes sources (CLE, REMP2) ;	*	*	*
	ETATDESC = 'source' ;	*	*	

fintable ;

PREMØDIF = PREMØDIF + 1 ;
LISTMØDIF (PREMØDIF) = CLE ;
NUMERØVERSION = NUMERØVERSIØN + 1 ;

CØ la procédure range le matricule de l'unité dans la liste des unités modifiées pour que le module de gestion des transitions d'état puisse en tenir compte CØ ;

si REF alors déterminer les successeurs dans M2 (CLE) ;
imprimer ('liste des successeurs de', REMP2) ;
pourchaque MAT de LIST1 faire
recherche nom dans dictionnaire (NØM, MAT) ;
imprimer (NØM) ; fpc ;
déterminer les prédecesseurs dans M2 (CLE) ;
imprimer (liste des prédecesseurs de ', REMP2) ;
pour chaque MAT de LIST2 faire
recherche nom dans dictionnaire (NØM, MAT) ;
imprimer (NOM) ; fpc ;
fsi ;
finproc ;

D.3) La procédure DETRUIRE (cf. 3.4.7.1.) :

La procédure DETRUIRE permet de supprimer un fichier permanent sans détruire l'identificateur de la classe d'application qui définit ce fichier.

```
procédure DETRUIRE ;
IDEN file (max = 10) caractère ;
IDEN = AF ;
recherche dans la TDS (IDEN) ;

si non identificateur présent alors imprimer ('identificateur inconnu') ;
                                   sortir ; fsi ;
si unité ≠ 'classe d'application' alors
            imprimer ('l'identificateur n'appartient pas à la classe
                       d'application') ; sortir ; fsi ;
si type ≠ 'fichier' alors imprimer ('l'identificateur ne désigne pas un
                                   fichier') ; sortir ; fsi ;
   rechercher la valeur dans la TDS (IDEN) ;
si NØMAPPL  ' ' alors imprimer ('le fichier n'appartient pas à l'application') ;
                                   sortir ; fsi ;
   GENER = vrai ;
   générer une carte ASSIGN ('EI', 'ØLD', IDEN) ;
si NUMGEN ≠ ' ' alors
   générer une carte DEFG DELETE ;

CØ cette carte de commande est décrite dans (les processeurs associés au
   SGF sous SIRIS7 - (45)) CØ ;

   sinon générer une carte FMGE DELETE ;

CØ cette carte est également décrite dans (les processeurs associés au
   SGF sous SIRIS7 - (45)) CØ ;

finproc ;
```

Le booléen GENER indique au système qu'une opération concernant les fichiers physiques est générée. Ceci lui permet d'inclure les cartes de commande au prochain travail lancé dans le train.

La procédure "générer une carte ASSIGN" permet de générer une carte de commande de SIRIS7 dont le rôle est d'assurer la connexion entre un fichier logique et un fichier physique. Le premier paramètre de cette procédure est l'étiquette logique de cette carte, le deuxième paramètre indique le statut du fichier ('NEW' pour un fichier en création, 'ØLD' pour un fichier existant, 'MØD' pour un fichier en rallongement).

D.4) La procédure SUPAPPLICATIØN (cf. 2.5.4.2.) :

Cette procédure a pour rôle de supprimer une ancienne application du système CIVA.

```
procédure SUPAPPLICATIØN ;
PASSAPPL file (max = 8) car ;
APPLICATIØN PRESENTE booléen ;
APPLICATIØN CØURANTE = AF (1) ;

recherche de l'application (APPLICATIØN CØURANTE, APPLICATIØN PRESENTE) ;
sélection
```

			ELSE
NUM (AF) = 1	V	-	
NUM (AF) = 2 et AF (2,1) = PASSE	-	V	
APPLICATION PRESENTE	V	V	
MØTPASSE = AF (2,2)	-	V	
MØTPASSE ≠ ' '	V	-	
imprimer ('ERREUR DANS L'APPEL') ; sortir ;			*
A1	*	*	

fintable ;

```
A1 : déterminer les noms des fichiers système de  l'application ;
     si INDICATIF (1) alors supprimer fichier descriptif ; fsi ;
     si INDICATIF (2) alors supprimer fichier des textes sources ; fsi ;
     si INDICATIF (3) alors supprimer fichier des textes compilés ; fsi ;
     si INDICATIF (4) alors supprimer fichier des textes édités ; fsi ;
     si INDICATIF (5) alors supprimer fichier dictionnaire ; fsi ;
     si INDICATIF (6) alors supprimer fichier catalogue ; fsi ;
     si INDICATIF (7) alors supprimer fichier des sauvegardes ; fsi ;
     si INDICATIF (8) alors
        pour chaque identificateur de type fichier de classe application
        faire supprimer le fichier de l'utilisateur correspondant ; fpc ;
        supprimer la partition dans le fichier des classes d'application
                    (APPLICATIØN CØURANTE) ; fsi ;
     si INDICATIF (9) alors
        supprimer la partition dans le fichier des tables des unités externes
                    (APPLICATION CØURANTE) ; fsi ;
     si NUM (AF) = 2 alors PASSAPPL = ' ' ; sinon PASSAPPL = AF (3,2) fsi ;
     supprimer l'application (APPLICATIØN CØURANTE, PASSAPPL) ;
        finproc ;
```


ANNEXE E : Description de quelques traitements réalisés par le module initial

E.1) Le module initial :

```
module module initial ;
utilise table des applications ;
utilise table des identificateurs ;
utilise définition du module de commande ;
initialisation du système ;
pour chaque instruction de module de commande faire
      interpréter l'instruction ; fpc ;
terminaison du système ;
finmod ;
```

E.2) Les opérations d'initialisations (cf. 5.1.) :

```
module initialisation des classes système et modification ;
TAILLE entier ;
      amener la table des identificateurs en mémoire ;
```

CØ ce module permet de restaurer la table des identificateurs en mémoire. Il demande au système d'exploitation de la place en zone dynamique commune et appelle le métamodule Ø restaurer qui restaure l'image de la table qui avait été sauvegardée. A ce niveau, cette table ne contient que les identificateurs des objets déclarés dans les classes système et modification CØ ;

```
TAILLE = taille de la classe système ;
demander de la place en zone dynamique commune (DEBSYS, TAILLE) ;
```

CØ l'appel de cette procédure permet d'obtenir des emplacements de mémoire pour implanter la classe système CØ ;

```
TAILLE = taille de la classe de modification ;
demander de la place en zone dynamique commune (DEBMØD, TAILLE) ;
```

CØ cet appel permet d'obtenir des emplacements de mémoire pour implanter les objets de la classe de modification CØ ;

initialiser les paramètres d'exploitation ;
initialiser les variables contrôlables ;

finmod ;

L'initialisation de la classe d'application est réalisée par la procédure APPLICATIØN (cf. annexe C.1)

E.3) Les opérations de terminaison (cf. 5.3.) :

module terminaison du système ;
si taille actuelle (LISTMØDIF) ≠ 0 alors transition d'états ; fsi ;

CØ il se peut très bien que depuis la demande d'exécution de la dernière unité de traitement, l'utilisateur ait réalisé des opérations de modification d'unités de son application. Le module initial doit donc mettre à jour l'application avant d'en terminer l'exploitation (cf. paragraphe 5.6.3.) CØ ;

si GENER alors lancer le travail généré dans le train des travaux ;

CØ si, depuis la demande d'exécution de la dernière unité de traitement, l'utilisateur a fait un ou plusieurs appels à la procédure CØPIER du système de gestion de fichiers, le module initial doit lancer l'exécution de ces opérations (cf. l'analyse de la procédure CØPIER en annexe C.8).

pour chaque élément de table des symboles faire
    si unité = 'module de commande' et type = 'fichier' alors
        détruire le fichier désigné ; fsi ; fpc ;

$$ sauvegarder les identificateurs de la classe d'application ;
recopier l'image de l'implantation des objets élémentaires de la classe d'application dans la partition du fichier des classes d'application de nom (APPLICATIØN CØURANTE) ;
$$ sauvegarder (' ' , table des unités externes) ;
recopier la table dictionnaire dans le fichier dictionnaire ;
finmod ;

E.4) Le module de gestion des transitions d'état :

Avant donc de lancer l'exécution d'une unité de traitement, le module initial doit gérer les transitions d'état définies ci-dessus.
Pour cela, il a besoin de connaître la liste des unités qui ont été modifiées par l'utilisateur. Cette liste appelée LISTMØDIF contient la liste des numéros de matricule des unités modifiées. Elle est garnie au moment de l'appel d'une des procédures CHANGER, CØRRIGER ou REMPLACER.
Pour chaque unité y modifiée par l'utilisateur, le module initial doit appliquer les règles de transitions d'état donnée précédemment à chacune des unités de :

$$\alpha = \hat{R}^{-1}(y)$$

Les unités de α sont les prédecesseurs de l'unité y dans le graphe G2 défini en 4.1.1.1.). Ce graphe est représenté en mémoire par la matrice de bits M2.

module transitions d'état ;
    utilise classe des compilations ;
TYPEY, ETATX entier ; TYPEX entier ;
si taille actuelle (LISTMØDIF) = 0 alors sortir ; fsi ;

CØ dans ce cas, aucune modification n'a été faite sur les unités de l'application. Celles-ci gardent donc leur état actuel CØ ;

créer la matrice M2 ;
utilisation (DESC, mise à jour) ;
pour chaque Y de LISTMØDIF faire
    pour chaque X de M2 (Y, *) telque M2 (Y, X) = 1
        faire
    soit DESCRIPTIF telque MATDESC = Y ;
    TYPEY = TYPEDESC ;
    soit DESCRIPTIF telque MATDESC = X ;
    ETATX = ETATDESC ;
    TYPEX = TYPEDESC ;
    décision
    conditions

				ELSE	
TYPEY = 'module'	F	F	F	V	
TYPEX = 'métamodule'	-	F	F	-	
TYPEX = 'module'	V	V	F	V	
ETATX = 'édité'	V	-	-	V	
ETATX = 'compilé'	-	V	V	-	
ETATX = 'compilé'					
ETATX = 'source'	*	*	*		
supprimer partition du fichier des textes édités (x) ;	*				
supprimer partition du fichier des textes compilés (x) ; NUMERØ VERSIØN = NUMERØ VERSIØN + 1 ;	*	*	*		
J = J + 1 PILEMØD (J) = x ;	*	*			
	(1)	(2)	(2)	(3)	(4)

fintable ;
fpc ; fpc ;
LISTMØDIF = vide ;
finmod ;

(1) correspond aux cas suivants :

$x \hat{R} y$: $x \in M$ et x est édité, $y \in C \cup_{MM}$

Si y est modifié, le module x doit revenir à l'état source et les textes compilés et édités doivent être supprimés. Le numéro de matricule de x est ajouté à la liste des modules à compiler (voir l'utilité de cette liste en 5.7.).

(2) correspond aux cas suivants :

$x \hat{R} y$: $x \in M \cup C$, x est compilé, $y \in C \cup_{MM}$

le module ou la classe x doit alors passer à l'état source et le texte compilé correspondant est supprimé. Si x est un module, son numéro de matricule est ajouté à la liste des modules à compiler.

(3) correspond au cas suivant :

$x \hat{R} y$: $x \in M$, x est édité, $y \in M$

le module x passe à l'état compilé et le texte édité est supprimé.
Dans tous les autres cas (4), l'unité x reste au même état.

E.5) Le module de lancement des compilations :

Pour lancer les compilations, le module utilise une pile dans laquelle sont rangés, au fur et à mesure des traitements, les matricules des modules à compiler. Le traitement s'arrête lorsque toute la pile a été explorée, c'est-à-dire lorsque l'ordre de compilation de toutes les classes et de tous les modules a été déterminé.

```
classe  classe des compilations ;
    PILEMØD file entier ;
    J entier ;
CØ J est l'indice du sommet de la pile PILEMØD CØ ;
    MATMØD entier ;
CØ  A un instant donné, MATMØD contient le matricule du module en cours de
    traitement CØ ;

finclasse ;
module lancer compilation ;
utilise classe des compilations ;
si taille actuelle (PILEMØD) = 0 alors
    PILEMØD (1) = CLE ; J = 1 ; fsi ;
CØ lorsque le module est appelé alors que la pile des modules à compiler est
   vide, il range dans la pile le numéro de matricule du module directeur CØ ;
```

```
pour K de 1 à * tantque J ≠ 0 faire
      MATMØD = PILEMØD (J) ;
      J = J - 1 ;
      Déterminer ordre des compilations ;
fp ;
lancer le compilateur ;
finmod ;
```

```
module déterminer ordre des compilations ;
utilise classe des compilations ; interface module de commande - compilateur ;
utilisation (DESC, lecture) ;
soit DESCRIPTIF telque MATDESC = MATMØD ;
décision
conditions
```

ETATDESC = 'édité'	V	-	F	F	F
ETATDESC = 'compilé'	-	V	F	F	F
taille actuelle (LISTMETA) = 0 (le module n'appelle aucun métamodule)	-	-	V	F	-
taille actuelle (LISTCLAS) = 0 (le module n'appelle aucune classe)	-	-	V	-	F
sortir ;	*				
ajouter MATMØD à la liste d'entrée du compilateur ;			*	2	2
déterminer l'ordre des classes ;				1	1

```
fintable ;
pour chaque X de LISTMØD faire
          J = J + 1 ; PILEMØD (J) = X ;
  fpc ;
```

CØ chaque module appelé par le module qui vient d'être traité est empilé sur la pile des modules à traiter CØ ;

```
finmod ;
```


Le module "déterminer l'ordre des classes" est présenté dans (CRIDLIG (6)). Il applique l'algorithme que nous avons présenté en 5.7.
Le résultat de ce module est l'adjonction des numéros de matricule des classes à compiler à la liste d'entrée du compilateur. L'adjonction d'un numéro n'est réalisée que si ce numéro ne figure pas déjà dans la liste.

E.6) La demande d'exécution d'une unité de traitement :

Lorsque le module initial rencontre le nom d'un module directeur, il doit lancer l'exécution de l'unité de traitement correspondante. C'est cette opération qui est analysée ici.

E.6.1.) Analyse de la demande d'exécution :

A la rencontre du nom du module directeur, le module initial passe le contrôle au module "demande d'exécution". Ce module a pour paramètre le nom du module directeur de l'unité de traitement. Ce nom est rangé dans la variable NOM déclarée dans la classe "unité de traitement".

```
module demande d'exécution ;
utilise unité de traitement, classe des compilations ;
recherche dans dictionnaire (NØM, CLE) ;
si non UNITE CREEE alors
                imprimer ('unité inconnue') ;
                sortir ; fsi ;
   utilisation (DESC, mise à jour) ;
   soit DESCRIPTIF telque MATDESC = CLE ;
si TYPEDESC ≠ 'module' alors
                imprimer ('l'unité appelée n'est pas un module') ;
                sortir ; fsi ;
transitions d'état ;
```

CØ le module met à jour les états des différentes unités de l'application comme nous l'avons vu en 5.6.2. En particulier, ce module relève également la liste des numéros de matricule des modules qui doivent être recompilés CØ ;

rechercher les unités externes ;

CØ ce module a pour rôle de reconnaître si l'unité de traitement emploie des unités externes à l'application. Le résultat est le positionnement du booléen "unité externe" à faux si l'unité de traitement n'emploie aucune unité externe, à vrai sinon CØ ;

si ETATDESC = 'édité' et UNITE EXTERNE alors
    vérifier les numéros de version des unités externes ;
    si VERSIØN DIFFERENTE alors
        supprimer partition fichier édité (CLE) ;
        ETATDESC = 'compilé' ; fsi ;
fsi ;

CØ si l'unité de traitement est déjà éditée, elle peut être directement exécutée. Mais si elle emploie des unités externes, il faut que ce soit les dernières versions de celles-ci qui figurent dans le module de chargement représentant l'unité de traitement. Si ce n'est pas le cas, les dernières versions doivent être incluses à l'unité de traitement qui doit alors être de nouveau traitée par le relieur CØ ;

si ETATDESC = 'source" ou taille actuelle (PILEMØD) ≠ 0
alors lancer compilation ; fsi ;

CØ si le module directeur ou un des modules de l'unité de traitement n'est pas compilé alors le module initial doit lancer les opérations de compilation des unités CØ ;

si ETATDESC ≠ 'édité' et UNITE EXTERNE alors
    inclure les unités externes ; fsi ;

CØ si l'unité de traitement n'est pas reliée et si elle utilise des unités externes, celles-ci doivent être introduites dans l'application. Le résultat de ce module est le positionnement du booléen INCLU à vrai si toutes les inclusions ont pu être réalisées, à faux sinon CØ ;

si non INCLU alors
    imprimer ('unités externes non incluses. exécution non lancée') ;
    sortir ; fsi ;
si ETATDESC = 'compilé' alors lancer la rèliure ;
si UNITE EXTERNE alors supprimer les partitions des unités externes dans le fichier des textes compilés ; fsi ; fsi ;

CØ lorsque l'unité de traitement est reliée, les unités externes sont supprimées de l'application CØ ;

sélection

conditions	NIVERR ⩾ 7	XINDEDIT = vrai	XTYPREC = 0	ELSE
actions	A1	A2	A3	A4

fintable ;

CØ une fois l'unité de traitement reliée, le module vérifie que l'exécution peut avoir lieu CØ ;

A1 : imprimer ('erreur maximale ⩾ 7 . exécution non lancée') ;
    sortir ;

A2 : imprimer ('erreur pendant la reliure . exécution non lancée') ;

A3 : imprimer ('espace mémoire insuffisant . exécution non lancée') ;
    sortir ;

A4 : lancer l'exécution ;
    si ATTENTE alors se mettre en attente ; fsi ;
finmod ;

E.6.2.) Analyse du lancement de l'exécution :

Comme nous l'avons vu en 5.5.1., le lancement de l'exécution d'une unité de traitement se fait par la création d'un travail et l'adjonction de ce travail dans le train des travaux du système d'exploitation. Le module initial doit donc générer des cartes de commande dans un fichier permanent ; chaque enregistrement de ce fichier contient l'image d'une carte de commande du travail qui va être lancé.

```
module lancer l'exécution ;
        générer une carte JØB ;

  CØ la première carte générée dans le fichier permanent est une carte
     de commande indiquant le début du travail. Cette carte permet
     également d'indiquer au système d'exploitation quel est le type de
     travail demandé. Ce type est connu grâce au paramètre TYPETRAV
     défini en 2.6.2.1. CØ ;

  générer une carte LIMIT ;

  CØ la deuxième carte générée dans le fichier permanent indique au
     système d'exploitation les valeurs maximales des ressources parta-
     geables. Ces valeurs sont obtenues à partir de TYPETRAV CØ ;

  générer les cartes ASSIGN ;

  CØ chaque fichier logique utilisé dans l'unité de traitement doit faire
     l'objet d'au moins une carte de commande ASSIGN. Cette carte réalise
     la connexion entre un ensemble d'opérations d'entrées-sorties défini
     dans l'unité de traitement et un fichier physique qui est défini
     dans le module de commande ou la classe d'application. Le module de
     génération des cartes ASSIGN parcourt le fichier catalogue et pour
     chaque fichier logique manipulé dans l'unité de traitement, il
     génère une carte de commande par type d'utilisation déclaré dans
     l'appel de la procédure LIAISØN CØ ;

  si GENER alors générer les appels d'utilitaires ; fsi ;
```



```
  CØ si l'utilisateur a fait appel à la procédure CØPIER du système de
     gestion de fichiers, alors le module génère des appels aux
     processeurs correspondant du système d'exploitation CØ ;

  générer une carte RUN ;

  CØ cette carte générée dans le fichier permanent est une demande au
     système d'exploitation de chargement en mémoire centrale du module
     de chargement qui représente l'unité de traitement et de lancement
     de son exécution CØ ;

  sauvegarder une image de la zone dynamique commune ;
  lancer le travail généré dans le train des travaux ;

finmod ;
```

E.7) Le traitement des unités externes :

Comme nous venons de le voir en annexe E.6, le module analysant la demande d'exécution doit savoir si l'unité de traitement utilise des unités externes et si c'est le cas, il doit inclure ces unités à l'application.

E.7.1.) La recherche des unités externes :

```
module rechercher les unités externes ;
utilise table des unités externes, unité de traitement ;

  CØ ce module recherche si l'unité de traitement de module directeur
     CLE utilise ou non des unités externes CØ ;

  LIAISØN booléen ;
  UNITE EXTERNE = faux ;
  si ETATDESC = 'édité' alors
    pour chaque ELEMENT de TABEXT faire
        pour I de 1 à taille actuelle (INCLUEXT) faire
            si INCLUEXT (I, 1) = CLE alors
            UNITE EXTERNE = vrai ; fsi ;
        fp ;
    fpc ;
```

CØ si l'unité de traitement est reliée, elle a été exécutée au cours d'une exploitation précédente. Toutes les unités externes ont donc été déjà incluses et le module de recherche peut savoir si l'unité de traitement utilise effectivement des unités externes grâce à la table CØ ;

```
sinon création de la matrice M2 ;
      pour chaque ELEMENT de TABEXT telque
            M2 (CLE, NUMMATINT) = 1 faire
      LIAISØN = faux ;
      UNITE EXTERNE = vrai ;
      pour I de 1 à taille actuelle (INCLUEXT) faire
          si INCLUEXT (I, 1) = CLE alors LIAISØN = vrai ; fsi ; fp ;
      si non LIAISØN alors INCLUEXT (I, 1) = CLE ;
                           INCLUEXT (I, 2) = 0 ; fsi ;
      fpc ;
fsi ;
```

CØ si l'unité de traitement n'est pas reliée, le module recherche, grâce à la matrice du graphe G2, les unités formant l'unité de traitement. Si une unité emploie une unité externe, il doit alors mettre à jour la table CØ ;

```
finmod ;
```

E.7.2.) La vérification des numéros de version :

Si l'unité de traitement est éditée et si elle utilise des unités externes, celles-ci doivent figurer dans le module de chargement sous leur dernière version. Le module que nous allons présenter a pour but de vérifier les numéros de version des unités incluses.

```
module vérifier les numéros de version des unités externes ;
utilise table des unités externes, unité de traitement ;
```



```
VERSIØN DIFFERENTE = faux ;
APPLICATIØN EXTERNE = ' ' ;
pour chaque ELEMENT de TABEXT par NØMAPPEXT croissant
    faire pour I de 1 à taille actuelle (INCLUEXT) faire
      si INCLUEXT (I, 1) = CLE alors
        si APPLICATIØN EXTERNE ≠ NØMAPPEXT alors
            APPLICATIØN EXTERNE = NØMAPPEXT ;
            déterminer le nom du fichier descriptif (APPLICATIØN
                                                    EXTERNE) ;
            assigner le fichier descriptif (APPLICATIØN EXTERNE) ;
        fsi ;
        utilisation (DESCRIPTIF - APPLICATIØN - EXTERNE, lecture) ;
        soit DESCRIPTIF de DESCRIPTIF - APPLICATIØN - EXTERNE
            telque MATDESC = NUMMATEXT2 ;
        si NUMEROVERSION = INCLUEXT (I, 2) alors
            VERSIØN DIFFERENTE = vrai ;
            INCLUEXT (I, 2) = 0 ; fsi ;
        fsi ;
      fp ;
    fpc ;
    finmod ;
```

E.7.3.) L'inclusion des unités externes à l'application :

Le module "inclure les unités externes" a pour rôle d'amener dans le fichier des textes compilés de l'application, les textes des unités externes utilisées par l'unité de traitement et de mettre à jour la table des unités externes en indiquant les nouveaux numéros de version des unités incluses.
Le texte de ce module n'est pas précisé ici car il est à peu près analogue à celui de la procédure AJØUTER qui est analysé en annexe C.3.

ANNEXE F : Exemples de modules de commande

1) Présentation générale de l'application "paie du lait" :

L'application que nous utiliserons à titre d'exemple concerne la "paie du lait" dans une société de transformation qui effectue le ramassage du lait auprès des agriculteurs de la région.
Cette application a été présentée dans (HERTSCHUH - (14) : annexes).

L'application consiste à établir mensuellement pour chaque agriculteur, un document (appelé bordereau de paie du lait) comportant :

- le relevé des quantités de lait livré ;
- la facture éventuelle des produits achetés par l'agriculteur à la laiterie ;
- le relevé du compte de l'agriculteur.

En plus du bordereau de paie de lait, on établit un état de décompte monétaire, un état de virement bancaire, un état statistique mensuel, un état statistique annuel.

Les documents d'entrée de cette application sont les suivants :

- le bordereau des achats effectués par l'agriculteur à la société,
- le bordereau des relevés réalisés par la société chez l'agriculteur,
- le bordereau d'analyse qui indique la qualité du lait collecté
- le bordereau des opérations financières diverses réalisées entre la société et l'agriculteur (remboursements d'acomptes ou d'avance, retenues pour achats divers, dûs antérieurs....).

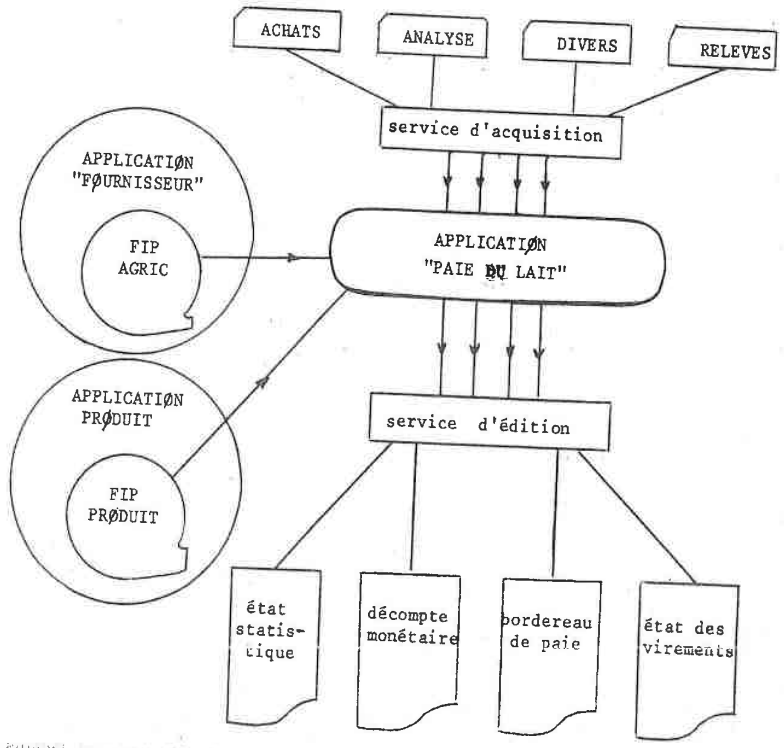
Cette application utilise également deux fichiers provenant d'autres applications :

- le fichier FIP-AGRIC qui contient des renseignements d'état civil des agriculteurs provient de l'application FOURNISSEUR.
- le fichier FIP-PRØDUIT qui représente le barème des prix des produits vendus par la société provient de l'application PRØDUIT.

Les programmes de l'application fournissent des résultats qui se présentent sous la forme de fichiers logiques internes. Ces résultats doivent faire l'objet d'un traitement d'un "service d'édition" pour être transformés en états de sortie (cf. CHABRIER - (5)).

Les bordereaux d'entrée contiennent des informations brutes qui ne peuvent être directement traitées par les programmes. Ces informations doivent être introduites dans le système par l'exécution d'un "service d'acquisition" (cf. CHABRIER - (5)).

L'application peut être schématisée par la représentation suivante :



F.2) La chaîne de traitement mensuelle :

L'analyse fonctionnelle de cette application met en évidence la présence de deux chaînes de traitement :

- une chaîne de traitement mensuelle qui permet d'établir les sorties suivantes : bordereau de paie du lait, état de décompte monétaire, état de virement bancaire et état statistique mensuel.

- une chaîne de traitement annuelle qui établit l'état statistique annuel.

Dans la suite de cet exposé, seule la première chaîne nous intéresse.
L'analyse organique de cette chaîne permet de déceler la présence de quatre unités de traitement :

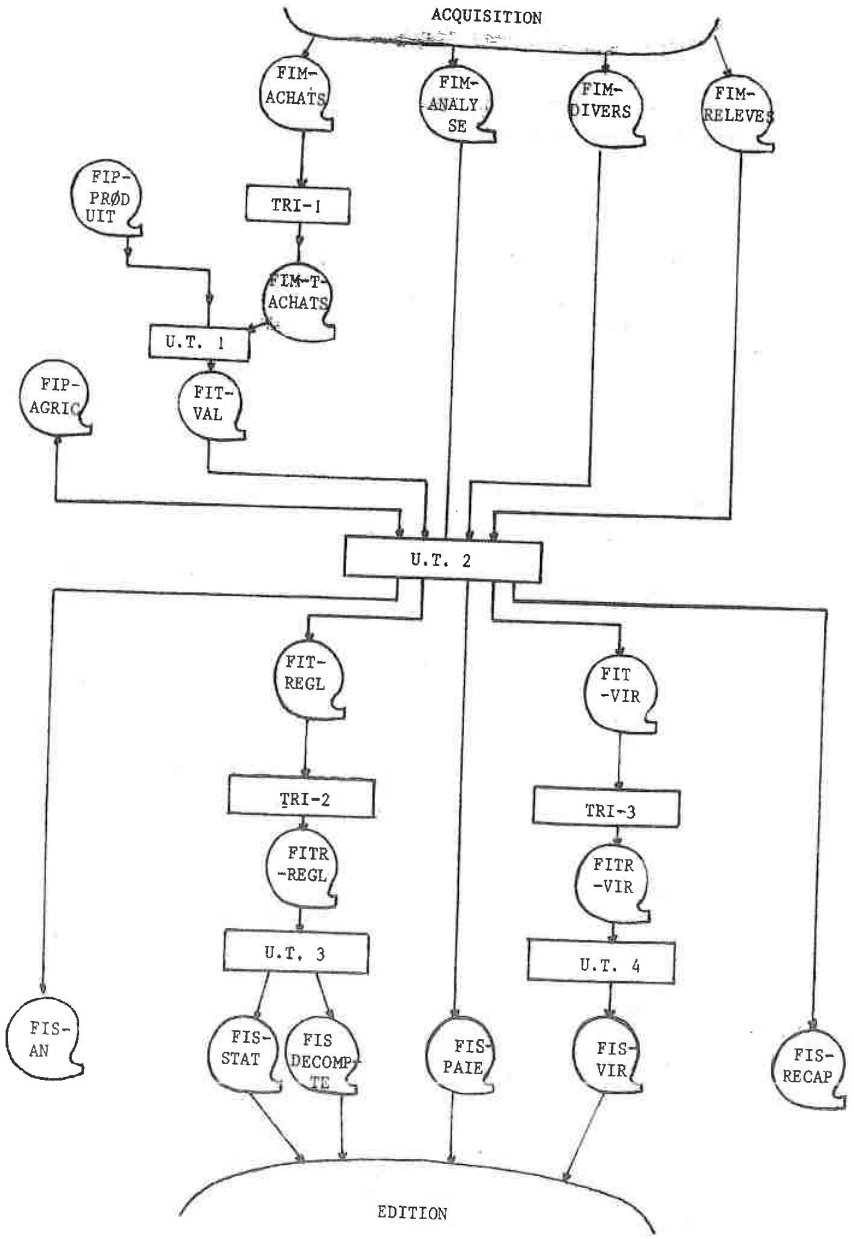
- l'unité de traitement 1 de module directeur UNITE-TRAIT1 permet de chiffrer en valeurs les achats réalisés par chaque agriculteur auprès de la Société.

- l'unité de traitement 2 de module directeur UNITE-TRAIT2 permet d'obtenir la paie du lait.

- l'unité de traitement 3 de module directeur UNITE-TRAIT3 fournit comme résultats les statistiques mensuelles et le décompte monétaire.

- l'unité de traitement 4 de module directeur UNITE-TRAIT4 fournit les informations de l'état des virements.

L'enchaînement général de ces unités peut être schématisé de la façon suivante :



F.3) La création de l'application :

L'opération de création de l'application consiste en :

- la création des unités formant l'application
- la création de variables d'exploitation permettant de contrôler l'évolution de l'exécution des chaînes de traitement.
- la déclaration des fichiers physiques manipulés
- la mise en place des liaisons permanentes entre fichiers logiques et fichiers physiques.

Création des unités de l'application :

Pour la chaîne de traitement mensuelle, les unités sont :

- les modules d'acquisition des bordereaux d'entrée
- les unités constituant l'unité de traitement 1
- les unités constituant l'unité de traitement 2
- les unités constituant l'unité de traitement 3
- les unités constituant l'unité de traitement 4
- les modules de tri (TRI-1, TRI-2, TRI-3, TRI-4)
- les modules d'édition des états.

Pour la création de chacune de ces unités, l'utilisateur doit se poser deux questions :

- l'unité existe-t-elle dans une autre application ?
- quelle est la protection inter-application que doit avoir l'unité ?

Création des variables d'exploitation :

D'après le schéma d'enchaînement des traitements que nous venons de donner, l'ordre d'exécution doit être le suivant :

$$\text{TRI-1, U.T.1, U.T.2, } \left\{ \begin{array}{l} \text{TRI-2, U.T. 3} \\ \text{TRI-3, U.T. 4} \end{array} \right\}$$

Si l'exécution d'une de ces unités de traitement ne se termine pas correctement, il est inutile d'entreprendre la suite du traitement de la chaîne.

Il paraît donc intéressant de mettre en place les contrôles d'exécution suivants :

- à chaque unité de traitement ou module de tri est associé un booléen de la classe d'application. Ce booléen est initialisé à faux. Si l'exécution d'une unité se termine de façon anormale, le booléen associé est positionné à vrai.

Déclaration des fichiers physiques :

Chaque fichier physique qui apparaît sur le schéma précédent doit être déclaré par un objet de type fichier. Le problème qui se pose est de savoir si cet objet doit appartenir à la classe d'application (dans ce cas, le fichier est catalogué) ou s'il doit être local au module de commande (dans ce cas, le fichier est temporaire).

- Les fichiers physiques correspondant aux bordereaux d'entrée et aux états de sortie sont temporaires : les objets qui les déclarent seront donc locaux aux modules de commande d'acquisition et d'édition.

- Les fichiers physiques provenant des autres applications sont permanents. Ils seront donc déclarés par des objets de la classe d'application.

- Les fichiers FIS-AN et FIS-RECAP utilisés par la chaîne de traitement annuel ont une durée de vie supérieure à celle d'une exploitation de la chaîne de traitement mensuel. Ils seront également déclarés par des objets permanents de l'application.

- Les autres fichiers utilisés par la chaîne de traitement mensuelle seront également catalogués afin d'éviter de déclarer, avant chaque exploitation de cette chaîne, les supports qu'ils repèrent.

Mise en place des liaisons permanentes :

Le fichier FIP-AGRIC est utilisé uniquement en lecture et nous pouvons supposer qu'il repère toujours le même support.

Le fichier FIP-PRODUIT est également utilisé uniquement en lecture et nous pouvons supposer que le barème des prix des produits est toujours implanté sur le même support.

Les fichiers FIS-AN et FIS-RECAP sont créés à la première exécution de la chaîne de traitement d'une certaine année. Au cours des exécutions suivantes, ils sont simplement mis à jour.

Tous ces fichiers feront donc l'objet d'une liaison permanente avec leur support.

Module de commande de création

application (PAIE-DU-LAIT, nouvelle) ;
créer
    ≋ unités d'acquisition

CØ les modules et métamodules d'acquisition des fichiers ACHATS, ANALYSE, DIVERS et RELEVES sont rendus internes au système CØ ;

créer (communs) ;
    ≋ module de tri-1

CØ le module TRI-1 permettant de trier le fichier FIM-ACHATS est créé dans l'application. Il pourra être utilisé par d'autres unités appartenant à d'autres applications CØ ;

créer (communs) ;
    ≋ unités constituant l'unité de traitement 1

créer ;
    ≋ unités constituant l'unité de traitement 2

créer (COMMERCIAL, STATISTIQUES) ;
    ≋ unités constituant l'unité de traitement 3

créer (FOURNISSEUR) ;
    ≋ unités constituant l'unité de traitement 4

CØ les unités de traitement de la chaîne de traitement mensuelle de l'application "PAIE-DU-LAIT" sont créées. L'unité de traitement 1 est utilisable à partir d'autres applications. L'unité de traitement 2 est protégée contre tout accès venant de l'extérieur. Les constituants de l'unité de traitement 3 peuvent être utilisés dans les applications "COMMERCIAL" et "STATISTIQUES" et seulement elles. Par contre, les constituants de l'unité de traitement 4 peuvent être utilisés dans l'application "FOURNISSEUR" et seulement elle CØ ;

ajouter (TRI-2, FØURNISSEUR, seul) ;
ajouter (TRI, PRØDUIT, seul, TRI-3) ;

CØ le module TRI-2 de l'application "FØURNISSEUR" est ajoutée à l'application courante. Le module TRI de l'application "PRØDUIT" est également ajouté. Dans l'application courante, il s'appelle TRI-3. TRI-2 permet de trier le fichier FIT-REGL . TRI-3 permet de trier le fichier FIT-VIR CØ ;

créer (communs) ;
≋ unités d'édition

CØ les modules et métamodules d'édition des fichiers FIS-STAT, FIS-DECØMPTE, FIS-PAIE et FIS-VIR sont créés dans l'application CØ ;

créer (UT1 booléen ; UT2 booléen ; UT3 booléen ; UT4 booléen ;
TRI1 booléen ; TRI2 booléen ; TRI3 booléen ;) ;

CØ ces booléens permettront de contrôler l'évolution de l'exécution de la chaîne de traitement CØ ;

créer (FIP-AGRIC-PHY type fichier ;
FIT-VAL-PHY type fichier ;
FIS-AN-PHY type fichier ; .....) ;

CØ déclaration de tous les fichiers physiques utilisés par la chaîne de traitement CØ ;

FIP-AGRIC-PHY = ((bande, (BQ1)), (sc, (lx, 120, 1000)), (FØURNISSEUR)) ;

CØ le fichier FIP-AGRIC-PHY est implanté sur la bande BO1 et appartient à l'application fournisseur CØ ;

FIT-VAL-PHY = ((bande, (BO3)), (x, (lx, 80, 900))) ;
FIS-AN-PHY = ((bande, (B09, B10)), (x, (lv, 100, 300)),, (CØMMUNS)) ;

CØ le fichier FIS-AN-PHY est un fichier multivolume : il est implanté sur les bandes BO9 et B10. De plus, tout programme peut lancer des opérations de lecture dans ce fichier CØ ;

≋ déclarations des autres fichiers physiques utilisés par la chaîne de traitement.

liaison (FIP-AGRIC, FIP-AGRIC-PHY, lecture, permanente) ;
liaison (FIP-PROD, FIP-PROD-PHY, lecture, permanente) ;
liaison (FIS-AN, FIS-AN-PHY, (écriture, mise à jour), permanente) ;
liaison (FIS-RECAP, FIS-RECAP-PHY, (écriture, mise à jour), permanente) ;

CØ les liaisons entre les fichiers physiques et les fichiers logiques indiquées sont permanentes : elles sont donc conservées par le système d'une exploitation à l'autre CØ ;

UT1 = faux ; UT2 = faux ; UT3 = faux ; UT4 = faux ;
TRI1 = faux ; TRI2 = faux ; TRI 3 = faux ;

CØ initialisations des booléens de contrôle de l'évolution de l'exécution de la chaîne de traitement CØ ;

F.3) L'acquisition des bordereaux d'entrée :

L'acquisition des bordereaux d'entrée consiste en :

- l'acquisition du fichier externe ACHATS,
- l'acquisition du fichier externe ANALYSE,
- l'acquisition du fichier externe DIVERS,
- l'acquisition du fichier RELEVES.

Ces opérations sont indépendantes les unes des autres : elles peuvent donc se dérouler colatéralement dans l'unité centrale.
Lorsque le module initial a lancé une opération d'acquisition d'un fichier, il n'a pas besoin d'attendre que cette dernière soit terminée. Il peut lancer aussitôt l'opération d'acquisition suivante. Dans le module de commande d'acquisition, le booléen ATTENTE de la classe système sera donc positionné à faux.

Module de commande d'acquisition

```
application (PAIE-DU-LAIT) ;
CARTES type fichier ;
CARTES = (('cartes'),('sq',('lx',80,80))) ;
attente = faux ;
liaison (ACHATS, CARTES, lecture) ;
liaison (FIM-ACHATS, FIM-ACHATS-PHY, écriture) ;
ACQUISITIØN-1 ;

CØ lancement de l'acquisition du fichier ACHATS. Le module initial continue
   l'interprétation sans attente CØ ;

liaison (ANALYSE, CARTES, lecture) ;
liaison (FIM-ANALYSE, FIM-ANALYSE-PHY, écriture) ;
ACQUISITIØN-2 ;

CØ lancement de l'acquisition du fichier ANALYSE. Le module initial continue
   l'interprétation CØ ;

liaison (DIVERS, CARTES, lecture) ;
liaison (FIM-DIVERS, FIM-DIVERS - PHY, écriture) ;
ACQUISITIØN-3 ;

CØ lancement de l'acquisition du fichier DIVERS ; le module initial continue
   l'interprétation CØ ;
```



```
liaison (RELEVES, CARTES, lecture) ;
liaison (FIM-RELEVES, FIM-RELEVES-PHY, écriture) ;
ACQUISITIØN-4 ;

CØ lancement de l'acquisition du fichier RELEVES CØ ;

finmod ;
```

F.4) L'exploitation de la chaîne de traitement mensuelle :

Comme nous l'avons dit en F.2, l'ordre d'exécution des unités est le suivant :

$$\text{TRI-1, U.T.1, U.T.2, } \left\{ \begin{matrix} \text{TRI-2, U.T.3} \\ \text{TRI-3, U.T.4} \end{matrix} \right\}$$

U.T. 1 doit attendre la fin de l'exécution de TRI-1 avant d'être lancée.
U.T. 2 doit attendre la fin de l'exécution de U.T. 1.

Une fois U.T. 2 exécutée, TRI-2 et TRI-3 peuvent être exécutés colatéralement.
Lorsque TRI-2 est exécuté, on peut lancer U.T. 3.
Lorsque TRI-3 est exécuté, on peut lancer U.T. 4.

Module de commande d'exploitation

```
application (PAIE-DU-LAIT) ;
liaison (FIM-ACHATS, FIM-ACHATS-PHY, lecture) ;
liaison (FIM-T-ACHATS, FIM-T-ACHATS-PHY, (lecture, écriture)) ;
TRI-1 ;

CØ lancement du tri du fichier FIM-ACHATS CØ ;

si TRI 1 alors imprimer ('ERREUR DANS TRI FIM-ACHATS) ;
              TRI A = faux ;
              sortir ; fsi ;
```

```
liaison (FIT-VAL, FIT-VAL-PHY, (lecture, écriture)) ;
UNITE-TRAIT 1 ;

CØ lancement de l'exécution de l'unité de traitement 1 CØ ;

si UT 1 alors imprimer ('UT 1 NØN EXECUTEE') ;
        UT 1 = faux ;
        sortir ; fsi ;

liaison (FIM-ANALYSE, FIM-ANALYSE-PHY, lecture) ;
liaison (FIM-DIVERS, FIM-DIVERS-PHY, lecture) ;
liaison (FIM-RELEVES, FIM-RELEVES-PHY, lecture) ;
liaison (FIT-REGL, FIT-REGL-PHY, (lecture, écriture)) ;
liaison (FIS-PAIE, FIS)PAIE-PHY, écriture) ;
liaison (FIT-VIR, FIT-VIR-PHY, (lecture, écriture)) ;

UNITE-TRAIT2 ;

CØ lancement de l'exécution de l'unité de traitement 2 CØ ;

si UT2 alors imprimer ('UT2 NØN EXECUTEE') ;
        UT 2 = faux ;
        sortir ; fsi ;

attente = faux ;
liaison (FITR-REGL, FITR-REGL-PHY, (lecture, écriture)) ;
TRI-2 ;

CØ lancement de l'exécution du tri du fichier FIT-REGL. Le module initial
   lance le tri et continue sans attendre l'interprétation du module de
   commande CØ ;

attente = vrai ;
liaison (FITR-VIR, FITR-VIR-PHY, (lecture, écriture)) ;
TRI-3 ;
```



```
si TRI 2 ou TRI 3 alors imprimer ('ERREUR DANS TRI  FIT-REGL OU FIT-VIR') ;
        TRI 2 = faux ;
        TRI 3 = faux ;
        sortir ;

attente = faux ;
liaison (FIS-VIR, FIS-VIR-PHY, écriture) ;
liaison (FIS-STAT,FIS-STAT-PHY, écriture) ;
liaison (FIS-DECØMPTE, FIS-DECØMPTE-PHY, écriture) ;
UNITE-TRAIT3 ;

CØ lancement de l'exécution de l'unité de traitement 3. Le module initial
   continue l'interprétation sans attente CØ ;

attente = vrai ;
UNITE - TRAIT 4 ;

CØ lancement de l'exécution de l'unité de traitement 4 CØ ;

si UT3 ou UT4 alors imprimer ('UT3 ou UT4 NØN EXECUTEE') ;
        UT3 = faux ;
        UT4 = faux ;
        fsi ;
finmod ;
```

F.5) L'édition des états de sortie :

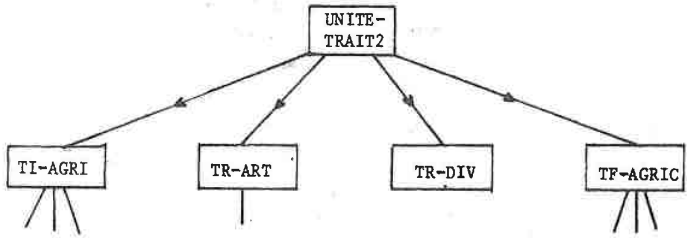
De même que pour les opérations d'acquisition, les opérations d'édition des états de sortie sont indépendantes les unes des autres : elles peuvent donc se dérouler simultanèment. Dans le module de commande d'édition, le booléen ATTENTE est positionné à faux.

Module de commande d'édition

```
application (PAIE-DU-LAIT) ;
IMPRIMANTE type fichier ;
IMPRIMANTE = (('impr'),('x',('lx,132,132))) ;
attente = faux ;
liaison (ETAT-STAT, IMPRIMANTE, écriture) ;
liaison (FIS-STAT, FIS-STAT-PHY, lecture) ;
EDITIØN - 1 ;

CØ lancement de l'édition de l'état statistique sans attente CØ ;

liaison (DEC-MØNET,IMPRIMANTE, écriture) ;
liaison (FIS-DECØMPTE, FIS-DECØMPTE-PHY, lecture) ;
EDITIØN-2 ;

CØ lancement de l'édition des décompte monétaires sans attente CØ ;

liaison (BØRD-PAIE, IMPRIMANTE, écriture) ;
liaison (FIS-PAIE, FIS-PAIE-PHY, lecture) ;
EDITIØN-3 ;

CØ lancement de l'édition des bordereaux de paie ; sans attente CØ ;

liaison (ETAT-VIR, IMPRIMANTE, écriture) ;
liaison (FIS-VIR, FIS-VIR-PHY, lecture) ;
EDITIØN-4 ;

CØ lancement de l'édition de l'état des virements CØ ;

finmod ;
```


F.6) Exemple d'opérations de maintenance :

A titre d'exemple, nous allons présenter un module de commande permettant de modifier le contenu de la deuxième unité de traitement. Celle-ci peut être schématisée de la façon suivante :



Les modifications à apporter sont les suivantes :

- remplacement du texte global du module TR-DIV sans changement de nom.
- changement du nom du module TF-AGRIC : celui-ci doit s'appeler désormais AGRIC.

Une fois ces modifications réalisées, on désire compiler en mode "mise au point" les unités ainsi modifiées et les relier entre elles.

Module de commande de maintenance

```
application (PAIE-DU-LAIT) ;
modification = 'AUTØRISE' ;
changer (TR-DIV) ;

        nouveau texte de TR-DIV

CØ l'ancien texte de TR-DIV est détruit. Le nouveau texte est introduit dans
   l'application. TR-DIV passe à l'état "source" CØ ;
```

```
remplacer (TF-AGRIC, AGRIC, référence) ;

CØ l'unité TF-AGRIC change de nom. Elle s'appelle désormais AGRIC CØ ;

corriger (UNITE-TRAIT 2) ;
        modifier ('TF-AGRIC ; ' , 1)
        AGRIC ;

CØ dans l'unité UNITE-TRAIT 2, l'appel du module TF-AGRIC est remplacé par
   un appel du module AGRIC CØ ;

compiler (environnement, mise au point, TR-DIV) ;

CØ le module TR-DIC est compilé en mode mise au point avec tout son environne-
   ment CØ ;

compiler (environnement, mise au point, (UNITE-TRAIT 2, AGRIC)) ;
relier (50, UNITE-TRAIT 2) ;

CØ l'unité de traitement 2 passe à l'état "édité" = les modules TR-DIV, UNITE-
   TRAIT 2 et AGRIC qui la composent sont en mode "mise au point" CØ ;

finmod ;
```

BIBLIOGRAPHIE

Publications CIVA

(1) AUBRY B. — Traduction des tables de décision. Thèse de 3ème cycle Université de Nancy 1. Mars 1973.

(2) BARTHELEMY C. PHILIPPE — Problème de documentation dans le projet CIVA. Thèse de 3ème cycle. Université de Nancy 1. A paraître.

(3) BENAMGHAR L. — Instructions d'affectation et définition d'un méta-langage dans le projet CIVA. Doctorat d'ingénieur. Université de Nancy 1. Juin 1973.

(4) BRETHES J. — Etude et réalisation du module initial. Rapport de DEA. Université de Nancy 1. A paraître.

(5) CHABRIER J.J. — Acquisition et édition des fichiers, analyse des données dans le projet CIVA. Thèse de 3ème cycle. Université de Nancy 1. Décembre 1973.

(6) CRIDLIG A. — Conséquences d'une modification d'un texte source dans une application CIVA ou METASYMBOL. Rapport de DEA. Université de Nancy 1. Juin 1972.

(7) DENDIEN J. — Gestion statique de mémoire dans un système de programmation modulaire . Doctorat d'ingénieur. Université de Nancy 1. Mars 1973.

(8) DERNIAME J.C. — Le projet CIVA, un système de programmation modulaire. Doctorat d'Etat. Université de Nancy I. Janvier 1974.

(9) DUCLOY J. — Compilation dans le projet CIVA. Doctorat d'ingénieur. Université de Nancy 1. Mars 1973.

(I0) FIEGEL C. Traduction des instructions d'itération dans le projet CIVA. Thèse de 3ème cycle. Université de Nancy 1. A paraître.

(II) FIEGEL C. Le relieur. Etude et réalisation. Document interne CIVA. Université de Nancy 1. Juillet I973.

(I2) FIEGEL C. Le langage de base $CIVA_0$. Document interne CIVA. Université de Nancy 1. Octobre I974.

(I3) HARMANT G. Métatraitement dans le projet CIVA. Thése de l'université de Nancy 1. A paraître.

(I4) HERTSCHUH N. L'analyse dans le projet CIVA. Thèse de 3ème cycle. Université de Nancy 1. Juin I974.

(I5) HUMBERT Traduction des expressions arithmétiques et booléennes dans le projet CIVA. Rapport de DEA. Université de Nancy 1. Juin I972.

(I6) LION F. Gestion des objets de taille variable dans le projet CIVA. Thèse de 3ème cycle. Université de NANCY 1. A paraître.

(I7) PAYAFAR M. Modifications des fichiers et des textes dans le projet CIVA. Thèse de l'université de Nancy 1. Avril 1972.

(I8) THUTEY J.P. Les tables de décision et de sélection. Etude et réalisation. Rapport de DEA. Université de Nancy 1. Juin I974.

(I9) VIAULT D. Les procédures de génération du compilateur CIVA. Rapport de DEA. Université de Nancy 1. Juin I973.

Etudes de quelques langages de commande

(20) ALSBERG P.A. Extensible data features in the opérating System language OSL/2. SIGOPS. June I972.

(2I) DE CALUWE R. Etude du langage de commande pour le réseau d'ordinateurs SOC. Thèse de 3ème cycle. Université scientifique et médicale de Grenoble. Septembre I973.

(22) C.I.I. Langage de commande sous SIRIS 7 / SIRIS 8. Manuel d'utilisation et d'opérations.

(23) CLARK B.L. The design of a system programming language. Thesis. University of Toronto. November I97I.

(24) CLARK B.L. HORNING J.J. The system language for project SUE. SIGPLAN. October I97I.

(25) DU MASLE M.J. NJCL, un langage de commande pour réseau d'ordinateurs. Université scientifique et médicale de Grenoble. Juillet I973.

(26) GURSKI A.F. Towards a hight level job control language. University of Bergen.

(27) SNOWDON R.A. PEARL : an interactive System for the preparation and validation of structured programs. Technical Report. University of Newcastle upon Tyne. November I97I.

(28) STEPHENSON C.J. On the structure and control of commands. ACM. Operating Systems Review. SIGOPS October I973.

Etudes générales sur les systèmes d'exploitation

(29) AUROUX A. HANS C. — Le système CP/67 et CMS. Centre scientifique IBM de Grenoble. Août 1968.

(30) CERT-ENSAE — Système à accès multiple SAM. Bibliothèque du centre d'études et de recherches de Toulouse. Mai 1971.

(31) C.I.I. — Système de gestion de fichier SGF sous SIRIS 7-SIRIS 8. Manuel d'utilisation.

(32) FERRIE, KAISER LANCIAUX, MARTIN — An extensible structure for protected systems design. International workshop on protection in operating systems. IRIA, août 1974.

(33) FLANIGAN L.K. — Introduction to the Computing Center and to MTS (Michigan Terminal System). Computing Center and Department of Computer and Communication Sciences. University of Michigan. September 1969.

(34) KAISER C. — Conception et réalisation de systèmes à accès multiple : gestion du parallélisme. IRIA Laboria. Note de travail Esope A 23. Février 1973.

(35) KRAKOWIAK S. — Conception et réalisation de systèmes à accès multiple : allocation de ressources. IRIA Laboria. Note de travail Esope A 24. Février 1973.

(36) NEBUT J.L. — Conception d'un système de langages de programmation. Thèse de docteur-ingénieur. Université de PARIS VI. Novembre 1974.

(37) ROUX Y. — Système MAJONC Version 3. Manuel SISI. Saclay n° 16. Janvier 1971.

(38) SPERRY RAND — Univac 1100 Série. Operating System EXEC 8. Sperry Rand Corporation 1971.

Etudes de méthodes de gestion d'applications

(39) BURNIAT J. — Principe d'action et d'organisation en informatique. DUNOD 1971.

(40) MALLET R. — La méthode informatique. Conception et réalisation de l'informatique de gestion. HERMANN 1971.

(41) REIX R. — L'analyse en informatique de gestion. DUNOD. Université et technologie 1971.

(42) SAVARY H. — Outils de mise au point pour langages de haut niveau : association de modules et contrôle de l'éxécution. Thèse de 3ème cycle. Université scientifique et médicale de Grenoble. Septembre 1973.

Aide à la réalisation du module initial

(43) CHLON J. CLEEMANN E. — Le langage ALGOL W. Initiation aux algorithmes. Presse universitaire de Grenoble 1973.

(44) C.I.I. — Procédures systèmes sous SIRIS 7 - SIRIS 8. Manuel d'utilisation.

(45) C.I.I. — Processeurs associés au SGF sous SIRIS 7 - SIRIS 8. Manuel d'opérations et d'utilisation.

(46) C.I.I. — Normes de programmation sous SIRIS 7. Manuel d'utilisation.

(47) DERNIAME J.C. PAIR C. — Problèmes de cheminement dans les graphes. DUNOD 1971.

(48) ROY B. Algèbre moderne et théorie des graphes. DUNOD finance et économie appliquée. I969.

(49) VERJUS J.P. Quelques propriétés des langages d'utilisation des systèmes. Ecole d'été AFCET. Grenade. Juillet I973.

NOM DE L'ETUDIANT : VIAULT Daniel

NATURE DE LA THESE : Doctorat de Spécialité en Informatique

VU, APPROUVE

& PERMIS D'IMPRIMER

NANCY, le 14 février 1975

LE PRESIDENT DE L'UNIVERSITE DE NANCY I

UNIVERSITÉ DE NANCY I Le Président

J.R. HELLUY