Un standard pour une représentation d'objets de type liste dans le domaine du traitement d'images. Manuel d'utilisation

Philippe Garnesson, Gérard Giraudon

INRIA

UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél (1) 39 63 55 11

Rapports Techniques

N° 82

UN STANDARD POUR UNE REPRÉSENTATION D'OBJETS DE TYPE LISTE DANS LE DOMAINE DU TRAITEMENT D'IMAGES

MANUEL D'UTILISATION

Philippe GARNESSON
Gérard GIRAUDON

Mars 1987

UN STANDARD POUR UNE REPRESENTATION D'OBJETS DE TYPE LISTE DANS LE DOMAINE DU TRAITEMENT D'IMAGES

MANUEL D'UTILISATION

Philippe GARNESSON

Gérard GIRAUDON

I.N.R.I.A. Sophia Antipolis
Route des Lucioles
06560 Valbonne

RESUME :

La Vision par Ordinateur nécessite souvent de manipuler de nombreux attributs d'image, tels que chaînes, segments, régions. Ces structures, bien que différentes d'un point de vue image, ont une organisation très similaire, de type liste, surtout vis à vis de la mémoire secondaire. On présente dans ce rapport technique une unification de la représentation de ces objets en mémoire secondaire afin d'obtenir un standard de programmation (fonctions de base écrites en C) qui en facilite l'exploitation et l'échange d'un point de vue utilisateur. Ce document a pour but de

- présenter les possibilités du standard
- montrer comment l'utiliser
- aider à le maintenir

A STANDARD FOR OBJECT LIKE-LIST REPRESENTATION IN COMPUTER VISION

USER'S GUIDE

ABSTRACT

Computer Vision often requires the manipulations of many different types of image features, such as chain-code, line segments or regions. As iconic objects, these features are different but they have a very similar list-like structure, especially towards secondary memory. A standardization of these objects representation in secondary memory is presented, in this report,. The goal is to obtain an adequate software tool (built-in functions in C language) which makes data processings and data transportability easier for end users. The document's aim is to :

- Present software possibilities
- Present how to use it
- Help for maintenance

SOMMAIRE

1. PRESENTATION

1.1. Les types de fichier concernés

Les fichiers concernés par cette proposition de standard sont tous ceux qui ont pour vocation de permettre de stocker des primitives images qui ont les propriétés suivantes :

1/ Elles sont composées d'un ensemble d'objets de même types que nous appellerons **éléments.** Ces éléments ont la caractéristique d'être composés d'un autre type d'objet que nous appellerons **maillons.**
Le nombre des maillons est variable suivant les éléments.

Schématiquement:

Fichier image

Eléments

Maillons

2/ Chacun des éléments est identifiable par une clef numérique unique.

3/ Le nombre de maillons que contient un élément doit être connu au moment de l'écriture sur disque et n'est pas susceptible d'être modifier.

4/ Il peut être attaché des informations dites variables
- à l'ensemble des éléments
- à chacun des élements
- à chacun des maillons

On n'impose aucune contrainte sur ces informations sinon qu'elles sont pour un type de fichier donné
- de même type pour tous les éléments
- de même type pour tous les maillons

Ne sont pas comprises dans ces informations variables
- le nombre d'éléments dans l'image
- les clefs numériques des éléments
- le nombre de maillons que contient un élément

1.2. Les primitives de base

1.2.1. Primitives manipulant les fichiers

On définit des fonctions d'accès aux fichiers analogues à celles qui existent en C. Ces fonctions offrent l'avantage de manipuler des objets plus complexes, à savoir des en_têtes de fichiers standards, des éléments et des maillons.

Attention : Il n'y a pas de fonctions qui manipulent un élément complet
(à savoir les informations attachées à l'élément et tous ses maillons).
Cela permet de rester complètement indépendant des représentations adoptées en mémoire centrale.
On a donc préféré des fonctions qui permettent de manipuler
- soit un en_tête d'élément (les infos attachées à l'élt)
- soit un maillon.

Par la suite, on parlera d'élément pour désigner un en_tête d'élément.

La liste des primitives :

- CR_FIC : création d'un nouveau fichier (ouverture implicite).
- OP_FIC : ouverture d'un fichier existant.
- CL_FIC : fermeture d'un fichier.
- WR_ELT : écriture de l'en_tête d'un élément.
- WR_MAI : écriture d'un maillon.
- RE_ELT : lecture de l'en_tête d'un élément.
- RE_MAI : lecture d'un maillon.
- RA_FIC : accès direct :
  permet de se positionner dans le fichier
  - sur un élément.
  - sur le maillon d'un élément.
- WR_FIC : permet de modifier les informations attachées au fichier.
- PR_FIC : visualisation des caractéristiques d'un fichier.

Les principales possibilités offertes par ces primitives :

- On peut tester l'existence d'un fichier, d'un élément ou d'un maillon.
- Quand on crée un fichier, il n'est pas nécessaire d'indiquer le nombre d'éléments qu'il contient.
- Quand on ouvre un fichier existant :
  - on peut connaître son nombre d'éléments.
  - on peut y ajouter de nouveaux éléments.
- on peut lire un élément ou un maillon en accès direct ou séquentiel.
- l'accès direct permet de modifier le contenu des informations variables attachées au fichier, à ses éléments ou ses maillons.
- Le read permet de détecter la fin de fichier ou la fin des maillons d'un élément.

1.2.2. Les descripteurs de zones variables

Un des problèmes majeurs du standard est qu'il doit permettre à l'utilisateur d'attacher n'importe quelles informations

- à l'ensemble du fichier
- aux éléments
- aux maillons

Le type et le nombre de ces informations sont variables suivant le type d'image, et sont susceptibles d'évoluer au cours du temps.
Afin d'avoir un standard qui permette d'unifier toutes les images qui nous intèressent on utilise 3 Descripteurs de Zones Variables.

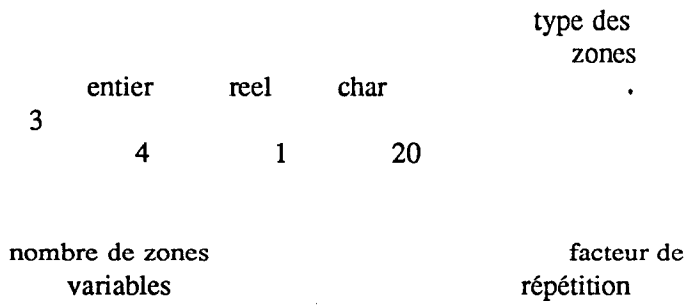
Un Descripteur de Zones Variables (DZV) est un objet qui permet de savoir comment les informations variables sont utilisées en mémoire centrale et par conséquent comment les y lire pour les stocker sur fichier et les restituer.

Exemple de descripteur :

Supposons qu'un utilisateur décide d'associer aux maillons les informations suivantes :

- une table de 4 entiers.
- 1 reel.
- 1 chaîne de 20 caractères.

Le descripteur sera alors :



En mémorisant 3 DZV pour chacun des fichiers, on se donne la possibilité de développer un standard d'une grande souplesse.
D'autre part, ce choix permet de stocker l'information de façon très dense. En effet les informations nécessaires à la gestion du fichier se limitent presqu'aux 3 descripteurs de zones variables.

Afin de permettre à l'utilisateur de décrire facilement les zones variables de ces fichiers et de lui permettre de savoir si le type d'un fichier donné est bien celui auquel il s'attend, on met à sa disposition trois primitives.

- CR_DZV : création d'un DZV .
- CP_DZV : comparaison de deux DZV.
  Cette primitive peut permettre de savoir si on utilise un fichier à bon escient.
- RE_DZV : permet de connaître les caractéristiques d'un DZV existant.

1.3. Les contrôles effectués par les primitives

En C, il existe une fonction ("perror") qui permet d'analyser le code retour d'une fonction "système" et d'écrire un message explicitant l'erreur correspondant au dernier appel.
On définit une fonction similaire qui permet de détecter 15 types d'erreurs auxquels s'ajoutent celles de la fonction "perror".

Exemples d'erreurs détectées :

- Si un fichier n'a pas été fermé,alors l'ouverture retourne un code l' indiquant.
- Si on écrit deux éléments qui ont la même clef.
- Si on essaie de lire ou écrire plus de maillons qu'ils n'en existent.

1.4. Les fonctionnalités du standard

1/ Il permet d'unifier de nombreux types de primitives symboliques en Vision par Ordinateur :
   - chaînes
   - métachaînes
   - segments
   - contours
   - régions
   ...

2/ Il est doté de primitives simples à utiliser.

3/ Les primitives permettent des accès séquentiels en écriture et lecture.

4/ Les primitives permettent des accès directs en lecture et modification.

5/ On peut créer un fichier sans en connaître le nombre d'éléments. On peut ajouter des éléments à un fichier existant.

6/ On obtient des fichiers qui peuvent évoluer sans remettre en cause le standard.

7/ La taille utilisée par le fichier est minimisée.

8/ En cas d'accès séquentiel, les fichiers sont "pipables" au sens UNIX.

9/ La possibilité de l'accès direct est résolue sans mettre en oeuvre un fichier "d'index".
On simplifie ainsi de nombreux aspects : cohérence des fichiers, copie ...

2. UTILISATION DES PRIMITIVES

Nous conseillons vivement à tout nouvel utilisateur du standard de lire très attentivement ce chapitre .
Les exemples d'utilisations des primitives de l'annexe I peuvent compléter ce chapitre.

2.1. Remarques générales

1/ Toutes les primitives du standard ont pour résultat un entier :
- positif ou nul : si tout c'est bien passé.
- négatif : en cas d'erreur.

Dans le cas d'un **op_fic()** ou d'un **cr_fic()** s'il n'y a pas eu d'erreur, l'entier qui a été retourné va servir par la suite de clef d'accès au fichier.

Exemple :

```
fd = op_fic ( "toto", ... );
wr_elt ( fd, ... );
```

2/ Il est conseillé dans une phase de mise au point de faire suivre systématiquement les appels des primitives du standard par la fonction **de_bug().**

Exemple :

```
cl ( fd );
de_bug( "commentaire optionnel" );
```

3/ Les procédures qui manipulent les DZV se chargent d'allouer la place nécéssaire. L'utilisateur ne doit donc déclarer que des pointeurs sur les DZV.

4/ Toutes les procédures qui utilisent des listes de paramètres variables doivent respecter :

- La liste des paramètres doit être complète, on ne peut se permettre de donner seulement les premiers paramètres.
- L'allocation de la place mémoire correspondant à ces paramètres est laissée à la charge de l'utilisateur.
- Ces paramètres (sauf dans le cas de cr_dzv()) doivent être des pointeurs.

2.2. Les différentes primitives

Pour chacune des primitives du standard, ce chapitre fournit une description complète, à savoir :

- La déclaration de la primitive
- Ce qu'elle fait
- Le type de ces paramètres
- Des remarques importantes
- Un exemple d'appel

Pour pouvoir utiliser les primitives du standard, il est nécéssaire de référencer la bibliothèque :

#include <lib_dzv_image>

Les types des différentes primitives sont ainsi définis ainsi que le type des DZV.
Le type des DZV est défini par un " typedef ... " et porte le nom "DZV", il suffit donc pour déclarer un DZV d'écrire :

DZV *dzv_exemple ;

Dans les exemples d'appels des primitives qui vont suivre, les arguments qui sont utilisés sont déclarés ainsi :

```
int     fd;
DZV     *dzv_fic, *dzv_elt, *dzv_mai;
int     nb_elt, num_elt, nb_maillon, moyenne, écart_type;
float   couleur
int     coordonnée[ 2 ];
char    *type;
char    date[10];
```

2.2.1. cr_dzv

DZV *cr_dzv(nb_zone, nb_elt_1, type_1, comment_1 ...
nb_elt_nb_zone, type_nb_zone, comment_nb_zone)

Action:

1/ Permet de créer un objet de type DZV.

Paramètres en entrée:

int	nb_zone;	nombre de zones du DZV, c'est aussi le nombre de triplets d'arguments qui suivent.
int	nb_elt_i;	facteur de répétition concernant la zone i du DZV.
char	*type_i;	type au sens C de la zone i. Exemple : "int"
char	*comment_i;	signification de la zone i, cet argument est obligatoire.

Notes:

1/ L'allocation mémoire est faite par la procédure.

2/ Les types possibles des zones sont :
"short" , "int" , "long" , "float" , "char" , "double"

3/ Les arguments "comment_i" peuvent être visualisés par la primitive pr_fic( ).

Exemples

```
dzv_fic = cr_dzv ( 1 , 10 , "char" , " date " ) ;
dzv_elt = cr_dzv ( 2 , 1 , "int" , "moyenne" ,
                                    1 , "int" , "ecart_type" );
dzv_mai = cr_dzv ( 2 , 1 , "float" , "niveau de gris" ,
                                    2 , "int" , "coordonnees" );
```

2.2.2. cp_dzv

int cp_dzv(dzv1 , dzv2)

Action:

1/ Compare deux DZV et retourne 0 si ils sont différents 1 sinon. Deux DZV sont différents si ils ont été créés avec des arguments différents. Les arguments concernant les commentaires ne sont pas significatifs.

Paramètres en entrée:

DZV *dzv1, *dzv2; Les deux DZV à comparer.

Note

1/ La primitive pr_fic() permet de visualiser les caractéristiques d'un fichier (y compris ses DZV).

Exemple

```
if ( cp_dzv ( dzv_mai, dzv_fic ) )
        printf (" les deux DZV sont identiques ");
else    printf (" les deux DZV sont différents ");
```

2.2.3. cr_fic:

int cr_fic(nom, type, nb_elt, dzv_fic, dzv_elt, dzv_mai, v0, ... , vn)

Actions:

1/ Permet de créer un fichier , si il existait déjà écrase l'ancien.

2/ Initialise les informations variables attachées au fichier.

3/ Retourne un entier qui servira de clef d'accès au fichier par la suite.

Paramètres en entrées:

char	*nom;	nom du fichier au sens UNIX "stdout" pour un pipe
char	*type;	type du fichier ex : "chaines"
int	nb_elt;	nombre d'éléments prévu dans le fichier
DZV	*dzv_fic;	DZV relatif au fichier
DZV	*dzv_elt;	DZV relatif aux éléments
DZV	*dzv_mai;	DZV relatif aux maillons
int	*v0,...,*vn;	liste de pointeurs sur les zones variables du fichier

Notes:

1/ Les protections associées au fichier (au sens Unix) sont prises par défaut :
- lecture et écriture pour le propriétaire
- lecture pour le groupe et les autres

Ces protections peuvent être modifier en affectant, avant d'appeler la primitive de création, la variable externe < **dzv_protection** >

Cette variable est initialisée par défaut à 0644 (en octal) et correspond au deuxième argument de la fonction creat() du langage C.

```
Exemple :   dzv_protection = 0600;
            cr_fic( "toto", ... );
```

2/ L'argument < nb_elt > peut être différent du nombre exact des éléments qu'on va écrire dans le fichier. Il a pour but d'optimiser la gestion de la table des accès directs.
Néanmoins, c'est ce nombre qu'on recupèrera dans la primitive op_fic() en cas d'utilisation d'un pipe. Dans les autres cas le nombre réel d'éléments dans le fichier sera automatiquement mis à jour.

3/ Les types possibles pour les fichiers sont :
"chaines", "contours", "metas", "segments", "regions"

Exemple:

```
fd = cr_fic("toto", "chaines", 1000, dzv_fic , dzv_elt, dzv_mai, "1986");
```

2.2.4. op_fic

int op_fic(nom, type, nb_elt, dzv_fic, dzv_elt, dzv_mai, v0, ... , vn)

Actions:

1/ Permet d'ouvrir un fichier existant respectant les normes du standard.

2/ On récupère toutes les caractéristiques du fichier.

3/ Retourne un entier qui servira de clef d'accès au fichier par la suite.

Paramètre en entrée:

char	*nom;	nom du fichier au sens UNIX "stdin" pour un pipe

Paramètres en sortie:

char	**type;	type du fichier ex : "chaines"
int	*nb_elt;	nombre d'éléments prévu dans le fichier
DZV	**dzv_fic;	DZV relatif au fichier
DZV	**dzv_elt;	DZV relatif aux éléments
DZV	**dzv_mai;	DZV relatif aux maillons
int	*v0,...,*vn;	liste de pointeurs sur les zones variables du fichier

Note:

1/ Le paramètre < nb_elt> est égal au nombre d'éléments du fichier. Néanmoins dans le cadre d'un pipe c'est la valeur qui a été donné lors de la création.
Il faut donc éviter d'écrire des programmes qui tiennent compte de ce nombre si on veut obtenir des programmes pipables.

Exemple:

```
fd = op_fic("toto", &type, &nb_elt, &dzv_fic , &dzv_elt, &dzv_mai,
                    date);
```

2.2.5. wr_elt

int wr_elt(fd, num_elt, nb_mai, v0, v1, ... vn)

Action:

1/ Permet d'ajouter un nouvel élément ou de modifier un élément existant.

Paramètres en entrée:

int	fd;	clef d'accès au fichier
int	num_elt;	numéro de l'élément
int	nb_mai;	nombre de maillons de l'élément
int	*v0,...,*vn;	liste de pointeurs sur les zones variables de l'élément

Notes:

1/ Dans le cas de l'adjonction d'un nouvel élément, il est indispensable que cette primitive soit suivie de < nb_mai > fois l'appel de la primitive wr_mai().

2/ Dans le cas de la modification d'un élément, on peut modifier seulement ses informations variables. C'est à dire qu'on ne peut modifier ni son numéro, ni son nombre de maillons.

3/ En cas d'adjonction d'un nouvel élément, celui_ci est stocké en fin de fichier. Cela signifie qu'en cas de lecture séquentiel on récupère les éléments dans l'ordre dans lequel on les a écrit.

Exemple:

```
wr_elt ( fd, num_elt, nb_maillon, &moyenne, &ecart_type);
```

2.2.6. wr_mai

int wr_mai(fd, v0, v1, v2 ..., vn)

Action:

1/ Permet d'écrire ou modifier un maillon du dernier élément référencé.

Paramètres en entrée:

int	fd;	clef d'accès au fichier
int	*v0,...,*vn;	liste de pointeurs sur les zones variables de l'élément

Notes:

1/ Dans le cas d'une création d'élément, le nombre d'appels à cette procédure doit être égal au nombre de maillons de l'élément.

2/ En cas d'accès séquentiel, l'ordre d'accès aux maillons sera le même en lecture que celui de l'écriture.

Exemple

```
wr_mai ( fd, &couleur, coordonnee );
```

2.2.7. re_elt

int re_elt(fd, num_elt, nb_mai, v0, v1, ... , vn)

Actions:

1/ Permet de lire un élément.(pas ses maillons)

2/ Permet de détecter une fin de fichier
(elle renvoie alors 0)

Paramètre en entrée:

int	fd;	clef d'accès au fichier

Paramètres en sortie:

int	*num_elt;	numéro de l'élément
int	*nb_mai;	nombre de maillons de l'élément
int	*v0,...,*vn;	liste de pointeurs sur les zones variables de l'élément

Note:

1/ Dans le cas d'un accès séquentiel, (c'est à dire qu'on n'a pas utilisé la primitive ra_fic()), il est indispensable que cette primitive soit suivie par < nb_mai > fois l'appel à < re_mai() > .

Exemple

```
re_elt ( fd, *num_elt, *nb_mai, &moyenne, &ecart_type );
```

2.2.8. re_mai

int re_mai(fd, v0, v1, ... , vn)

Actions:

1/ Permet de lire un maillon du dernier élément accéder.

2/ Permet de détecter le dernier maillon d'un élément (elle renvoie alors 0)

Paramètre en entrée:

int	fd;	clef d'accès au fichier

Paramètres en sortie:

int	*v0,...,*vn;	liste de pointeurs sur les zones variables du maillon

Note:

1/ Dans le cas d'un accès séquentiel, (c'est à dire qu'on n'a pas utilisé la primitive ra_fic()), il est indispensable que cette primitive soit appelée autant de fois qu'il y a de maillons dans l'élement.

Exemple

```
re_mai ( fd, &couleur, coordonnee );
```

2.2.9. cl_fic

int cl_fic (fd)

Action:

1/ ferme un fichier.

Paramètre en entrée:

int fd; clef d'accès au fichier

Notes:

1/ Un fichier doit être impérativement fermé si il a été ouvert (cr_fic() ou op_fic()).

2/ Un fichier qui n'a pas été refermé n'est plus accessible par le standard. Attention aux programmes qui modifie des fichiers ...

Exemple

```
cl ( fd );
```

2.2.10. ra_fic

int ra_fic(fd, num_elt, num_mai)

Actions:

1/ Fonctionnement similaire à celui de la fonction lseek() du langage C.
Permet de positionner le pointeur courant du fichier sur
- un élement existant.
- un élement inexistant (adjonction d'un élément).
- un maillon existant.

Pour se positionner sur un élément, on donne son numéro et un numéro de maillon egal à 0.
Pour se positionner sur un maillon, on donne le numéro de l'élément et l'indice du maillon (de 1 à nombre de maillons de l'élément).

2/ Si on se positionne sur un élément qui existe, (la primitive retourne 1) on peut alors le lire en utilisant re_elt() ou le modifier en utilisant wr_elt().

3/ Si on se positionne sur un élément qui n'existe pas, (la primitive retourne 0) on peut alors l'ajouter en fin de fichier en utilisant wr_elt().

4/ Si on se positionne sur un maillon qui existe (la primitive retourne 1) on peut alors le lire en utilisant re_mai() ou le modifier en utilisant wr_mai().

5/ Si on se positionne sur un maillon qui n'existe pas la primitive retourne un nombre inférieur a 0.

Paramètres en entrée:

int	fd;	clef d'accès au fichier
int	num_elt;	numéro de l'élément
int	num_mai;	indice du maillon

Note:

1/ L'utilisation de cette primitive implique un accès direct et signifie donc que son utilisation rend le programme non pipable.

Exemples:

```
ra_fic ( fd, num_elt, 0 );
ra_fic ( fd, num_elt, 3 );
```

2.2.11. wr_fic

int wr_fic(fd, v0, ... , vn)

Action:

1/ Permet de modifier les informations variables attachées à un fichier.

Paramètres en entrée:

int	fd;	clef d'accès au fichier
int	*v0,...,*vn;	liste de pointeurs sur les zones variables du fichier

Note:

1/ L'utilisation de cette primitive implique un accès direct et signifie donc que son utilisation rend le programme non pipable.

Exemple

```
wr_fic ( fd, "20/7/1986" );
```

2.2.12. de_bug

int de_bug(commentaire)

Action:

1/ Cette primitive peut permettre de détecter une mauvaise utilisation d'une des primitives du standard.
En effet, si le dernier appel à une primitive du standard c'est mal passé, l'appel de la fonction de_bug() provoquera l'impression sur < STDERR > d'un message expliquant l'erreur sinon elle n'a aucun effet.

Paramètres en entrée:

char	*commentaire	message optionnel qui s'imprimera avant le message

Exemples

```
if ( cr_dzv ( 1 , 10 , "char" ) < 0 )
        de_bug( "message facultatif" );

dzv_elt = cr_dzv ( 2 , 1 , "int" , 1 , "int" );
de_bug ( );
```

2.2.13. pr_fic

int pr_fic(output , nom)

Action:

1/ Permet d'écrire les caractéristiques d'un fichier du standard. Le fichier du standard à pour nom < nom > et les caractéristiques sont visualisées dans le fichier < output >.

Paramètres en entrée:

output	*char;	Fichier de sortie
nom	*char;	Fichier en entrée

Notes

1/ Le fichier de sortie peut faire référence aux sorties standard

- "output"
- "stderr"

2/ En cas d'anomalie un message sera affiché sur "stderr".

3/ Il existe une fonction système analogue < pr_fic > qui prend comme argument un nom de fichier du standard et qui affiche ses cractéristiques sur stdout.

Exemples

```
pr_fic ( "stdout", "essai1" );
pr_fic ( "stderr", "essai1" );
pr_fic ( "trace", "essai1" );
```

2.2.14. re_dzv

int re_dzv (dzv, nb_zone, taille, type)

Action:

1/ Permet de récupérer les caractéristiques des DZV. Cela signifie que pour un DZV donné la primitive retourne son nombre de zone (nb_zone), et pour chacune des zones ses caractéristiques :
- taille[num_zone] = la taille de la zone num_zone
- type [num_zone] = le type de la zone num_zone

Le numéro de la zone est un nombre compris entre 1 et nb_zone.
Le type de la zone est codifié par un caractère afin de faciliter des comparaisons. Cette codification consiste à prendre le premier caractère des types C prédéfinis (ex: 'i' pour "int").

Paramètres en entrée:

DZV	*dzv	le DZV en donnée.
int	*nb_zone;	nombre de zones du DZV
int	**taille;	tableau des tailles des différentes zones
char	**type;	tableau des types des différentes zones

Notes:

1/ L'allocation mémoire des tableaux "type" et "taille" est faite par la procédure. L'utilisateur doit passer des adresses de pointeurs.

2/ Les types des zones peuvent être :
'i' pour "int"
's' pour "short"
'f' pour "float"
'c' pour "char"
'd' pour "double"
'l' pour "long"

3/ Les premiers éléments des tableaux "taille" et "type" (taille[0] et type[0]) n'ont aucune signification.

Exemple

```
int   nb_zone;
int   *taille;
char  *type;
DZV   *dzv_elt;
dzv_elt = cr_dzv ( 2 ,  5 , "int" , "moyennes" ,
                          1 , "float" , "ecart_type" );
re_dzv( dzv_elt, &nb_zone, &taille, &type );
```

On a alors :

```
nb_zone  = 2
taille[ 1 ] = 5          type[ 1 ] = 'i'
taille[ 2 ] = 1          type[ 2 ] = 'f'
```

3. Maintenance du logiciel

3.1. La gestion des informations

Afin de permettre une compréhension plus facile des sources du logiciel, nous allons fournir dans les chapitres suivants un bref résumé des informations qui permettent de gérer un fichier du standard.
Ces informations ne sont pas accessibles par l'utilisateur. On assure ainsi une plus grande robustesse du logiciel.

3.1.1. Gestion simultanée de plusieurs fichiers.

Tout d'abord, le standard ne permet pas de gérer un fichier mais un ensemble de fichier. L'utilisateur identifie ses fichiers par un numéro qui lui est retourné par les primitives cl_fic (...) et cr_fic (...) .
Ce numéro est en fait retourné par les fonctions open() et creat() du langage C. On assure ainsi l'unicité de ces numéros.
Les informations gérées par le logiciel sur chacun des fichiers sont mémorisées dans une table. C'est bien entendu le numéro du fichier qui sert d'entrée dans cette table.
Cette table < INFO > , est une table de pointeurs sur une structure appelée
< INFO_FIC >.

Note: Quand on fait un < cl_fic(...) > on libère cette entrée qui pourra de nouveau servir pour un autre fichier.

3.1.2. Les informations attachées à un fichier.

A chaque ouverture ou création de fichier, on alloue dynamiquement une structure < INFO_FIC > qu'on initialise en fonction des paramètres donnes par l'utilisateur.

On trouve dans cette structure des informations qu'on peut classifier en 4 catégories :

1/ Les mêmes informations que celles du header de fichier.
- clef_ouverture = clef de contrôle pour identifier un fichier du standard
- clef_fermeture = clef de contrôle pour vérifier qu'un fichier est bien fermé.
- nb_elt = c'est le nombre d'élt dans le fichier
- adr_nb = La taille de la table d'accès direct
- adr_last = Le dernier élt du fichier
- type_fic = Le type du fichier (ex: "chaines")
- nb_fic_var = Le nombre de zones dans le DZV du fichier
- nb_elt_var = Le nombre de zones dans le DZV des éléments
- nb_mai_var = Le nombre de zones dans le DZV des maillons

2/ Des informations sur les dernières opérations effectuées :
- mode_open = Le mode d'ouverture du fichier
- elt_seek = Le dernier élt accédé avec un accès direct < ra_fic >
- mai_seek = Le dernier maillon accédé avec un accès direct

- at_eof = Fin de fichier
- at_eor = Le nombre de maillons qu'il reste à lire ou à écrire
- nb_elt_reel = Le nombre d'élt dans le fichier

2/ Des informations concernant la taille du fichier, des élts, des maillons.

- taille_head = Taille de l'en-tête du fichier
- taille_fic = Taille de la zone variable attachée au fichier
- taille_elt = Taille d'un élt (non compris ses maillons)
- tab_fic = Taille des différents
- tab_elt champs des informations variables
- tab_mai (calculé à partir des DZV)
- debut_fic = Adresse du début de fichier

4/ Des informations concernant la table d'accè direct

- adr = La table permettant les accès direct
- adr_taille = Sa taille en Mémoire Centrale

3.1.3. Comment sont détectées les erreurs.

L'utilisation de la primitive < de_bug (...) > permet de mettre en évidence un certain nombre d'erreurs commises par l'utilisateur du standard.
Ce sont les premières instructions de chacune des primitives qui permettent cette détection d'erreurs.

Au premier abord, les tests effectués peuvent poser quelques problèmes de compréhension. En fait, il s'expliquent quand on connaît la signification exacte des variables < mode_open > et < at_eor >.
Ces variables permettent de savoir quelles ont été les dernières opérations et par conséquent ce qu'il est possible de faire.

1/ mode_open = 0 en cas de pipe
= 1 en cas de lecture séquentiel
= 2 en cas d'écriture séquentiel
= 3 en cas d'accès direct sur un élément (lecture ou écriture)
= 4 en cas d'accès direct sur un maillon (lecture ou écriture)

2/ at_eor = C'est le nombre de maillons qu'il reste à lire ou à écrire pour le dernier élément accédé.
On peut ainsi savoir si tous les maillons d'un élément ont été ecrit, ou si il reste des maillons à lire.

La compréhension des erreurs qui ne dépendent pas de ces variables est suffisament simple pour ne pas être détaillée.

3.2. La structure des fichiers

Elle a été choisie afin de permettre de concilier les accès directs et séquentiels.

3.2.1. L'accès direct

L'accès direct est réalisé en utilisant un seul fichier. Quand on n'est pas dans le cadre d'un pipe, on gère une table < ADR > qui contient l'adresse relative des éléments par rapport au début du fichier.
On ne peut connaître à l'ouverture la place qui sera nécessaire à cette table, cette remarque implique que cette table soit stockée à la fin du fichier. Cette table est donc gérée en mémoire centrale tant que le fichier n'est pas fermé.
Si on n'est pas dans le cadre d'un "pipe", avant la fermeture du fichier, on va écrire cete table en fin de fichier.

Remarque: La taille allouée en mémoire centrale à cette table peut en cas d'adjonction d'élément ne plus être suffisante. On est alors obliger de réallouer de la place. Cette opération est chère, aussi pour en minimiser la fréquence, on alloue généralement plus de place que nécéssaire. Cette taille supplémentaire est égale à la constante SUP_ELT.

3.2.2. L'accès séquentiel

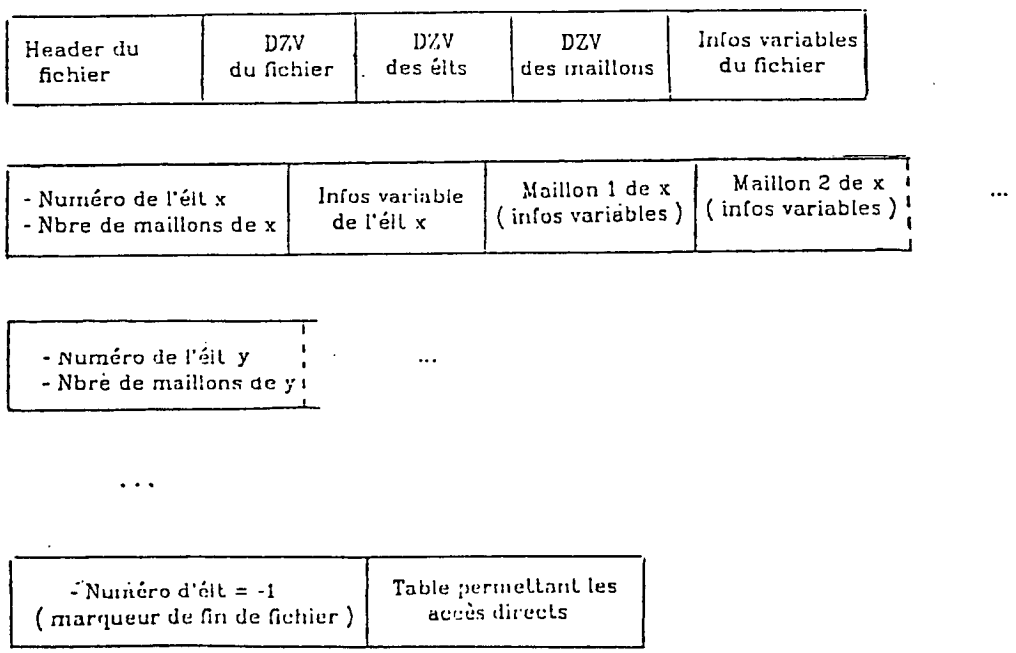
Pour permettre au fichier du standard d'être "pipable" l'accès à l'information doit pouvoir être fait séquentiellement.
Cette remarque impose l'organisation suivante pour le standard :

On trouve au début des informations générales. Ensuite sont stockés les éléments et les maillons dans l'ordre dans lequel ils ont été donnés par l'utilisateur. En fin de fichier on trouve la table qui permet les accès directs.

3.2.3. Structure du fichier

L'organisation du fichier est imposée par les possibilités d'accès séquentiels et directs.

Shématiquement:



3.3. Maintenance

3.3.1. Avertissement

Il faut prendre garde de ne pas effectuer des modifications qui rendraient incohérents les fichiers existants du standard...
Nous proposons dans la suite de ce chapitre quelques modifications qu'on pourrait avoir envie de réaliser sur le standard.

3.3.2. L'adjonction d'un type de fichiers , ex: "chaines"

Cette opération peut se faire simplement mais nécessite de modifier le souce du standard. Il faut modifier la fonction < d_type_fic( ) > dans le fichier < divers.c >.

Attention :

Le nom du type ne doit pas dépasser 20 caractères, y compris celui de fin de chaînes de caractères.

3.3.3. Comment réaliser la suppression d'éléments.

Il n'y a pas de méthode permettant de supprimer physiquement un élément.
La solution qui consiste à recréer un fichier n'est pas acceptable, on préferera donc une solution qui consiste à laisser des "trous" dans le fichier.
Cela signifie que quand on détruit un élément, on l'invalide en lui mettant par exemple une clef égale à -2. On doit auusi supprimer son entrée dans la table d'accès direct.
Il ne faut pas oublier de modifier les procédures de lecture qui doivent en cas d'accès séquentiel laisser la lecture d'un élément invalidé transparente a l'utilisateur.

Note:

La destruction d'un élément est un moyen "simple" de permettre à l'utilisateur de modifier le nombre de maillons d'un élément. En effet cela peut être simuler en invalidant l'élément et en en créant un nouveau qui aurait les mêmes caractéristiques (excepté bien entendu le nombre de maillons).
Si cela est réalisé, il sera certainement nécessaire d'écrire une procédure permettant de tasser des fichiers ... On s'expose sinon à obtenir des fichiers dont la taille ne correspond plus à celle de l'information.

3.3.4. Les problèmes de portabilités

Le logiciel a été ecrit en langage C sous le système UNIX et ne devrait donc pas poser trop de problèmes de portabilités.

Les procédures présentant un risque :

De nombreuses procédures ont un nombre variable de paramètres.
Nous nous sommes attachés à rendre l'utilisation de ces paramètres indépendantes du compilateur utilisé. Nous allons néanmoins détailler comment ils sont utilisés au cas où surviendrait un problème.

1/ cr_dzv(...)

Les paramètres variables de cette fonction peuvent être de différents types :

- des entiers passés par valeur.
- des chaînes de caractères (donc passage par adresse).

On a supposé que le compilateur genère du code qui au moment de l'appel empile la valeur des entiers et l'adresse des chaînes. Ces entiers et ces adresses sont

supposés avoir pour taille respective celle d'entiers et celle de pointeurs.

2/ Les autres procédures.
Tous les paramètres variables sont de types pointeurs. On suppose alors que quel que soit le type d'un pointeur il utilise la même taille en mémoire.
Les paramètres variables de toutes ces procédures sont traités par les procédures d_écrire_variable(...) et d_lire_variable(...) .

Note :

On essaie de tester, dans le cas des paramètres de type pointeur, si il n'en manque pas lors de l'appel.
Cela est simple à détecter quand le code généré par le compilateur empile le pointeur NULL à la fin de la liste des paramètres. Dans ce cas on pourrait même tester si il n'y a pas eu trop de paramètres lors de l'appel.
On suppose que ce n'est pas le cas, mais on vérifie néanmoins si ce type de paramètre n'est pas par hasard NULL, cela signifie alors qu'il y a eu une erreur.

3.3.5. Modifications des constantes du standard

On peut sans problème modifier certaines constantes du logiciel.

1/ SUP_ELT .
C'est la dimension supplémentaire qui est accordée à la table d'accès direct en cas de débordement de celle ci. A chaque fois que le logiciel détecte un débordement il fait un realloc( ). Cette valeur à donc intérêt à être la plus grande possible à condition qu'on ait suffisamment de place en mémoire centrale ...

2/ MAX_FICHIERS.
C'est le nombre maximum de fichiers autorisé par le standard, il est fixé a 30 et ne devrait donc pas être sujet à modification ...

3/ PROTECTION.
C'est le mode de protection par défaut associé aux fichiers.
- lecture et écriture pour l'utilisateur.
- lecture pour tous les autres.

Sa codification est la même que celle du paramètre de la fonction creat(...) du langage C sous UNIX.

4. CONCLUSION

Nous proposons dans ce rapport une standardisation pour la représentation de toutes les primitives extraites par les algorithmes de segmentation d'image. Les grandes classes de primitives sont les chaînes de contour, les segments de droite et les régions. La caractéristique principale de ces classes est la représentation sous forme de listes d'éléments. L'utilisation de ce standard permet de simplifier la programmation et résoud le problème de la portabilité des données.

Après une brève présentation des caractéristiques du standard, on donne dans le chapitre II tout ce qu'il est utile de connaître pour pouvoir l'utiliser sans problèmes. Les nombreux exemples donnés en annexe I devraient compléter efficacement cette description. La programmation en C permet de rester proche de l'esprit d'Unix en particulier pour le retour des erreurs. La possibilité d'avoir aussi bien l'approche de lecture séquentielle donc "pipable", que l'approche de lecture directe (aléatoire) semble à notre avis une particularité très intéressante de ce standard.

Ce manuel d'utilisation permet de répondre à toutes les questions que peut se poser un utilisateur du standard pour la création de ses programmes ainsi que l'échange de données. Il répond aux trois questions principales :

- présenter les possibilités du standard
- montrer comment l'utiliser
- aider à le maintenir

Note : les sources en langage C des programmes de ce standard sont disponibles. La demande doit en être faite à G. Giraudon à l'adresse indiquée.

ANNEXES

ANNEXE I

Exemples d'utilisations des primitives du standard.

ANNEXE II

Les messages d'erreurs de la primitive de-bug().

ANNEXE I

Exemples d'utilisations des primitives du standard.

- Création et construction d'un fichier avec des accès séquentiels
- Ouverture et relecture d'un fichier avec des accès séquentiels
- Utilisation de fichiers "pipables"
- Utilisation des accès directs sur les éléments, tests d'existence
- Utilisation des accès directs sur les maillons
- Modification des informations variables attachées au fichier
- Test des caractéristiques d'un fichier
- Visualisation des caractéristiques d'un fichier

ANNEXE I

Cette annexe à pour but de montrer sur quelques exemples comment utiliser les primitives permettant de manipuler les fichiers.
Tous ces exemples manipulent un même type de fichier dont voici la description des zones variables.

- Informations attachées à l'ensemble du fichier :

    - 1 libellé de 40 caractères.

- Informations attachées aux éléments :

    - 1 entier : "moyenne".
    - 1 entier : "écart–type".

- Informations attachées aux maillons :

    - 1 tableau de 2 réels : "coordonnée".
    - 1 entier : "couleur".

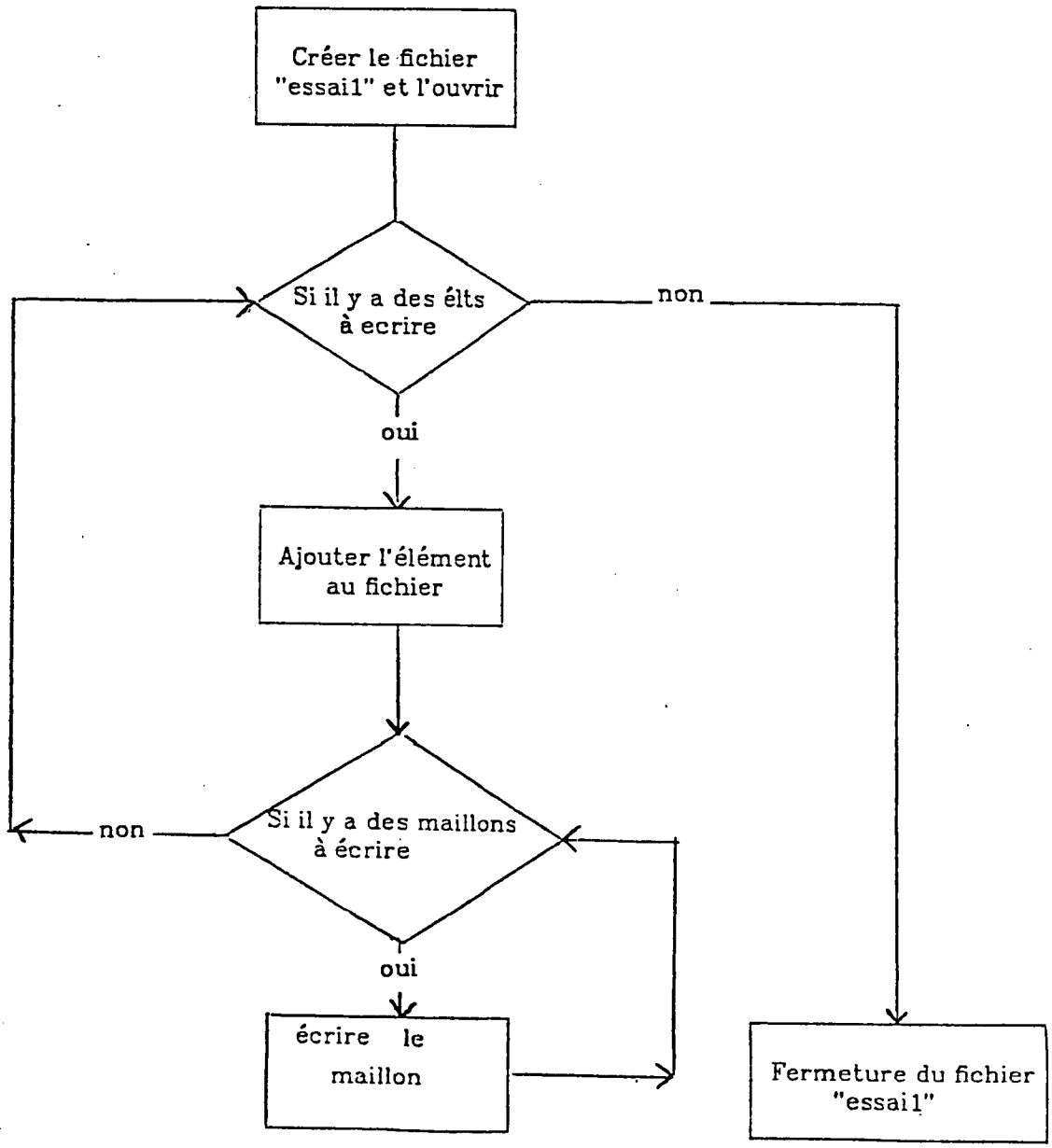
Les différentes variables qui représentent en mémoire centrale ces informations sont :

- moyenne, écart–type.
- coordonnée, couleur.
- nb–elt : c'est le nombre d'élements dans le fichier.
- nb–maillon : c'est le nombre de maillons de l'élément courant.

Dans les exemples, on simulera l'initialisation de ces variables par
variable = ... ;

EXEMPLE D'ECRITURES SEQUENTIELS

Ce que fait le programme qui suit :



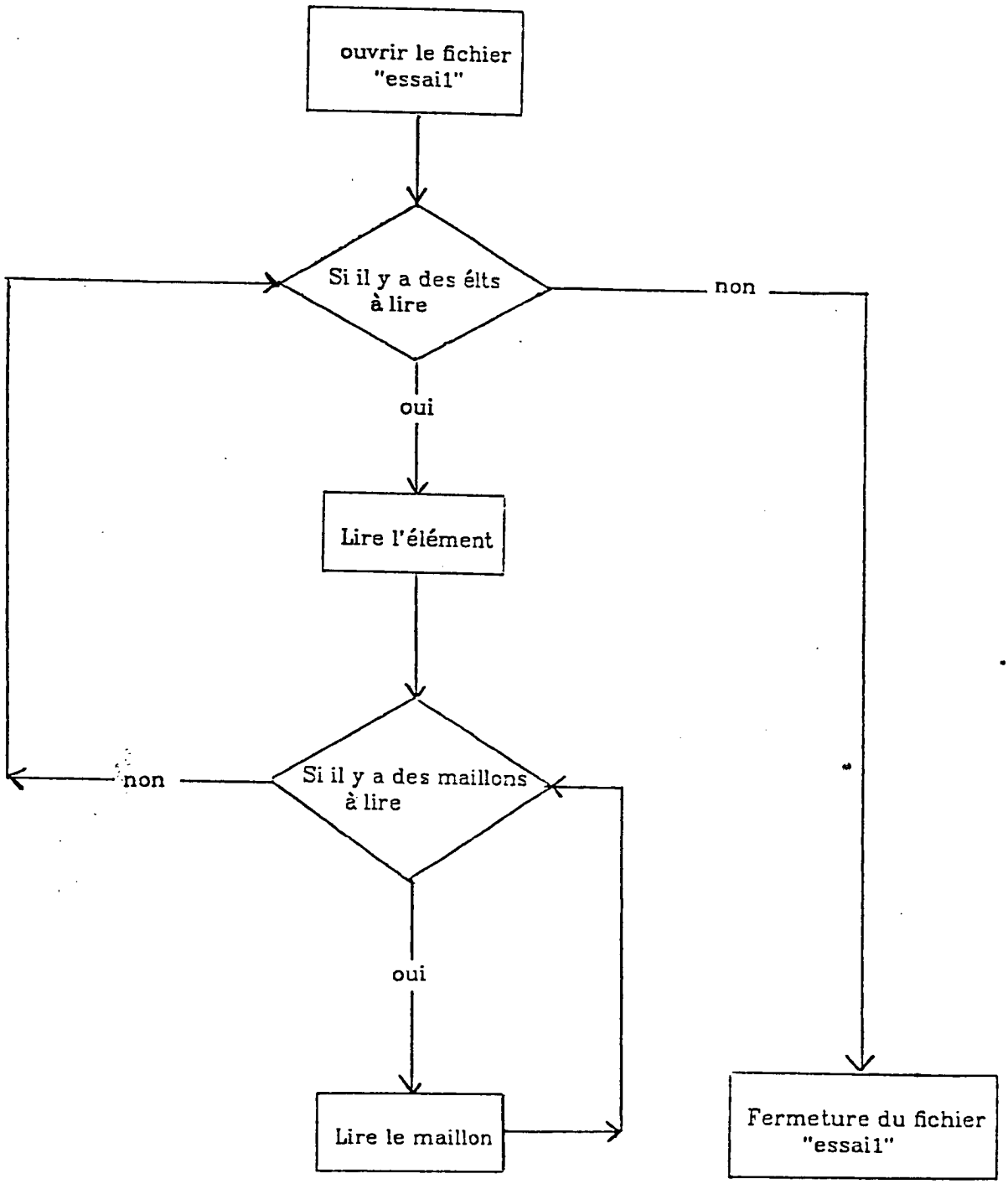
EXEMPLE D'ACCES SEQUENTIELS, L'ECRITURE.

```
#include <lib_dzv_image>

main()

{ /*....Déclaration d'un identificateur de fichier .....*/

 int fd;

 /*....Déclaration de pointeurs sur des descripteurs de zones variables..*/

 DZV *dzv_fic, *dzv_elt, *dzv_mai;

 /*....Variables de travail .....*/

 int i, j;
 int num_elt, nb_maillon, moyenne, écart_type, couleur;
 int coordonnée[ 2 ];


 /*....Initialisation des descripteurs de zones variables .....*/

 dzv_fic  = cr_dzv ( 1,   40, "char" , "date" );
 dzv_elt  = cr_dzv ( 2,   1, "int" , "moyenne",    1, "int" ,"ecart_type" );
 dzv_mai = cr_dzv ( 2,    2, "float", "coordonnees" ,    1, "int" , "niveau de gris"

 /*....Création d'un fichier de nom "essai1" de type "chaines" ..*/
 /*....qui contient environ 100 elts.....*/

 fd = cr_fic ( "essai1", "chaines", 100, dzv_fic, dzv_elt, dzv_mai, "Infos");
```

```
for ( i=1; i < ...; i++)

    { num_elt     = ... ;
      nb_maillon = ... ;
      moyenne     = ... ;
      écart_type = ... ;

      /*.... Ecriture de l'entête et des zones variables d'un elt .....*/

      wr_elt ( fd, num_elt, nb_maillon, &moyenne, &écart_type );

      /*.... Ecriture de ses maillons .....*/

      for ( j=1; j <= nb_maillon, j++ )

          { coordonnée[ 0 ] = ...;
            coordonnée[ 1 ] = ...;
            couleur          = ... ;
            wr_mai ( fd, coordonnée, &couleur );
          }
    }
  cl_fic ( fd );
}
```

EXEMPLE DE LECTURES SEQUENTIELS

Ce que fait le programme qui suit :



EXEMPLE D'ACCES SEQUENTIELS, LA LECTURE.

```
#include <lib_dzv_image>

main()
{ /*....Déclaration d'un identificateur de fichier .....*/
  int fd;
  /*....Déclaration de pointeurs sur des descripteurs de zones variables..*/
  DZV *dzv_fic, *dzv_elt, *dzv_mai;
  /*....Variables de travail .....*/
  int   nb-elt, num_elt, nb_maillon, moyenne, écart_type, couleur;
  int   coordonnée[ 2 ];
  char zv_fic[40];
  char *type;

  /*....Ouverture d'un fichier de nom "essai1" .....*/
  fd = op_fic ("essai1", &type, &nb_elt, &dzv_fic, &dzv_elt, &dzv_mai,
            zv_fic );
  /*.... Lecture de l'entête et des zones variables d'un elt .....*/
  while ( re_elt ( fd, &num_elt, &nb_maillon, &moyenne, &écart_type ))
     {
       /*.... Lecture ses maillons .....*/
       while ( re_mai ( fd, coordonnée, &couleur ) )
          {
            ...
          }
     }
  cl_fic ( fd );
}
```

L'UTILISATION DU PIPE.

Le standard permet d'écrire des programmes pipables.
Ces programmes doivent néanmoins remplir plusieurs conditions :

1/ Ne pas utiliser la primitive ra-fic( )

2/ Ne pas utiliser la primitive wr-fic( )

3/ Eviter d'utiliser le nombre d'élt retourner par op-fic( ). Celui ci serait en effet le nombre qui a été donné lors de la création et pas forcément le nombre d'élément du fichier.

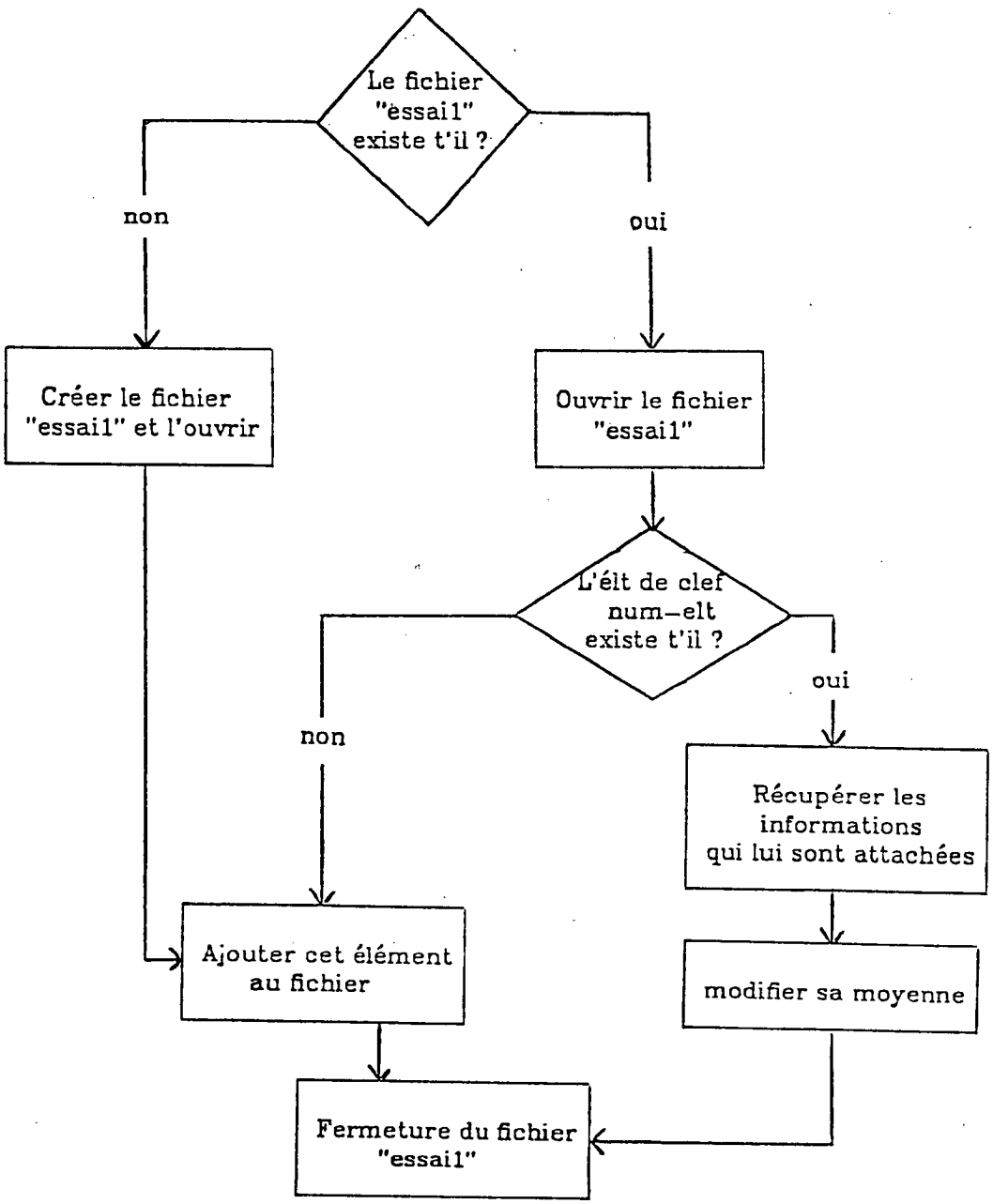
4/ Le fichier en sortie doit s'appeler "stdout".
Exemple : cr-fic("stdout", ...);

5/ Le fichier en entrée doit s'appeler "stdin".
Exemple : op-fic("stdin", ...);

Les deux premiers exemples de l'annexe I, sont de bons exemples de programmes pipables.

EXEMPLES D'ACCES DIRECTS.

Ce que fait le programme qui suit :



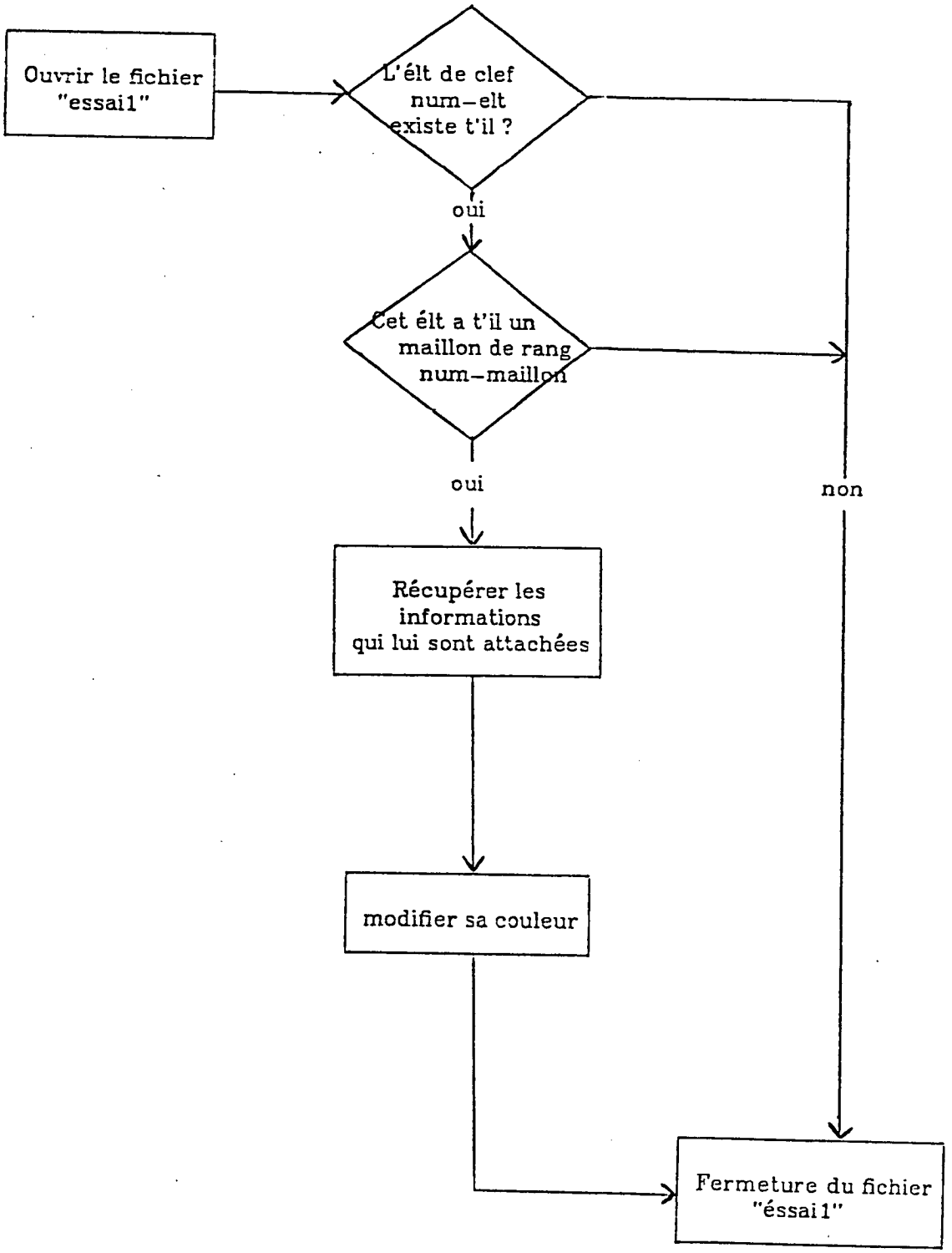
EXEMPLE D'ACCES DIRECTS.

```
#include <lib_dzv_image>

main()

{ /*....Déclaration d'un identificateur de fichier .....*/

  int fd;

  /*....Déclaration de pointeurs sur des descripteurs de zones variables....*/

  DZV *dzv_fic, *dzv_elt, *dzv_mai;

  /*....Variables de travail .....*/

  int     nb-elt, num_elt, nb_maillon, moyenne, écart_type;
  char   zv_fic[40];
  char   *type;

  /*....Ouverture d'un fichier de nom "essai1" si il existe .....*/
  /*....sinon creation.....*/

  if ( ( fd = op_fic ("essai1", &type, &nb_elt,
                      &dzv_fic, &dzv_elt, &dzv_mai, zv_fic )) < 0)
      {
       /*....On doit créer un fichier .....*/
       /*....Initialisation des descripteurs de zones variables .....*/

       dzv_fic  = cr_dzv ( 1,   40, "char" , "date" );
       dzv_elt  = cr_dzv ( 2,   1, "int" , "moyenne",    1, "int" ,"ecart_type" );
       dzv_mai = cr_dzv ( 2,    2, "float", "coordonnees" ,
                                  1, "int" , "niveau de  gris" );

       /*....Création d'un fichier de nom "essai1" de type "chaines" qui .....*/
       /*....contient environ 100 elts.....*/

       fd = cr_fic ( "essai1", "chaines", 100, dzv_fic, dzv_elt, dzv_mai,
                     "Infos");

      }
```

```
else  if ( ra_fic ( fd, num-elt, 0 ) )
      {
      /*.... L'élt existe,lecture de l'entête et des zones variables .....*/

      re_elt ( fd, &num_elt, &nb_maillon, &moyenne, &écart_type ))

      /*.... modification de sa moyenne .....*/
      moyenne = ... ;
      ra_fic ( fd, num-elt, 0 ) ;
      }

 /*.... modification ou écriture de l'elt suivant le cas .....*/

 wr_elt ( fd, num_elt, nb_maillon, &moyenne, &écart_type );

 cl_fic ( fd );
}
```

EXEMPLES D'ACCES DIRECTS.

Ce que fait le programme qui suit :

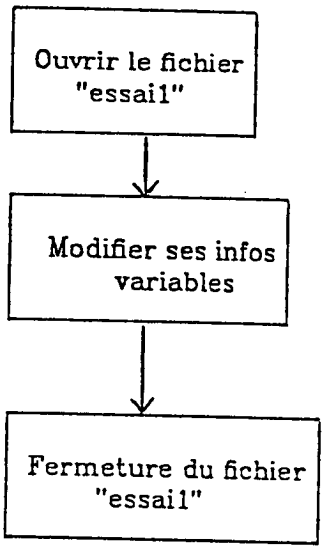


EXEMPLE D'ACCES DIRECTS SUR LES MAILLONS.

```
#include <lib_dzv_image>

main()

{ /*....Déclaration d'un identificateur de fichier .....*/

  int fd;

  /*....Déclaration de pointeurs sur des descripteurs de zones variables....*/

  DZV *dzv_fic, *dzv_elt, *dzv_mai;

  /*....Variables de travail .....*/

  int     nb-elt, num_elt, num_maillon, couleur;
  int     coordonnée[ 2 ];
  char    zv_fic[40];
  char    *type;

  /*....Ouverture d'un fichier de nom "essai1" .....*/


    op_fic ("essai1", &type, &nb_elt,
                    &dzv_fic, &dzv_elt, &dzv_mai, zv_fic )


    if ( ra_fic ( fd, num-elt, num-maillon ) )
       {
        /*.... Le maillon existe,lecture de ses zones variables .....*/

        re_mai ( fd, coordonnee, &couleur ))

        /*.... modification de sa couleur .....*/

        couleur = ... ;

        ra_fic ( fd, num-elt, num-maillon ) ;


        re_mai ( fd, coordonnee, &couleur ))
       }
  cl_fic ( fd );
}
```

EXEMPLE DE MODIFICATION DES INFOS VARIABLES DU FICHIER.

Ce que fait le programme qui suit :

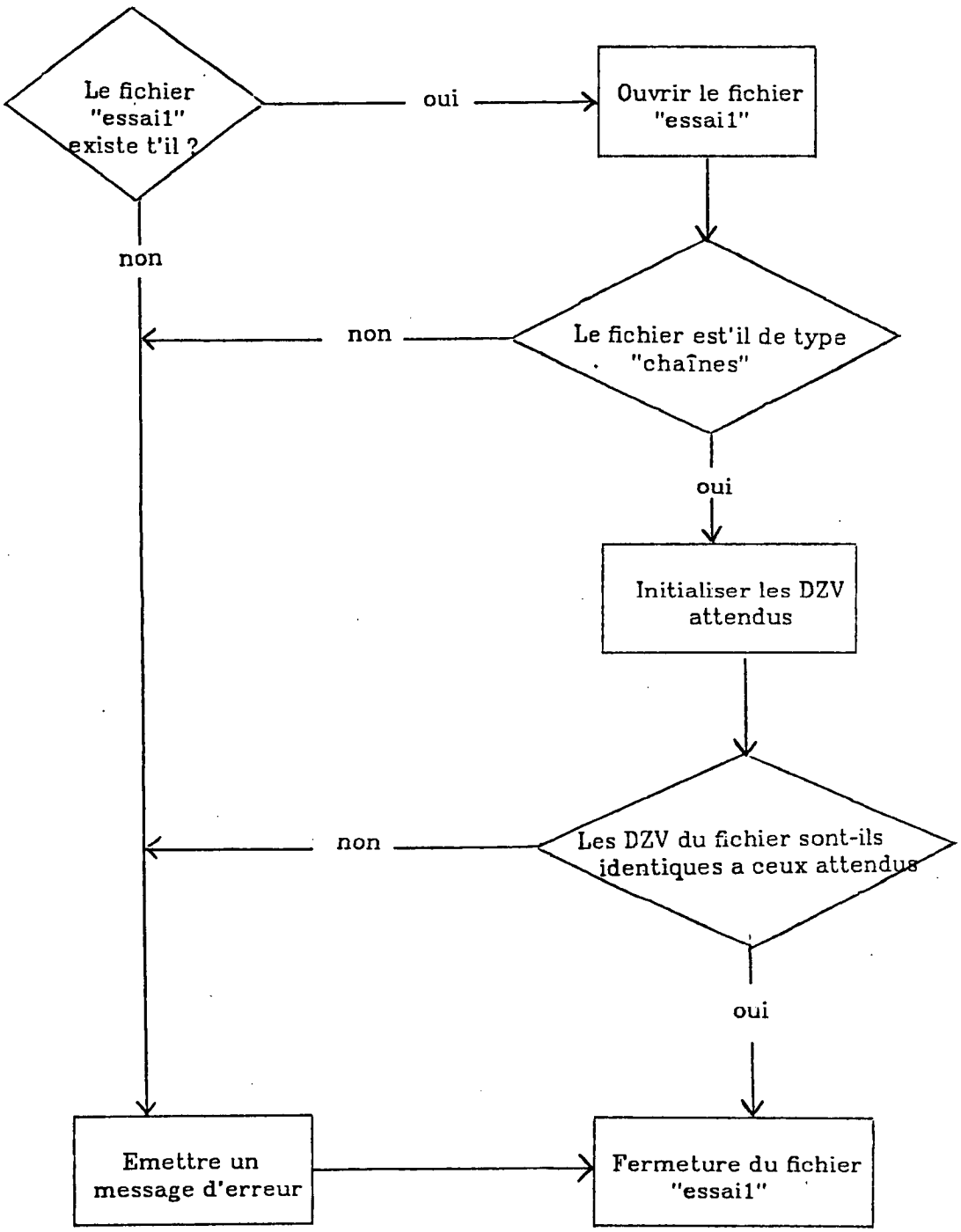


EXEMPLE DE MODIFICATION DES INFOS VARIABLES ATTACHEES AU FICHIER.

```
#include <lib_dzv_image>

main()

{ /*....Déclaration d'un identificateur de fichier .....*/

  int fd;

  /*....Déclaration de pointeurs sur des descripteurs de zones variables....*/

  DZV *dzv_fic, *dzv_elt, *dzv_mai;

  /*....Variables de travail .....*/

  int     nb-elt;
  char   zv_fic[40];
  char   *type;

  /*....Ouverture d'un fichier de nom "essai1" .....*/


    op_fic ("essai1", &type, &nb_elt,
                      &dzv_fic, &dzv_elt, &dzv_mai, zv_fic )



  /*.... Modification des infos variables.....*/

  wr_fic ( fd, "new infos" )

  cl_fic ( fd );
}
```

EXEMPLES DE CONTROLES SUR LES CARACTERISTIQUES D'UN FICHIER

Ce que fait le programme qui suit :



EXEMPLES DE CONTROLES SUR LES CARACTERISTIQUES D'UN FICHIER.

```
#include <lib_dzv_image>

main()

{ /*....Déclaration d'un identificateur de fichier .....*/

  int fd;

  /*....Déclaration de pointeurs sur des descripteurs de zones variables....*/

  DZV *dzv_fic, *dzv_elt, *dzv_mai;
  DZV *dzv_fic2, *dzv_elt2, *dzv_mai2;

  /*....Variables de travail .....*/

  int     nb-elt;
  char   zv_fic[40];
  char   *type;

  /*....Ouverture d'un fichier de nom "essai1" si il existe .....*/
  /*....sinon émission d'un message d'erreur.....*/

  if ( ( fd = op_fic ("essai1", &type, &nb_elt,
                      &dzv_fic, &dzv_elt, &dzv_mai, zv_fic )) < 0)
      {
       printf( " le fichier n'existe pas ..." );
      }

  if ( strcmp( "chaines", type ) )
      {
       printf( " le type du fichier n'est pas celui attendu ...");
      }
```

```
/*.... Initialisation des DZV attendus ..... */
 dzv_fic2  = cr_dzv ( 1,   40, "char" , " " );
 dzv_elt2  = cr_dzv ( 2,    1, "int" , " ",     1, "int" ," " );
 dzv_mai2 = cr_dzv ( 2,    2, "float", " " ,    1, "int" , " " );


 if ( ! ( cp_dzv ( dzv_fic, dzv_fic2 ) )  ||
    ( ! ( cp_dzv ( dzv_elt, dzv_elt2 ) )  ||
    ( ! ( cp_dzv ( dzv_mai, dzv_mai2 ) )  )
          {
           printf( "un des DZV ne correspond pas à ceux attendus ..." );
          }
 cl_fic ( fd );
}
```

EXEMPLE DE VISUALISATION DES CARACTERISTIQUES D'UN FICHIER.

```
#include <lib_dzv_image>

main()

{

 /*....Visualisation sur stderr d'un fichier de nom "essai1" ..... */

   pr_fic ("stderr", "essai1" )

}
```

Execution du programme :

```
OUVERTURE DU FICHIER essai1

   type du fichier  = chaines
   nombre d'elements = 500
   le dernier elt   = 500

LE DESCRIPTEUR DE 1 ZONES VARIABLES ASSOCIE AU FICHIER

   40  char  => commentaire

LE DESCRIPTEUR DE 1 ZONES VARIABLES ASSOCIE AUX ELEMENTS

   1  int   => moyenne
   1  int   => ecart_type

LE DESCRIPTEUR DE 1 ZONES VARIABLES ASSOCIE AUX MAILLONS

   2  float => coordonnees
   1  int   => niveau de gris
```

Remarque :

On aurait pu obtenir le même résultat en effectuant la requête système suivante :

```
> pr-fic essai1
```

Le résultat aurait été affiché sur "stdout".

ANNEXE II

Cette annexe fournit des explications supplémentaires qui peuvent permettre de mieux comprendre les messages d'erreurs écrits par la primitive < de-bug >.

Il est à noter que la primitive < de-bug > retourne deux types d'erreurs.

1/ Les premiers sont des messages en Anglais qui sont en réalité des messages générés par la fonction < perror > du langage C.
Les primitives du standard font en effet appel à des fonctions C telles que open(), read(), write(), seek() et close().
Au cas où ces fonctions retourne le code "-1" la primitive du standard mise en cause retourne également "-1".

2/ Les autres sont des messages en Français et correspondent à des erreurs détectées directement par les primitives du standard.
Dans ce cas la primitive du standard mise en cause retourne un entier négatif différent de "-1".

On va détailler dans la suite chacun des messages d'erreurs en Français en indiquant

- son numéro.
- son libelle.
- une explication complémentaire.

-2= "paramètre de description zone variable incorrect"
Il se peut qu'on ait essayé de créer un DZV en indiquant un nombre de zones négatif ou plus probablement que l'un des types de zone variable ne fait pas parti des types prédéfinis en C.
Les types possibles : "int", "float", "short", "char", "long", "double".

-3= "Commentaire incorrect dans un cr_dzv()."
Chaque zone d'un descripteur de zone variable est caractérisée par un facteur de répétition, un type de zone et un commentaire. Ce commentaire est obligatoire et ne doit pas dépassé 256 caractères.

-4= "Liste de paramètres variables incorrecte"
La liste des paramètres variables fournie à la primitive ne correspond pas au DZV. La liste complète des paramètres doit être donnée.

-5= "type de fichier incorrect"
Le type du fichier indiqué lors de l'ouverture ne correspond pas aux types prédéfinis.
Les types prédéfinis: "chaines", "contours", "metas", "segments", "regions"

-7= "RE.. impossible en creation , utilise ra-fic.."
Alors que la dernière opération à consisté à créer un fichier ou à y ajouter un nouvel élément, on essaie de faire une lecture. Ce n'est pas possible à moins d'utiliser un accès direct.

-8= "WR.. impossible en ouverture , utilise ra-fic.."
Alors que la dernière opération à consisté à ouvrir un fichier ou à y lire un élément, on essaie de faire une écriture. Ce n'est pas possible à moins d'utiliser un accès direct.

-9= "operation impossible avec un pipe ..."
On a essayé d'utiliser la primitive "ra-fic" ou "wr-fic" alors que le fichier correspond à "stdin" ou "stdout".

-10="organisation du fichier incoherente ... fichier close?"
Deux possibilités :
- le fichier qu'on ouvre n'est pas un fichier creer par le standard.
- le fichier n'a pas été refermé.

-11="ra-fic : paramètre numéro de maillon incorrect."
On essaie de faire un accès direct sur un maillon qui n'existe pas, ou bien le numéro de l'élément correspond à un nouvel élément mais le numéro de maillon était différent de 0.

-12="nombre incorrect d'accès aux maillons d'un elt ."
En utilisation séquentiel, c'est à dire quand on fait un READ() ou un WRITE() non précédé d'un RA-FIC() on est obligé de lire (ou écrire) tous les maillons.
Quand on ajoute un nouvel élément, on est obligé d'écrire tous ses maillons (en accès séquentiel et direct).

-13="tentative d'ecriture de deux éléments de meme clef"
On ne peut écrire deux éléments de même clef. Pour modifier un élément, il faut faire preceder la primitive d'écriture par un RA-FIC().

-14="requete incompatible avec le dernier seek"
Deux possibilités :
- On demande de lire ou écrire un élément alors que l'on vient de demander un accès direct sur un maillon.
- On essaie d'écrire un élément dont la clef d'accès est différente de la demande d'accès direct qui a précédé.

-15="tentative de lecture d'elt out of file"
On demande la lecture d'un élément alors que la derniere lecture a déjà indiqué qu'on était en fin de fichier.

-20="clef d'accès au fichier incorrecte"
Le premier paramètre (la clef d'accès au fichier) n'est pas valide. Le fichier a t'il été correctement ouvert ou créé ?

-30="Nombre de fichiers maximum atteint ..."
On essaie d'utiliser simultanément plus de fichiers que ne le permet le langage C.

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique