Henri Lesourd
~
Implémentation du système Hammourabi - *le moteur*

Toute technique est mise au point, utilisée, importante, obsolète, normalisée puis comprise.

Introduction

Dans ce papier, nous discutons de l'implémentation du moteur de notre système *Hammourabi*. Dans la partie 1, nous nous intéressons d'abord à la mémoire, nous discutons du moteur lui-même dans la partie 2, et nous en décrivons le sous-système d'entrées-sorties dans la partie 3.

1. La mémoire

1.1. Un aperçu succinct sur le typage ; lien avec la représentation

La représentation sous forme de *listes attribut-valeur* est, sous une forme ou sous une autre, une des plus fréquemment utilisées en informatique. Elle est une des manières possibles de représenter des *objets* dans un ordinateur. Dans les langages compilés tels que le Pascal ou le C, on a l'habitude de définir de nouveaux *types* en définissant des descripteurs d'enregistrement, qui sont la définition de tels types, et servent ensuite de patrons pour la construction des objets.

Par exemple, dans les deux langages C et Pascal :

```
typedef struct {
   char *Nom,*Prenom;
   int Age;
}
TPersonne;
```

```
Type
   TPersonne=Record
      Nom,Prenom : String;
      Age : Integer;
   End;
```

Ces déclarations de types sont ensuite en pratique utilisés comme des descripteurs de format qui servent de moule pour la construction des instances. Les instances d'objet sont alors représentées sous la forme de vecteurs contigus dans la mémoire, par exemple :

```
main() {
   Personne *p=malloc(sizeof(Personne));
   p->Nom="Dupont";
   p->Prenom="Jean-Pierre";
   p->Age=23;
}
```

nous donne :

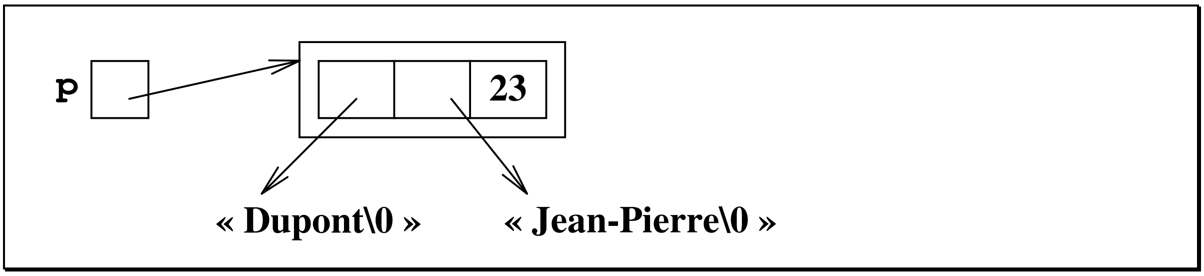

Figure 1 : la représentation des objets classique dans les langages typés statiquement

Dans les langages dits *à objets*, le descripteur de type est utilisé non seulement pour décider du format qu'aura l'objet en mémoire, mais aussi pour décider des actions autorisées ou non sur cet objet, et pour effectuer le choix de la bonne procédure à employer quand on demande l'exécution d'une action en donnant son nom (ceci est aussi appelé un *envoi de message*). Le polymorphisme à la C++ ou à l'ADA est une autre manière d'aborder le problème, dans ce cas on se sert du *prototype* de l'appel de fonction plutôt que de la hiérarchie des classes pour en déduire la procédure à laquelle on doit se référer. L'appel de clause dans Prolog est lui aussi basé sur un principe similaire. Mais nous ne discuterons pas plus ici du problème de la liaison et de la désambiguïsation des messages ; nous notons simplement que la notion de type peut être vue comme ayant un lien avec ce problème-là, et est effectivement exploitable pour les aborder en pratique.

Ce que nous pouvons dire, c'est que la représentation habituelle des objets dans la mémoire d'un ordinateur est bien adaptée au traitement de ces objets du point de vue de leur création et de l'accès rapide à leurs champs. Elle est bien adaptée aussi à la *modification* des champs et à la *destruction* des objets. Elle est par contre beaucoup moins bien adaptée à l'*indexation* de ces objets, ainsi qu'à l'*ajout et à la suppression dynamique* de champs dans ces objets. C'est ce dont nous allons parler à présent.

1.2. Indexations snarkiennes

La *liste attribut-valeur* à proprement parler se définit classiquement comme une liste chaînée, dans laquelle chaque élément de liste sert à coder un champ de l'objet, et contient d'une part une référence sur le nom de ce champ (l'*attribut*), et d'autre part une référence sur la *valeur* de ce champ. De telles listes attribut-valeur se représentent facilement par des triplets de pointeurs, dans un environnement où on utilise d'autre part fréquemment une table de symboles servant à factoriser les chaînes de caractères.

Ainsi, l'instance d'objet que nous avons décrite ci-dessus :

```
P:(NOM:DUPONT PRENOM:JEAN-PIERRE AGE:23)
```

Peut se représenter comme ceci sous forme de LAV :

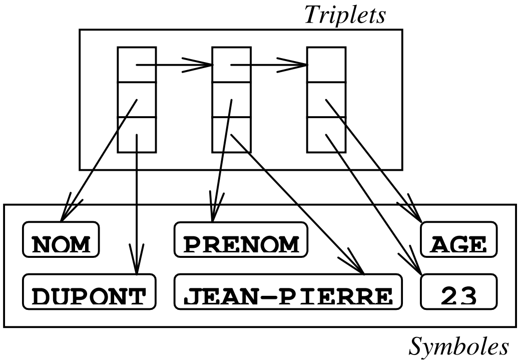

Figure 2 : la représentation LAV classique

Tout comme la représentation LISP sous forme de paires CONS, la représentation LAV est mieux adaptée à la programmation symbolique que celles utilisée dans les langages typés statiquement : en particulier, les objets n'ont pas de forme rigide et déterminée à l'avance, et il est possible d'y ajouter de nouveaux champs ou d'en supprimer facilement. Intégrer des fonctionnalités équivalentes à un langage statiquement typé serait extrêmement problématique, à cause de l'accès précalculé et positionnel aux champs des objets qui y est mis en œuvre. En revanche, ce que l'on gagne de ce côté-là, on le perd de l'autre, puisque dans un tel cadre, l'accès aux champs ne se fait plus en temps constant.

Une manière élégante d'aborder ce problème consiste à choisir une représentation complètement symétrique pour les triplets constituant les objets et d'indexer les composants en y associant simplement la liste de leurs occurrences dans la base. Dans le cas de notre exemple, nous supposerions qu'existent dans la base :

- ♦ Les symboles `NOM`,`PRENOM`,`AGE`,`DUPONT`,`JEAN-PIERRE` et `23` ;
- ♦ L'objet `P`, qui peut être vu comme un symbole anonyme.

Et nous représenterions la LAV :

```
P:(NOM:DUPONT PRENOM:JEAN-PIERRE AGE:23)
```

Par :

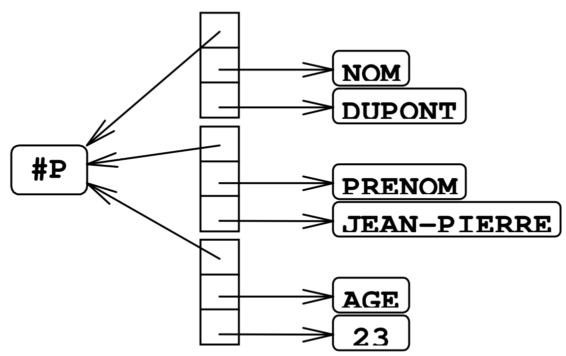

Figure 3 : la représentation snarkienne déclarative classique

Nous obtenons ici une représentation de type réseau sémantique, qui *dissocie clairement* l'aspect déclaratif, purement informationnel associé aux données, de l'aspect procédural qui sert à faire effectivement fonctionner l'ensemble. A notre connaissance, ce type de représentation a été utilisé pour la première fois dans le moteur *Snark* [LAU86]. En Snark, notre ami *Jean-Pierre Dupont* serait représenté par un objet `#P` comme ceci :

```
#P NOM DUPONT
#P PRENOM JEAN-PIERRE
#P AGE 23
```

D'autre part, la représentation est complètement symétrique ; on ne parle plus d'objet contenu dans un champ d'un autre objet : les triplets sont utilisés pour créer des *associations* liant trois objets qui sont respectivement l'objet contenant, l'attribut, et la valeur (voir par exemple [LEE63], [SIK82]). En pratique, il faut ensuite trouver un moyen pour éviter de devoir parcourir toute la mémoire de façon séquentielle quand l'on cherche à accéder un objet. Ceci se fait facilement en stockant dans chaque objet la liste de ses occurrences, c'est à dire la liste des triplets qui mentionnent cet objet. Ainsi, quand l'on rajoute un triplet concrétisant l'association entre 3 objets A, B et C, on crée ce triplet et l'on ajoute dans chacune les listes d'occurrences associées à A, à B et à C un élément pointant ce triplet. Si ces listes d'occurrence sont concrétisées par des listes chaînées, nous obtenons :

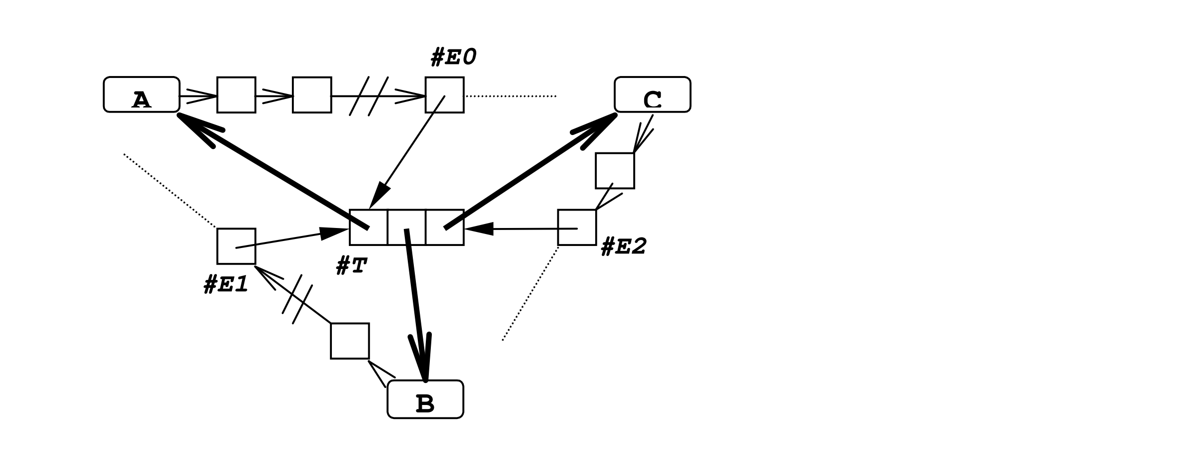

Figure 4 : la représentation snarkienne indexée (1)

Donc pour créer une association entre les objets A, B et C (donnée dans cet ordre dans le schéma ci-dessus), nous avons du créer le triplet `#T`, et les trois éléments de liste chaînée `#E0`, `#E1` et `#E2`. Comme ces quatre éléments interviennent toujours ensemble, nous pouvons les agréger en un seul (en supposant des listes d'occurrences doublement chaînées) :

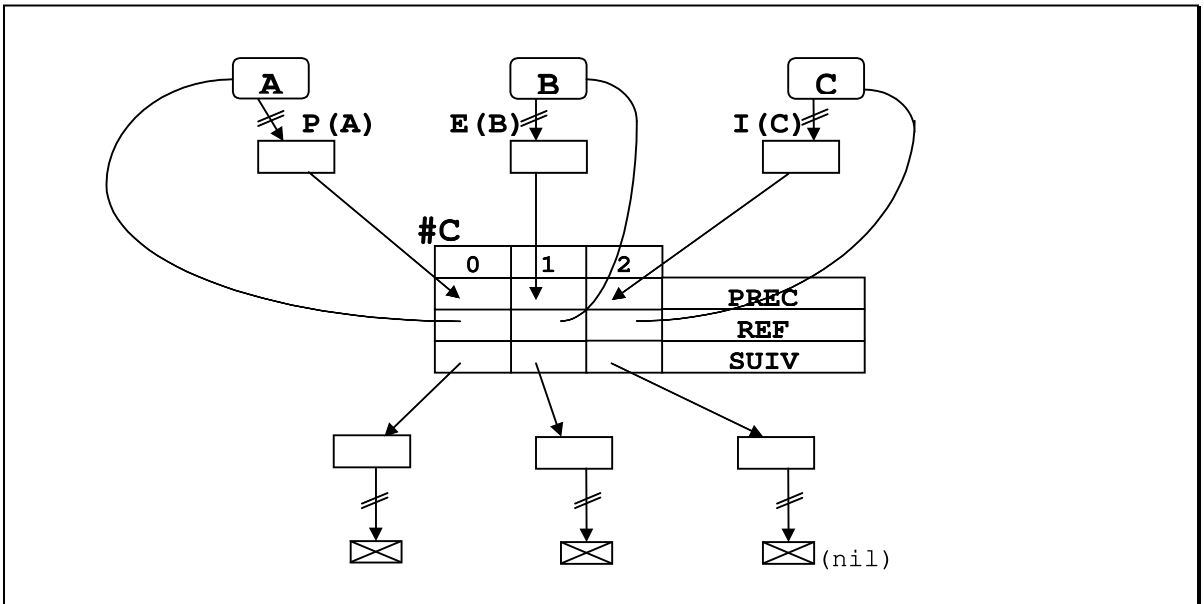

Figure 5 : la représentation snarkienne indexée (2)

Nous appelons l'objet agrégé `#C` un *codon*. Dans un tel codon, les éléments de liste doublement chaînée `#Ei` se retrouvent concrétisés par les cases `PREC(i)` et `SUIV(i)`, et l'on n'a plus besoin du pointeur correspondant au champ CAR d'une liste LISP dans ces éléments, puisqu'ils sont directement donnés à partir du codon. Le triplet `#T` se retrouve pour sa part dans les trois champs `REF(0)`, `REF(1)` et `REF(2)`.

Nous pouvons alors décrire les objets *symbole* et *codon* comme suit (à la C++) :

```
class SYMBOLE {
   char *Nom;
   CODON *I,*E,*P;
};
class CODON:SYMBOLE {
   CODON PREC[3];
   CODON SUIV[3];
   SYMBOLE REF[3];
};
```

Le *symbole* est en réalité utilisé pour représenter tous les types d'objets, y compris des objets anonymes (dans ce cas-là, le champ *Nom* est laissé vide). Dans un symbole S, les listes de codons I, E et P pourraient être fusionnées en une seule, qui contiendrait la liste de tous les codons mentionnant S dans le tableau *REF*. Nous avons adopté un système un peu plus sophistiqué, dans lequel cette liste de tous les codons mentionnant S est répartie en trois listes :

- *La liste P*, qui est la liste des codons dont *REF[0]* vaut S ;
- *La liste E*, qui est la liste des codons dont *REF[1]* vaut S ;
- *La liste I*, qui est la liste des codons dont *REF[2]* vaut S.

En pratique dans *Hammourabi*, nous avons donné aux trois positions possibles dans le tableau *REF* une signification bien précise, liée au fait que les objets sont construits comme des hiérarchies de parties :

- *La liste P* est utilisée pour coder la liste des *parties* de l'objet S ; à l'intérieur d'un codon, le champ *REF[0]* désigne l'objet *contenant* ce codon ;
- *La liste E* est la liste de tous les codons dans lesquels le symbole S joue le rôle d'étiquette ;
- *La liste I* est utilisée pour coder la *liste des pointeurs inverses* de l'objet S, qui est la liste de tous les codons *contenant* S ; à l'intérieur d'un codon, le champ *REF[2]* désigne l'objet « contenu » dans ce codon, c'est à dire celui qui joue le rôle de la valeur dans la relation liste attribut-valeur. C'est par conséquent à ce champ *REF[2]* que l'on se réfère pour définir la liste des pointeurs inverses.

Nous discuterons plus loin de l'intérêt de cette répartition de la liste des codons mentionnant l'objet S en trois listes I,E et P. Nous empruntons l'expression *pointeur inverse* à François Pachet [PAC92], et n'en donnons d'ailleurs ici qu'une version appliquée au codage des données LAV dans la mémoire d'un ordinateur. De façon plus générale, c'est du concept d'*objet utilisé par* un autre objet dont il est question ; ce concept est à notre avis un concept important en informatique, qu'on retrouve dans bien d'autres contextes que l'indexation d'une base de faits, par exemple dans le domaine de la synchronisation d'agents lors de l'accès à une ressource partagée [GUE96, pp 60 et suivantes]. Enfin, notons que la première implémentation connue de Snark était d'abord faite pour fonctionner en chaînage avant, et utilisait des triplets réduits au tableau *SUIV* de notre représentation [VIA85].

<u>*Méta*</u> : La représentation donnée en figure 4 date de la première année de notre thèse, à l'époque d'un premier système très inspiré de Snark. La représentation donnée en figure 5 est une reformulation personnelle de la précédente, mais elle est le fruit de nombreuses et anciennes discussions entre nous et M. V. Lecerf, qui nous ont amené à définir le concept de *groupe* d'informations : les LAV dont nous avons parlé jusqu'à présent sont de tels groupes au sens où il est possible d'insérer un objet dans plusieurs groupes à la fois, *sans qu'aucun de ces groupes soit privilégié* (cf. *à contrario* la notion de lien dans UNIX, qui n'est pas équivalente à celle d'inclusion d'un fichier dans un directory). La pratique actuelle en matière de conception de systèmes et langages considère plutôt la notion de *lien*, de *désignation*, les problèmes de *nommage* [KRA87, pp 202 et suivantes], et est donc plutôt centrée sur une notion de type *pointeur*. Ce concept de groupe est à notre avis un concept récurrent, et est important pour qui s'intéresse à la conception des machines, des systèmes, langages et systèmes de représentation : avec le groupe, nous structurons les objets stockés en machine en utilisant des éléments de *multigraphe*, et non plus en utilisant de simples liens, qui sont des éléments de graphe.

1.3. Pointeurs inverses

Il reste que la représentation que nous avons présentée au paragraphe précédent est assez coûteuse en place mémoire. Nous aurions besoin de pouvoir indexer aussi des objets du type de ceux donnés en figure 1. Pour ce faire, une manière simple de procéder consiste à stocker dans l'objet, contiguement à chaque occurrence d'un champ pointeur, le *pointeur inverse* qui lui correspond, concrétisé par un pointeur de type *SUIV* dans la représentation de la figure 1. Il faut aussi que l'objet pointé soit muni d'un pointeur vers la liste de ses pointeurs inverses, c'est à dire vers le champ inverse du premier objet contenant une occurrence de cet
objet pointé. Si nous supposons que l'objet pointé est le symbole « Dupont », et trois instances de *TPersonne* appartenant à la famille dupont, nous écririons le type *TPersonne* comme ceci :

```
typedef struct {
   void *i_Nom;
   symbole *Nom;
   void *i_Prenom;
   symbole *Prenom;
   int Age;
}
TPersonne;
```

```
typedef struct {
   void *i;
   char *Valeur;
}
symbole;
```

Et les trois personnes *Jean-Pierre*, *Samantha* et *Siméon Dupont* seraient alors représentées comme ceci en mémoire :

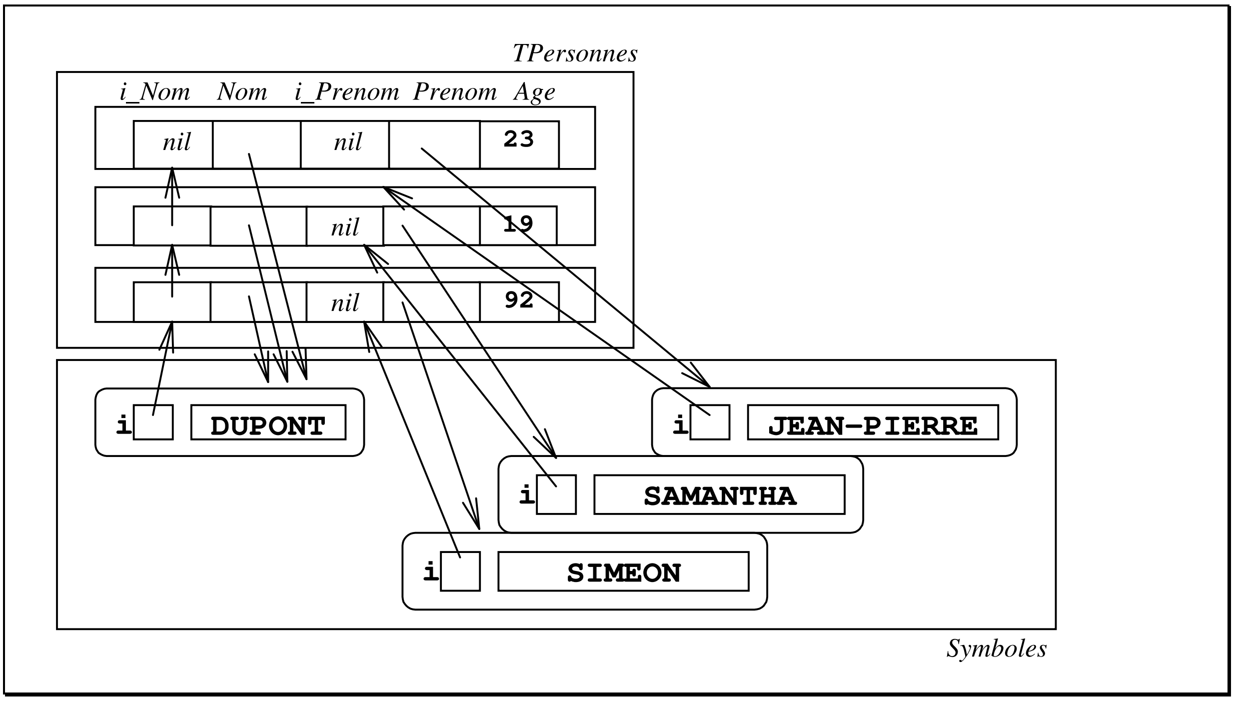

Figure 6 : la représentation snarkienne indexée (3-1)

Il nous manque encore deux éléments importants. Le premier est que les chaînages inverses sous forme de liste simplement chaînée sont trop inefficaces si l'on considère le fait que cette liste chaînée doit être mise à jour à chaque modification de la valeur d'un champ pointeur, ce qui implique le fait d'enlever l'objet de la liste inverse de l'ancienne valeur pointée, et donc un parcours linéaire en temps si cette liste est simplement chaînée. Il est bien préférable d'utiliser des chaînages doubles :

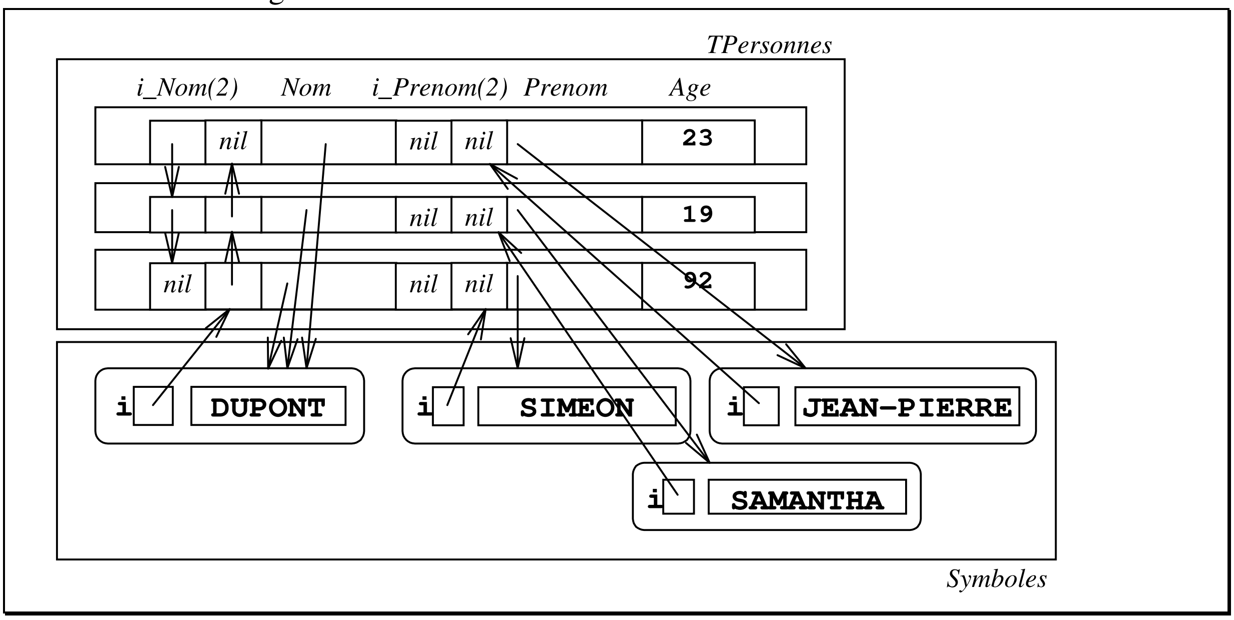

Figure 7 : la représentation snarkienne indexée (3-2)

Ce codage est fortement réminiscent de celui donné dans [VIA85, p. 83], qui code les objets en utilisant des triplets ne contenant que des pointeurs inverses, comme ceci :

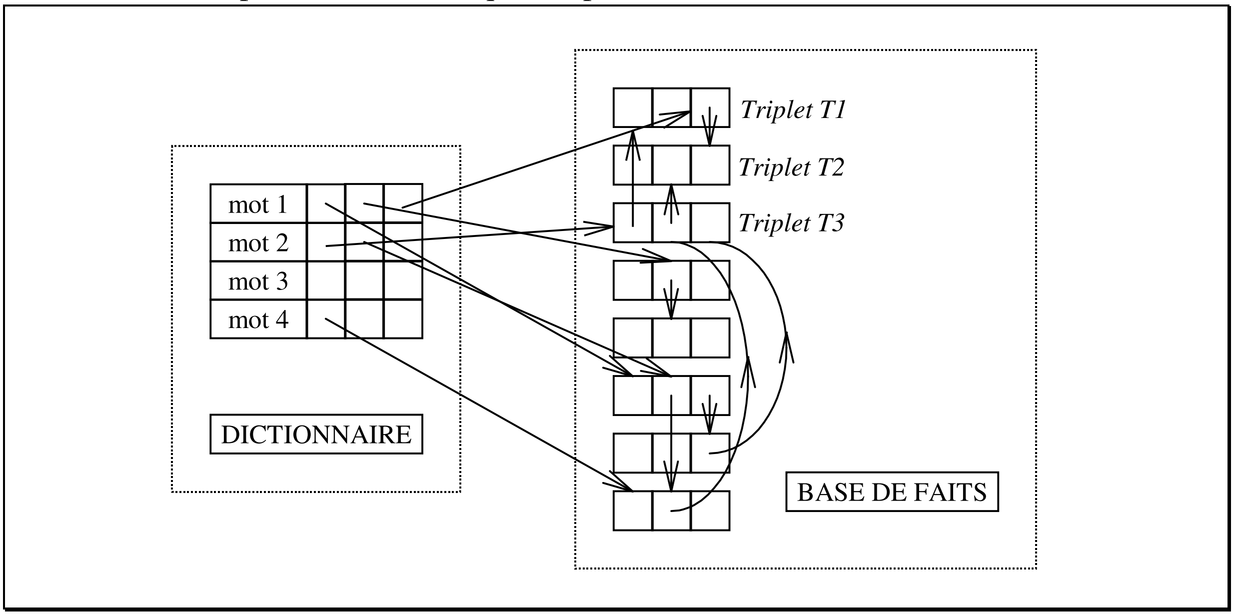

Figure 8 : la représentation snarkienne originale

Le second élément important que nous mentionnions plus haut est le suivant : avec ce type de système, les pointeurs désignent directement des champs et pointent donc *à l'intérieur des objets*. Par exemple, si les adresses des triplets *T1*, *T2* et *T3* ci-dessus sont respectivement 12, 24 et 36, les valeurs des objets correspondants sont (l'emploi du *point d'interrogation* signifiant la valeur inconnue, et *nil* la valeur vide des pointeurs) :

```
12:(?,?,32)
24:(?,?,?)
36:(12,28,nil)
```

Et les contenus des entrées de dictionnaire correspondant à *mot1* et *mot2* sont :

```
("mot1",?,?,20)
("mot2",36,?,?)
```

Dans le cas de Snark où il n'y a qu'un type d'objet, ceci n'est pas gênant. Mais s'il y a plusieurs types d'objets possibles, et que l'interpréteur ne connaît pas ces types d'objets à l'avance, il faut un moyen général pour pouvoir retrouver rapidement le début d'objets tels que ceux montrés en figure 6 et 7 à partir d'un pointeur désignant un champ *à l'intérieur* de ces objets. Ceci se fait classiquement en utilisant un mécanisme d'allocation *par page*, où l'on regroupe les objets de même classe en pages de taille standardisée (typiquement, 1024, 2048 octets). De telles pages sont alignées sur la taille standard, et connaissant un pointeur dans une page, il suffit d'annuler les bits de poids faible correspondant à la taille standard des pages pour avoir le début de la page, où sont situées les informations sur la classe commune à tous les objets de la page, à partir desquelles on peut tirer en particulier la taille d'une instance (*les instances sont supposées avoir toutes la même taille*). A partir de là, il devient facile de retrouver le début d'un objet connaissant un pointeur à l'intérieur de l'objet, puisqu'on connaît sa taille, et qu'on sait que les objets sont eux-mêmes alignés à l'intérieur de la page par rapport à la taille d'une instance. Supposons par exemple une page d'objets située à l'adresse 1024, les objets ayant tous une taille de 12 octets, et étant alloués à l'offset 24 dans la page. La zone 0-23 située en début de page servant à stocker les informations sur la classe :

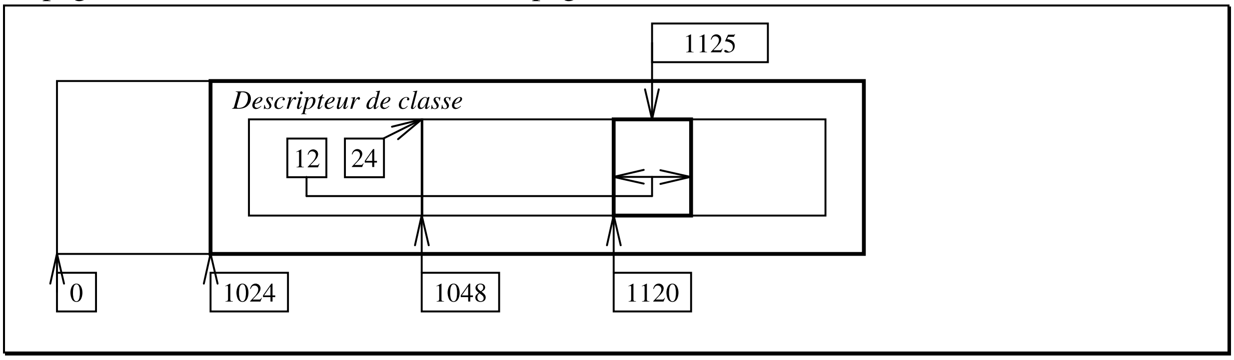

Figure 9 : allocation par page

Si nous supposons un pointeur égal à 1125, pour calculer le début de l'objet, nous effectuons les calculs suivants :

- *Annuler les 10 bits de poids faible* (le 10 correspond au log à base 2 de la taille des pages, qui est ici 1024 ; $\log_2(1024)=10$) de la valeur 1125 du pointeur. On obtient l'adresse du debut de page, qui est 1024 ;
- *Accéder aux informations* sur la taille des instances (12) et à celle donnant le début de la zone d'allocation de la page (24) ;
- *Calculer l'offset du pointeur 1125* par rapport au début de la zone d'allocation des instances dans la page : 1125-1024-24=77 ;
- *Calculer la division entière* de cet offset par la taille commune aux instances : 77 div 12=6 ;
- *Nous savons à présent* que le pointeur 1125 pointe à l'intérieur du sixième objet de la page, dont l'offset dans la page est donc 6*12=72 (en supposant les objets directements consécutifs les uns aux autres dans la page), et dont l'adresse absolue est alors 1024+24+72=1120.

Notons enfin qu'il est souhaitable de ne pas (comme nous l'avons fait pour les besoins de l'explication) répéter dans chaque page les informations concernant la classe des instances de cette classe contenues dans la page. Il convient de factoriser le descripteur *DC* d'une classe *C*, en le désignant dans les différentes pages dédiées aux instances de cette classe *C* sous la forme d'un pointeur pointant *DC*, situé en début de page.

Nous avons supposé une compilation statique des classes et n'avons pas parlé d'attributs au sens des figures 4 et 5, mais il va de soi qu'on pourrait rajouter dans les objets des champs symboles correspondant aux attributs, indexés par le système de pointeurs inverses décrit ici. Nous obtenons alors la *représentation LAV compactée*.

Si nous faisons le bilan, nous obtenons pour l'objet *(NOM:DUPONT PRENOM:JEAN-PIERRE AGE:23)* les tailles suivantes, en nombre de pointeurs consommés (sans compter la taille des symboles) :

Représentation statique (figure 1)	3
LAV classique (figure 2)	9
Snarkienne indexée 1 (figure 4)	36
Snarkienne indexée 2 (figure 5)	27

Snarkienne indexée 3-1 (figure 6)	6
Snarkienne indexée 3-2 (figure 7)	9
Snarkienne originale (figure 8)	9
LAV compactée	18

1.4. Gestion de la persistance

Quand la base de faits se met à grossir, le problème du temps de chargement et du recalcul des données au démarrage commence à se poser. Nous avons donc implémenté la base de faits de notre moteur sous la forme d'une véritable base de données persistante, en utilisant des fichiers mappés UNIX. Il est assez facile de réaliser soi-même les allocations mémoire des objets à l'intérieur d'un gros bloc mémoire, qui sera ensuite directement associé à un fichier au moment du boot du moteur, qui redémarrera donc en utilisant le *core image* de la session précédente : à part la première fois, il n'y a donc plus besoin de réinitialiser la mémoire, de lire des fichier d'init, *exit* les insupportables méthodes *LOAD* et *SAVE* dont on oublie toujours une occurrence, qu'on ne sait pas toujours quand et comment déclencher, qui nécessitent des parcours, etc.

Pour pouvoir faire ceci, il vaut bien mieux que les références d'objets ne soient plus des pointeurs, mais des indices par rapport au début du bloc mémoire qui concrétise la base. De cette manière, la base est *relogeable* et peut être réallouée à n'importe quel emplacement dans la mémoire de l'ordinateur. Ceci est extrêmement utile, parce que les primitives fournies pour les fichiers mappés ou la mémoire partagée ne garantissent justement pas que d'une session à l'autre, la base sera remappée au même endroit. C'est d'autre part indispensable dès que l'on veut que deux programmes *différents* puissent mapper la même base (puisque dans ce cas-là, l'adresse à laquelle la base est mappée dépend en fait de la taille de l'exécutable). Ceci nous amène à décrire nos types comme ceci :

```
typedef int ID;
class SYMBOLE {
   ID Nom;
   ID I,E,P;
};
...........
```

au lieu de :

```
class SYMBOLE {
   char *Nom;
   CODON *I,*E,*P;
};
...........
```

En C, nous devrons alors écrire :

```
ID symbol_id;
printf("%s\n",(char*)&MEM[((SYMBOLE *)&MEM[i])->Nom]);
```

au lieu de :

```
SYMBOLE *s;
printf("%s\n",s->Nom]);
```

Ceci est assez illisible, mais n'est pas forcément un grave problème, nous pouvons le résoudre en écrivant une fonction :

```
char *symboleLireNom(ID i) {
   return (char*)&MEM[((SYMBOLE *)&MEM[i])->Nom];
}
```

puis:

```
ID symbol_id;
printf("%s\n",symboleLireNom(symbol_id));
```

Le second problème qui se pose, c'est alors d'être en mesure de tester dynamiquement le type d'un objet : supposons par exemple qu'existe simultanément le type *PERSONNE*, contenant lui aussi un champ *Nom* :

```
class PERSONNE {
   int Age;
   ID Prenom,Nom;
};
```

et qu'on dispose d'un *ID* dont on ne sait pas si il désigne une *PERSONNE* ou un *SYMBOLE*. Mis à part le cas (improbable et de toute manière inmaintenable) où les champs *Nom* ont le même emplacement dans les deux types d'objet, on est coincé. Il est donc nécessaire que le premier mot de tous les objets contienne leur type effectif. Soit :

```
enum {
   typeSYMBOLE,typePERSONNE;
};
class OBJET {
   int Type;
};
```

```
class PERSONNE:OBJET {
   int Age;
   ID Prenom
   ID Nom;
};
class SYMBOLE:OBJET {
   ID Nom
   ID I,E,P;
};
```

Nous pouvons alors écrire une version de *LireNom* valable pour tous les types :

```
char *LireNom(ID i) {
   OBJET *o=&MEM[i];
   switch (o->Type) {
      case typePERSONNE: return (char*)&MEM[((PERSONNE*)o)->Nom];
      case  typeSYMBOLE: return (char*)&MEM[((SYMBOLE *)o)->Nom];
   }
}
```

Par la suite, dans les programmes, nous utiliserons systématiquement des *ID* au lieu d'utiliser des *SYMBOLE** et des *PERSONNE** ; bien que nous programmions en C, la façon d'écrire les programmes devient ainsi beaucoup plus adaptée au traitement de données symboliques, et en fait très proche de la façon dont on peut écrire du LISP (les parenthèses en moins). Tous les langages à objets compilés font en réalité quelque chose d'analogue à ce que nous décrivons ici avec notre champ *Type* (c'est du moins certain dès qu'il y a des appels de méthodes dynamiques), mais ils nous posent le problème de coder l'équivalent du champ *Type* à leur manière, codage qui est à chaque fois dépendant de l'implémentation du compilateur. D'autre part, un appel de méthode comme :

```
CLASSE_RACINE *o;
o->Afficher();
```

Nécessite que la variable o soit un *pointeur* au sens C++ du terme, ce qui, nous l'avons vu, pose des problèmes avec les fichiers mappés (nous n'utilisons que des *ID* dans nos programmes, et aussi en lieu et place des pointeurs dans les champs des objets, pour la relogeabilité de la base). Nous avons donc choisi de procéder autrement, afin d'éliminer cette gestion des types dynamique qui s'est révélée être une boîte noire gênante dans le cœur même de l'accès mémoire de notre système.

Il nous reste à décrire la façon d'initialiser le tableau *MEM*, par exemple :

```
int *MEM;
void BOOT() {
   MEM=malloc(100000);
   INIT(MEM);
}
```

pour une utilisation basique, et

```
int *MEM;
void BOOT(boolean effacerBase) {
   MEM=mmap("base",filesize("base"));
   if (effacerBase) INIT(MEM);
}
```

pour une utilisation avec *core image* où l'espace mémoire utilisé est couplé avec le fichier "base" (version linux : on utilise un *mmap()*), et peut donc être récupéré à partir d'une session précédente.

2. Le moteur

2.1. Lecture et pattern-matching

La représentation que nous avons définie pour les listes attribut-valeur doit ensuite être complétée par des *méthodes* classiques pour la manipulation des listes, comme :

```
typedef int ID;
typedef ID IDSYMB,IDCODON;
IDSYMB listeCreer();
  void listeAjouterDebut(IDSYMB liste,IDSYMB attr,IDSYMB val);
  void listeAjouterFin(IDSYMB liste,IDSYMB attr,IDSYMB val);
  void listeDetruireUnSeulNiveau(IDSYMB liste);
  void listeDetruireRecursivement(IDSYMB liste);

enum { indI,indE,indP } IND;
IDCODON listeLirePremier(IDSYMB liste,IND i);

boolean codonEstPremier(IDCODON c);
     IDCODON codonPrecedent(IDCODON c,IND i);
     IDCODON codonSuivant(IDCODON c,IND i);
          ID codonValeur(IDCODON c,IND i);

void codonDetruire(IDCODON c);

IDCODON ChoisirPlusCourtContenu(IDSYMB obj,IDSYMB attr,IDSYMB val,IND *i);
```

Nous utilisons en fait le type *SYMBOLE* pour représenter tout type d'objet, ce qui fait que le symbole n'est pas forcément lié à une table de symboles sous sa forme habituelle (comme dans un interpréteur LISP ou un compilateur, par exemple). Dans le cas de types d'objets quelconques, comme des noeuds de sous-expression, par exemple, nous utilisons des objets *SYMBOLE* anonymes, qui peuvent être créés et détruits facilement. Les symboles véritables, non anonymes, sont en sus indexés dans une table de symboles hcodée, et on ne les supprime jamais. Ils sont supposés peu nombreux.

Comme nous l'avons dit plus haut, nous aurions pu utiliser simplement une seule liste, fusionnant les trois listes *I*, *E* et *P*. Mais avec ces trois listes, quand l'on recherche le codon correspondant à un triplet de symboles, il devient possible d'optimiser la recherche en choisissant celui de ces trois symboles qui contient la liste la plus courte. Par exemple, si l'on recherche le triplet *[A,B,C]*, que la liste *I* de *C* est de taille *12*, la liste *E* de *B* de taille *12* et la liste *P* de *A* de taille 6, alors il vaut mieux parcourir la liste des parties de *A* si l'on veut économiser du temps, puisque de toutes les façons, le codon éventuel correspondant au triplet *[A,B,C]* doit, s'il existe, figurer *simultanément* dans *P(A)*, *E(B)* et *I(C)*. C'est le premier codon de la liste des parties de *A* que renverrait alors la fonction *ChoisirPlusCourtContenu()* ci-dessus, dans cet exemple.

Pour que l'ensemble puisse fonctionner, il faut prévoir de stocker la taille courante des listes *I*,*E* et *P* dans l'objet symbole, soit :

```
class SYMBOLE:OBJET {
   ID Nom
   ID I,E,P;
   int TailleI,TailleE,TailleP;
};
```

Ces tailles courantes doivent être tenues à jour par les fonctions de manipulation des listes. La fonction *ChoisirPlusCourtContenu()* s'écrit alors :

```
IDCODON ChoisirPlusCourtContenu(IDSYMB obj,IDSYMB attr,IDSYMB val,IND *i) {
   int NP=TailleI(obj),NE=TailleE(obj),NI=TailleP(obj);
   if (NP<=NE && NP<=NI) return listeLirePremier(obj, *i=indP);
   else
   if (NE<=NP && NE<=NI) return listeLirePremier(attr,*i=indE);
                    else return listeLirePremier(val, *i=indI);
}
```

Nous pouvons ensuite étendre le procédé à plusieurs codons simultanément. Par exemple, supposons que nous recherchions :

```
(NOM:CALMENT PRENOM:JEANNE AGE:110)
```

Si notre base ne contient que des enregistrements décrivant des personnes décrites comme ci-dessus, il est fort probable que la liste des pointeurs inverses du nombre « 110 » est très courte : un moteur a donc intérêt à utiliser cette liste-là plutôt que la liste des pointeurs inverses de « JEANNE », qui est un prénom très fréquent. Il a encore moins intérêt à utiliser les symboles « NOM », « PRENOM » et « AGE », qui jouent le rôle d'étiquette et dont la liste *E* a une taille égale au nombre de personnes présentes dans la base.

A présent que nous avons défini notre mémoire et les méthodes qui servent à manipuler les objets qu'elle contient, nous allons montrer la façon dont fonctionne la partie « pattern matching » de l'interpréteur. Quand nous recherchons un objet en mémoire, nous souhaitons écrire la requête sous la même forme que celle donnée pour les objets, en remplaçant les parties non précisées par des variables (qui sont notées comme des symboles préfixés par le caractère '$'), par exemple :

```
$O:(NOM:$N PRENOM:$P AGE:110)
```

pour parcourir les listes de codons, nous définissons un objet itérateur :

```
class ITERATEUR:OBJET {
   ID Obj,Attr,Val;
   boolean oSpec,aSpec,vSpec;
   ID Cour;
   IND indCour;
};
```

Une requête telle que celle donnée ci-dessus est décomposée sous forme de triplets :

```
$O NOM $N
$O PRENOM $P
$O AGE 110
```

Chacun de ces triplets sera concrétisé par un itérateur. Les champs *Obj*,*Attr* et *Val* sont les symboles composant le triplet, y compris si ce sont des symboles de variable. Les flags *oSpec*, *aSpec* et *vSpec* servent à indiquer si les positions respectives correspondant aux trois champs *Obj*,*Attr* et *Val* étaient auparavant non instanciées, et l'ont été par cet itérateur (ces flags n'ont de signification que dans le cas des variables). A partir de là, nous pouvons écrire :

```
boolean EstVariable(o) {
   char *n=Nom(Obj(iter));
   return n!=NULL && *n=='$'
}
boolean &xSpec(ID iter,IND pos)
   switch (pos) {
      case indI: return &oSpec(iter);
      case indE: return &aSpec(iter);
      case indP: return &vSpec(iter);
   }
}
void iterateurReset(ID iter) {
   IND i;
   boolean b,oVar,aVar,vVar;
   oSpec(iter)=aSpec(iter)=vSpec(iter)=FALSE;
   oVar=EstVariable(Obj(iter));
   aVar=EstVariable(Attr(iter));
   vVar=EstVariable(Val(iter));
   Cour(iter)=ChoisirPlusCourtContenu(oVar?NIL:Obj(iter),
                                      aVar?NIL:Attr(iter),
                                      vVar?NIL:Val(iter),&i
                                     );
   indCour(iter)=i;
}
```

```
boolean Matcher(ID iter,IND pos,ID var,ID val) {
   if (EstVariable(var)) {
      if (xSpec(iter,pos)) EcrireValeurVariable(var,NIL),
                           xSpec(iter,pos)=FALSE;
      if (LireValeurVariable(var)!=NIL) {
         return LireValeurVariable(var)==val;
      }
      else {
         EcrireValeurVariable(var,val);
         xSpec(iter,pos)=TRUE;
         return TRUE;
      }
   }
   else return var==val;
}
boolean iterateurAvancer(ID iter) {
   ID o=Obj(iter),a=Attr(iter),v=Val(iter),c=Cour(iter);
   boolean resu=TRUE;
   if (Cour(iter)==NIL) return FALSE;
   resu&=Matcher(iter,indI,o,codonValeur(c,indI));
   resu&=Matcher(iter,indE,a,codonValeur(c,indE));
   resu&=Matcher(iter,indP,v,codonValeur(c,indP));
   Cour(iter)=codonSuivant(Cour(iter),indCour(iter));
   return resu;
};
```

Une *instruction de lecture* est ensuite représentée par une liste d'itérateurs ; les symboles de variable donnent accès à un environnement permettant de lire et d'écrire des valeurs (fonctions *Lire* et *EcrireValeurVariable*). Quand un itérateur réussit, l'itérateur courant devient le suivant. Quand le suivant est *NIL*, on récupère le sous ensemble des variables de l'environnement qui sont instanciées, et on a ainsi une solution à la requête. Quand il n'y a plus de possibilités pour un itérateur, on le réinitialise et l'itérateur courant redevient l'itérateur précédent dans la liste. Si l'itérateur précédent est *NIL*, la requête est terminée.

Il est aussi possible de grouper plusieurs instructions en un *bloc d'instructions*. Ceci devient nécessaire dès que les données deviennent suffisamment complexes pour qu'il ne soit plus possible de les représenter sous la forme de simples objets : en d'autres termes, ceci advient dès qu'on est amené à utiliser d'autres relations que la relation *partie-de* qui sert à agglomérer les champs des objets. Le bloc d'instructions est alors utilisé pour écrire des *requêtes complexes*, au moyen desquelles on fait des recherches dans la base de faits conçue comme un *graphe* d'objets. De telles requêtes complexes sont tout à fait semblables à celles qu'on rencontre en bases de données, construites au moyen de *jointures*. La jointure dans notre système passe par l'utilisation de variables désignant les objets eux mêmes, à partir desquelles il est possible de parcourir les listes inverses de ces objets. Supposons par exemple une organisation découpée en *DEPARTEMENTS*, et des *PERSONNES*, décrites comme ceci :

```
(TYPE:PERSONNE
   NOM:$N PRENOM:$P JOB:$J
   TRAVAILLE-DANS:$DEPARTEMENT
)
```

Nous pouvons aussi bien rechercher les *PERSONNES* dont l'attribut *JOB* vaut *MANAGER* en parcourant les départements :

```
$D:(TYPE:DEPARTEMENT NOM:$ND)
$P:(TYPE:PERSONNE TRAVAILLE-DANS:$D
    JOB:MANAGER NOM:$NP PRENOM:$PP)
   (_PRINT
      "NOM= " $NP "; " "PRENOM= " $PP "DEPT= " $ND
   )
```

En faisant ceci, nous parcourons la liste des départements, et parcourons ensuite la liste des liens inverses pour chacun de ces départements, à la recherche de triplets liant une personne et le département courant par la relation *TRAVAILLE-DANS*.

Mais nous pourrions aussi bien écrire :

```
$P:(TYPE:PERSONNE TRAVAILLE-DANS:$D
    JOB:MANAGER NOM:$NP PRENOM:$PP)
$D:(TYPE:DEPARTEMENT NOM:$ND)
   (_PRINT
      "NOM= " $NP "; " "PRENOM= " $PP "DEPT= " $ND
   )
```

En SQL on écrirait :

```
SELECT P.NOM,P.PRENOM,D.NOM
  FROM PERSONNES,DEPARTEMENTS
 WHERE P.DEPTNO=E.DEPTNO
```

Notons qu'en utilisant des itérateurs explicites de la façon qui a été décrite plus haut, nous avons la possibilité d'implémenter le pattern matching de façon *non récursive*. Cette remarque est importante, car une fois que ceci a été fait pour l'opération de lecture, on peut réutiliser le même schéma pour les autres opérations, et on aboutit finalement à dérécursiver tout le moteur. A partir de là, il devient possible de borner le temps maximal consommé par une étape d'exécution, ce qui nous amène à la perspective de doter notre moteur de capacités *temps-réel*. D'autre part, il est possible d'obtenir ainsi un entrelacement très fin des étapes d'exécution des acteurs, propriété utile à la bonne mise en œuvre de notre système d'espionnage (ceci était notre motivation de départ).

Une *INSTRUCTION* est représentée comme ceci dans le moteur :

```
enum {
   etatEnCours,etatSucces,etatTermine;
}
etat;
class INSTRUCTION:OBJET {
   ID Premier;
   ID Cour;
   etat Etat;
};
```

Le champ *Premier* est la liste des itérateurs constituant l'instruction. Le champ *Etat*, pour sa part, sert à indiquer l'état de l'instruction. L'instruction est alors initialisée comme ceci :

```
void instructionReset(ID instr) {
   ID l=Premier(instr);
   while (l) iterateurReset(Val(l)),l=Suivant(l);
   Etat(instr)=etatEnCours;
   Cour(instr)=Premier(instr);
}
```

Pour faire avancer l'instruction d'une étape, nous écrivons ensuite :

```
void instructionAvancer(ID instr) {
   if (Etat(instr)==etatEnCours) {
      ID cour=Cour(instr);
      ID iter=Val(cour);
      if (iterateurAvancer(iter)) {
         ID suiv=Suivant(cour);
         if (suiv!=NIL) Cour(instr)=suiv; else Etat(instr)=etatSucces;
      }
      else {
         if (Cour(iter)==NIL) {
            iterateurReset(iter);
            cour=Precedent(cour);
            if (cour!=NIL) Cour(instr)=cour; else Etat(instr)=etatTermine;
         }
      }
   }
}
```

L'instruction avance d'étape en étape, et les variables présentes dans cette instruction reçoivent des valeurs au fur et à mesure que le pattern matching progresse, ou sont au contraire remises à *NIL* en cas de retour arrière.

Ces variables appartiennent à un *environnement*, et les instructions dont nous parlons sont, comme nous l'avons dit regroupées en listes, *en blocs d'instructions*. Un *acteur* est pour nous l'assemblage d'un tel environnement et d'un bloc d'instructions.

2.2. Les acteurs dans le moteur

Les acteurs sont implémentés comme ceci dans le moteur :

```
class ACTEUR:SYMBOLE {
   ID Environnement,Instructions;
   ID Cour;
   etat Etat;
};
```

On peut remarquer que les acteurs dérivent du type *SYMBOLE*. Ce sont des objets de première classe dans notre système. Il existe un état *Stoppé* pour les acteurs, qui est celui qu'ils atteignent quand une de leurs instructions est interceptée par une *interdiction* (un "¿■"). La *fonction acteurAvancer()* est au départ implantée de façon analogue à *instructionAvancer()*, la seule différence notable étant qu'au début, elle affecte l'environnement courant à l'environnement de l'acteur qu'on lui passe en paramètre :

```
void acteurAvancer(ID act) {
   ID cour,instr;
   cour=Cour(act),instr=Val(cour);
   if (Etat(act)==etatEnCours) {
      EnvironnementCourant=Environnement(act);
      instructionAvancer(instr);
      if (Etat(instr)==etatSucces) {
         ID suiv=Suivant(cour);
         if (suiv!=NIL) Cour(act)=suiv; else Etat(act)=etatSucces;
      }
      else
      if (Etat(instr)==etatTermine) {
         instructionReset(instr);
         cour=Precedent(cour);
         if (cour!=NIL) Cour(act)=cour; else Etat(act)=etatTermine;
      }
      EnvironnementCourant=NIL;
   }
   else if (Etat(act)==etatSucces) {
      Etat(act)=etatEnCours;
      Etat(instr)=etatEnCours;
   }
   else Erreur("Acteur non actif");
}
```

La boucle principale du moteur s'écrit alors à peu près comme ceci :

```
ID Actifs,Stoppes;
void Scheduler() {
   ID act=Premier(Actifs);
   while (act!=NIL) {
      listeRetirer(&Actifs,act);
      acteurAvancer(Val(act));
      switch (Etat(Val(act))) {
         case etatStoppe: listeAjouterFin(&Stoppes,act);
                          break;
         case etatSucces: break;
        case etatEnCours: listeAjouterFin(&Actifs,act);
                          break;
        case etatTermine: acteurDetruire(Val(act));
                          Detruire(act);
                          break;
      }
      act=Premier(Actifs);
   }
}
```

A ce stade de la description, nous disposons d'un environnement dans lequel le traitement de plusieurs *requêtes* peut être entrelacé au cours du fonctionnement d'un moteur d'inférence. Nous allons voir maintenant comment il est possible d'enrichir cet environnement pour qu'il soit possible de dire qu'existent en son sein de véritables acteurs interagissants, qui sont des *programmes qui manipulent d'autres programmes*.

2.3. Communication inter acteurs

Telle que nous l'avons décrite jusqu'à présent, la fonction *instructionAvancer()* utilise des itérateurs pour parcourir la mémoire et en extraire des données. Ces itérateurs, regroupés en listes, sont des requêtes, ou encore des *patterns*. Un *espion* est pour nous un pattern capable de détecter non plus l'existence de données ayant une certaine forme syntaxique, mais *l'activation de certains patterns* au cours de l'interprétation des acteurs. Un espion est donc un *métapattern*.

Pour mettre ceci en œuvre, nous introduisons *une phase supplémentaire* dans l'interprétation des instructions : une fois qu'une instruction *I* contenue dans un acteur *A* a matché un ensemble d'objets dans la base de faits, et *avant* de passer à l'instruction suivante, l'instruction *I* passe dans une phase de *méta interprétation*, prioritaire sur la précédente, dans laquelle sont recherchés tous les espions déclenchables par le succès de *I*. Une fois que tous ces espions ont été activés, et si aucun espion n'est bloquant, l'interprétation de l'acteur *A* reprend. S'il existe un espion bloquant, l'acteur *A* passe dans l'état *Stoppé*.

2.3.1. Mise en place d'espions

En pratique, nous avons choisi de concrétiser le mécanisme d'espionnage en introduisant *plusieurs types d'instructions*. Il y en a au minimum 2 :

- ♦ D'une part, l'instruction de *lecture*, symbolisée par le caractère « ? » ;
- ♦ D'autre part, l'instruction d'*espionnage*, symbolisée par le caractère « ¿ ».

Le jeu d'instructions est ensuite enrichi par :

- ♦ L'*écriture*, et l'*envoi de message*, symbolisées respectivement par les caractère « ! » et le caractère « ¡ ». Ces instructions sont de la même nature que la lecture, qui est une instruction « non méta » ;
- ♦ L'*interdiction*, symbolisée par « ¿■ ». Cette instruction appartient à la catégorie des métainstructions.

Soit :

```
enum {
   typeLecture,typeEvenement,typeEcriture,typeEspion,typeInterdiction
}
typeInstruction;
class INSTRUCTION:OBJET {
   ID Premier;
   ID Cour;
   typeInstruction Type;
};
```

Pour implémenter l'espionnage, nous avons ensuite besoin de rajouter un nouvel ensemble d'acteurs à l'ensemble des acteurs *Actifs* et *Stoppes*, qui est l'ensemble des *Espions* :

```
ID Actifs,Stoppes,Espions;
void Scheduler() {
   ...............
}
```

Quand l'instruction courante d'un acteur *A* est une instruction d'espionnage *E*, un clone *A2* de *A* est créé. L'instruction courante de l'acteur *A2* est l'instruction *E*, alors que cette instruction *E* échoue dans *A*, qui effectue un retour en arrière. L'acteur *A2* est alors placé dans la liste *Espions*, ou liste des *acteurs femelles*, alors que l'acteur *A* continue son exécution, et est replacé dans la liste *Actifs*, liste des *acteurs mâles*.

Le *clonage* d'un acteur est l'opération consistant à créer une copie complète d'un acteur. Ceci comprend la duplication de la liste des variables locales de l'acteur (ce que nous avons appellé son environnement), *mais aussi* la duplication de sa liste d'instructions, puisque ces instructions sont des ensembles d'itérateurs, et ont donc elles aussi un état.

C'est la fonction *acteurAvancer()* qui se charge de tester le type de l'instruction courante et d'effectuer les opérations dont il vient d'être question :

```
void acteurAvancer(ID act) {
   ID cour,instr;
   cour=Cour(act),instr=Val(cour);
   switch (Type(instr)) {
      case typeLecture: {
         if (Etat(act)==etatEnCours) {
            ........................
         }
         else if (Etat(act)==etatSucces) {
            ........................
         }
         else Erreur("Acteur non actif");
      }
      break;
      case typeEspion: {
         ID act2=acteurCloner(act);
         listeAjouterFin(&Espions,listeNouveau(act2,NIL,NIL));
         Etat(instr)=etatTermine;
      }
      break;
   }
}
```

Résumons ce qui a été dit sous la forme d'une figure :

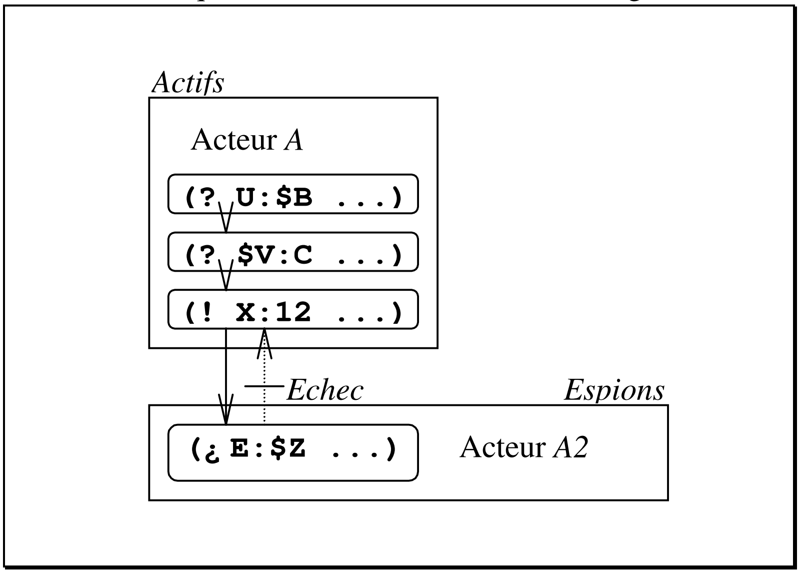

Figure 10 : mise en place d'espions

2.3.2. Espions asynchrones et événements

Quand des espions sont en place, ils peuvent être contactés par des acteurs, au cours de la phase de métainterprétation des instructions :

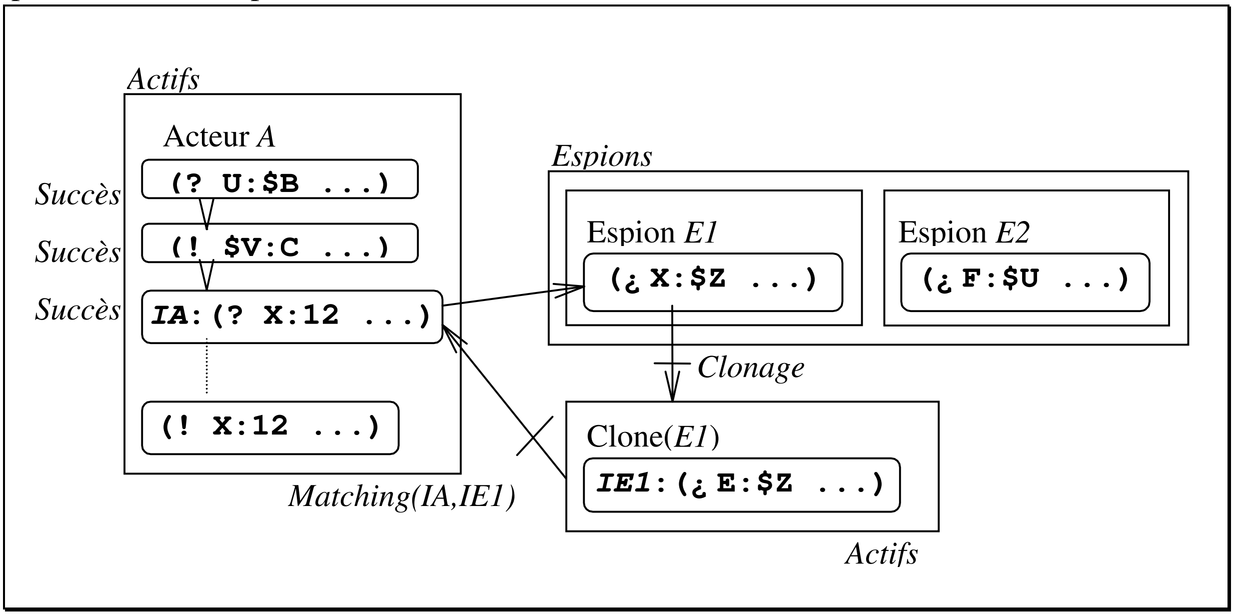

Figure 11 : métainterprétation des instructions

Pendant toute la phase de métainterprétation d'une instruction *IA* de l'acteur *A*, cet acteur *A* est dans l'état *etatMeta*, dans lequel il doit consulter tous les espions activables par le succès de *IA*. L'interprétation de cette phase méta est elle aussi itérative, c'est pourquoi il faut aussi ajouter un champ *EspionCourant* dans le type *ACTEUR* :

```
enum {
   etatEnCours,etatSucces,etatTermine,etatMeta;
}
etat;
.........................
class ACTEUR:SYMBOLE {
   ID Environnement,Instructions;
   ID Cour;
   etat Etat;
   ID EspionCourant;
};
```

A chaque acteur espion est associée l'instruction d'espionnage qui le caractérise. Sur la figure ci-dessus, quand l'instruction *IA* de l'acteur *A* réussit, l'espion *E1* est contacté, et après clonage, l'on confronte l'instruction *IE1* de l'acteur clone à l'instruction *IA*. Si les deux patterns se correspondent, l'acteur *Clone(E1)* continue, il meurt sinon. Quel que soit le résultat, l'acteur *A* contacte ensuite l'espion *E2* qui suit l'espion *E1* dans la liste des espions, et le processus est réitéré jusqu'à ce que tous les espions aient été consultés. Une fois que tous les espions ont été consultés et s'il n'existe pas d'espion bloquant, l'acteur *A* entame l'exécution de l'instruction suivante, autrement il passe dans l'état *etatStoppe*.

L'unification entre une instruction d'espionnage et une autre instruction est elle aussi implémentée de façon non récursive, mais de façon plus simple que dans les procédures *iterateurAvancer()* et *instructionAvancer()* qui ont été décrites plus haut, puisque les deux objets à matcher étant connus, il n'y a aucune recherche à faire (cette recherche est en fait effectuée par le biais du parcours de la liste des espions). De plus, les éléments à matcher dans les deux instructions sont supposés donnés dans le même ordre, ce qui accélère encore les calculs. Un élément important doit par contre être introduit, c'est le fait que l'unification met cette fois-ci en contact *deux environnements différents*, alors qu'auparavant, les itérateurs étaient confrontés directement à des données. Nous sommes amenés à faire évoluer la fonction *Matcher()* :

```
boolean &iterateurValeur(ID iter,IND pos)
   switch (pos) {
      case indI: return &Obj(iter);
      case indE: return &Attr(iter);
      case indP: return &Val(iter);
   }
}
boolean Matcher(ID iter1,ID iter2,IND pos) {
   ID var1=iterateurValeur(iter1,pos),
      var2=iterateurValeur(iter2,pos),
      val1=EstVariable(var1)?LireValeurVariable(var1):var1,
      val2=EstVariable(var2)?LireValeurVariable(var2):var2;
   if (EstVariable(var1)) {
      if (xSpec(iter1,pos1)) EcrireValeurVariable(var1,NIL),
                             xSpec(iter1,pos)=FALSE;
      if (val1!=NIL) return val1==val2;
      else {
         EcrireValeurVariable(var1,val2);
         xSpec(iter,pos)=TRUE;
         return TRUE;
      }
   }
   else return val1==val2;
}
```

2.3.3. Espions synchrones et récursivité

La création d'acteurs par l'intermédaire d'espions que nous avons montré jusqu'ici était de nature *asynchrone*, puisque la continuation de l'acteur espionné n'était aucunement liée à la terminaison ou au succès du ou des acteurs espionneur(s). L'appel récursif *à la Prolog* peut s'implémenter en utilisant la même technique au niveau des déclenchements : une tête de clause Prolog est pour nous une instruction d'espionnage particulière, symbolisée par « ¿> ». Nous rajoutons alors une seconde phase dans l'exécution des instructions de lecture : après avoir recherché toutes les solutions possibles dans la base de faits, une instruction de lecture doit ensuite parcourir la liste des têtes de clause qu'elle peut éventuellement unifier.

Quand une instruction de lecture réussit en s'unifiant avec une tête de clause, il est possible de procéder exactement de la même manière que lorsque cette instruction de lecture réussit et matche une donnée en mémoire, à savoir de rechercher les espions qui peuvent être concernés. Cette fois-ci, nous n'espionnerions plus seulement les accès mémoire, mais nous espionnerions aussi les appels de fonction, ce qui serait un peu plus compliqué, mais ne présente aucune difficulté nouvelle. Nous n'avons pas mis en œuvre ce type d'espionnage-là, faute de temps et parce que ce n'était pas indispensable.

L'appel de clause Prolog en lui-même nécessite enfin la mise en œuvre d'une communication *dans les deux sens* entre les environnements de l'appelant et de l'appelé, ainsi que l'implémentation d'un mécanisme pour *revenir en arrière* sur les instanciations des variables. Les moyens pour faire ceci sont nombreux et bien étudiés par ailleurs, nous nous en tiendrons donc aux explications succinctes données ici.

3. Les entrées sorties du moteur

Les communications entre le moteur et le monde extérieur ont lieu par l'intermédiaire de *ports de communication* à la scheme [STE89]. Nous les avons adaptés et étendus pour en obtenir une version conforme aux besoins de notre système : nous discutons ici de la communication entre le moteur d'inférence et des processus attentionnels *externes*, de nature totalement asynchrone par rapport au fonctionnement du moteur. Ces processus attentionnels externes sont soit liés à des événements pouvant avoir lieu dans le système (timer, par exemple), soit à des lectures de périphérique (clavier, socket, etc.). Dans les deux cas, le problème est le même : comme ils doivent être concrétisés par des processus UNIX distincts, le moteur d'inférence ne peut y lire de données : il faut donc que les intervenants externes soient eux-mêmes en mesure de venir lire et écrire dans la base de faits du moteur, qui joue alors pleinement son rôle de mémoire partagée. Mais ceci doit avoir lieu d'une manière bien précise et bien réglementée : c'est la raison d'être des ports de communication.

Ces ports de communication sont des objets situés dans la base de faits, et sont chacun lié de façon biunivoque à un processus UNIX externe et distinct du moteur. Ils sont accessibles par tous les processus partageant la base, tout le problème étant ensuite de synchroniser correctement les accès. Ces accès ont lieu en deux phases, d'abord une phase dite de *sérialisation*, qui consiste à stocker les demandes d'accès au moteur dans une file d'attente, suivie d'une phase de *transaction*, où les données sont échangées sur le port.

3.1. Phase de transaction

Le seul processus qui a le droit de manipuler directement les données situées dans la base de faits est le moteur, qui joue ici le rôle d'un *serveur*. Les autres processus, les *clients*, ont simplement le droit d'accéder le port qui leur est alloué. Les ports peuvent fonctionner alternativement dans le sens Client→Serveur ou dans le sens inverse (mais pas en *full duplex* dans notre implémentation actuelle). Ces ports sont constitués de deux listes, une liste *Priv* d'objets située dans la zone de mémoire *privée* de ce client, alimentée ou décodée exclusivement par lui, et un bloc *Comm* situé en mémoire partagée, contrôlé par le serveur, dans lequel ce dernier décode la liste *Priv*, ou dépose des données devant être lues par le client :

```
class PORT:SYMBOLE {
   boolean Mode; // Input ou Output
   List *Priv;   // Messages en attente dans la memoire du client
   int Taille;   // Taille des donnees a transmettre
   ID Comm;
   etatPORT Etat;
};
```

La liste *Priv* est constituée de *messages* qui sont des chaînes de caractères. Le bloc *Comm* contient pour sa part des *CODONs* en bonne et due forme. Les transactions ne s'effectuent enfin pas en une seule étape. Les ports passent par une série successive d'états :

```
enum {
   VIDE,NONALLOUE,ALLOUE,REMPLI,LU
}
etatPORT;
```

a. Transaction dans le sens Client→Serveur

- Tant que le port n'est pas retourné dans l'état *VIDE*, le client bufferise dans *Priv* les données qu'il a à transmettre, et l'on est par ailleurs assuré que cet état *VIDE* surviendra, puisque dans le mode *Input* dont il est question ici, le serveur passe son temps à lire des données sur le port, alors que le client l'alimente ;
- Une fois que le port est retourné à l'état *VIDE* (et est donc redevenu inactif) le client calcule la taille dont il a besoin pour allouer toutes les données présentes dans sa liste *Priv*, puis informe le moteur de la quantité de mémoire dont il a besoin pour transmettre son message, en écrivant la taille nécessaire dans le slot *Taille* de l'objet *PORT* prévu à cet effet. Il modifie ensuite l'état de ce port. Le port passe de l'état *VIDE* à l'état *NONALLOUE* (*les changements d'état des ports sont atomiques*) ;
- Au moment où il détecte le changement d'état du port, le moteur alloue en mémoire partagée un bloc de la taille demandée, puis stocke la référence dans le champ *Comm* du port de communication. Le port change encore une fois d'état, et passe à l'état *ALLOUE* ;
- Le client tranvase les données de *Priv* vers *Comm*, et fait passer le port dans l'état *REMPLI*. Pendant tout le temps que dure cette opération, on a la garantie que le serveur (ni aucun des acteurs qu'il interprète) n'accédera le bloc *Comm* en aucune manière ;
- Une fois que *Comm* est rempli, le serveur décode son contenu sous forme de listes attribut-valeur dans la base de faits. Le bloc *Comm* est détruit, et le port retourne enfin dans l'état *VIDE* d'origine.

b. Transaction dans le sens Serveur→Client

- ♦ Le serveur alloue un bloc *Comm* de la taille adéquate, et copie les données à l'intérieur. Le port passe directement de l'état *VIDE* à l'état *REMPLI* ;
- ♦ Le client décode alors les données dans sa liste *Priv* et fait passer le port dans l'état *LU* ;
- ♦ Le serveur détruit le bloc *Comm*, et le port repasse dans l'état *VIDE*.

Nous pouvons résumer ce qui a été dit par le schéma suivant (cas *Client→Serveur*) :

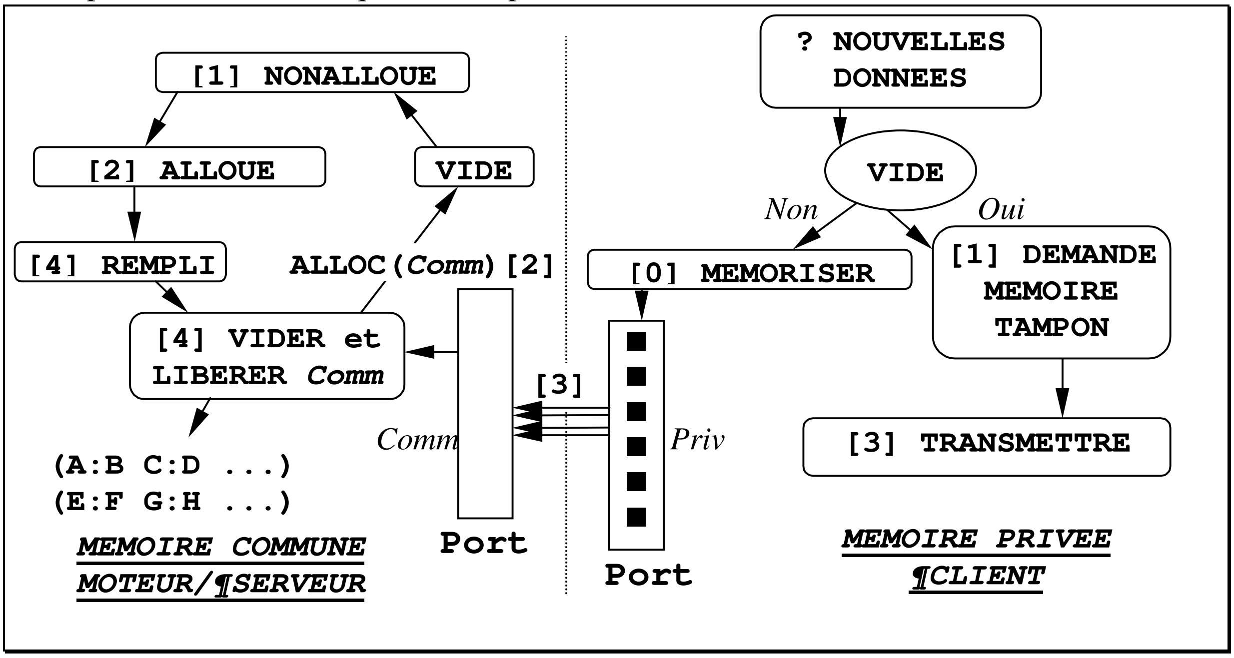

Figure 12 : communication entre le moteur et des acteurs exogènes (*input*)

3.2. Phase de sérialisation

Avec ce qui a été montré, la synchronisation a lieu en utilisant une variable de contrôle, et on pourrait donc s'en tenir là. Il reste à savoir comment et quand clients et serveur vont tester l'état du port. Dans le cas du client en attente, il suffit d'une simple boucle testant l'état attendu du port, par exemple :

```
while (Etat(port)!=ALLOUE) sched_yield();
```

Le cas du serveur ne se ramène par contre pas à du simple pooling, le nombre de ports qu'il peut avoir à surveiller n'étant pas borné à l'avance : ce que l'on souhaite, c'est qu'en un cycle, il ne consulte et traite que les ports qui ont été modifiés. Pour faire ceci, il est nécessaire de disposer d'une *file d'attente* en mémoire partagée, contenant des références sur des ports, associées chacune avec le type de la commande effectuée (*Input* ou *Output*). Quand une étape de communication a lieu sur un port, ce port est ajouté dans la file d'attente. Ainsi, à chaque cycle, le moteur traite le contenu d'une file d'attente ne contenant que les ports modifiés, et non tous les ports existants.

L'accès à la file d'attente est protégé de la façon suivante :

```
// Pointent deux emplacements reserves dans la memoire partagee
int *ENTRE=NULL,*SORTI=NULL;
/*
 * Effectue un swap atomique entre une case memoire et
 * une valeur, et renvoie l'ancienne valeur de la case.
 */
int atomSWAP(int *ptr,int val) {
   asm("pushl %ebx");
   asm("pushl %ecx");
   asm("movl %0,%%ebx"::"D"(ptr):"ax");
   asm("movl %0,%%ecx"::"2"(val):"ax");
   asm("xchg (%ebx),%ecx");
   asm("movl %%ecx,%0"::"2"(val):"ax");
   asm("popl %ecx");
   asm("popl %ebx");
   return val;
}
void IOENTER() {
   int enteredPID=atomSWAP(ENTRE,getpid());
   while (enteredPID!=*SORTI) sched_yield();
}
void IOLEAVE() {
   *SORTI=SELFPID;
}
```

puis :

```
IOENTER();
// Manipulation de la file d'attente
....................................
IOLEAVE();
```

Conclusion

Dans ce papier, nous avons discuté de l'implémentation du moteur de notre système *Hammourabi*, d'abord de la représentation mémoire, puis de l'interpréteur, et enfin des entrées-sorties. Nous voulions créer au départ un environnement dont on puisse réellement dire qu'en son sein interagissent des programmes manipulant d'autres programmes. Bien que le résultat soit certainement encore bien loin d'être parfait, nous croyons avoir atteint notre but initial de façon relativement satisfaisante, et avons pu utiliser le moteur obtenu pour mener nos expériences.

Bibliographie

[GUE96] Guessoum Zahia, *Un environnement opérationnel de conception et de réalisation de systèmes multi-agents*, Thèse de l'université Paris 6, 1992.

[KRA87] Krakowiak Sacha, *Principes des systèmes d'exploitation des ordinateurs*, DUNOD 1987.

[LAU86] Laurière J.L., *SNARK : a language to represent declarative knowledge and an inference engine which uses heuristics*, IFIP 86, Elsevier science publications, pp 811-816, 1986.

[LEE63] Lee C.Y., Paul M.C., *A content addressable Distributed-Logic Memory with Applications to Information Retrieval*, IEEE Proceedings 51, pp 924-932, 1963.

[PAC92] Pachet F., *Représentation des connaissances par objets et règles : le système NéOpus*, Thèse de l'université Paris 6, 1992.

[SIK82] Siklossy L., Laurière J.L., *Removing restrictions in the relational database model. An application of problem-solving techniques*, Proc. Nat. Conf. Art. Intel. (AAAI), Pittsburgh PA, 1982.

[STE89] G.L Steele, G.J. Sussman et al., *Revised$^{3.99}$ Report on the Algorithmic Language Scheme (R4RALS)*, W. Clinger, J. Rees, Eds, 1989.

[VIA85] Vialatte M., *Description et applications du moteur d'inférences Snark*, Thèse de l'Université Paris 6, 1985.

Espionnage et interdictions dans UNIX

1. Description

En plus du mécanisme d'espionnage/interdictions utilisé pour faire communiquer des processus interprétés *à l'intérieur* du moteur, nous en avons mis au point un autre, dont l'objet est cette fois-ci de faire communiquer des processus UNIX réels, situés donc *à l'extérieur* du moteur. Ce mécanisme peut en particulier être utilisé par le moteur pour *contrôler* le comportement d'autres processus au sein d'UNIX.

Pour faire ceci, nous avons donc ajouté au noyau UNIX un *module contrôleur* servant à gérer les espionnages. Lors de l'appel d'une primitive du noyau UNIX, le contrôleur est contacté, et les trois informations suivantes lui sont fournies :

- *Le processus* courant ;
- *La primitive* qui est sur le point d'être exécutée ;
- *Le ou les objets* sur lequel(s) porte l'action.

Ceci constitue ce que nous appelons par abus de langage un *événement*. D'autre part, un *espion* dans le contrôleur est constitué par la donnée d'un processus UNIX, d'une action devant être effectuée par ce processus, et de l'objet sur lequel doit avoir lieu cette action. Les éléments intervenant dans cette description de l'événement définissant un espion peuvent être spécifiés de façon partielle. Enfin, l'espion peut être une *interdiction* (ou *espion bloquant*), c'est à dire un espion qui stoppe et met en attente les processus dont il détecte les actions.

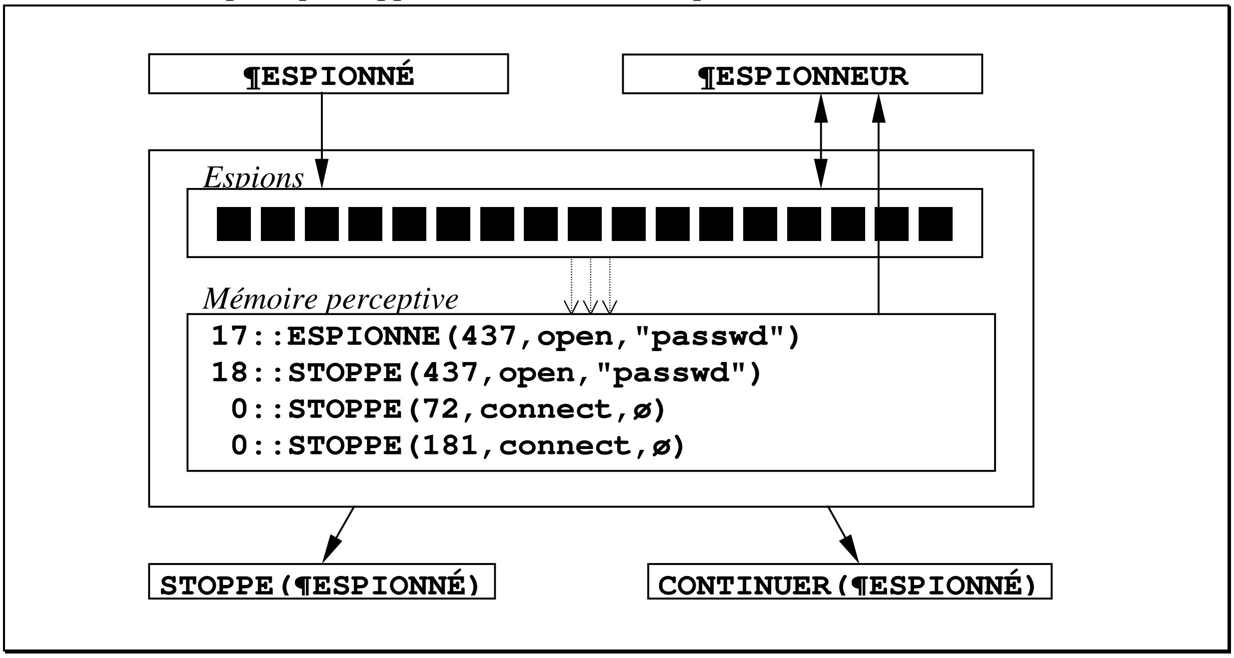

Figure 1 : fonctionnement de l'espionnage dans UNIX

Le contrôleur est averti de l'imminence de l'exécution d'une action issue du processus courant *¶ESPIONNÉ*, et consulte alors la liste des espions actifs en son sein. S'il existe un ou plusieurs espions matchant l'action, l'événement est mémorisé. S'il existe un espion bloquant parmi les espions matchant l'action issue de *¶ESPIONNÉ*, ce processus espionné est stoppé, sinon il continue. Le processus espionneur *¶ESPIONNEUR* a pour sa part la possibilité de poser et d'enlever des espions dans le contrôleur, et de consulter les événements détectés. Au cas où certains de ces événements ont été détectés par des espions bloquants, il peut faire redémarrer les processus stoppés par ces espions.

2. Implémentation

Le déclenchement du contrôleur se fait par l'intermédiaire d'une fonction *do_spy()*, faisant office de point d'entrée, dont le prototype est le suivant :

```
do_spy(int primitive_no,char *nom_objet,int taille_nom)
```

Un appel à cette fonction *do_spy()* est inséré au début du code de chacune des primitives espionnables, à un endroit où aucune des opérations à effets de bord mises en jeu dans l'exécution de cette primitive n'a encore été invoquée. Le lecteur attentif aura peut-être remarqué que le paramètre correspondant au processus courant ne fait pas partie du prototype de *do_spy()*. Il est en effet inutile de passer un tel paramètre, puisque les appels à *do_spy()* étant situés à l'intérieur du noyau, le processus courant est toujours disponible à travers la variable globale *current* de ce même noyau.

L'appel à *do_spy()* est par exemple effectué comme ceci dans le cas du *open()* :

```
....................
#include <linux/mm.h>
#include <linux/spy.h>
...
int open_namei(const char *pathname,int flag,int mode,
               struct inode **res_inode,struct inode *base)
{
   const char *basename;
   int namelen,error;
   struct inode *dir,*inode;

   do_spy(PRIMopen,(char*)pathname,strlen(pathname)+1);
   mode&=S_IALLUGO&~current->fs->umask;
   .............
   return error;
}
...
```

Cette fonction *do_spy()* doit ensuite vérifier si l'action qu'on lui soumet concerne un espion actif. Ces espions actifs sont représentés comme ceci dans le contrôleur :

```
typedef struct {
   int spy_no;            // Numéro de l'espion
   int process_no;        // Si -1, signifie "tous les processus"
   int primitive_no;      // Si -1, signifie "toutes les primitives"
   char *nom_objet;       // Si NULL, signifie "tous les objets"
   int actif;             // Si TRUE, l'espion est actif
   int interdiction;      // Si TRUE, il s'agit d'un ¿■
   char *nom_programme;   // Si non NULL, espionner les processus dont
                               le programme exécutable a pour nom la
                               valeur de la variable nom_programme.
}
spy_struct;
```

Le module contrôleur *spy.c* maintient la liste des espions :

```
spy_list *espions;
```

En consultant cette liste, le contrôleur vérifie si une instance d'appel de primitive concerne un des espions existants. Il est possible d'espionner des événements partiellement indéterminés comme « tous les processus qui écrivent dans le fichier */etc/passwd* », ou encore « tous les fichiers ouverts par le processus *P* ». Il est aussi possible d'*interdire* de tels événements, ce qui a pour effet de *stopper* les processus effectuant ces actions sujettes à interdiction. Les événements sont ensuite enregistrés :

```
typedef struct {
   int spy_no;            // L'espion ayant provoqué l'enregistrement
   int process_no;        // Le processus ayant émis l'action
   int primitive_no;      // La primitive ayant été exécutée
   char *nom_objet;       // Le nom de l'objet manipulé
}
spy_event;
```

Les processus sont stoppés à l'intérieur du noyau, à l'aide d'un *wait()* modifié, qui stoppe inconditionnellement sans être lié à la terminaison de processus enfants. Ceci parce qu'on ne dispose d'aucune information pour savoir si des processus enfants des processus qu'on stoppe se sont déjà terminés, ou même si ces processus ont eu des processus enfants : notre espion doit pouvoir stopper *n'importe quels* processus, et ne peut donc à priori pas faire d'hypothèse sur eux. Pour cette même raison, *le réveil* de ces processus stoppés doit (quand il a lieu) être effectué à l'aide d'une fonction *kill()*[1] modifiée, cette fois-ci parce que la survenue d'un signal tue un processus qu'on essaie de réveiller s'il ne gère pas ce signal. Là aussi, nous ne pouvons

[1] Dans UNIX, *kill()* est la fonction servant à émettre un signal ! La fonction *signal()* sert pour sa part à établir un handler pour un signal...

pas faire d'hypothèses sur les signaux gérés par les processus que nous stoppons. Ceci revient à rajouter un signal ayant toujours pour effet de réveiller les processus, sans jamais les tuer.

Une fois que le noyau est programmé pour espionner et interdire un certain nombre d'actions, il faut qu'un processus *espionneur* puisse venir consulter la liste des événements, et puisse surtout être averti quand de nouveaux événements sont disponibles. Les communications entre le noyau et ce processus espionneur se font par l'intermédiaire d'une nouvelle primitive *sys_spy(int cmd,spy_struct *s)*, qui donne aux processus espionnés la possibilité de passer des *commandes* au contrôleur. Il est possible de placer des espions en utilisant la commande *SPYcreate*, d'en détruire en utilisant *SPYdelete*, et surtout de lire les événements détectés par les espions, en utilisant *SPYread*. La commande *SPYconnect* est la commande de connexion au contrôleur, qui transforme un processus en processus espionneur. A partir du moment où elle a été exécutée, le processus espionneur est averti par un signal *SIGSPY* dès qu'il y a des événements disponibles. Le contrôleur n'envoie pas un signal par événement (ce qui risquerait de conduire à l'oubli silencieux de certains signaux par l'espionneur au cas où le débit est trop important), mais *un seul signal* au moment où survient le premier événement. Ces événements sont stockés dans *deux* listes, une liste *spy_alire* qui est mise à la disposition de l'espionneur, et une liste *spy_memo* qui continue d'être alimentée par le contrôleur. Quand l'espionneur a fini de vider *spy_alire* en utilisant des *SPYread*, les deux listes sont permutées, et la lecture reprend sur l'ancienne liste *spy_memo*. L'ancienne liste *spy_alire* remplace *spy_memo* et continue d'être remplie. Quand les deux listes sont vides, le signal peut être réarmé, puisqu'on sait que l'espionneur n'est certainement plus en train de le traiter, et le cycle recommence au moment de la détection d'un nouvel événement par un espion :

```
enum {
   pasla,pasprevenu,prevenu,enlecture
};
int spy_state;
struct task_struct *spy_process; // procedures pour UN SEUL spy
typedef struct spy_list {
   void *buf;
   int n;
   struct spy_list *suiv;
}
spy_list;
spy_list *spy_memo=NULL,*spy_alire=NULL;
// Processus espionneur
asmlinkage int sys_spy(int cmd,int p) {
   switch (cmd) {
   /* Connexion */
      case SPYconnect: {
         if (!spy_process) {
            spy_state=pasprevenu,spy_process=current; return 1;
         }
         else return 0;
      }
```

```
   /* Deconnexion */
      case SPYdeconnect: {
         if (spy_process) {
            exit_spy();
            return 1;
         }
         else return 0;
      }
   /* Savoir s'il existe un autre espion */
      case SPYhere: {
         return spy_process!=NULL;
      }
   /* Connexion en lecture : le processus doit vider la file */
      case SPYread: {
         if (spy_process && current==spy_process && spy_state!=pasprevenu)
         {
            if (spy_state==prevenu) {
switchlistes:
               if (spy_alire) panic("spy::alire");
               spy_alire=spy_memo,spy_memo=NULL;
               spy_state=enlecture;
               goto lecture;
            }
            else {
               int resu;
               spy_list *nouveau;
lecture:
               resu=0;
               nouveau=spy_alire;
               if (spy_alire) {
                  nouveau=spy_alire,spy_alire=spy_alire->suiv;
                  memcpy_tofs((void*)p,(void*)nouveau->buf,nouveau->n);
                  kfree(nouveau->buf),kfree(nouveau);
                  resu=1;
               }
               else {
                  if (spy_memo) goto switchlistes;
                  spy_state=pasprevenu;
               }
               return resu;
            }
         }
         else return 0;
      }
      .................
      default: return 0;
   }
}
```

```
// Controleur
void do_spy(int prim,void *p,int n) {
   boolean yaeuuntruc=FALSE;
   if (spy_process && spy_process!=current) {
   // Detection et enregistrement des evenements
      .......................................
   // Envoi du signal s'il a ete rearme et si un evenement est arrive
      if (yaeuuntruc && spy_state==pasprevenu) {
         spy_state=prevenu;
         send_sig(SIGSPY,spy_process,1);
      }
   }
}
```

3. Application 1

Nous donnons ici deux programmes C utilisant l'espion. Le premier programme espionne les ouvertures de fichier, on obtient avec lui un résultat semblable à ce qu'on aurait avec *strace()*. Le second programme interdit les suppressions de fichier le temps de copier ces fichiers dans un répertoire de backup.

Programme 1

```
#include <signal.h>
#include <linux/unistd.h>
#include <linux/spy.h>
#include <stdio.h>

int buf[1000];
spy_struct *msg=(spy_struct *)&buf;
void handler(int sig) {
   while (spy(SPYread,(int)msg)) {
      printf("%d|%s\n",msg->process_no,msg->nom_objet),
      fflush(stdout);
   }
   signal(SIGSPY,handler);
}
main() {
   signal(SIGSPY,handler);      /* SIGSPY pour l'espion   */
   msg->actif=1;
   msg->interdiction=0;         /* Espion non bloquant    */
   msg->process_no=-1;          /* Tous les processus     */
   msg->primitive_no=PRIMopen; /* Ouvertures de fichiers */
   msg->nom_objet=NULL;         /* Tous les fichiers      */
   spy(SPYcreate,(int)msg);     /* Creer l'espion         */
   if (spy(SPYconnect,0)) {     /* Connexion              */
      while (1);
   }
}
```

Programme 2

```
#include <signal.h>
#include <linux/unistd.h>
#include <linux/spy.h>
#include <stdio.h>

// Copie le fichier ficSrc a l'emplacement ficDest
void ficcp(char *ficSrc,char *ficDest) {
.......................................
}
int buf[1000];
spy_struct *msg=(spy_struct *)&buf;
void handler(int sig) {
   static backupno=0;
   static char str[256],fic[256];
   while (spy(SPYread,(int)msg)) {
      if (msg->primitive_no==PRIMdelete) {
         strcpy(fic,"/poubelle/backup");
         sprintf(str,"%d",backupno++);
         strcat(fic,str);
         ficcp(msg->nom_objet,fic);
         spy(SPYresume,msg->process_no);
      }
   }
   signal(SIGSPY,handler);
}
main() {
   signal(SIGSPY,handler);        /* SIGSPY pour l'espion     */
   msg->actif=1;
   msg->interdiction=1;           /* Espion bloquant          */
   msg->process_no=-1;            /* Tous les processus       */
   msg->primitive_no=PRIMdelete;  /* Suppressions de fichiers */
   msg->nom_objet=NULL;           /* Tous les objets          */
   spy(SPYcreate,(int)msg);       /* Creer l'espion           */
   if (spy(SPYconnect,0)) {       /* Connexion                */
      while (1);
   }
}
```

4. Application 2

Le mécanisme d'espionnage *UNIX* est ensuite intégré à l'interpréteur. Pour faire ceci, nous rajoutons dans cet interpréteur une procédure *handler()* similaire à celle des deux programmes ci-dessus, qui alimente une file d'attente. Cette file d'attente est décodée à l'intérieur de la boucle de contrôle principale de l'interpréteur (fonction *Scheduler()*). Les processus *UNIX* apparaissent sous la forme d'un nouveau type d'objet dans la base de faits, le type *UNIX*. Les événements *UNIX* apparaissent sous la forme d'objets de type *PERCEPT*. Enfin, le moteur doit être immunisé contre les effets pervers que peuvent avoir les interdictions UNIX quand il se les applique à lui-même. Cette fonctionnalité est nécessaire si l'on veut que le mécanisme des interdictions puisse concerner aussi des processus *interprétés* par le moteur. Une solution simple au problème consiste à dédoubler le mécanisme d'espionnage, et à tester *avant exécution* si une action UNIX au sein du moteur est sujette à interdiction. Dans ce cas-là, si elle concerne un acteur *A* interprété, et non les actions du moteur lui-même, cet acteur *A* est intercepté directement dans l'interpréteur et un pattern *(PERCEPT ...)* est directement émis. Autrement, les interdictions doivent être inhibées. Il existe aussi un certain nombre de processus UNIX exogènes pour lesquels il est nécessaire d'inhiber les interdictions (par exemple, le processus *init*, le swapper, etc.). Ces processus, de quelque nature qu'ils soient, pour lesquels l'interdiction doit être inhibée, sont appellés les *processus vitaux* du système. Nous donnons ci-dessous les deux programmes précédents, réécrits sous forme de scripts *Hammourabi*.

Script 1

```
(CLASSE:SCRIPT NOM:S1
   CODE:(
      (¿ PERCEPT ACTEUR:$A ACTION:unixOPEN OBJET:$O)
   $A:(? CLASSE:ACTEUR TYPE:UNIX PID:$PID)
   $O:(? CHEMIN:$C)
      (_PRINT $PID "|" $C "\n")
   )
)
```

Script 2

```
V:(CLASSE:NOMBRE VALEUR:0)
  (CLASSE:SCRIPT NOM:S2
     CODE:(
        (¿■ PERCEPT ACTEUR:$PR ACTION:unixDELETE OBJET:$O)
    $PR:(? CLASSE:ACTEUR TYPE:UNIX)
     $O:(? CHEMIN:$FIC)
      V:(? VALEUR:$V)
        (_FICCP $FIC (_+ "/poubelle/backup" $V))
      V:(-! VALEUR:(_+ $V 1)) §REMPLACER VALEUR(V) PAR ($V+1)§
    $PR:(↑)                   §RESUME($PR)§
     )
  )
```