CHAPTER 13

REALIZATION OF DATABASE MANAGEMENT SYSTEMS

This chapter continues the application of VDM to database applications. As explained above, the various architectures correspond to programming languages. The implementation of the architecture corresponds to an interpreter or, in cases, to a compiler. Just as VDM can be used to justify the design of an interpreter or compiler with respect to the definition of the language, this chapter briefly illustrates how a database management system implementation can be related to the specification of a data model. Implementations of both hierarchic and network architectures are discussed in this chapter; some of the standard (for example HDAM) IMS representations are discussed for the former. This chapter again illustrates the concepts of object transformations reviewed in chapters 10 and 11.

CONTENTS

13.0 INTRODUCTION

In this last chapter we shall briefly roundoff the ideas of chapters 10, 11 and 12. We illustrate the transition from some of the formal, architecturally abstract database data models of chapter 12 towards more concrete specifications of the database management systems (DBMS) which implement these models. We exemplify only the object transformations involved. Chapter 10 has amply exemplified the related operation transformations. And we display aspects of only hierarchical and network database systems.

13.1 HIERARCHICAL DATABASE MANAGEMENT SYSTEMS

We assume a slightly different data model than the one illustrated in section 12.2.1. And we focus attention only on the data part of the data model. (That is: we omit consideration of the catalogue part.) This subsection presents four models, one abstract and three concrete (HSAM, $HDAM_{ct}$, $HDAM_{fp}$). First we restate the abstract model. This model is considered an abstraction of an appropriate part of IBMs IMS DBMS. IBM does not deliver abstractions. IBM delivers concretizations. In fact IBM offers three variants of IMS. These are referred to as the hierarchical sequential access method (HSAM), the hierarchical direct access method (HDAM) with child-twin pointers $HDAM_{ct}$, and HDAM with file pointers $HDAM_{fp}$.

13.1.1 Abstract IMS Data Part

Informally an IMS data part, *DP*, is abstracted as follows: A *DP* consists of a number of uniquely named physical database records, *P*, with names in *F*. Each *P* consists of an ordered sequence of segments, *S*. Each segment, *S*, has three components: a sequence field which is an integer, a number of other uniquely named fields (of some type), and a data part. (The last component description refers to what is being defined. From this recursion, then, stems the 'hierarchy'.)

1. $DP = F \overrightarrow{m} P$
2. $P = S^*$
3. $S :: Int \ (Sn \overrightarrow{m} VAL) \ DP$

Within a given data part there is, according to the above informal and

formal descriptions, no ordering among the uniquely (F) named physical database records.

13.1.2 Hierarchical Sequential Access Methods, HSAM

In IMS, not just in its HSAM version, but in general, there is an ordering among the Ps of any DP. It is straightforward to model such an ordering:

4. $DP_{hsam} = (F\ P_{hsam})^*$
5. $P_{hsam} = S_{hsam}^*$
6. $S_{hsam} :: Intg\ (Sn \xrightarrow{m} VAL)\ DP_{hsam}$

7.0 $is\text{-}wf\text{-}DP_{hsam}(dp) \triangleq$
.1 $\quad is\text{-}unique(\langle s\text{-}F(dp[i]) \mid 1 \leq i \leq \underline{len}\ dp\rangle)$
.2 $\quad \wedge\ (\forall(,pdbr) \in \underline{elems}\ dp)$
.3 $\quad\quad (\forall mk\text{-}S_{hsam}(,,dp') \in \underline{elems}\ pdbr)$
.4 $\quad\quad\quad is\text{-}wf\text{-}DP_{hsam}(dp')$

8.0 $retr\text{-}DP(dp) \triangleq [f \mapsto retr\text{-}P(pdbr) \mid (f,pdbr) \in \underline{elems}\ dp]$

9.0 $retr\text{-}PDBR(sl) \triangleq \langle retr\text{-}S(sl[i]) \mid 1 \leq i \leq \underline{len}\ sl\rangle$

10.0 $retr\text{-}S(mk\text{-}S_{hsam}(i,svm,dp)) \triangleq mk\text{-}S(i,svm,retr\text{-}DP(dp))$

There is almost an inverse to the retrieve, or abstraction, functions. That is there is an injection relation ($\rightarrowtail$):

$inj\text{-}DP(mk\text{-}DP_{hsam}(dp)) \triangleq \langle(f,inj\text{-}SL(dp(f))) \mid f \in \underline{dom}\ dp\rangle$
$\underline{type}$: $DP \rightarrowtail DP_{hsam}$

$inj\text{-}P(sl) \triangleq \langle inj\text{-}S_{hsam}(sl[i]) \mid 1 \leq i \leq \underline{len}\ sl\rangle$
$\underline{type}$: $S^* \rightarrowtail P$

$inj\text{-}S(mk\text{-}S(i,svm,dp) \triangleq mk\text{-}S_{hsam}(i,inj\text{-}FVAL(svm),inj\text{-}DP(dp))$
$\underline{type}$: $S \rightarrowtail S_{hsam}$

$inj\text{-}FVAL(svm) \triangleq \langle(s,v) \mid s \in \underline{dom}\ svm \wedge v = svm(s)\rangle$
$\underline{type}$: $(Sn \xrightarrow{m} VAL) \rightarrowtail (Sn\ VAL)^*$

Tape Formatting

The HSAM organization is well-suited for, and conceived of in connection with, storing data parts on a magnetic tape. The hierarchical structure:

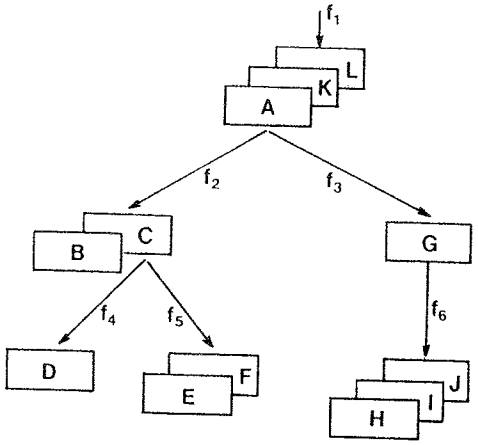

Fig. 1

-- omitting important details, thus is basically "stored" as:

Fig. 2

That is as the concrete object:

```
<(f1,
  <mk-S(a,
        <(f2,
          <mk-S(...B...),
           mk-S(c,
                <(f4,
                  <mk-S(...D...)>),
                 (f5,
                  <mk-S(...E...),
                   mk-S(...F...)>)>)>),
         (f3,
          <mk-S(q,
                <(f6,
                  <mk-S(...H...),
                   mk-S(...I...),
                   mk-S(...J...)>)>)>)>),
  mk-S(...K...),
  mk-S(...L...)>)>
```

[Here we have abbreviated S_{hsam} into S, and the sequence field integer and other field values into one (lower case roman) component.] The problem we wish to briefly consider is that of designing a concrete linear representation of DP objects. Basically that problem is always one of representing the abstract syntax for DP_{hsam} by a concrete, e.g. BNF, grammer DP_{bnf}. Sequence (S_{hsam}*) of objects are usually BNF-"programmed" as left- or right-recursive structures; but we do not care whether it is left- or right-leaning. In this "exercise" we decide to design the linear representation as a fully "bracketed" structure using marked parantheses and marked comma delimiters:

DP_{bnf} ::= DP< $File_{bnf}$ { DP, $File_{bnf}$ }* >PD |
$File_{bnf}$ ::= F(*Fname* , $PDBR_{bnf}$)F
$PDBR_{bnf}$ ::= SL< $SegList_{bnf}$ >LS
$SegList_{bnf}$ ::= *Segm* { SL, *Segm* }* |
$Segm_{bnf}$ ::= S(*IntgSVAL* s, DP_{bnf})S

(Here we have lumped into one object, *IntgSVAL*, the sequence field integer and other segment field values.) The previously shown abstract object now can be given the linear representation below:

```
DP< F( f1 f,
      SL< S( a s,
           DP< F( f2 f,
                SL< S(... B ...)S SL,
                   S( c s,
                      DP< F( f4 f,
                           SL< S(... D ...)S >LS )F DP,
                        F( f5 f,
                           SL< S(... E ...)S >LS )F >PD )S >LS DP,
             F( f3 f,
                SL< S( g s,
                     DP< F( f6 f,
                          SL< S(... H ...)S SL,
                             S(... I ...)S SL,
                             S(... J ...)S >LS )F >PD )S >LS )F >PD )S SL,
        S(... K ...)S SL,
        S(... L ...)S >LS )F >PD
```

In un-indented, i.e., non-structured textual form the above becomes:

DP< F(f_1 f, SL< S(a s, DP< F(f_2 f, SL< S(... B ...)S SL, S(c s, DP< F(f_4 f, SL< S(... D ...)S >LS)F DP, F(f_5 f, SL< S(... E ...)S)LS)F >PD)S >LS DP, F(f_3 f, SL< S(g s, DP< F(f_6 f, SL< S(... H ...)S SL, S(... I ...)S SL, S(... J ...)S >LS)F >PD)S >LS)F >PD S(... K ...)S SL, S(... L ...)S >LS)F >PD

Using conventional syntax analysis tools one can easily generate a parser which reads such strings and, by means of suitable, simple actions connected to appropriate productions, effects the operations of *mk-* or *s-* function decomposition as used in our various elaboration functions of e.g. section 2.2 of chapter 12.

13.1.3 Hierarchical Direct Access Methods, HDAM

The ordering relation suggested by the physical juxtaposition relation of magnetic tape offsets, or addresses can, however, be achieved in other ways. The *DP* part of segments, in the abstract *S*, as well as in the sequential S_{hsam}, models, can be considered as being contained in segments. Containment will now be replaced by designation. That is: proper parts of segments will contain pointers to *DP* structures, and these together with "proper" components will be physically allocated to any storage fragment. "Proper" components of segments will be those not involving parts. The following figure should illustrate our point:

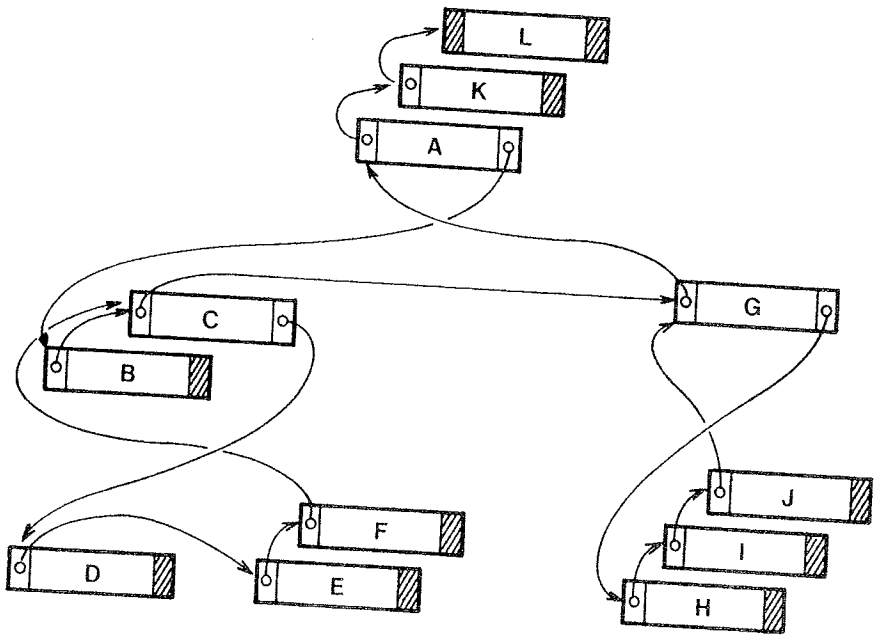

Fig. 3

Several pointer organizations are possible. The above embodies a notion of the "left" pointer designating (1) the "next segment" within a "segment list", (2) the "first segment of the next (twin) brother/sister segment list", or (3) the "parent segment". The "right" pointer of any segment designates a "first" child segment, if any.

Instead of having a pointer designating a child segment of some such first child file, we permit segments to individually designate the "first" segment of all children files:

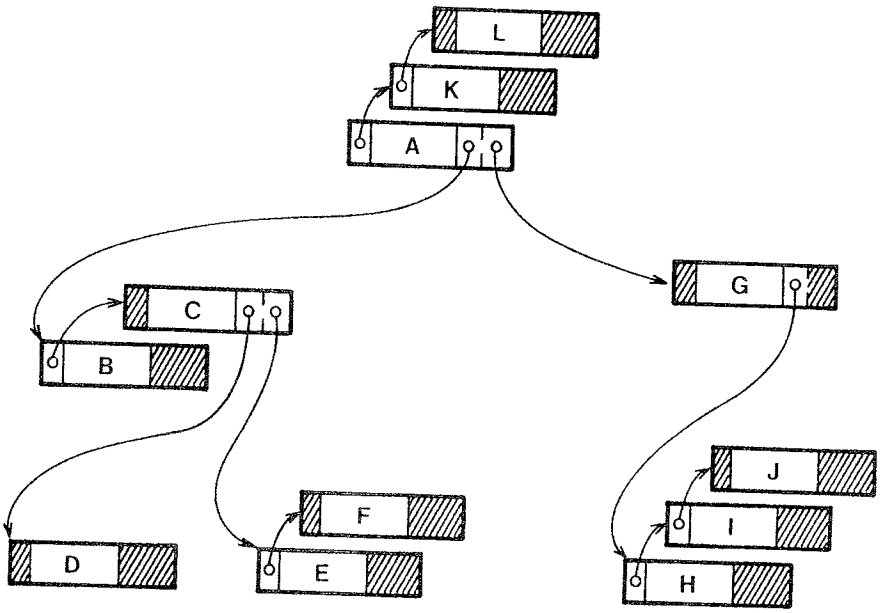

Fig. 4

These two organizations will now be formalized.

The Child/Twin Pointer Scheme, $HDAM_{ct}$

$$
\begin{array}{lcl}
DP_{ct} & :: & [\ Ptr\ ]\ (Ptr \xrightarrow{m} F)\ STG_{ct} \\
STG_{ct} & = & Ptr \xrightarrow{m} S_{ct} \\
S_{ct} & :: & [\ PTR\ ]\ (Intg(Sn \xrightarrow{m} VAL))\ [\ Ptr\ ] \\
PTR & = & NsPtr \mid TwPtr \mid PaPtr \\
NsPtr & :: & Ptr \\
TwPtr & :: & Ptr \\
PaPtr & :: & Ptr
\end{array}
$$

The particular object shown earlier now has the following $HDAM_{ct}$ representation:

$mk\text{-}DP_{ct}(p_a,$
$[pa \mapsto f_1,\ p_b \mapsto f_2,\ p_c \mapsto f_4,\ p_e \mapsto f_5,\ p_g \mapsto f_3,\ p_n \mapsto f_6],$
$[p_a \mapsto mk\text{-}S_{ct}(mk\text{-}NsPtr(p_k),\ a\ ,p_b),$
$p_b \mapsto mk\text{-}S_{ct}(mk\text{-}NsPtr(p_c),\ b\ ,\underline{nil}),$
$p_c \mapsto mk\text{-}S_{ct}(mk\text{-}TwPtr(p_g),\ c\ ,p_d),$
$p_d \mapsto mk\text{-}S_{ct}(mk\text{-}TwPtr(p_e),\ d\ ,\underline{nil}),$
$p_e \mapsto mk\text{-}S_{ct}(mk\text{-}NsPtr(p_f),\ e\ ,\underline{nil}),$
$p_f \mapsto mk\text{-}S_{ct}(mk\text{-}PaPtr(p_c),\ f\ ,\underline{nil}),$
$p_g \mapsto mk\text{-}S_{ct}(mk\text{-}PaPtr(p_a),\ g\ ,p_h),$
$p_h \mapsto mk\text{-}S_{ct}(mk\text{-}NsPtr(p_i),\ h\ ,\underline{nil}),$
$p_i \mapsto mk\text{-}S_{ct}(mk\text{-}NsPtr(p_j),\ i\ ,\underline{nil}),$
$p_j \mapsto mk\text{-}S_{ct}(mk\text{-}PaPtr(p_g),\ j\ ,\underline{nil}),$
$p_k \mapsto mk\text{-}S_{ct}(mk\text{-}NsPtr(p_l),\ k\ ,\underline{nil}),$
$p_l \mapsto mk\text{-}S_{ct}(\underline{nil},\ l\ ,\underline{nil})])$

Exercise: We leave it, as a non-trivial exercise, for the reader to define the $inv\text{-}DP_{ct}$ and $retr\text{-}DP$ functions.

The File Pointer Scheme, $HDAM_{fp}$

$$DP_{fp} :: (F \twoheadrightarrow Ptr)\ \ STG_{fp}$$
$$STG_{fp} = Ptr \twoheadrightarrow S_{fp}$$
$$S_{fp} :: [Ptr]\ \ (Intg(Sn \twoheadrightarrow VAL))\ \ (F \twoheadrightarrow Ptr)$$

-- and now the object representation is:

$mk\text{-}DP_{fp}([f_1 \mapsto p_a],[p_a \mapsto mk\text{-}S_{fp}(p_k,\ a\ ,[f_2 \mapsto p_b, f_3 \mapsto p_g]),$
$p_b \mapsto mk\text{-}S_{fp}(p_c,\ b\ ,[\,]),$
$p_c \mapsto mk\text{-}S_{fp}(\underline{nil},\ c\ ,[f_4 \mapsto p_d, f_5 \mapsto p_e]),$
$p_d \mapsto mk\text{-}S_{fp}(\underline{nil},\ d\ ,[\,]),$
$p_e \mapsto mk\text{-}S_{fp}(p_f,\ e\ ,[\,]),$
$p_f \mapsto mk\text{-}S_{fp}(\underline{nil},\ f\ ,[\,]),$
$p_g \mapsto mk\text{-}S_{fp}(\underline{nil},\ g\ ,[f_6 \mapsto p_h]),$
$p_h \mapsto mk\text{-}S_{fp}(p_i,\ h\ ,[\,]),$
$p_i \mapsto mk\text{-}S_{fp}(p_j,\ i\ ,[\,]),$
$p_j \mapsto mk\text{-}S_{fp}(\underline{nil},\ j\ ,[\,]),$
$p_k \mapsto mk\text{-}S_{fp}(p_l,\ k\ ,[\,]),$
$p_l \mapsto mk\text{-}S_{fp}(\underline{nil},\ l\ ,[\,])])$

Exercise: Formulate the $inv\text{-}DP_{fp}$ and $retr\text{-}DP$ functions.

The "inverse" of the *retr-DP* function, a so-called injection function, can be constructed. It is not "an exact" inverse. Its type is relational:

$$\underline{type}: \quad inj\text{-}DP_{fp}: DP \nrightarrow DP_{fp}$$

It satisfies:

$$(\forall dp \in DP)(retr\text{-}DP(inj\text{-}DP_{fp}(dp)) = dp)$$

but not:

$$(\forall dp_{fp} \in DP_{fp})(inj\text{-}DP_{fp}(retr\text{-}DP(dp_{fp})) = dp_{fp})$$

We design our $inj\text{-}DP_{fp}$ in stages, bottom up. First we define:

$$\underline{type}: \quad alloc: \quad \rightarrow (\Sigma \rightarrow \Sigma\ Ptr)$$
$$\underline{type}: \quad Allocate\ F\text{-}set \rightarrow (\Sigma \rightarrow \Sigma\ (F \overrightarrow{m}\ Ptr))$$

$$alloc() \triangleq (\underline{def}\ p \in Ptr\ \underline{dom}\ \underline{c}\ \text{Stg};$$
$$\text{Stg} := \underline{c}\ \text{Stg} \cup [p \mapsto \underline{undefined}];$$
$$\underline{return}(p))$$

$$Allocate(fs) \triangleq \underline{if}\ fs = \{\}$$
$$\underline{then}\ [\,]$$
$$\underline{else}\ (\underline{let}\ f \in fs\ \underline{in}$$
$$\underline{def}\ p: alloc();$$
$$\underline{def}\ m: Allocate(fs - \{f\});$$
$$\underline{return}([[f \mapsto p] \cup m]))$$

The model now is imperative:

$$\underline{dcl}\ \text{Stg} := [\,]\ \underline{type}\ STG_{fp};$$
$$\Sigma \quad = \quad (\text{Stg} \overrightarrow{m}\ STG_{fp})$$

$$\underline{type}: \quad inj\text{-}DP_{fp}: DP \rightarrow (\Sigma \rightarrow \Sigma\ (F \overrightarrow{m}\ Ptr))$$

$$inj\text{-}DP_{fp}(dp) \triangleq$$
$$(\underline{def}\ m: Allocate(\underline{dom}\ dp);$$
$$\underline{for\ all}\ f \in \underline{dom}\ dp\ \underline{do}\ inj\text{-}SL_{fp}(dp(f))(m(f), \underline{nil}, \underline{nil});$$
$$\underline{return}(m))$$

```
inj-SL_fp(sl)(cp,pp,s) ≜
  if sl=<>
    then  if pp=nil then I else Stg := c Stg + [pp ↦ s]
    else (if pp=nil
            then I
            else (let mk-S_fp(nil,r,m) = s in
                    Stg := c Stg + [pp ↦ mk-S_fp(cp,r,m)]);
          let mk-S(i,vm,dp) = hd sl in
          def m  : inj-DP_fp(dp);
          def np : if tl sl=<> then nil else alloc();
          inj-SL_fp(tl sl)(np,cp,mk-S_fp(nil,(i,vm),m)))
```

type: $inj\text{-}SL_{fp}: S^* \to ((Ptr \times [\ Ptr\ ] \times [\ S_{fp}\ ]) \to (\Sigma \to \Sigma))$

13.2 NETWORK DATABASE MANAGEMENT SYSTEMS

We no expand on section 12.3.1. The data model was:

$$DSD_0 :: (Fid \xrightarrow{m} R_0\text{-}set)(Sid \xrightarrow{m} ((Fid \times Fid)(R_0 \xrightarrow{m} R_0\text{-}set)))$$

We focus on just one "arrow", i.e. "DBTG-set":

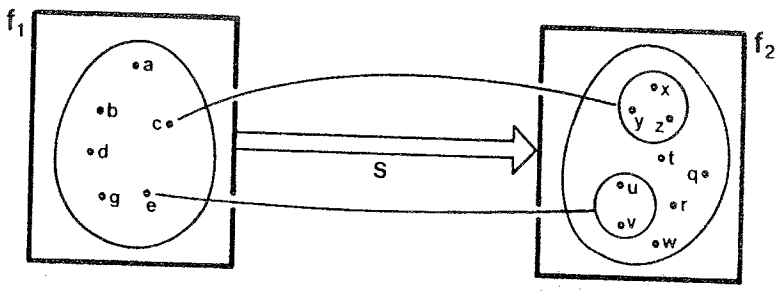

Fig. 5

As a *DSD* object it is represented by:

```
mk-DSD([f1 ↦ {a,b,c,d,e,g},
        f2 ↦ {x,y,z,u,v,w,q,r,t}],
       [s  ↦ ((f1,f2),[c↦{x,y,z},e ↦ {u,v}])])
```

We note a seeming duplication of objects: $x, y, z, y, v, e,$ and e, in both file objects and "DBTG-sets"s.

For the purposes of the development in this section we redraw the above example, ascribing a "box" to each record:

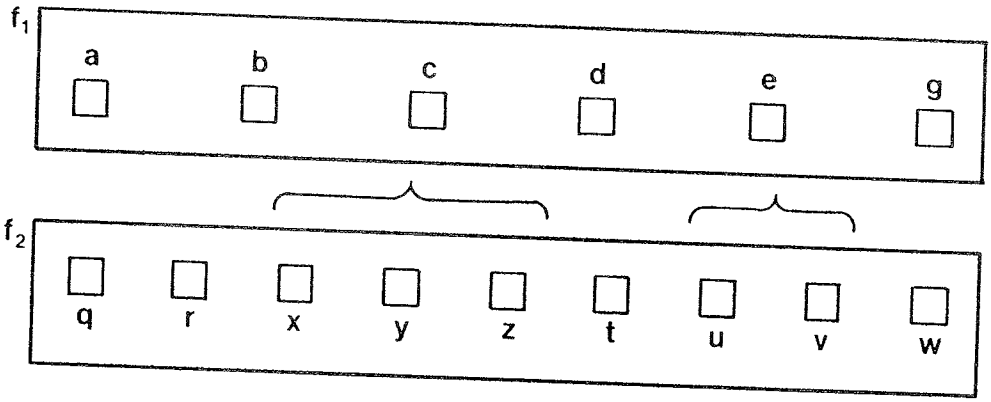

Fig. 6

We think, in a first step of development, of the "boxes" as allocated to disjoint storage cells. Each such record cell is uniquely designated by a pointer. Thus the R_0-*set* part of the files component, i.e. $(Fid \xrightarrow{m} R\text{-}set)$, is itself realized by a map: $(Ptr \xrightarrow{m} R)$. The $(R_0 \xrightarrow{m} R_0\text{-}set)$ component of the "DBTG-set" component, i.e. the braces of the above figure, is in this tentative, "trial" stage of development, realized as follows: with each owner record in the domain of $(R_0 \xrightarrow{m} R_0\text{-}set)$ is associated a pointer field to a potential arbitrary "first" member record (in the range of that map). And to each member record is associated a pointer field to a potential "next" member record:

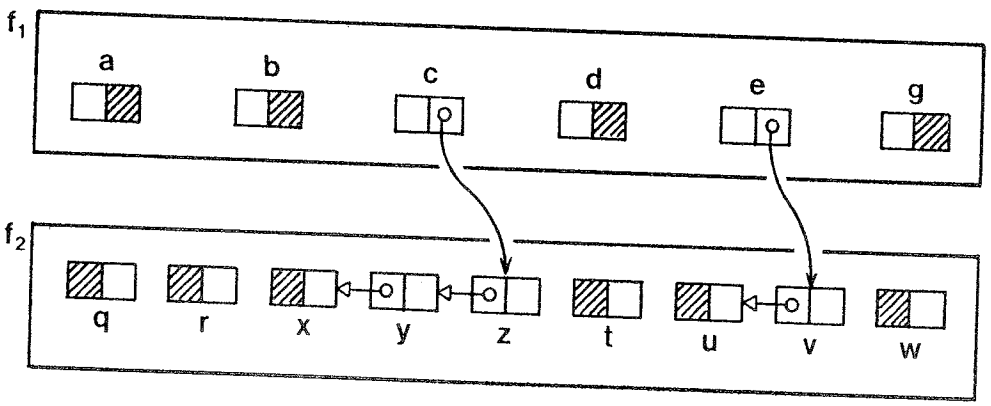

Fig. 7

This tentative implementation corresponds to a Domain definition of the realized DSD_0:

$$
\begin{array}{lll}
DSD_1 & :: & FILES_1 \quad SETS_1 \\
FILES_1 & = & (Fid \xrightarrow{m} (Ptr \xrightarrow{m} R_1)) \\
SETS_1 & = & (Sid \xrightarrow{m} (Fid \times Fid)) \\
R_1 & :: & R \ [\ Ptr\ ]
\end{array}
$$

-- with the above illustrated *DSD* object being represented by:

$$
\begin{aligned}
mk\text{-}DSD_1([\ f_1 \mapsto [\ & p_a \mapsto mk\text{-}R_1(\mathrm{a},\underline{nil}),\ p_d \mapsto mk\text{-}R_1(\mathrm{d},\underline{nil}),\\
& p_b \mapsto mk\text{-}R_1(\mathrm{b},\underline{nil}),\ p_e \mapsto mk\text{-}R_1(\mathrm{e},p_v),\\
& p_c \mapsto mk\text{-}R_1(\mathrm{c},p_z),\ p_g \mapsto mk\text{-}R_1(\mathrm{g},\underline{nil})],\\
f_2 \mapsto [\ & p_q \mapsto mk\text{-}R_1(\mathrm{q},\underline{nil}),\ p_t \mapsto mk\text{-}R_1(\mathrm{t},\underline{nil}),\\
& p_r \mapsto mk\text{-}R_1(\mathrm{r},\underline{nil}),\ p_n \mapsto mk\text{-}R_1(\mathrm{u},\underline{nil}),\\
& p_x \mapsto mk\text{-}R_1(\mathrm{x},\underline{nil}),\ p_r \mapsto mk\text{-}R_1(\mathrm{v},p_n),\\
& p_y \mapsto mk\text{-}R_1(\mathrm{y},p_x),\ p_w \mapsto mk\text{-}R_1(\mathrm{w},\underline{nil}),\\
& p_z \mapsto mk\text{-}R_1(\mathrm{z},p_y)\ ]\ ],\\
[s \mapsto (f_1,f_2)\ ])&
\end{aligned}
$$

But there is a problem, in fact three, the above realization is constrained: (A) it only permits at most one "arrow" incident upon or emanating from a file, (B) it makes the representation of records of any file dependent on their possibly partaking in some "DBTG-set", and (C) it does not permit the overlapping, or sharing of member records.

To solve all three problems at once we propose another realization. We first show an example of what the realization results in:

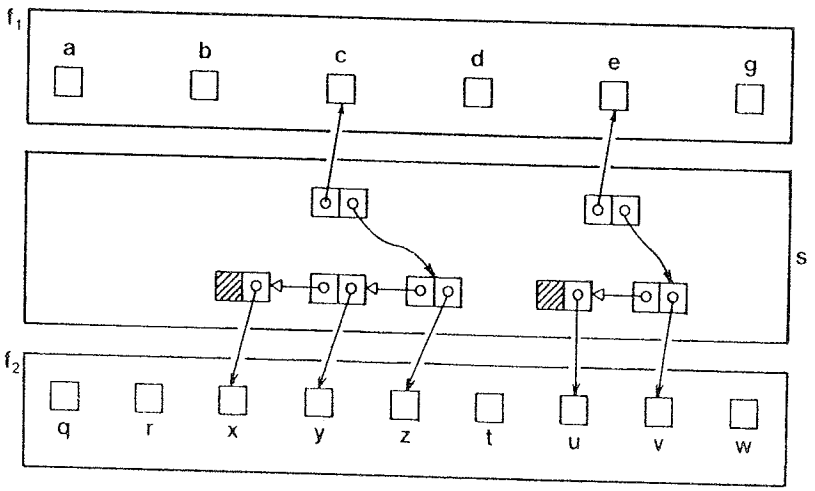

Fig. 8

We then explain the realization with respect to the previous "tentative" (but insufficient "realization") in terms of some surgical "operations": (i) we have cut away, from the owner and member records their pointer fields; (ii) we have then made these pointer parts into "records" themselves, (iii) joining to the new pointer fields containing pointers "back to" the owner, respectively the member records they were separated from. Now [(B)] owner- and member record representations are independent of the "DBTG-sets" they potentially partake of. Now [(A)] any file may take part in several "DBTG-sets". In fact [(C)]: to each such "DBTG-set" (or "arrow") there is exactly one pointer constellation as illustrated above, and shown "sandwiched" between the two files.

The Domain specification becomes:

$$
\begin{array}{lll}
DSD_2 & :: & FILES_2 \times SETS_2 \\
FILES_2 & = & (Fid \xrightarrow{m} (Ptr \xrightarrow{m} R)) \\
SETS_2 & = & (Sid \xrightarrow{m} ((Fid \times Fid) \times OWN \times MBR)) \\
OWN & = & Ptr_0 \xrightarrow{m} OR \\
MBR & = & Ptr_m \xrightarrow{m} MR \\
OR & :: & Ptr_r \times Ptr_m \\
MR & :: & [\ Ptr_m\ ] \times Ptr_r
\end{array}
$$

-- with the above illustrated *DSD* object being represented by:

$$
\begin{array}{l}
mk\text{-}DSD_2([\ f_1 \mapsto [\ p_a \mapsto a,\ p_b \mapsto b,\ p_c \mapsto c \\
\qquad\qquad p_d \mapsto d,\ p_e \mapsto e,\ p_g \mapsto g\ ], \\
\qquad f_2 \mapsto [\ p_q \mapsto q,\ p_r \mapsto r,\ p_x \mapsto x,\ p_y \mapsto y, p_z \mapsto z, \\
\qquad\qquad p_t \mapsto t,\ p_u \mapsto u,\ p_v \mapsto v,\ p_w \mapsto w\ ]], \\
\quad [\ s \mapsto ((f_1,f_2), \\
\qquad [op_c \mapsto mk\text{-}OR(p_c, mp_z), op_e \mapsto mk\text{-}OR(p_e, nil)], \\
\qquad [mp_z \mapsto mk\text{-}MR(mp_y, p_z), mp_y \mapsto mk\text{-}MR(mp_x, p_y), \\
\qquad\ mp_x \mapsto mk\text{-}MR(\underline{nil}, p_x), mp_v \mapsto mk\text{-}MR(mp_w, p_v), \\
\qquad\ mp_w \mapsto mk\text{-}MR(\underline{nil}, p_w)])])
\end{array}
$$

We have illustrated a stage of development: from the abstract *DSD* to the pointer-based, more concrete DSD_2. Subsequent development steps could now allocate all records: file records and owner and member pointer records of all files and all "DBTG-sets/arrows" in one pointer-based storage medium. Models of the CODASYL/DBTG notions of "areas" and "current-of" can, and probably should, be entered into our development before doing such an ultimate concretization.