

FORMALIZATION OF DATA MODELS

This chapter illustrates the use of VDM on database applications. Database architectures can be likened to programming languages; implementations can be likened to interpreters and compilers and are discussed in chapter 13. Although not the main purpose, this chapter introduces the reader to many of the current concepts in the database world (relational, hierarchical, and network approaches). The material comprises a careful demonstration of how models are constructed; their components; the interrelation of the components; and the choice of abstraction principles.

This chapter is based on [Bjørner 80c], but is extensively revised and enlarged. Other papers which apply VDM specification techniques to database problems include [Hansal 76a, Nilsson 76a, Owlett 77a, Owlett 79a, Bjørner 80e, Lamersdorf 80ab, Neuhold 80a, Lindenau 81a, Olnhoff 81a, Neuhold 81a, Bjørner 82c].

CONTENTS

12.0	Introductory Characterizations.....	381
12.1	The Relational Data Model.....	382
12.1.1	The Data Aggregates.....	382
12.1.2	The Operations.....	383
	An Algebraic Query Language.....	384
	A Predicate Calculus Query Language.....	390
12.2	The Hierarchical Data Model.....	396
12.2.1	Concepts of the Hierarchical Model.....	396
	Modelling the Schema.....	397
	The Hierarchical Model.....	401
12.2.2	Hierarchy Oriented Languages.....	406
	A Hierarchy Oriented Query Language.....	407
	Towards IMS.....	413
12.2.3	Selection Languages.....	416
	A Small Selection Language.....	422
	A Boolean Selection Language.....	425
12.2.4	Concluding Remarks on the Hierarchical Data Model.....	429
12.3	The Network Data Model.....	430
12.3.1	The Data Aggregate.....	430
12.3.2	The Operations.....	437

12.0 INTRODUCTORY CHARACTERIZATIONS

By a Data Model (DM) we shall simply understand a data type; that is: a set of objects and operations.

By a Database Management System (DBMS) we shall (corresponding simply) understand a system which supports the storing of the DM objects and the execution of the DM operations.

By a Database (DB) we shall understand a particular such collection of data being administered by a particular such DBMS.

Thus a DM is to us like a programming language. A DBMS, then, is like a processor (for example interpreter) for the DM. Finally a DB is like a specific program formulated in the DM language and interpreted on the DBMS processor for that language.

Like we can distinguish, in the area of "ordinary" programming languages, among "equivalence classes" of so-called ALGOL-like, LISP-like, SNOBOL-like, etc. languages (that is languages built up around rather distinct data type and programming construct ideas), so we will, at the moment distinguish between three kinds of Data Models.

Our treatment, in this chapter, will therefore focus on abstract specifications of these three Data Models. They are the so-called Relational, the Hierarchical and the Network Data Models. Chapter 13 will then sketch stages of developments towards Database Management Systems for the latter two of the Data Models, namely an IBM IMS-like DBMS for the hierarchical DM and a CODASYL/DBTG-like DBMS for the network DM.

Common to all three DMs is the ability to speak of their objects in isolation from the operations applicable to them. Our presentation will therefore, within each of the three models, be subdivided into a first part dealing with representational abstractions of data objects, followed by a second part dealing with operational abstractions of operations on data.

Operations on data, that is on the entire aggregate of the usually composite data object of a database, fall in two classes: meta-functions and functions. Meta-functions are operations concerned with manipulating a description of the data of the database; functions are operations

concerned with operating upon the data "itself", that is the non-descriptive parts. Some Data Models do not elaborate on this (so-called Schema-based) distinction. Our treatment will mostly focus on ordinary data functions. Such functions fall in two groups: data querying- and data manipulation commands. Data querying commands denote the evaluation, or extraction of data, $d \in D$, from, but no change to, the database data aggregate, $db \in DB$. Data manipulation commands denote that is interpret to a change of the database aggregate (from db to db').

type: *Val-Query*: $DB \rightarrow D$

type: *Int-Manip*: $DB \rightarrow DB$

Given a definition of the Domain of database objects one can, in general, define a variety of classes of for example query operations. This variety can be characterized by two extremes. These are sometimes referred to as procedural versus non-procedural operations. We shall term them algebraic, respectively logic operations. An algebraic operator basically speaking specifies how to extract data -- based on its form. A logic operator roughly speaking specifies what to extract -- based on its content, that is on properties. (We could as well call the operation classes: syntactic, respectively semantic.)

In our treatment we shall exemplify both kinds of operations in the relational DM. The sections on the two other DMs will only exemplify algebraic operations. We invite the reader to propose and properly formalize for example predicate calculus based query operations on the hierarchical and/or network DMs. There is nothing intrinsic in these latter DMs which prevent such a set of operations.

12.1 THE RELATIONAL DATA MODEL

12.1.1 The Data Aggregates

The major, or main, data (structure or) aggregate of the RDM is that of a set of uniquely named relations:

$$1. \quad RDS = Rnm \text{ } \overline{\text{REL}}$$

Each relation consists of an unordered collection of rows:

2. $REL = ROW\text{-}set$

(Rows are often referred to as 'tuples' -- not to be confused with the VDM specification language data type tuples.) Each row consists of a fixed number of distinctly attributed, that is 'named' element values. Two ways of modelling rows are possible:

3'. $ROW' = VAL^+$

models rows as tuples of values with these being individually 'named' by their index position. Since ordering among row elements is of no importance and since, moreover, one mostly prefers to refer to individual row elements by a more freely chosen attribute name we usually prefer the model rows by:

3". $ROW'' = \text{Attr} \rightarrow VAL$

All rows of any relation have the same number of similarly named elements:

4. 'is-wf-REL'(rel) $\underline{\Delta} (\forall r_1, r_2 \in rel)((\underline{len} \ r_1 = \underline{len} \ r_2) \wedge (...))'$

respectively:

4. "is-wf-REL"(rel) $\underline{\Delta} (\forall r_1, r_2 \in rel)((\underline{dom} \ r_1 = \underline{dom} \ r_2) \wedge (...))''$

We also illustrate, but do not detail, the well-formedness constraints which express, (...), that values from corresponding fields of any two rows must be of the same primitive ("scalar") type:

$(...)'' : (\forall n \in \underline{dom} \ r_1)(type(r_1(n)) = (type(r_2(n))))$

The issues at stake when choosing between ROW' and ROW'' will be illustrated as we turn to an explication of algebraic operations on and between relations.

12.1.2 The Operations

We shall illustrate two kinds of operations: algebraic and logic. The latter are embedded in a rather general form of a predicate calculus.

AN ALGEBRAIC QUERY LANGUAGE

The relational algebra consists, besides relations, of the following operations: select, project, θ -join and divide. We consider these to be denoted by objects of the like-named command Domains:

5. $Cmd = Sel \mid Proj \mid Join \mid Div$

We can now either base our further description on the tuple, that is ROW' , explication of rows, or on the map, that is ROW'' , or attribute named explication of rows. To illustrate the consequences of choosing one over the other we exemplify both alternatives, thus illuminating their consequences.

Before, however, detailing the formal definition of each of the individual command Domains we informally define their semantics.

Informal Semantics and Formal Syntax

Selection operates on a single relation, rel , and delivers the relation of all those rows, in rel , whose elements in given attribute positions equals correspondingly given values:

6.' Sel' $:: Rnm \quad (Nat1^+ VAL)$

6." Sel'' $:: Rnm \quad (Anm^+ VAL)$

Projection operates on a single relation, rel , and delivers the relation of rows each of which is a sub-segment of the rows of rel :

7.' $Proj'$ $:: Rnm \quad Nat1^+$

7." $Proj''$ $:: Rnm \quad Anm\text{-}set$

Join operates on two, not necessarily distinct relations, rel_1 and rel_2 . It forms the "composition" of exactly those rows, $r_1 \otimes r_2$, from rel_1 and rel_2 , which in pairwise given positions have equal values:

8.' $Join'$ $:: (Rnm \times Nat1^+) \quad (Nat1^+ \times Rnm)$

8." $Join''$ $:: (Rnm \times Anm^+) \quad (Anm^+ \times Rnm)$

(The \otimes operation is meta-linguistic. Its particular meaning will be defined, below, relative to each of the two row-variants.)

Division operates on two relations: the dividend, rel_a , and the divisor, rel_b . The quotient is a relation. It is a projection of rel_a with respect to the complement of dividend fields, il_a , with only those rows, symbolically: $x \oplus y$ in rel_a , contributing a sub-segment x all of whose corresponding (il_a -projected) y components form a relation which includes the il_b projected divisor. Thus to specify a divide command we require, besides the names of the dividend and divisor relations, the respective (complementing) row element positions (field selectors):

- 9.' Div' :: $(Rnm \times Nat1^+) (Nat1^+ \times Rnm)$
 9." Div'' :: $(Rnm \times Anm^+) (Anm^+ \times Rnm)$

Syntactic Well-formedness

Given just the syntactic command objects certain obvious constraints must be satisfied:

10. $is-wf-Proj'[mk-Proj'((,il))] \underline{\Delta} unique(il)$

where $unique$ is a function which checks that the elements of its argument list, al , are all unique, for example:

11. $unique(al) \underline{\Delta} (\forall i, j \in \underline{inds} \ al) (al[i]=al[j] \supset i=j)$

- 12.0 $is-wf-Join[mk-Join((,al_1),(al_2,))]$ $\underline{\Delta}$

- .1 $(\underline{len} \ al_1 = \underline{len} \ al_2)$
 [.2 $\wedge (unique(al_1) \wedge unique(al_2))]$

Where, optionally ([...] around line 12.2), we have expressed uniqueness of individual join fields.

- 13.0 $is-wf-Div[mk-Div((,al_1),(al_2,))]$ $\underline{\Delta}$

- .1 -- same as for $Proj$!

Types of $is-wf-...$ and $unique$ functions are:

14. $type: is-wf-... \ Cmd \rightarrow \text{BOOL}$
 15. $type: unique: (Nat1^+ \mid Anm^+) \rightarrow \text{BOOL}$

Semantic Constraints

Given commands in the context of a relational data system, $rds \in RDS$, additional constraints must be satisfied:

- 16.0 $pre-E_{cmd}[c](rds) \triangleq$
- .1 $\text{cases } c:$
 - .2 $\text{mk-Sel}'(r, ivm) \rightarrow ((r \in \underline{dom} \ rds) \wedge (rds(r) \neq \{\}))$
 - .3 $\wedge (\text{let } row \in rds(r) \text{ in}$
 - .4 $\underline{dom} \ ivm \subseteq \underline{inds} \ row))$,
 - .5 $\text{mk-Proj}''(r, ans) \rightarrow ((r \in \underline{dom} \ rds) \wedge (rds(r) \neq \{\}))$
 - .6 $\wedge (\text{let } row \in rds(r) \text{ in}$
 - .7 $ans \subseteq \underline{dom} \ row))$,
 - .8 $\text{mk-Join}'((r_1, l_1), (l_2, r_2)) \rightarrow ((r_1 \in \underline{dom} \ rds) \wedge (r_2 \in \underline{dom} \ rds)$
 - .9 $\wedge (rds(r_1) \neq \{\}) \wedge (rds(r_2) \neq \{\}))$
 - .10 $\wedge (\text{let } rw_1 \in rds(r_1),$
 - .11 $rw_2 \in rds(r_2) \text{ in}$
 - .12 $(\underline{elems} \ l_1 \subseteq \underline{inds} \ rw_1) \wedge$
 - .13 $(\underline{elems} \ l_2 \subseteq \underline{inds} \ rw_2)))$,
 - .14 $\text{mk-Div}'((r_1, l_1), (l_2, r_2)) \rightarrow ((r_1 \in \underline{dom} \ rds) \wedge (r_2 \in \underline{dom} \ rds)$
 - .15 $\wedge (rds(r_1) \neq \{\}) \wedge (rds(r_2) \neq \{\}))$
 - .16 $\wedge (\text{let } rw_1 \in rds(r_1),$
 - .17 $rw_2 \in rds(r_2) \text{ in}$
 - .18 $(\underline{elems} \ l_1 \subseteq \underline{dom} \ rw_1)$
 - .19 $\wedge (\underline{elems} \ l_2 \subseteq \underline{dom} \ rw_2)))$
 - 16.20 $\text{type: } pre-E_{cmd}: Cmd \rightarrow (RDS \rightarrow BOOL)$

The constraint $r \in \underline{dom} \ rds$ expresses that the relation name names a relation in the actual system (rds), and $rds(r) \neq \{\}$ that this relation is non-empty. The latter requirements are strictly speaking not necessary, but are included here to be able to test syntactic correctness of remaining command components. Normally the pre -checking could be, or is, done "against" (not the proper data part, but) a 'catalogue' describing all relations. We have not modelled such a 'schema' facility, one which could itself be a (specifically designated) relation. But we could easily do so.

Semantic Functions

Given the informal description semantics descriptions of the meaning of individual command categories it should now be easy to decipher:

17.0' $E\text{-Sel}'[mk\text{-Sel}'(r, ivm)](rds) \underline{\Delta}$
 .1' $\{ row \mid row \in rds(r) \wedge (\forall i \in \underline{dom} \ ivm)(row[i] = ivm(i)) \}$

17.0" $E\text{-Sel}''[mk\text{-Sel}''(r, avm)](rds) \underline{\Delta}$
 .1" $\{ row \mid row \in rds(r) \wedge (\forall a \in \underline{dom} \ avm)(row(a) = ivm(a)) \}$

Here the distinction between the two (the row-tuple and the row-map) models was almost invisible ($row[i]$ versus $row(a)$).

18.0' $E\text{-Proj}'[mk\text{-Proj}'(r, il)](rds) \underline{\Delta}$
 .1' $\{ \langle row[il[j]] \mid 1 \leq j \leq \underline{len} \ il \rangle \mid row \in rds(r) \}$

in contrast to:

18.0" $E\text{-Proj}''[mk\text{-Proj}''(r, as)](rds) \underline{\Delta}$
 .1" $\{ [a \mapsto row(a) \mid a \in as] \mid row \in rds(r) \}$

Next:

19.0' $E\text{-Join}'[mk\text{-Join}'((r_1, l_1), (l_2, r_2))](rds) \underline{\Delta}$
 .1' $\{ rw_1 \hat{=} rw_2 \mid (rw_1 \in rds(r_1)) \wedge (rw_2 \in rds(r_2))$
 .2' $\wedge (\forall i \in \underline{inds} \ l_1)(rw_1[l_1[i]] = rw_2[l_2[i]]) \}$

in sharper contrast to:

19.0a $E\text{-Join}''[mk\text{-Join}''((r_1, l_1), (l_2, r_2))](rds) \underline{\Delta}$
 .1a $\{ rw_1 \cup rw_2 \mid (rw_1 \in rds(r_1)) \wedge (rw_2 \in rds(r_2))$
 .2a $\wedge (\forall i \in \underline{inds} \ l_1)(rw_1[l_1[i]] = rw_2[l_2[i]]) \}$

This latter (specifically: $rw_1 \cup rw_2$) only works, that is is only well-defined provided line 20.4:

20.0 $pre\text{-}E\text{-Join}''[mk\text{-Join}''((r_1, l_1), (l_2, r_2))](rds) \underline{\Delta}$
 .1 $(\dots \text{repetition of 16.8-9} \dots)$
 .2 $\wedge (\underline{let} \ rw_1 \in rds(r_1),$
 .3 $\quad \quad \quad rw_2 \in rds(r_2) \quad \underline{in}$
 .4 $\quad ((\underline{dom} \ rw_1 \cap \underline{dom} \ rw_2 = \{\})$
 .5 $\quad \wedge (\underline{elems} \ l_1 \in \underline{dom} \ rw_1) \wedge (\underline{elems} \ l_2 \subseteq \underline{dom} \ rw_2))$

If we permit attribute (that is column) names of joined relations to be common then we must "invent" some "renaming" scheme, for example:

- 19.1b { [(r_1, a_1) \mapsto $rw_1(a_1)$ | $a_1 \in \underline{\text{dom}}\ rw_1$]
 .2b \cup [(r_2, a_2) \mapsto $rw_2(a_2)$ | $a_2 \in \underline{\text{dom}}\ rw_2$]
 .3b | ($rw_1 \in \text{rds}(r_1)$) \wedge ($rw_2 \in \text{rds}(r_2)$)
 .4b $\wedge (\forall i \in \underline{\text{inds}}\ l_1)(rw_1(l_2[i])) = rw_2(l_2[i]))$ }

which then replaces 19.1a"-2a". The map merge, \cup , in 19.1a" and the relation-attribute name pairing and merge, (\dots, \dots) and \cup , in 19.1b-2b, then "explains" the previously unexplained \otimes meta-operator.

Finally we formalize division. Without proof (of equivalence) we rephrase the earlier stated description of the division operator. Let $\text{mk-Div}((r_1, l_1), (l_2, r_2))$ be the divided command in question. First we construct an auxiliary relation rel from the dividend relation $\text{rds}(r_1)$ by projecting on the positions complementary to those listed in l_1 . Then we select only those rows, row , from rel , for which the following condition is satisfied. Namely the relation, rel' , formed from the divisor relation, $\text{rds}(r_2)$, by projections on l_2 , must be wholly contained in, that is a subset of, the relation formed by projections on l_1 of those rows row' in the dividend, $\text{rds}(r_1)$, whose complementing positions equal row .

- 21.0 $E\text{-Div}'[\text{mk-Div}'((r_1, l_1), (l_2, r_2))](\text{rds}) \underline{\Delta}$
 .1 $(\underline{\text{let}}\ (\text{dvd}, \text{dsr}) = (\text{rds}(r_1), \text{rds}(r_2)) \text{ in}$
 .2 $\text{cl} = \text{complement}(l_1, \text{dvd}) \text{ in}$
 .3 $\underline{\text{let}}\ (\text{rel}, \text{rel}') = (\text{project}(\text{dvd}, \text{cl}), \text{project}(\text{dsr}, l_2) \text{ in}$
 .4 $\{\text{row} \mid (\text{row} \in \text{rel})$
 .5 $\wedge \text{rel}' \subseteq \{p(\text{row}', l_1) \mid \text{row}' \in \text{dvd} \wedge \text{row} = p(\text{row}', \text{cl})\}) \}$

- 22.0 $\text{complement}(\text{rel}, l, \text{rel}) \underline{\Delta} (\underline{\text{let}}\ \text{rw} \in \text{rel} \text{ in}$
 $\langle i \mid 1 \leq i \leq \text{len}\ \text{rw} \wedge i \notin \underline{\text{elems}}\ l \rangle)$

- 23.0 $\text{project}(\text{rel}, l) \underline{\Delta} \{p(\text{row}, l) \mid \text{row} \in \text{rel}\}$

- 24.0 $p(\text{row}, l) \underline{\Delta} \langle \text{row}[l[i]] \mid 1 \leq i \leq \text{len}\ l \rangle$

where:

22. $\underline{\text{type}}:$ $\text{complement}: \text{Nat1}^+ \times \text{REL}' \rightarrow \text{Nat1}^+$
 23. $\underline{\text{type}}:$ $\text{project}: \text{REL}' \times \text{Nat1}^+ \rightarrow \text{REL}'$
 24. $\underline{\text{type}}:$ $p: \text{ROW}' \times \text{Nat1}^+ \rightarrow \text{ROW}'$

and, in general:

17-19. type: $E\text{-Cmd}$: $\text{Cmd} \leadsto (\text{RDS} \leadsto \text{REL})$

Other ways of specifying the semantic functions could be given. We next illustrate function definition by so-called pre-/post- conditions. If:

25. type: F : $A \rightarrow B$

then:

26. type: $\text{pre-}F$: $A \rightarrow \text{BOOL}$

27. type: $\text{post-}F$: $A \times B \rightarrow \text{BOOL}$

are the types of functions which determine the applicability of an A argument to F , that is specifies F 's domain, respectively give the property that any result, $b \in B$, must satisfy relative to an applicable argument $a \in A$:

28. $\text{pre-}F(a) \supset (\exists! b \in B)(F(a) = b)$

29. $\text{pre-}F(a) \wedge \text{post-}F(a, b) \supset (F(a) = b)$

30.0 $\text{pre-E-Sel}'([\text{mk-Sel}'(r, \text{ivm})], \text{rds}) \triangleq$

.1 $((r \in \text{dom } \text{rds})$

.2 $\wedge ((\text{rds}(r) \neq \{\}) \supset (\text{let } \text{row} \in \text{rds}(r) \text{ in } (\text{dom } \text{ivm} \subseteq \text{inds } \text{row}))))$

-- which we already specified in 16.2-4.

31.0 $\text{post-E-Sel}'([\text{mk-Sel}'(r, \text{ivm})], \text{rds}, \text{rel}) \triangleq$

.1 $((\text{rel} \subseteq \text{rds}(r))$

.2 $\wedge (\forall \text{row} \in \text{rel})(\forall i \in \text{dom } \text{ivm})(\text{row}[i] = \text{ivm}(i)))$

32.0 $\text{post-E-Proj}'([\text{mk-Proj}'(r, \text{il}), \text{rds}], \text{rel}) \triangleq$

.1 $(\forall \text{row} \in \text{rel})$

.2 $(\exists \text{rw} \in \text{rds}(r))$

.3 $((\text{len } \text{row} = \text{len } \text{il})$

.4 $\wedge (\forall i \in \text{inds } \text{row})(\text{row}[i] = \text{rw}[\text{il}[i]]))$

33.0 $\text{post-E-Join}'([\text{mk-Join}'((r_1, l_1), (l_2, r_2))], \text{rds}, \text{rel}) \triangleq$

.1 $(\forall \text{rw}_1 \in \text{rds}(r_1), \text{rw}_2 \in \text{rds}(r_2))$

.2 $((\text{rw}_1 \hat{\wedge} \text{rw}_2 \in \text{rel})$

.3 $\equiv (\forall i \in \text{inds } l_1)(\text{rw}_1[l_1[i]] = \text{rw}_2[l_2[i]]))$

etc.

A PREDICATE CALCULUS QUERY LANGUAGE

The predicate calculus based query language now to be illustrated is the basis for the SQL query language of IBMS System/R relational DBMS, but is otherwise based on DSL-alpha.

Syntax and Informal Semantics

A program in this language is a query. A query consists of three parts. The first part is a specification of the names of relations, possibly projected to individual columns, that is attributes, to be delivered as the result. This part is called a target specification list. The second part is a specification of which properties rows of these, possibly projected, relations must satisfy. This part is a predicate expression in which arbitrary identifiers may be used to stand for virtual relations. Which relations they stand for is specified in the third part, the virtual relation identifier to virtual relation expression. A virtual relation expression is a possibly operator/ operand expression which evaluates to a relation:

1. *Query* :: *Targ*⁺ (*Vid* \vec{m} *Range*) *Pred*
2. *Targ* :: *Vid* [*Nat1*]

Range expressions either denote existing relations directly or relations which are the set theoretic union, intersection or non-symmetric complement of two, possibly virtual relations:

3. *Range* = *Rnm* | *InfixR*
4. *InfixR* :: *Range* *SOp* *Range*
5. *SOp* = UNION | INTERSECTION | COMPLEMENT

Finally we analyze the syntax of predicates. Four kinds are provided: quantified, infix and negated propositional and atomic:

6. *Pred* = *QPred* | *IPred* | *NPred* | *APred*
7. *QPred* :: ((ALL|EXIST) *Tid* *Rnm*) *Pred*
8. *IPred* :: *Pred* (AND|OR) *Pred*
9. *NPred* :: *Pred*
10. *APred* :: *Term* *ROp* *Term*
11. *ROp* = LESEQ | LESS | EQ | NEQ | LAREQ | LARG

Quantified predicates $mk-QPred((q, t, r), p)$ representationally abstract $(\forall t \in r)(p)$ or $(\exists t \in r)(p)$ where $q = \underline{ALL}$, respectively $q = \underline{EXIST}$. Semantically they express whether p is true for all rows, respectively at least one, row, t , in the relation named r . Infix predicates express the conjunction or disjunction of two predicates. Negate predicates the negation of a predicate. Finally an Atomic predicate expresses relations between row elements and/or constant values, that is:

12. $Term = Elem \mid VAL$
13. $Elem :: (Vid \mid Tid) \times Nat1$

Well-formedness, or Context Constraints

It is clear that certain constraints on queries must be satisfied. These constraints are of two kinds referred to as internal and external constraints. Internal constraints express well-formedness of one query part with respect to another query part. External constraints express well-formedness with respect to existing relations of the database.

Examples of constraints are: [internal:] only virtual relation identifiers defined in the range part of a query can be referred to in the target specifications, [external:] and the row position used there must be in the interval of positions for tuples, that is rows, of the identified virtual relation. Similar for virtual relations mentioned in atomic predicates. In these latter, if a term refers to a row identifier, in Tid , [internal:] then the term must be in the scope of a quantification defining that row identifier, and [external:] the row element position of the term must likewise be in the internal of positions for tuples, that is rows, of the 'range' relation correspondingly named in the quantification, that is of r in $mk-QPred((q, t, r), p)$.

We therefore define a function, *Interval*, which when applied to an actual relation name, Rnm , and a relational data system, RDS , yields the index set of row tuples of the named relation:

13. type: $Interval: Rnm \rightarrow (RDS \rightarrow Nat1-set)$

- 13.0 $Interval(r)(rds) \underline{\Delta}$
 - .1 $(\underline{let} \text{ row} \in rds(r) \text{ in } \underline{inds} \text{ row})$
 - .2 $\underline{pre}: rds(r) \neq \{\}$

We express both internal and external constraints in one function:

- 14.0 $pre-E-Query[mk-Query(tl, rm, p)](rds) \underline{\Delta}$
 .1 $wfRanges[rm](rds)$
 .2 $\wedge wfTargl[tl](D(rm, rds))$
 .3 $\wedge is-wf-Pred[p](rds)(D(rm, rds))$
- 15.0 $WfRanges[rm](rds) \underline{\Delta}$
 .1 $(\forall r \in \underline{rng} \ rm)(is-wf-Range[r](rds))$
- 16.0 $is-wf-Range[rng](rds) \underline{\Delta}$
 .1 $\underline{cases} \ rng:$
 .2 $(mk-InfixR(r1, , r2) \rightarrow (is-wf-Range[r1](rds)$
 .3 $\wedge is-wf-Range[r2](rds)$
 .4 $\wedge I(r1, rds) = I(r2, rds))$
 .5 $T \rightarrow rng \in \underline{dom} \ rds)$
17. $\underline{type}: I: Range \ RDS \rightarrow Nat1-set$
- 17.0 $I(rng, rds) \underline{\Delta}$
 .1 $\underline{cases} \ rng: (mk-InfixR(r1, ,) \rightarrow I(r1, rds),$
 .2 $T \rightarrow Interval(rds(rng)))$
- In expressing well-formedness of target specification list and predicate we make use of a dictionary-like construction built from the range specification and recording the row tuple intervals of pertinent (virtual) relations:
18. $DICT = ((Vid \mid Tid) \multimap Nat1-set)$
19. $\underline{type}: D: (Vid \multimap Range) \ RDS \rightarrow DICT$
- 19.0 $D(rm, rds) \underline{\Delta} [r \mapsto I(rm(r), rds) \mid r \in \underline{dom} \ rm]$
- 20.0 $WfTargl[tl](\delta) \underline{\Delta}$
 .1 $(\forall t \in \underline{elems} \ tl)(is-wf-Targ[t](\delta))$
- 21.0 $is-wf-Targ[mk-Targ(v, i)](\delta)$
 .1 $(v \in \underline{dom} \ dict) \wedge (i \in \delta(v))$

- 22.0 $is-wf-Pred[p](rds)(\delta) \underline{\Delta}$
- .1 $\text{cases } p:$
 - .2 $(mk-QPred((,t,r),p') \rightarrow (r \in \underline{dom} \ rds)$
 - .3 $\wedge (\text{let } \delta' = \delta + [t \mapsto I(r,rds)] \text{ in}$
 - .4 $is-wf-Pred[p'](rds)(\delta'))),$
 - .5 $mk-IPred(p_1,,p_2) \rightarrow (is-wf-Pred[p_1](rds)(\delta)$
 - .6 $\wedge is-wf-Pred[p_2](rds)(\delta)),$
 - .7 $mk-NPred(p') \rightarrow is-wf-Pred[p'](rds)(\delta),$
 - .8 $mk-APred(t_1,,t_2) \rightarrow (is-wf-Term[t_1](\delta)$
 - .9 $\wedge is-wf-Term[t_2](\delta))$
- 23.0 $is-wf-Term[t](\delta) \underline{\Delta}$
- .1 $\text{cases } t: (mk-Elem(id,i) \rightarrow (id \in \underline{dom} \ \delta)$
 - .2 $\wedge (id \in \delta(id)),$
 - .3 $T \rightarrow \underline{true})$
14. $type: \ pre-E-Query: \ Query \rightarrow (RDS \rightarrow BOOL)$
15. $WfRanges: \ (Vid \ \overline{m} \ Range) \rightarrow (RDS \rightarrow BOOL)$
16. $is-wf-Range: \ Range \rightarrow (RDS \rightarrow BOOL)$
20. $WfTargl: \ Targ^+ \rightarrow (DICT \rightarrow BOOL)$
21. $is-wf-Targ: \ Targ \rightarrow (DICT \rightarrow BOOL)$
22. $is-wf-Pred: \ Pred \rightarrow (RDS \rightarrow (DICT \rightarrow BOOL))$
23. $is-wf-Term: \ Term \rightarrow (DICT \rightarrow BOOL)$

We have highlighted some, but not all aspects of checking the consistency of queries of a predicate-based language. We next turn to their semantics.

Formal Semantics

The general form of a query can be given schematically:

24. $mk-Query(<mk-Targ(v_i, in_i), mk-Targ(v_j, in_j), \dots, mk-Targ(v_k, in_k)>, \\ [v_1 \mapsto rng_1, v_2 \mapsto rng_2, \dots, v_n \mapsto rng_n] \\ predicate)$

We refer to the informal semantics subsection of this section for an informal wording of the semantics of such a query. We "translate" that (incomplete) specification into the complete formalization below. This formal definition is based on the following auxiliary constructs: For each of the virtual relation names, v , we construct the relation denoted

by its corresponding range expression, *rng*. We collect these in a table *rm* (25.1). This table, or map, thus pairs names to sets of rows. For each combination, *m*, of rows, one from each named virtual relation, we check (*E-Pred*) whether the *predicate* is satisfied. If so, we construct (*C*), from *m*, a projection based on the target list specification *tl*. And we do so for all combinations. Applicable such *m* contribute to the final answer which itself is a relation:

25.0 $E\text{-Query}[mk\text{-Query}(tl, irm, p)](rds) \underline{\Delta}$
 .1 $(\underline{\text{let}} \ rm = [v \mapsto E\text{-Range}[irm(v)](rds) \mid v \in \underline{\text{dom}} \ irm] \ \underline{\text{in}}$
 .2 $\{\underline{\text{conc}}(C(tl, m)) \mid m \in G(rm) \wedge E\text{-Pred}[p](m)(rds)\})$

25. $\underline{\text{type}}: \text{Query} \rightsquigarrow (RDS \rightsquigarrow REL')$

26.0 $E\text{-Range}[r](rds) \underline{\Delta}$
 .1 $\underline{\text{cases}} \ r:$
 .2 $(mk\text{-InfixR}(r_1, o, r_2)$
 .3 $\rightarrow (\underline{\text{let}} \ rel_1 = E\text{-Range}[r_1](rds),$
 .4 $\quad \quad \quad rel_2 = E\text{-Range}[r_2](rds) \ \underline{\text{in}}$
 .5 $\quad \quad \quad \underline{\text{cases}} \ o: (\underline{\text{UNION}} \rightarrow rel_1 \cup rel_2,$
 .6 $\quad \quad \quad \underline{\text{INTERSECTION}} \rightarrow rel_1 \cap rel_2,$
 .7 $\quad \quad \quad \underline{\text{COMPLEMENT}} \rightarrow rel_1 \setminus rel_2)),$
 .8 $\quad \quad \quad T \rightarrow rds(r))$

26. $\underline{\text{type}}: \text{Rang} \rightsquigarrow (RDS \rightarrow REL')$

27. $\underline{\text{type}}: E\text{-Pred}: \text{Pred} \rightarrow (TABLE \rightarrow (RDS \rightarrow \text{BOOL}))$

27.0 $E\text{-Pred}[p](map)(rds) \underline{\Delta}$
 .1 $\underline{\text{cases}} \ p:$
 .2 $(mk\text{-QPred}((q, t, r), p')$
 .3 $\rightarrow (\underline{\text{let}} \ rel = rds(r) \ \underline{\text{in}}$
 .4 $\quad \quad \quad \underline{\text{cases}} \ q:$
 .5 $\quad \quad \quad (\underline{\text{ALL}} \rightarrow (\forall row \in rel$
 .6 $\quad \quad \quad \quad \quad \quad (E\text{-Pred}[p'](map + [t \mapsto row])(rds),$
 .7 $\quad \quad \quad \underline{\text{EXISTS}} \rightarrow (\exists row \in rel$
 .8 $\quad \quad \quad \quad \quad \quad (E\text{-Pred}[p'](map + [t \mapsto row])(rds))),$
 .9 $\quad \quad \quad mk\text{-IPred}(p_1, o, p_2)$
 .10 $\quad \quad \quad \rightarrow (\underline{\text{let}} \ b_1 = E\text{-Pred}[p_1](map)(rds),$
 .11 $\quad \quad \quad \quad \quad \quad b_2 = E\text{-Pred}[p_2](map)(rds) \ \underline{\text{in}}$
 .12 $\quad \quad \quad \underline{\text{cases}} \ o: (\underline{\text{AND}} \rightarrow b_1 \wedge b_2, \underline{\text{OR}} \rightarrow b_1 \vee b_2)),$


```

.13      mk-NPred(p')
.14      → ¬E-Pred[p'](map)(rds),
.15      mk-APred(t1, o, t2)
.16      → (let e1 = E-Term[t1](map),
.17           e2 = E-Term[t2](map) in
.18      cases o: (LESEQ → e1 ≤ e2, LESS → e1 < e2
.19                EQ → e1 = e2, NEQ → e1 ≠ e2
.20                LAREQ → e1 ≥ e2, LARG → e1 > e2)))

```

where:

```

28.      TABLE = ((Vid | Tid) ↦ ROW')

```

```

29.0      E-Term[t](tbl) Δ
.1      cases t:
.2      (mk-Elem(id, i) → (tbl(id))[i],
.3      T → t)

```

```

29.      type: Term ↦ (TABLE ↦ VAL)

```

The two auxiliary functions G and C are finally defined. G takes an object in $(A \multimap B\text{-set})$ and delivers an object in $(A \multimap B)\text{-set}$. For each combination of $(a_1, b_{1k}), \dots, (a_m, b_{mj})$ in:

$$[a_1 \mapsto \{b_{11}, \dots, b_{1n_1}\}, \dots, a_m \mapsto \{b_{m1}, \dots, b_{mn_m}\}]$$

G delivers an element map:

$$[a_1 \mapsto b_{1k}, \dots, a_m \mapsto b_{mj}]$$

in the set $G(rm)$ of such maps:

```

30.      type: G: (Vid ↦ ROW-set) → (Vid ↦ ROW)-set
30.0      G(rm) Δ
.1      if rm = []
.2      then {}
.3      else {[v ↦ row] ∪ m | v ∈ dom rm ∧ row ∈ rm(v)
.4              ∧ m ∈ G(rm \ {v})}

```

and:

```

31.      type: C: Targ+ TABLE ↦ ROW+
31.0      C(tl, map) Δ
.1      <cases tl[i]:
.2      mk-Targ(v, nil) → map(v),
.3      mk-Targ(v, j) → (map(v))[i] > | 1 ≤ i ≤ len tl >

```

This concludes our treatment of the semantics of the [SQL-] (or DSL-alpha) predicate-based query language as basically used in the IBM System/R DBMS.

12.2 THE HIERARCHICAL DATA MODEL

In this section we describe the development of various models of database systems using the hierarchical model.

First, we introduce the hierarchical model and set up a data model of a very simple hierarchical database. Using this model, the semantics is defined for two simple query languages corresponding to the languages of the Information Management system (IMS) of IBM, and the SYSTEM 2000 of MRI.

12.2.1 Concepts of the Hierarchical Model

The aim of this section is to explain the concepts of the hierarchical data model, and to show how these may be formalized. We shall do this on basis of a traditional example from which we isolate the main concepts: the schema and the data hierarchy.

A Traditional Presentation

Hierarchical Database concepts are very often explained on the basis of a sample or "snapshot" database like the one in the figure below:

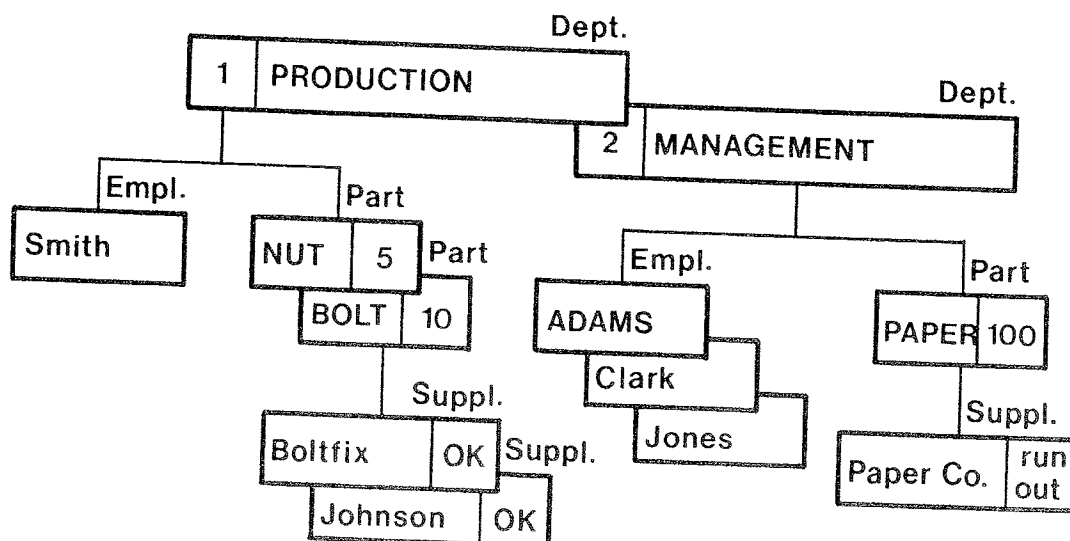


Fig. 1

Usually, one is told that the example is simple and unrealistic. But one can deduct the following laws from the diagram:

- (1) Data are grouped into "boxes" called records.
- (2) The records are arranged in a tree structure.
- (3) Some records share the same structure, called record type.
- (4) Records having the same type occur at the same level of the database.
- (5) Records of one type have children out of a certain set of other types. No record of another type has children of this type.
- (6) For all records of the same type, the parents of these share a type too.

(4) and (5) are consequences of (6). To summarize the specific relationships among types, a Hierarchical Definition tree, Hierarchy Chart, or Hierarchy Diagram is often given:

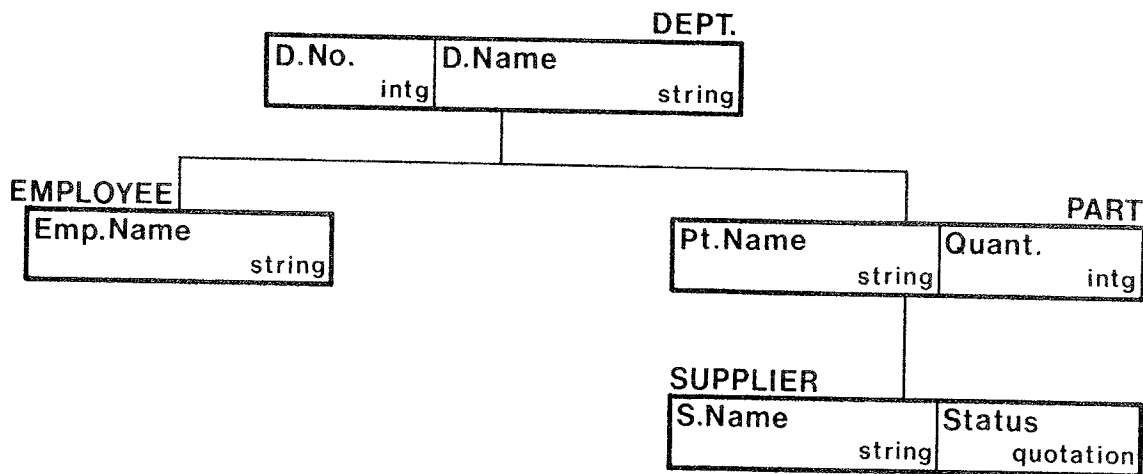


Fig. 2

The diagram shows the "pattern" of data in the database, and is sometimes called the schema of the database.

MODELLING THE SCHEMA

We start our modelling by considering a simplified Hierarchy Diagram like the left one on the next page:

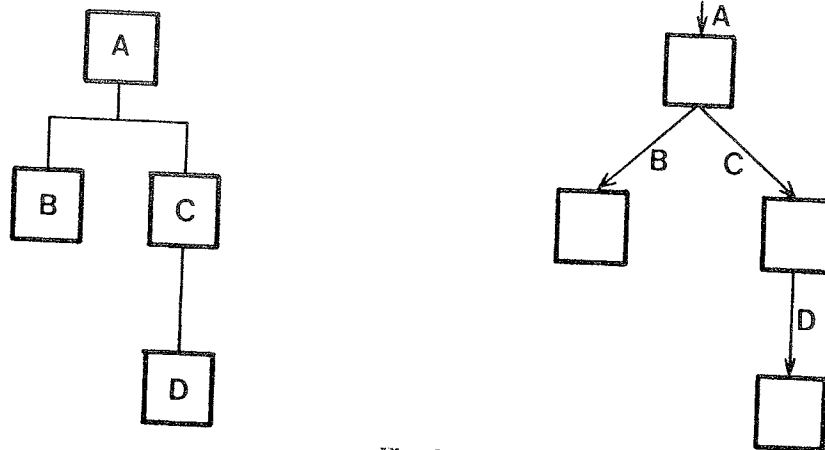


Fig. 3

We see that it is made up of record type names arranged in a tree structure. How do we model this? From the right hand diagram, which is just another way of picturing the one on the left, we more readily see that a Hierarchy Diagram (*HD*) can be modelled by:

$$1. \quad HD = RTId \rightarrow HD$$

where *RTId* are Record Type Identifiers which may be considered to be *Token*'s. An *HD*-object corresponding to the diagram above is thus:

$$2. \quad [A \mapsto [B \mapsto [], C \mapsto [D \mapsto []]]]$$

-- Extension

In this model it is possible to have several record types at the uppermost level, thereby introducing a forest of hierarchies corresponding to several databases. The tree view may be retained by introducing an imaginative anonymous "system" type being the parent of all upper level types:

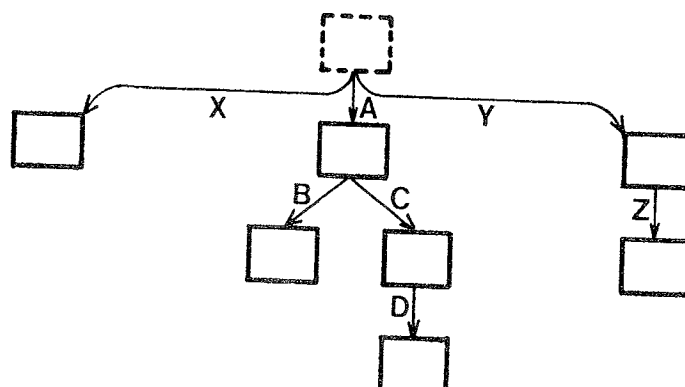


Fig. 4

The *HD* model thus shows the tree below the "system" type.

Inclusion of Record Types

The schema should also record the various record structures, the record types. How these may be included in the model depends on the constraints on the type names. Three policies arise:

- (a) Duplicate names allowed; they denote the same type.
- (b) Duplicate names allowed; they may denote different types.
- (c) Duplicate names not allowed.

In actual systems, case (c) is usually dominant.

Now, turning to the problem of including Record Types two alternatives seem to emerge. Either we can give the association of Record Types to Record Type Names in conjunction with the hierarchy, or the Record Types can be inserted in the hierarchy (at the place of the "boxes"). Introducing the term catalogue for the data structure that describes the "shape" of a database or the "pattern" of its data, we get:

- 3. $CTLG' :: HD (RTId \multimap RecTp)$
- 4. $CTLG = RTId \multimap (RecTp \times CTLG)$

We see that *CTLG'* is able to handle cases (a) and (c), but that *CTLG* can handle all three cases. We therefore choose *CTLG* being the most general model.

-- The Record Type

The Record Type should describe the structure of all records of a record class. Since a record is a collection of named data values, the Record Type must include the names of the data items, and the type of values allowed for each named item. We shall call the names for field identifiers (*FieldId*'s). The Types may be INTEGER, STRING, etc.:

- 5. $RecTp = FieldId \multimap TYPE$
- 6. $FieldId = Token$
- 7. $TYPE = \dots | \underline{INTEGER} | \dots$

Well-formedness

Having defined the whole catalogue we must decide whether to allow all such catalogues. Although not necessary for the model, we choose to apply name-constraint (c) since this is typical for actual systems and seems to be part of the hierarchical data model.

Thus, we do not allow any two type names in the whole catalogue to be identical. Using an auxiliary function which collects all names of a (sub) catalogue this can be formalized by:

```

8.1   $inv-CTLG(ctlg) \triangleq$ 
      .2     $(\forall id \in dom\ ctlg)$ 
      .3       $(\underline{let}\ (rectp, ctlg') = ctlg(id)\ \underline{in}$ 
      .4         $inv-RecTp(rectp) \wedge inv-CTLG(ctlg'))$ 
      .5     $\wedge (\forall id_1, id_2 \in dom\ ctlg)$ 
      .6       $(\underline{let}\ ids_1 = collect-names(s-CTLG(ctlg(id_1))),$ 
      .7         $ids_2 = collect-names(s-CTLG(ctlg(id_2)))\ \underline{in}$ 
      .8         $(ids_1 \cap dom\ ctlg = \{\}) \wedge (id_1 \neq id_2 \supset ids_1 \cap ids_2 = \{\}))$ 
      .9     $\underline{type}: CTLG \rightarrow Bool$ 

```

```

9.1   $collect-names(ctlg) \triangleq$ 
      .2     $dom\ ctlg \cup \underline{union}\ \{collect-names(s-CTLG(ctlg(id))) \mid id \in dom\ ctlg\}$ 
      .3     $\underline{type}: CTLG \rightarrow RTId-set$ 

```

$inv-RecTp$ will be only partially specified. Here we require at least one field to be present:

```

10.1   $inv-RecTp(rectp) \triangleq ((rectp \neq []) \wedge \dots)$ 

```

The Hierarchical Path Concept

A hierarchical path (HP) is a useful notion by which we shall understand a sequence of record type names which starts at (one of) the root(s) of the hierarchical diagram, and follows the branches of the tree. Thus:

```

11.   HP    = RTId+

```

The validity of such a path must be checked against a given catalogue:

- 12.1 $pre-HP(hp)(ctlg) \underline{\Delta}$
- .2 $\text{cases } hp:$
- .3 $\langle \rangle \rightarrow \underline{true},$
- .4 $\langle id \rangle^{\wedge hp'} \rightarrow id \in \underline{dom} \text{ } ctlg \wedge pre-HP(hp')(s-CTLG(ctlg(id)))$
- .5 $\underline{type}: HP \rightarrow (CTLG \rightarrow Bool)$

-- Some Hierarchical Path Operations

A Hierarchical Path may be used to select the corresponding sub-catalogue and record type:

- 13.1 $sub-catalogue(hp,ctlg) \underline{\Delta}$
- .2 $\text{cases } hp: \langle \rangle \rightarrow \underline{ctlg},$
- .3 $\langle id \rangle^{\wedge hp'} \rightarrow sub-catalogue(hp',s-CTLG(ctlg(id)))$
- .4 $\underline{type}: RTId * \times CTLG \rightarrow CTLG$

- 14.1 $lookup-rectp(hp,ctlg) \underline{\Delta}$
- .2 $(\underline{let} \ hp'^{\wedge \langle id \rangle} = hp \ \underline{in} \ s-Rectp(sub-catalogue(hp',ctlg)(id)))$
- .3 $\underline{type}: HP \times CTLG \rightarrow Rectp$

Finally, we will utilize our unique record type names to find the complete Hierarchical Path corresponding to such a name.

- 15.1 $find-hp(id,ctlg) \underline{\Delta}$
- .2 $(id \in \underline{dom} \text{ } ctlg \rightarrow \langle id \rangle,$
- .3 $T \rightarrow (\underline{let} \ id' = (\Delta id' \in \underline{dom} \text{ } ctlg)$
- .4 $(id \in \underline{collect-names}(s-CTLG(ctlg(id')))) \ \underline{in}$
- .5 $\langle id' \rangle^{\wedge find-hp(id,s-CTLG(ctlg(id')))) \)$
- .6 $\underline{type}: RTId \times CTLG \rightarrow HP$
- .7 $\underline{pre}: id \in \underline{collect-names}(ctlg)$

THE HIERARCHICAL MODEL

Record Instances

A record is a collection of named data values:

- 16. $Rec = FieldId \ \dot{m} \ VAL$
- 17. $VAL = \dots | Int | \dots$

A record is said to be an instance or occurrence of a record type if it has the "structure" prescribed by the record type, that is, it has

exactly the same field-names as present in the type, and the value of each named field belongs to the type associated with the name in the record type. Assuming a function *type-of* that returns the type of a value, we can formalize the above statement:

- 18.1 $\text{inv-Rec}(\text{rec})\text{rectp} \triangleq$
 .2 $(\text{dom } \text{rec} = \text{dom } \text{rectp}) \wedge$
 .3 $(\forall \text{fid} \in \text{dom } \text{rec}) (\text{type-of}(\text{rec}(\text{fid})) = \text{rectp}(\text{fid}))$
 .4 $\text{type: } \text{Rec} \rightarrow (\text{RecTp} \leadsto \text{Bool})$
19. $\text{type: } \text{type-of: VAL} \rightarrow \text{TYPE}$

Observe that the name-set is fixed (18.2) thereby prohibiting varying records etc.

Hierarchy Model

Consider part of the hierarchy diagram above:

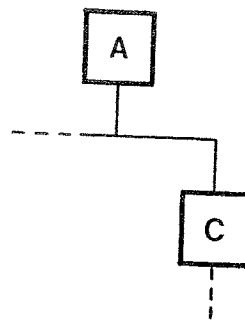


Fig. 5

The branch from *A* down to *C* indicates a 1:*n* relationship between *A*-records and *C*-records, that is to say:

- With each *A* record is associated a number (possibly zero) of *C* records called the *A* record's children of type *C*.
- With each *C*-record is associated exactly one *A* record called the parent of the *C*-record.

To illustrate these relations, a sample or "snapshot" database is often drawn:

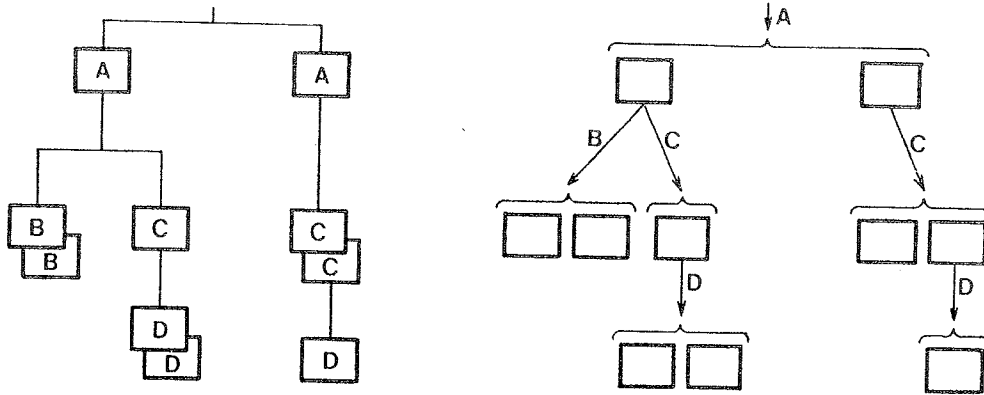


Fig. 6

The right side refiguring shows that we may model the database by:

20. $DB = RTid \# (Rec \times DB)\text{-set}$

21. $HDBS :: CTLG \quad DB$

In fact, the model covers several databases under the imaginary system-record. Finally, a Hierarchical Database System (*HDBS*) is defined as consisting of a Catalogue and a Database.

Well-formedness

Of course we will not accept all *DB*-objects as databases; only database s corresponding to some catalogue are allowed. We may try to exploit the database structure to see that sub, and sub-sub-records have the same structure etc. However it is much more convenient only to consider databases in connection with a catalogue. We therefore define a function to check that a database has the structure given by a catalogue, that is we define the invariant over the *HDBS* Domain:

- 22.1 $inv\text{-}HDBS(mk\text{-}HDBS(ctlg, db)) \underline{\Delta}$
- .2 $\underline{dom} \ db = \underline{dom} \ ctlg \wedge$
- .3 $(\forall id \in \underline{dom} \ db)$
- .4 $(\underline{let} \ (rectp, ctlg') = ctlg(id) \ \underline{in}$
- .5 $(\forall (rec, db') \in db(id))$
- .6 $(inv\text{-}Rec(rec)rectp \wedge$
- .7 $inv\text{-}HDBS(mk\text{-}HDBS(ctlg', db'))))$
- .8 $\underline{type}: HDBS \rightarrow Bool$

Note in 22.2 that we demand all Record Type names to be present in the database. As a result, the database part $[id \mapsto \{\}, \dots]$ is not equivalent to $[\dots]$.

Unique Identification

When defining the data manipulation languages, it is convenient to be able to uniquely identify each record occurrence. Since our model is a top-down model of the tree, such an identification must reflect this. That is, the identification should indicate how to reach the record starting at the database root. Therefore, to identify a record R we must at the top-level indicate which record occurrence contains R in its associated sub-database. For this sub-database, we must again identify the record of which R is a descendant.

At each level, the record occurrence is given by a record type t and an indication of which record of the record set of type t is chosen. To uniquely identify records within a record set we may either choose to require unique key fields of the records, or we may choose to assign a unique name or label to each record in the set. To lead up to the retrieval languages defined later in which no key fields are required, we choose the latter technique.

When Record labels are introduced, the database domain is redefined to:

23. $DB = RTId \overline{\mapsto} (RecLab \overline{\mapsto} (Rec \times DB))$
 24. $RecLab = Token$

We shall call the unique record identification a path:

25. $Path = (RTId \times RecLab)^*$

The record identifications from the upper level to the record level are given from left to right. The empty path identifies the imaginary "system record". Thus, the set of all records in a database can be represented by the set of all possible paths in the database:

- 26.0 $all_paths(db) \triangleq$
 .1 $\{ \langle \rangle \} \cup \{ \langle (id, l) \rangle^p \mid id \in dom\ db \wedge l \in dom\ db(id) \wedge$
 .2 $p \in all_paths(s-DB(db(id)(l))) \}$
 .3 type: $DB \rightarrow Path-set$

Given a path, it is often convenient to look up the record it designates and its associated sub-database:

- 27.0 $sub_database(p, db) \underline{\Delta}$
- .1 $cases\ p: \langle \rangle \rightarrow db$
 - .2 $\langle (id, lab) \rangle^{\wedge} p' \rightarrow sub_database(p', s_DB(db(id)(lab)))$
 - .3 $type: Path\ DB \rightleftharpoons DB$
 - .4 $pre: p \in all_paths(db)$
- 28.0 $lookup_rec(p, db) \underline{\Delta}$
- .1 $(let\ p'^{\wedge} \langle (id, lab) \rangle = p\ in$
 - .2 $s_Rec(subdatabase(p', db(id)(lab)))$
 - .3 $type: Path \rightarrow (DB \rightleftharpoons Rec)$
 - .4 $pre: p \in all_paths(db) \setminus \{\langle \rangle\}$

Note that the "system" record cannot be retrieved.

With the path identification of records, we see that one record is an ancestor of another if its path is an prefix of the other path. We shall say that two records are independent if none of them is an ancestor of the other. This may be formalized by:

- 29.1 $is_prefix(p_1, p_2) \underline{\Delta} (\exists p_3 \in Path) (p_2 = p_1^{\wedge} p_3)$
- .2 $type: Path \times Path \rightarrow Bool$
- 30.1 $indep(p_1, p_2) \underline{\Delta} \neg is_prefix(p_1, p_2) \wedge \neg is_prefix(p_2, p_1)$
- .2 $type: Path \times Path \rightarrow Bool$

SUMMARY OF A SIMPLE HIERARCHICAL DATABASE SYSTEM

These semantic domains form the basis for the languages defined in section 12.2.2 and 12.2.3.

- 31. $HDBS :: CTLG\ DB$
- 32. $CTLG = RTId \multimap (RecTp \times CTLG)$
- 33. $DB = RTId \multimap (RecLab \multimap (Rec \times DB))$
- 34. $RecTp = FieldId \multimap TYPE$
- 35. $Rec = FieldId \multimap VAL$
- 36. $TYPE = \underline{INTEGER} \mid \dots$
- 37. $VAL = Intg \mid \dots$
- 38. $RTId = Token$
- 39. $RecLab = Token$
- 40. $FieldId = Token$

12.2.2 A Hierarchy Oriented Query Language

In this section, we define a simple language that uses hierarchical paths as a basic concept. The DL/1 language of IMS does so. However, our language will be less procedural than DL/1.

Search String

The main idea in the hierarchy-oriented languages is that the records to be considered are denoted by essentially a hierarchical path augmented with qualifications at each level. The records selected are those of the last type of the path for which the qualifications for themselves and all their ancestors are satisfied. The syntactical construct whose purpose is to select records in the database for further actions is here called a Search String. We here assume that either the qualification is omitted, or it demands a given field to have a certain value.

44. $SearchStr = (RTId \times [Qual])^*$
 45. $Qual = (FieldId \times VAL)$

(The empty search string can (as usual) be considered to select the imaginary "system record".)

For a search string to be valid, the record types must follow a hierarchical path from the root and the last record type. Furthermore, at each level, the qualification must, if present, use a field of the corresponding record type and the value must be of the right type:

- 43.1 $pre-SearchStr[ss]ctlg \underline{\Delta}$
 .2 cases ss :
 .3 $(\langle \rangle \rightarrow true,$
 .4 $\langle (id, qual) \rangle^{\wedge ss'} \rightarrow id \in \underline{dom} \text{ ctlg} \wedge$
 .5 $(\underline{let} (rectp, \text{ctlg}') = \text{ctlg}(id) \text{ in}$
 .6 $(qual = \underline{nil} \vee pre-Qual[qual]rectp) \wedge$
 .7 $pre-SearchStr[ss']ctlg'))$
 .8 type: $SearchStr \rightarrow (CTLG \rightarrow Bool)$
- 44.1 $pre-Qual[qual]rtp \underline{\Delta}$
 .2 $(\underline{let} (fid, v) = qual \text{ in } fid \in \underline{dom} \text{ rtp} \wedge \text{type-of}(v) = rtp(fid))$
 .3 type: $Qual \rightarrow (RecTp \rightarrow Bool)$

-- Selection

As all the ancestors have to fulfil their qualification for an "end" record to be selected, we may start our search at the root of the database. Since the selected set is given by the set of paths identifying the records, the meaning of evaluating a search string may then be formally defined by:

- 45.1 $eval-SearchStr[ss]db \triangleq$
 - .2 $cases\ ss:$
 - .3 $(\langle \rangle \rightarrow \{\langle \rangle\},$
 - .4 $\langle (id, q) \rangle \wedge ss'$
 - .5 $\rightarrow (\underline{let}\ rs = db(id) \quad \quad \quad \underline{in}$
 - .6 $\quad \underline{let}\ rs' = rs | \{ l \mid l \in dom\ rs \wedge$
 - .7 $\quad \quad \quad satisfy(s-Rec(rs(l), q) \} \underline{in}$
 - .8 $\quad \langle (id, l) \rangle \wedge p | l \in dom\ rs' \wedge$
 - .9 $\quad \quad \quad p \in eval-SearchStr[ss']s-DB(rs'(l)))$
 - .10 $type: SearchStr \ni (DB \ni Path-set)$
 - .11 $pre: (\exists ctlg \in CTLG)(inv-HDBS(mk-HDBS(ctlg, db))$
-
- 46.1 $satisfy(r, q) \triangleq ((q = nil) \vee (\underline{let}\ (fid, v) = q \underline{in}\ r(fid) = v))$
 - .2 $type: Rec \times [Qual] \ni Bool$
 - .3 $pre: (\exists rtp \in RecTp)(inv-Rec(r)rtp) \wedge pre-Qual[q]rtp)$

A HIERARCHICY ORIENTED LANGUAGE

Now, having defined the basic concept of our language, we are ready to set up a full set of commands covering retrieval, insertion, deletion, and updating.

47. $Cmd = Search \mid Insert \mid Delete \mid Update$
48. $Search :: SearchStr$
49. $Insert :: SearchStr\ RTId\ Rec$
50. $Delete :: SearchStr$
51. $Update :: SearchStr\ FieldId\ Op$
52. $Op = \dots$

All of these commands are data functions, that is they do not modify, or directly list catalogue information although they use it for accessing the database properly. Informally we wish the semantics of the commands to be as follows:

Search: The result is the set of records identified by the search string.

Insert: The record, which must be of the given type, is inserted directly below each record denoted by the search string.

Delete: Each record denoted by the search string is deleted, and with it all of its descendants.

Update: The given field of each record denoted by the search string is modified by the given operation. *Op* is a syntactical expression which denotes a value modifying function (for example "+5" denoting $\lambda x.x+5$). However, we are not interested in the abstract syntax of this function, and will assume functions to check and evaluate such an object.

Well-formedness, Pre-Conditions

The commands must satisfy certain constraints as indicated above. The precondition predicate is defined by cases below. It uses only catalogue information:

53. type: $pre-Cmd: Cmd \rightarrow (CTLG \rightarrow Bool)$

Of course, for all commands the search string must be well-formed. For the *Search* command, this is the only condition to be checked:

54.1 $pre-Cmd[mk-Search(ss)]ctlg \underline{\Delta} pre-SearchStr[ss]ctlg$

For an *Insert* command it must be checked that the given record type is in continuation of the search string, that is that the type is immediately below the type designated by the hierarchical path of the search string (55.5). Also, the given record must belong to this type:

55.1 $pre-Cmd[mk-Insert(ss, rtpid, rec)]ctlg \underline{\Delta}$
 .2 $pre-SearchStr[ss]ctlg \wedge$
 .3 $(\underline{let} \ hp = extract-hp[ss] \quad \underline{in}$
 .4 $\underline{let} \ ctlg' = sub-catalogue(hp, ctlg) \quad \underline{in}$
 .5 $rtpid \in \underline{dom} \ ctlg' \wedge$
 .6 $(\underline{let} \ (rectp,) = ctlg'(rtpid) \quad \underline{in}$
 .7 $inv-Rec(rec)rectp))$

- 56.1 $extract-hp[ss] \underline{\Delta} < id \mid 1 \leq i \leq len\ ss \wedge ss[i] = (id,) >$
 .2 $\underline{type}: SearchStr \rightarrow HP$

With the *Delete* command, one is not allowed to delete the "system record":

- 57.1 $pre-Cmd[mk-Delete(ss)]ctlg \underline{\Delta} ss \neq \langle \rangle \wedge pre-SearchStr[ss]ctlg$

For an *Update* command it must be checked that the search string does not designate the system record, that is is empty. Furthermore, the field must belong to the record type of the search string, and the type of the operator must be applicable to the field. Given:

58. $\underline{type}: is-wf-Op: Op \rightarrow Bool$ left unspecified
 and
 59. $\underline{type}: operator\ type: Op \ni Type$ left unspecified
 we get:

- 60.1 $pre-Cmd[mk-Update(ss, fid, op)]ctlg \underline{\Delta}$
 .2 $ss \neq \langle \rangle \wedge pre-SearchStr[ss]ctlg \wedge$
 .3 $(\underline{let}\ hp = extract-hp[ss] \underline{in}$
 .4 $\underline{let}\ rectp = lookup-rectp(hp, ctlg) \underline{in}$
 .5 $\underline{fid} \in \underline{dom}\ rectp \wedge rectp(fid) = operator\ type[op] \wedge pre-Op[op]$

Semantic Functions

The meaning of a command depends on its kind:

- 61.1 $elab-Cmd[cmd]hdb \underline{\Delta}$
 .2 $(is-Search(cmd) \rightarrow eval-Search[cmd]hdb)$
 .3 $is-Insert(cmd) \rightarrow int-Insert[cmd]hdb$
 .4 $is-Delete(cmd) \rightarrow int-Delete[cmd]hdb$
 .5 $is-Update(cmd) \rightarrow int-Update[cmd]hdb$
 .6 $\underline{type}: Cmd \rightarrow (HDBS \ni (Rec-set \mid HDBS))$
 .7 $\underline{pre}: pre-Cmd[cmd]hdb$

The commands are divided into two groups: the retrieval command and the modifying commands, respectively *Search* and *Insert*, *Delete* & *Update*.

The retrieval command only extracts information from the database:

62. $\underline{type}: eval-Search: Search \rightarrow (HDBS \ni Rec-set)$

whereas the modifying commands only alters the data of the *HDBS*:

63. type: *int-Insert*: *Insert* \rightarrow (*HDBS* \leadsto *HDBS*)
 64. type: *int-Update*: *Update* \rightarrow (*HDBS* \leadsto *HDBS*)
 65. type: *int-Delete*: *Delete* \rightarrow (*HDBS* \leadsto *HDBS*)

-- Retrieval

The result of the *Search* command is the set of records identified by the search string. Since the search string does not specify any order among the selected records, the result is likewise unordered, that is a set:

- 66.1 *eval-Search*[*mk-Search*(*ss*)] *mk-HDBS*(*ctlg*, *db*) Δ
 .2 (let *paths* = *eval-SearchStr*[*ss*]*db in*
 .3 {*lookup-rec*(*p*, *db*) | *p* \in *paths* \ {*<>*}})

Note that the "system record" cannot be yielded.

-- Modification Functions

The modification commands usually change some sub-database, but since our model is top down, this has to be reflected all the way up to the root. Therefore, it seems convenient to have one function which, given a "reference" to a sub-database (that is a path) and a change-function for this subdatabase, propagates the change all the way up to the root. Suppose only one sub-database is to be changed. Then the required function could be:

- 67.1 *modify1*(*p*, *mod*, *db*) Δ
 .2 cases *p*:
 .3 (*<>* \rightarrow *mod*(*db*),
 .4 *<(id, lab)>^p' \rightarrow (let (*rec*, *db'*) = (*db*(*id*))(*lab*)* in
 .5 let *db''* = *modify1*(*p'*, *mod*, *db'*) in
 .6 *db* + [*id* \mapsto *db*(*id*) + [*lab* \mapsto (*rec*, *db''*)]])
 .7 type: *Path* \times (*DB* \rightarrow *DB*) \times *DB* \leadsto *DB*
 .8 pre: *p* \in *all-paths*(*db*)

This function may also be specified indirectly, using:

- 69.1 *indep-paths*(*p*, *db*) Δ {*p'* | *p'* \in *all-paths*(*db*) \wedge *indep*(*p*, *p'*)}
 .2 type: *Path* \times *DB* \leadsto *Path-set*
 .3 pre: *p* \in *all-paths*(*db*)

- 68.1 type: $\text{modify1'}: \text{Path} \times (\text{DB} \rightarrow \text{DB}) \times \text{DB} \leadsto \text{DB}$
- .2 $\text{pre-modify1'}(p, \text{mod}, \text{db}) \underline{\Delta} p \in \text{all-paths}(\text{db})$
- .3 $\text{post-modify1'}(p, \text{mod}, \text{db})(\text{db}') \underline{\Delta}$
- .4 $(p \in \text{all-paths}(\text{db}'))$
- .5 $\wedge \text{indep-paths}(p, \text{db}) = \text{indep-paths}(p, \text{db}')$
- .6 $\wedge (\forall p' \in \text{indep-paths}(p, \text{db}))$
- .7 $(\text{lookup-rec}(p', \text{db}) = \text{lookup-rec}(p', \text{db}'))$
- .8 $\wedge \text{sub-database}(p, \text{db}') = \text{mod}(\text{sub-database}(p, \text{db}))$

Here, the case will be that many sub-databases should be changed using the same change function. The sub-databases should be independent to get a deterministic effect. Such a modification function may be written in many ways. We could do the changes one at a time, but in unspecified order:

- 70.1 $\text{modify}(ps, \text{mod}, \text{db}) \underline{\Delta}$
- .2 if $ps = \{\}$
- .3 then db
- .4 else (let $p \in ps$ in
- .5 let $db' = \text{modify1}(p, \text{mod}, \text{db})$ in
- .6 $\text{modify}(ps \setminus \{p\}, \text{mod}, db')$)
- .7 type: $\text{Path-set} \times (\text{DB} \rightarrow \text{DB}) \times \text{DB} \leadsto \text{DB}$
- .8 pre: $ps \subseteq \text{all-paths}(\text{db}) \wedge (\forall p, p' \in ps) (p \neq p' \supset \text{indep}(p, p'))$

A rather mechanic solution. An implicit specification may easily be given changing (in 68.) p to ps , (in 68.2,4) \in to \subseteq , extending indep-paths to handle sets, and finally changing (68.8) to:

- 68.8' $\wedge (\forall p \in ps) (\text{sub-database}(p, \text{db}') = \text{mod}(\text{sub-database}(p, \text{db})))$

-- Modification Commands

Having defined the modification function we are now able to give the semantics of the modifying commands. For the *Insert* command, the change is to add a new record associated with an empty subdatabase:

```

71.1  int-Insert[mk-Insert(ss,rtpid,rec)] mk-HDBS(ctlg,db) Δ
      .2    (let hp = extract-hp[ss]^<rtpid>
      .3    let ps = eval-SearchStr[ss]db in
      .4    let empty = [id → [] | id ∈ dom sub-catalogue(hp,ctlg)] in
      .5    let mod(subdb) = in
      .6    (let lab ∈ RecLab \ dom subdb(rtpid) in
      .7    subdb + [rtpid ↦ subdb(rtpid) ∪ [lab ↦ (rec,empty)]]) in
      .8    mk-HDBS(ctlg,modify(ps,mod,db)))

```

For the *Delete* command we have to view the search string as consisting of two parts. The first $len\ ss - 1$ elements which identify the records which are to stay and which records in their sub-database are to be deleted, and the last element which selects the records of the sub-database to be deleted. Using:

```

72.1  satisfying-labels(rs,q) Δ { l | l ∈ dom rs ∧ satisfy(rs(l),q) }
      .2  type: (RecLab → (Rec × HDB)) × [Qual] → RecLab-set

```

we get:

```

73.1  int-Delete[mk-Delete(ss)] mk-HDBS(ctlg,db) Δ
      .2    (let ss'^(id,qual) = ss
      .3    let ps = eval-SearchStr[ss']db in
      .4    let mod(subdb) = in
      .5    (let rs = subdb(id)
      .6    let rs' = rs \ satisfying-labels(rs,qual) in
      .7    subdb + [id ↦ rs']) in
      .8    mk-HDBS(ctlg,modify(ps,mod,db))) in

```

The *Update* command goes almost the same way:

```

74.1  int-Update[mk-Update(ss,fid,op)] mk-HDBS(ctlg,db) Δ
      .2    (let ss'^(id,qual) = ss
      .3    let ps = eval-SearchStr[ss']db in
      .4    let f(r) = r + [fid ↦ eval-Op[op](r(fid))] in
      .5    let mod(subdb) = in
      .6    (let rs = subdb(id)
      .7    let ls = satisfying-labels(rs,qual) in
      .8    let rs' = rc + [l ↦ (f(r),db) | l ∈ ls ∧ (r,db) = rs(l)] in
      .9    subdb + [id ↦ rs']) in
      .10   mk-HDBS(ctlg,modify(ps,mod,db))) in

```

TOWARDS IMS

We shall, as the final subject of our treatment of hierarchical search languages, indicate how the principles given here may be extended to the so-called traversal languages like DL/1 of IMS. An IMS based model on these ideas can be found in [Bjørner 82c]. Here too, we shall use IMS as a reference for our discussion of traversal languages.

The main differences between the language given so far and IMS is that in IMS, the records are accessed one at a time, and in a certain order called the hierarchical order or sequence. Here we shall show how this will influence our model. For brevity, we will consider only record retrieval.

Hierarchical Order

The hierarchical ordering of the records corresponds to a parent-first, left-to-right traversal of the database tree when drawn as a diagram. The parent-first part of the ordering is given by the top-down structure, but the left-to-right ordering must be established explicitly at each level, that is in each sub-database. The records of a sub-database are first of all ordered by their record types, in the same way as the record types in each sub-catalogue are ordered as from left to right in the hierarchical diagram. Within each record type of a sub-database, the records must be ordered by some means, for example by the value of a key field.

There are various ways to incorporate the hierarchical ordering into our data model. To determine the ordering among record types in each sub-catalogue, we could add an extra component to the catalogue:

75. $CTLG :: (RTId \rightarrow RecTp) \rightarrow Ord$

The order component should impose some ordering on the record types of the catalogue, for example by associating an ordinal number to each type, or by establishing a "chain":

76. $Ord = RTId \rightarrow Nat$ or

77. $Ord = RTId \rightarrow [RTId]$

The ordering of records within each record type could be done through a

key field, but since IMS does not require unique keys this is not applicable here. The ordering could also be given by an ordering component as above, where the ordinal number case would be equivalent to arranging the records in a list:

78. $DB = RTId \rightarrow (Rec \times DB)^*$

However, since we do not wish to change our established data model and associated operations here, we shall simply assume that we implicitly, at each level, have access to a total, irreflexive, asymmetric, and transitive ordering operation among record types \ll_{tp} , and to one among record labels \ll_{lab} , where $x \ll y$ models that x is to the left of y in the diagram.

It is easily seen that the hierarchical ordering among the records corresponds to a lexicographical ordering on the paths identifying the records. Therefore, we can define that one record given by the path p_1 precedes a record with path p_2 in the hierarchical sequence by:

79.1 $precedes(p_1, p_2) \triangleq$
 .2 $(p_2 = \langle \rangle \rightarrow \underline{false},$
 .3 $p_1 = \langle \rangle \rightarrow \underline{true},$
 .4 $T \rightarrow (\underline{let} \langle id_1, lab_1 \rangle \wedge p_1' = p_1 \text{ in}$
 .5 $\underline{let} \langle id_2, lab_2 \rangle \wedge p_2' = p_2 \text{ in}$
 .6 $(id_1 \neq id_2 \rightarrow id_1 \ll_{tp} id_2,$
 .7 $lab_1 \neq lab_2 \rightarrow lab_1 \ll_{lab} lab_2,$
 .8 $T \rightarrow precedes(p_1', p_2'))))$
 .9 $\underline{type}: Path \times Path \rightarrow Bool$

We shall also need a function that given a non-empty set of paths returns the first of these according to the hierarchical ordering:

80.1 $first(paths) \triangleq (\Delta p \in paths)(\forall p' \in paths)(p \neq p' \rightarrow precedes(p, p'))$
 .2 $\underline{type}: Path\text{-}set \rightarrow Path$

Record retrieval

In IMS, only one record is retrieved at a time. Therefore, in order to achieve the effect of our Search command, that is to get all records satisfying the search string, two commands are provided:

81. *GetUnique* :: *SearchStr*
 82. *GetNext* :: [*SearchStr*]

Get Unique will return the first record in the hierarchical sequence satisfying the search string. It will furthermore establish what is called current position of this record. This position is used in *Get Next* commands which will return the first record following the current position and satisfying the search string, and will set 'current position' to this record. If no search string is given, *Get Next* will return the record immediately after 'current position' thereby allowing a complete traversal of the database in hierarchical order. The 'current position' is held as a separate component of the database system:

83. *HDBS'* :: *CTLG DB POS*
 84. *POS* = *Path*

If no record can be found, we indicate this by returning the nil object. Since 'current position' is changed by the retrieval, the type of the interpretations functions for these commands become:

85. type: *int-GetUnique*: *GetUnique* \ni (*HDBS'* \ni *HDBS'* \times [*Rec*])
 86. type: *int-GetNext*: *GetNext* \ni (*HDBS'* \ni *HDBS'* \times [*Rec*])
 87.1 *int-GetUnique*[*mk-GetUnique(ss)*]*hdb*s Δ
 .2 (let *mk-hdb*s'(*ctlg*,*db*,*pos*) = *hdb*s in
 .3 let *paths* = *eval-SearchStr*[*ss*]*db* in
 .4 *paths*={ } \rightarrow (*hdb*s,nil),
 .5 *T* \rightarrow (let *path* = *first*(*paths*) in
 .6 (*mk-HDBS'*(*ctlg*,*db*,*path*),*lookup-rec*(*path*,*db*))
 88.1 *int-GetNext*[*mk-GetNext(ss)*]*hdb*s Δ
 .2 (let *mk-HDBS'*(*ctlg*,*db*,*pos*) = *hdb*s in
 .3 let *paths* = (*ss*=nil \rightarrow *all-paths*(*db*),
 .4 *T* \rightarrow *eval-SearchStr*[*ss*]*db* in
 .5 let *paths'* = { *p* | *p* \in *paths* \wedge *precedes*(*pos*,*p*) } in
 .6 *paths'*={ } \rightarrow (*hdb*s,nil),
 .7 *T* \rightarrow (let *path* = *first*(*paths*) in
 .8 (*mk-HDBS'*(*ctlg*,*db*,*path*),*lookup-rec*(*path*,*db*))

IMS also maintains a position called Parent Position which is used by a so-called *Get Next Within Parent* command. Furthermore, parts of the

search string (which is called Search Argument List in IMS) may be left out. The detailed modelling of these facilities can be found in [Bjørner 82c].

12.2.3 Selection Languages

In this section we describe and model the characteristics of languages used in hierarchical database systems as for example SYSTEM 2000 of MRI. Due to their nature, such languages are often called "selection languages". They are high-level languages and given a proper syntax the semantics may be quite close to the intuitive meaning of the construct interpreted as a sentence. In SYSTEM 2000 the language is even called the "Natural Language Feature".

First, we give some examples of selective queries to introduce the idea behind these languages. On the basis of these we then introduce some basic views on the database and give some basic operations including the important *broom* concept. These are used in the following definition of a small language including only single selection. We then show how single selections may be combined in two different ways using boolean operators, and finally we discuss some disadvantages of the so-called tree and how they can be remedied by a special selection construct.

Note, that since selection languages are characterized by the way they select records for operation and not by the operations themselves, only retrieval expressions are covered here.

A Selection Example

In order to let you have an idea of how selective languages work, we shall show a few selection queries based on the following hierarchical diagram:

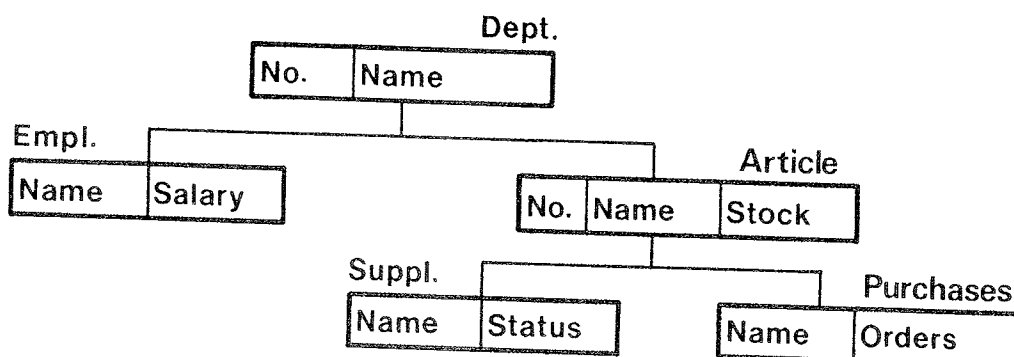


Fig. 7

The diagram describes the database structure of a larger trading firm divided into departments which each deals with certain articles.

In general, a selection language request or command has the form:

<action part> where <selection part>

The intended semantics is that the action in the action part is applied to the set of nodes designated by the selection part.

We start with one of the most simple kinds of query, for example "print the names of articles for which there are more than 1000 items in stock". This may be achieved by the command:

print Article.Name where Article.Stock >= 1000

All the employees in a certain department, for example the Food Department, may be found by:

print Empl.Name where Dept.Name = "Food"

Note that we select on the basis of a Department field, but actually use an Employee field. Thus, the selection of Department records automatically gives access to all descendants of the selected records. This is often called "downward normalization". In the same way all suppliers that supply a certain department can be found. Another query could be:

print Dept.Name where Article.No = 123

which will be interpreted as "all departments which deal with article no. 123". Thus, in this case we see that the selection of an article on the basis of its number automatically includes its parent (in general: ancestors). This may be called "upward normalization". Changing the command to:

print Suppl.Name where Article.No = 123

will give us all suppliers which supply the specific article to one or more departments. If we are interested in only those which supply a specific department, we can use:

```
print Suppl.Name where Article.No = 123 and Dept.Name = "Food"
```

As we see, selections may in general be combined by boolean operators. Another example is to "find the departments which trade with the firm 'Smith'".

```
print Dept.Name where Suppl.Name = "Smith"  
      and Purchases.Name = "Smith"
```

The semantics of boolean combinators, which seems rather natural, is discussed later. Finally, suppose we are interested in those articles which may be supplied by "Jones & Co." and may be sold to "Printall". We may try:

```
print Article.Name where Suppl.Name = "Jones & Co."  
      and Purchaser.Name = "Printall"
```

Unfortunately, this command will return the desired article names in some systems, whereas other systems will tell us that there are no such articles. This problem is further treated in the section named "The Has Clause".

Regarding the Database as a Tree

The readers who are familiar with the IMS System may have wondered why we have chosen such an abstract view on the hierarchical tree in the model presented (31.-40.), knowing that, in IMS it suffices to make a pointer structure imposing both a hierarchical structure on the records, and a sequential order in which they are to be retrieved. The reason is that such a model (which is really one kind of IMS implementation, see Chapter 13) is not a good starting point for explaining and modelling selection languages. The main view of the database in these languages is that of a tree of nodes where each node has an associated record. Now, as stated above, a selection command first qualifies a set of nodes (and thereby records) which are then considered (but not necessarily used) by some action. Therefore, a good data model should easily adapt to the tree view.

-- The Database as a Tree of Records

Recalling the model of the database part (33.):

$$89. \quad DB = RTId \overline{\overline{\overline{}}} (RecLab \overline{\overline{\overline{}}} (Rec \times DB))$$

we see that it does not immediately regard the database as a tree of records. However, changing the model to:

$$90. \quad TDB = (RTId \times RecLab) \overline{\overline{\overline{}}} (Rec \times TDB)$$

we get a model where each record is directly connected to its children by branches labelled by (id, lab) pairs. A given DB may easily be transformed into a "pure" tree TDB by:

$$\begin{aligned} 91.1 \quad & tree-view(db) \triangle \\ .2 \quad & [(id, lab) \mapsto (rec, tree-view(db')) \mid id \in \underline{dom} \ db \wedge lab \in \underline{dom} \ db(id) \wedge \\ .3 \quad & (rec, db') = db(id)(lab) \quad] \\ .4 \quad & \underline{type}: \quad DB \rightarrow TDB \end{aligned}$$

Being aware of this relationship we retain the original model, but speak in terms of the tree view!

-- Interpreting a Path

The path concept was introduced to enable unique identification of the records in the database thereby making it possible to speak of a record from the "outside" of the database in spite of its top-down structure. Speaking in tree terms, a Path (28.) is the sequence of branch-labels on the way from the root record ("system record") to a certain record occurrence. Therefore, a path may have two immediate interpretations:

- A. The path designates one record: the record at the end of the path. This is the view which motivated the path concept.
- B. The path designates a set of records: all the records along the path, including the "system" record and the record at the end of the path. It may also be considered as a subtree; one without branches.

As both interpretations will be used in this section, we rename the Path domain to emphasize this differentiation. Thus, when speaking of single records or nodes we use the domain *Node*. The second interpretation will be used only for paths ending in leaf nodes of the database tree. In this case, the path will be called a stem:

92. $Node = Stem = Path = (RTid \times RecLab)^*$

-- Node and Stem Operations

Stems are important since they form the basis for regular trees defined below. We therefore introduce a few node and stem operations here. The first one is for pragmatic reasons only.

93.1 $all-nodes(db) \triangleq all-paths(db)$

94.1 $node-type(n) \triangleq cases\ n: (<> \rightarrow \underline{SYSTEM}, n'^(id,) \rightarrow id)$
 .2 $type: Node \rightarrow (\underline{SYSTEM} \mid \underline{TYPE})$

A stem must end in a leaf node of the database (see figure below). In terms of our model this requirement can be expressed by:

95.1 $inv-Stem(st)db \triangleq st \in all-paths(db) \wedge sub-database(st,db)=[]$
 .2 $type: Stem \rightarrow (DB \rightarrow Bool)$

All the stems of a database can be determined in the same way:

.1 $all-stems(db) \triangleq \{st \mid st \in all-paths(db) \wedge sub-database(st,db)=[]\}$
 .2 $type: DB \rightarrow Stem-set$

We shall also be interested in the nodes represented in a stem:

95.1 $nodes-of-stem(st) \triangleq \{p \mid p \in Node \wedge is-prefix(p,st)\}$
 .2 $type: Stem \rightarrow Node-set$

(Note that the type: $Path \rightarrow Path-set$ might have been confusing.)

-- Brooms

A broom is a general tree concept which turns out to be very important in many selection languages. Given a node in a tree, the broom of this node is generally defined by:

$$broom(n) \triangleq \{n\} \cup \{n' \mid n' \text{ is a descendant of } n\} \\ \cup \{n' \mid n' \text{ is an ancestor of } n\}$$

See illustration below. In our model we may use the dependency concept:

- 96.1 $\text{broom}(n) \text{ db} \triangleq \{ n' \mid n' \in \text{all-nodes}(\text{db}) \wedge \neg \text{indep}(n, n') \}$
- .2 $\text{type: Node} \rightarrow (\text{DB} \rightarrow \text{Node-set})$
- .3 $\text{pre: } n \in \text{all-paths}(\text{db})$

-- Regular Trees

The notion of regular trees given here is a slightly modified version of the one given in [Hardgrave 72a] upon which much of the following material based. A regular tree with respect to a database tree is a sub-tree whose leaf nodes are also leaf nodes in the database. See illustration below. As it can be seen, the nodes of a regular tree is the union of the stems leading to the leaves. A regular tree may therefore be represented by the stems of its leaves:

- 97. $\text{RegTree} = \text{Stem-set}$

We also see that a broom of some node is a regular tree. In this representation the broom is given by:

- 98.1 $\text{broom}_T(n) \text{ db} \triangleq \{ n^{\wedge} \text{stem} \mid \text{stem} \in \text{all-stems}(\text{sub-database}(n, \text{db})) \}$
- .2 $\text{type: Node} \rightarrow (\text{DB} \rightarrow \text{RegTree})$
- 99.1 $\text{nodes-of-tree}(\text{rt}) \triangleq \text{union} \{ \text{nodes-of-stem}(\text{st}) \mid \text{st} \in \text{rt} \}$
- .2 $\text{type: RegTree} \rightarrow \text{Node-set}$

We now define the regular tree operations intersection, union, and negation as the corresponding set operations on the *Stem-sets* representing the trees. In this way the operations always results in regular trees.

-- Illustration of Tree Concepts

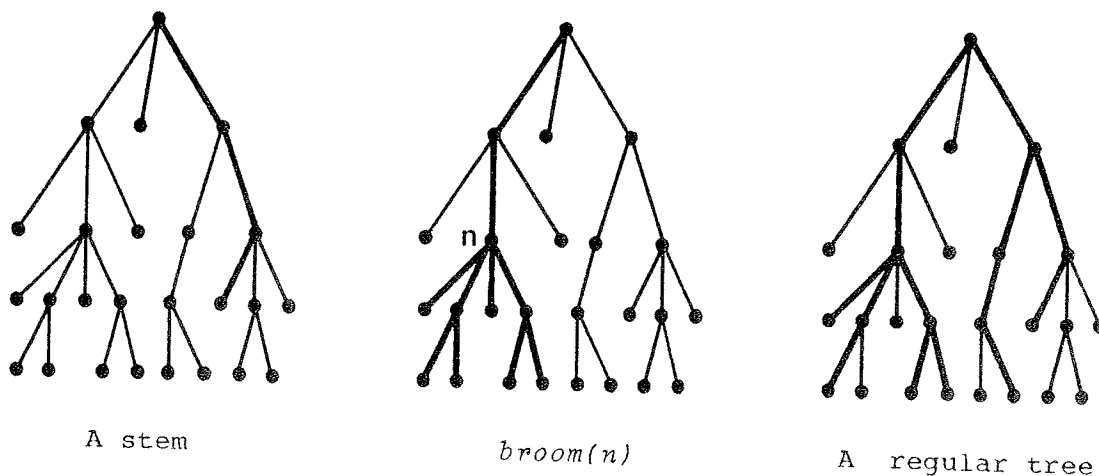


Fig. 8

A SMALL SELECTION LANGUAGE

We are now ready to define and model a small language involving only selection on the basis of one field.

Abstract Syntax

- 100. *Cmd* $::$ *Action* *Where*
- 101. *Where* $::$ *SelExp*
- 102. *SelExp* $=$ *FieldSel* | ...
- 103. *FieldSel* $::$ *FieldDesig* *Qual*
- 104. *Action* $=$ *Print* | ...
- 105. *Print* $::$ *FieldDesig-set*
- 106. *FieldDesig* $::$ *RTId* *FieldId*
- 107. *Qual* $=$ *Eq* | ...
- 108. *Eq* $::$ *Val*

(The purpose of the extra *Where*-level will become apparent later.)

Well-formedness

The only constraints which we shall impose on the constructs are that *Field*-designators must exist in the catalogue, and that the qualification is of the right type.

- 109.1 $pre-Cmd[mk-Cmd(act, mk-Where(se))]ctlg \underline{A}$
- .2 $pre-Action[act]ctlg \wedge pre-SelExp[se]ctlg$
- 110.1 $pre-Action[act]ctlg \underline{A}$
- .2 cases *act*:
- .3 $(mk-Print(fds) \rightarrow (\forall fd \in fds) pre-FieldDesig[fd]ctlg, \dots)$
- 111.1 $pre-SelExp[se]ctlg \underline{A}$
- .2 cases *se*:
- .3 $(mk-FieldSel(,) \rightarrow pre-FieldSel[se]ctlg, \dots)$
- 112.1 $pre-FieldSel[mk-FieldSel(fd, qual)]ctlg \underline{A}$
- .2 $pre-FieldDesig[fd]ctlg \wedge$
- .3 $(let\ hp = find-hp(s-RTId(fd), ctlg)$
- .4 $let\ fieldtp = sub-catalogue(hp, ctlg)(s-FieldId(fd))$ *in*
- .5 $pre-Qual[qual]fieldtp)$ *in*

- 113.1 $\text{pre-Qual}[\text{qual}]\text{fieldtp} \underline{\Delta}$
 .2 $\text{cases qual: (mk-Eq(val) } \rightarrow \text{type-of(val)=fieldtp, ...)}$
- 114.1 $\text{pre-FieldDesig}[\text{mk-FieldDesig(id,fid)}]\text{ctlg} \underline{\Delta}$
 .2 $\text{id} \in \text{collocat-names(ctlg)}$
 .3 $\wedge \text{fid} \in \text{sub-catalogue(find-hp(id,ctlg),ctlg)}$
115. $\text{type: pre-X: } X \rightarrow (\text{CTLG} \rightarrow \text{Bool})$ where X is *Action*, *SelExp*,
FieldSel, resp. *FieldDesig*
116. $\text{type: pre-Qual: Qual} \rightarrow (\text{TYPE} \rightarrow \text{Bool})$

Action interpretation

The semantics of a command is to select a set of records by the Where-clause, and then impose the specified action on these.

- 117.1 $\text{int-Cmd}[\text{mk-Cmd(act,wh)}]\text{hdbs} \underline{\Delta}$
 .2 $(\text{let nodes} = \text{eval-Where[wh]hdbs in}$
 .3 $\text{int-Action[act](nodes)hdbs})$
 .4 $\text{type: Cmd} \rightarrow (\text{HDBS} \rightarrow (\text{HDBS} \mid \text{Table} \mid \dots))$

The type of the result depends on the specific action. For example, the result of a Print command may be a Table giving the different values occurring in the specified fields of the selected records.

118. $\text{Table} = \text{FieldDesig} \multimap \text{Val-set}$
- 119.1 $\text{int-Action}[\text{mk-Print(fds)}](\text{nodes}) \text{mk-HDBS(ctlg,db)} \underline{\Delta}$
 .2 $[\text{fd} \mapsto \text{select-field-vals(nodes,fd,db)} \mid \text{fd} \in \text{fds}]$
 .3 $\text{type: Action} \rightarrow (\text{Node-set} \rightarrow (\text{HDBS} \rightarrow \text{Table}))$
- 120.1 $\text{select-field-vals(nodes,mk-FieldDesig(id,fid),db)} \underline{\Delta}$
 .2 $\{ \text{lookup-rec}(n)(\text{fid}) \mid n \in \text{nodes} \wedge$
 .3 $n \neq \langle \rangle \wedge n[\text{len } n] = (\text{id},) \}$
 .4 $\text{type: Node-set} \times \text{FieldDesig} \times \text{DB} \rightarrow \text{Val-set}$

Where evaluation

The Where-clause is evaluated in two steps. First, a number of nodes are directly selected according to the Field Selector. To achieve the effect of upward and downward "normalization" mentioned in the introduction

these nodes are then qualified to give an extended set of nodes.

121.1 $eval-Where[mk-Where(se)]\ mk-HDBS(db) \triangleq eval-SelExp[se]db$
 .2 type: $Where \rightarrow (HDBS \rightarrow Node-set)$

122.1 $eval-SelExp[se]db \triangleq$
 .2 cases se :
 .3 $(mk-FieldSel(fd, qual) \rightarrow (let\ nodes = select(fd, qual)db\ in$
 .4 $qualify(nodes)db),$
 .5 $\dots)$
 .6 type: $SelExp \rightarrow (DB \rightarrow Node-set)$

123.1 $select(mk-FieldDesig(id, fid), qual)\ db \triangleq$
 .2 $\{ n \mid n \in all-nodes(db) \wedge node-type(n) = id \wedge$
 .3 $match((lookup-rec(n, db)(fid), qual) \}$
 .4 type: $FieldDesig \times Qual \rightarrow (DB \rightarrow Node-set)$

124.1 $match(val, qual) \triangleq \text{cases } qual: (mk-Eq(val') \rightarrow val=val', \dots)$
 .2 type: $VAL \times Qual \rightarrow Bool$

-- Qualification

The purpose of qualification is to extend the set of selected nodes by including nodes which are in a certain relationship to those directly selected. These nodes are called indirectly selected. In almost all systems, the relation is given by:

A node is indirectly selected if at least one of its ancestors or descendants is directly selected.

This notion of qualification corresponds well to an intuitive interpretation of upward and downward normalization. Furthermore, it is easily seen that the nodes qualified by a directly selected node are those of the broom of the node. Thus, the total set of qualified nodes may be found by

125.1 $qualify(nodes)db \triangleq \text{union } \{ broom(n)db \mid n \in nodes \}$
 .2 type: $Node-set \rightarrow (DB \rightarrow Node-set)$

A BOOLEAN SELECTION LANGUAGE

We now extend our language by constructs which will allow us to select on the basis of more than one field as indicated in the introduction. The new select constructs are:

- 126. *SelExp* = *FieldSel* | *And* | *Or* | *Not* | ...
- 127. *And* :: *SelExp SelExp*
- 128. *Or* :: *SelExp SelExp*
- 129. *Not* :: *SelExp*

These constructs are well-defined if their components are.

In order to achieve the "natural" interpretation effect on these constructs, all systems combine the nodes of their components. Also, the systems agree that directly selected nodes should still be qualified before they are combined. However, there are two essentially different ways to do these combinations: the set-theoretic and the tree-theoretic which we shall deal with in turn.

Set-Theoretic Combination

According to this principle, the nodes designated by the component expressions are simply combined using the usual set operations. Therefore we simply get the following new cases of the *eval-SelExp* function:

- 130.1 *eval-SelExp*[mk-And(*se*₁, *se*₂)]db Δ
- .2 (let *nodes*₁ = *eval-SelExp*[*se*₁]*db*,
- .3 *nodes*₂ = *eval-SelExp*[*se*₂]*db* in
- .4 *nodes*₁ ∩ *nodes*₂)

The *Or* case is of course similar.

- 131.1 *eval-SelExp*[mk-Not(*se*)]db Δ
- .2 (let *nodes* = *eval-SelExp*[*se*]*db* in
- .3 *all-nodes*(*db*) \ *nodes*)

Tree-Theoretic Combination

It is easily seen that the set combinations may result in node-sets which are not sub-trees of the database, and upward and downward normalization

may therefore become difficult to define properly. To avoid this, some systems (including SYSTEM 2000) apply another combination principle based on regular trees and the associated tree operations.

In this approach, the denotation of a Select Expression is no longer a set of nodes, but instead (more restrictively) a regular tree. The semantics of the select operators may then be defined by (or as) the corresponding tree operations. In order to incorporate this in our model, a few functions and types need to be redefined:

- 132.1 $qualify_T(ns)db \triangleq \underline{\text{union}} \{ broom_T(n)db \mid n \in ns \}$
 .2 type: $Node\text{-}set \rightarrow (DB \rightarrow RegTree)$

The type of *eval-SelExp* changes to:

133. type: $eval\text{-}SelExp_T: SelExp \rightarrow (DB \rightarrow RegTree)$

However, the function definition remains the same, except that "nodes" for pragmatic reasons should be renamed "stems" everywhere except in 122. Finally we are now ready to use the extra Where-level in the syntax since the result of evaluating a Where-clause should still be a Node-set.

- 134.1 $eval\text{-}Where_T[mk\text{-}Where(se)] mk\text{-}HDBS(,db) \triangleq$
 .2 $nodes\text{-}of\text{-}tree(eval\text{-}SelExp[se]db)$

Differences Between Set and Tree Combination

Here we shall look at the semantics and pragmatics of the two kinds of boolean combination. As already stated, the set operations work on node sets whereas the tree operations work on regular trees. We shall say that a node set is similar to a regular tree if it equals the nodes of the tree. We see immediately that the result of a Field Selection is similar in the two systems. It is also seen that the or operator yields similar results provided the operands are similar. Thus, the origin of any differences must be the and and the not operators which are discussed below.

-- The And Operator

Analyzing the and operation we find that as long as the fields combined are different and on the same hierarchical path, the results will be

similar. This was the case in the introductory example (now a little less concrete):

```
print Suppl.Name where Dept.Name = D and Article.No = n
```

where the semantics in both cases corresponds to "those suppliers which supply article *n* to department *D*". However, in the last example of the introduction:

```
print Article.No where Suppl.Name = S and Purchaser.Name = P
```

the field selectors are no longer on the same hierarchical path. In the tree combination case, this implies that they cannot have any stems in common and therefore the result is an empty regular tree. Using set combination, we see that the result is those records on the common path which have descendants satisfying the qualifications. Thus, the command above will result in exactly the articles we want. However, consider:

```
print Dept.Name where Suppl.Name = S and Purchaser.Name = P
```

This command will give the "departments which are supplied by *S* and sell to *P*", but there need not be a single article in the department for which this is the case. Thus, the latter command may give results even though the first does not, that is the effect of upward normalization has been lost, and it is not possible to get those departments where the condition is satisfied by at least one single article. We may also try to get "those suppliers that supply articles also supplied by *S* and sold to *P*" using:

```
print Suppl.Name where Suppl.Name = S and Purchaser.Name = P
```

However, this command will not give any results even using set combination since we have lost the descendants of the selected articles. Finally we try to conjoin two selections on the same field:

```
print Dept.Name where Suppl.Name = S1 and Suppl.Name = S2
```

(*S*₁+*S*₂). Using tree combination we get no result, justified by the fact that no supplier can have two names. In the set approach we get "those departments which are supplied by both *S*₁ and *S*₂". Again we cannot get the departments which deal with an article supplied by both suppliers.

-- The Not Operator

Since the not operator in both approaches may be distributed according to De Morgans Laws, we need only consider negation of single Field Selectors. A command like:

```
print Dept.Name where Article.No = n
```

is generally interpreted as "those departments under which there exists an article with number n ". Now consider:

```
print Dept.Name where not Article.No = n
```

Using set combination this will result in "those departments which do not deal with article n ", whereas the tree principle will give "those departments under which there is anything different from article number n ". Note the difference, and that none of them makes "not Article.No = n " equivalent to "Article.No $\neq n$ ". Another problem is that:

```
print Empl.Name where not Article.No = n
```

will in both cases give all employees although they have nothing to do with article numbers. This problem may be solved by using type constrained negation, where the negation returns only nodes with types such as those negated. Here we consider only the tree combination case:

```
135.1 eval-SelExp[mk-Not(se)]hdb in
      .2 (let stems = eval-SelExp[se]db in
      .3 let types = { node-type(st) | st  $\in$  stems } in
      .4 { st' | st'  $\in$  all-stems(db) \ stems  $\wedge$  node-type(st')  $\in$  types } )
```

Now "not Article.No = n " becomes equivalent to "Article.No $\neq n$ ".

The Has-Clause

To sum up, both combination principles have some advantages:

Set: Useful, although restricted and operation. Negation corresponds to universal quantification.

Tree: Always regular trees as result, that is no normalization lost.

Not operation negates select condition.

In practice, many systems use the tree combination principle (among others SYSTEM 2000). The main disadvantage is the restricted and operation. To compensate for this, these systems also offer a so-called Has-clause which enables re-selection/qualification at any level:

```

136. SelExp      = ... | Has
137. Has         :: RTId SelExp

138.1 eval-SelExp[mk-Has(id,se)]db Δ
    .2  (let nodes = nodes-of-tree(eval-SelExp[se]db) in
    .3  let nodes' = { n | n ∈ nodes ∧ node-type(n)=id } in
    .4  qualify(nodes')db )

```

The Has-clause gives us many new possibilities. For example we can get "the employees in the departments where at least one single article is supplied by *S* and sold to *P*", and we may achieve the universal quantification effect:

```

print Empl.Name where Dept has Article has (Suppl.Name = S and
                                         Purchaser.Name = P )

```

```

print Dept.Name where not Dept has Article.No = n

```

12.2.4 Concluding Remarks on the Hierarchical Data Model

We have in 12.2.1 build up a top-down model of a hierarchical database. This model was used directly for the hierarchy-oriented language defined in 12.2.2. Although a little more abstract than needed, the model could also be used for the more procedural traversal languages like IMS. In 12.2.3 we formalized a number of concepts (brooms, etc.) on top of our top-down model in order to enable a more global tree-view of the database. Using these, we could easily give the semantics of the most important selection language constructs. All this seems to indicate that our model is a reasonable starting point for hierarchical database modeling -- as also evidenced by [Bjørner 82c]. However, for use in connection with selection languages only, a more global tree-like model may turn out to be more suitable.

We shall finally stress that a primary concern of this section has been to illustrate a model development process. This has been tried through many concrete examples and careful model description. The length of this section, compared to those on the relational model and on the network model, is thus not an indication of a special interest in, or support of the hierarchical model, but rather of a general emphasis on modelling pragmatics.

12.3 THE NETWORK DATA MODEL

The network data model to be formalized in this section is based on Bachmann's 'Data Structure Diagrams'. We present abstractions of the data model behind the CODASYL/DBTG proposal and, hence of the data model underlying such DBMSs as IDS/2, IDMS, DMS1100, etc.

By a network we understand a directed graph. By a network data model we understand an interpretation and utilization of the nodes and edges as follows: nodes denotes aggregates of records; edges denotes relations between the records denoted by the connected nodes; and the paths of the graph enable operations to extract collections of records of the "end" node of a path based on properties satisfied by records of all nodes of the path. Exactly what is meant by aggregates, relations and extraction is then the purpose of our formalization.

12.3.1 The Data Aggregate

Our presentation of the relational data model data aggregate was 'matter-of-factly': we merely stated the model. The hierarchical data aggregate model was arrived at as the result of an analysis of given example pictures. In presenting the network data aggregate model we shall proceed in yet a third way. Whereas our analysis of hierarchical database "snapshots" lead to a top-down presentation, we shall now start with the primitives and end up with their synthesis into networks. As was the case in our presentation of the hierarchical model we shall also use pictures, but now in an a-posteriori supporting rôle.

Our objective is to describe the syntax and semantics of so-called data structure diagrams. These consists, pictorially of boxes (nodes) and arrows (directed edges). We next explain the meaning of boxes, the meaning of arrows, and finally the syntactic rules for well-formed data

structure diagrams.

A box denotes, \sim , a set of Records. We take records as our first primitive:

1.  \sim $R\text{-set}$

Fig. 9

Arrows will (come to) play the rôle of operators infix between two, not necessarily distinct boxes:


2. 

Fig. 10

The meaning of arrows, in general, is that of a map from records to sets of records:

3.  \sim $(R \xrightarrow{\sim} R\text{-set})$

Fig. 11

The meaning of an arrow, in particular from a box denoting the record set rs_f to a box denoting the record set rs_t , is a map, m , whose domain is, in general, a subset of rs_f , and whose range is a set of sets of records whose union is, in general, a subset of rs_t :

4. $m \in (R \xrightarrow{\sim} R\text{-set});$
5. $\underline{\text{dom}}\ m \subseteq rs_f, \underline{\text{union}}\ \underline{\text{rng}}\ m \subseteq rs_t$

A data structure diagram is a collection of uniquely named boxes and uniquely named arrows, the latter infix between existing boxes. Box and arrow names are then our remaining primitives. The records of the Box from which an arrow emanates and which are in the domain of the map

denoted by the arrow are called owner records, while the records of the file to which an arrow is incident and which are in some set(s) of records of the range of the arrow denoted map are called member records. The arrow denotation is, in CODASYL/DBTG rather confusingly, called a 'settype'; we shall call it a relation.

The Domain of data structure diagrams can syntactically then be formalized as:

$$6. \quad DSD_{syn} :: Fid\text{-}set \times (Sid \rightarrow (Fid \times Fid))$$

The set of file names represent the boxes, and the map from arrow names to the pairs of from box and to box names represent the arrows and where they are infixes. The diagram:

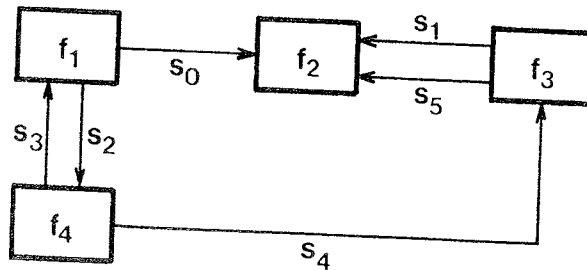


Fig. 12

then corresponds to the abstract object denoted by the following expression:

$$\begin{aligned}
 7.0 \quad & mk\text{-}DSD_{syn}(\{f_1, f_2, f_3, f_4\}, \\
 .1 \quad & [s_0 \mapsto (f_1, f_2), s_1 \mapsto (f_3, f_2), s_2 \mapsto (f_1, f_4), \\
 .2 \quad & s_3 \mapsto (f_4, f_1), s_4 \mapsto (f_4, f_3), s_5 \mapsto (f_3, f_2)])
 \end{aligned}$$

We observe the constraint that "to each arrow there corresponds two boxes in the data structure diagram":

$$\begin{aligned}
 8.0 \quad & inv\text{-}DSD_{syn}(mk\text{-}DSD_{syn}(fs, sffm)) \triangleq \\
 .1 \quad & (\forall s \in \text{dom } sffm) \\
 .2 \quad & (\text{let } (f_f, f_t) = sffm(s) \text{ in } \{f_f, f_t\} \subseteq fs)
 \end{aligned}$$

Semantically the data structure diagrams can be formalized, using (1.) and (3.):

$$9. \quad DSD_{sem} :: (Fid \multimap R\text{-set}) \quad (Sid \multimap (R \multimap R\text{-set}))$$

which we annotate: to file names, in *Fid*, correspond sets of records; and to arrow names, in *Sid*, maps from records to (sub)sets of records. Combining the syntactic and semantic abstractions we get:

$$10. \quad DSD \quad :: (Fid \multimap R\text{-set}) \quad (Sid \multimap ((Fid \times Fid) \times (R \multimap R\text{-set})))$$

which we annotate: to file names correspond, as in the semantic "view", sets of records; and to arrow names correspond two things: (syntactically) the pair of from/to identification, and (semantically) the map from 'from' records to sets of 'to' records.

Combining the two constraints: (3.) and (5.) we get:

$$\begin{aligned} 11.0 \quad & inv\text{-}DSD(mk\text{-}DSD(fm, sm)) \underline{\Delta} \\ .1 \quad & (\forall s \in \underline{dom} \ sm) \\ .2 \quad & (\underline{let} \ ((f, t), m) = sm(s) \ \underline{in} \\ .3 \quad & (\{f, t\} \subseteq \underline{dom} \ fm) \\ .4 \quad & \wedge \ ((\underline{dom} \ m \subseteq fm(f) \wedge (\underline{union} \ rng \ m \subseteq fm(t)))) \end{aligned}$$

Given the holding of these constraints we can express the following relations between *DSD*, on one side, and *DSD_{syn}* and *DSD_{sem}*, on the other side:

$$\begin{aligned} 12.0 \quad & retr\text{-}DSD_{syn}(mk\text{-}DSD(fm, sm)) \underline{\Delta} \\ .1 \quad & mk\text{-}DSD_{syn}(\underline{dom} \ fm, [s \mapsto (f, t) \mid s \in \underline{dom} \ sm \wedge ((f, t),) = sm(s)]) \end{aligned}$$

$$12. \quad \underline{type}: \quad DSD \rightsquigarrow DSD_{syn}$$

$$\begin{aligned} 13.0 \quad & retr\text{-}DSD_{sem}(mk\text{-}DSD(fm, sm)) \underline{\Delta} \\ .1 \quad & mk\text{-}DSD_{sem}(fm, [s \mapsto m \mid s \in \underline{dom} \ sm \wedge (, m) = sm(s)]) \end{aligned}$$

$$13. \quad \underline{type}: \quad DSD \rightsquigarrow DSD_{sem}$$

$$\begin{aligned} 14.0 \quad & inj\text{-}DSD(mk\text{-}DSD_{syn}(fs, ss), mk\text{-}DSD_{sem}(fm, sm')) \underline{\Delta} \\ .1 \quad & mk\text{-}DSD(fm, [s \mapsto (ss(s), sm') \mid s \in \underline{dom} \ ss]) \\ .2 \quad & \underline{pre}: (fs = \underline{dom} \ fm) \wedge (\underline{dom} \ ss = \underline{dom} \ sm') \\ .3 \quad & \wedge \ inv\text{-}DSD_{syn}(mk\text{-}DSD_{syn}(fs, ss)) \\ .4 \quad & \wedge \ (\forall s \in \underline{dom} \ ss) (\underline{let} \ (f, t) = ss(s) \ \underline{in} \\ .5 \quad & (\underline{dom} \ sm'(s) \subseteq fm(f)) \\ .6 \quad & \wedge (\underline{union} \ rng \ sm'(s) \subseteq fm(t))) \end{aligned}$$

$$14. \quad \underline{type}: \quad DSD_{syn} \times DSD_{sem} \rightarrow DSD$$

where:

$$15.0 \quad (\forall dsd \in DSD) (inv-DSD(dsd) \supset \\ .1 \quad (inj-DSD(retr-DSD_{syn}(dsd), retr-DSD_{sem}(dsd)) = dsd))$$

Restrictions and Extensions to the Data Aggregate Model

First we deal with a restriction. Historically the following constraint had been imposed: under any arrow denoted map, two distinct records of its domain map into disjoint sets of records. We express this (further) constraint by joining an additional line (11.5) to formula (11.):

$$11.5 \quad (\forall r_1, r_2 \in \underline{dom} \, m) ((r_1 \neq r_2) \supset (m(r_1) \cap m(r_2) = \{\}))$$

One (logical) reason for this restriction is that it allows a network database user to model (tree-like) hierarchies. We shall attempt in chapter 13, section 2 to give another (physical) reason for this restriction.

Then we turn to generalizations on the theme of variations on the syntax of arrows and correspondingly denoted meanings. Up to now we have dealt with arrows (i) emanating from exactly one box and being incident upon exactly one box. We now formalize the syntax and semantics of arrows which (ii and iii) emanate from one but are incident upon more than one box, or (iv, v and vi) emanate from several and are incident upon exactly one, respectively more than one box:

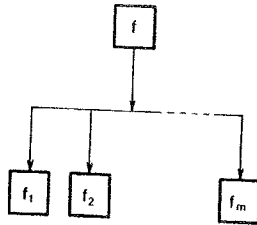


Fig. 13

$$mk-FT_{ii}(f, \{f_1, f_2, \dots, f_m\})$$

$$mk-FT_{iii}(f, \{f_1, f_2, \dots, f_m\})$$

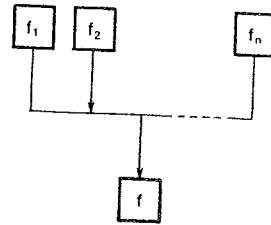


Fig. 14

$$mk-FT_{iv}(\{f_1, f_2, \dots, f_n\}, f)$$

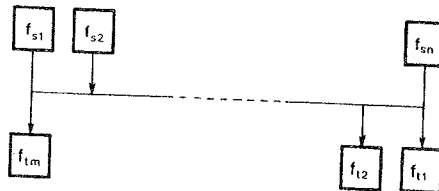


Fig. 15

$$mk-FT_{v/vi}(\{f_{s1}, f_{s2}, \dots, f_{sn}\}, \{f_{tm}, \dots, f_{t2}, f_{t1}\})$$

In DSD_{syn} [DSD] we mapped Sid into $(Fid\ Fid) [(Fid\ Fid)\ (R \multimap R-set)]$
we now map Sid into $FT\ (SET)$. The Domain FT has the six forms:

$$16. \quad FT = FT_{11} \mid FT_{1M1} \mid FT_{1MK} \mid FT_{N1} \mid FT_{NM1} \mid FT_{NMK}$$

where:

$$17. \quad FT_i \quad \equiv \quad FT_{11} \quad \quad \quad :: \quad Fid \quad Fid$$

$$18. \quad FT_{ii} \quad \equiv \quad FT_{1M1} \quad \quad \quad :: \quad Fid \quad Fid-set$$

$$19. \quad FT_{iii} \quad \equiv \quad FT_{1MK} \quad \quad \quad :: \quad Fid \quad Fid-set$$

$$20. \quad FT_{iv} \quad \equiv \quad FT_{N1} \quad \quad \quad :: \quad Fid-set \quad Fid$$

$$21. \quad FT_v \quad \equiv \quad FT_{NM1} \quad \quad \quad :: \quad Fid-set \quad Fid-set$$

$$22. \quad FT_{vi} \quad \equiv \quad FT_{NMK} \quad \quad \quad :: \quad Fid-set \quad Fid-set$$

The associated meaning of arrows are given next. Basically their meaning is a map from records of the source box(es) to sets of records of the target box(es). [Having already disposed of (i) we "start" with (ii and iii).] Either of two meanings can be attached to the arrow which forks out to many boxes. One associates with a record of F , the set of records denoted by f , a non-empty set of records of some F_i ; the other associates with a record of F , for each target box (file), F_i , a set of its records:

$$23.0 \quad SET = SET_{11} \mid SET_{1M1} \mid SET_{1MK} \mid SET_{N1} \mid SET_{NM1} \mid SET_{NMK}$$

$$24. \quad SET_{11} \quad :: \quad (Fid \times Fid) \quad (R \multimap R-set)$$

$$25. \quad SET_{1M1} \quad :: \quad Fid \quad (R \multimap (Fid \times R-set))$$

$$26. \quad SET_{1MK} \quad :: \quad Fid \quad (R \multimap (Fid \multimap R-set))$$

Turning next to arrows which fork inwards (iv). To each record of each source box there corresponds a set of records of the same target box:

$$27. \quad SET_{N1} \quad :: \quad ((Fid \times R) \multimap R-set) \times Fid$$

Finally consider the multiple-source/multiple-target arrows. Again two meanings are possible -- corresponding, for each record of some source box, to the situations (ii and iii).

$$28. \quad SET_{NM1} \quad :: \quad (Fid \times R) \multimap (Fid \times R-set)$$

$$29. \quad SET_{NMK} \quad :: \quad (Fid \times R) \multimap (Fid \multimap R-set)$$

For each of these 'new' arrows (ii-vi) we must express suitable constraints:

- 30.0 $inv-DSD(mk-DSD(fm, sm)) \underline{\Delta}$
 .1 $(\forall s \in \underline{dom} \ sm)(inv-SET(sm(s))(fm))$
- 31.0 $inv-SET_{1M1}(mk-SET_{1M1}(f, m))(fm) \underline{\Delta}$
 .1 $((f \in \underline{dom} \ fm)$
 .2 $\wedge (\forall r \in \underline{dom} \ m)((r \in fm(f))$
 .3 $\wedge (\underline{let} \ (t, rs) = m(r) \ \underline{in} \ (t \in \underline{dom} \ fm)$
 .4 $\wedge (rs \subseteq fm(t))))$
- 32.0 $inv-SET_{1MK}(mk-SET_{1MK}(f, m))(fm) \underline{\Delta}$
 .1 $((f \in \underline{dom} \ fm)$
 .2 $\wedge (\forall r \in \underline{dom} \ m)((r \in fm(f))$
 .3 $\wedge (\underline{let} \ m' = m(r) \ \underline{in}$
 .4 $(\forall t \in \underline{dom} \ m')((t \in \underline{dom} \ fm)$
 .5 $\wedge (m'(t) \subseteq fm(t))))))$
- 33.0 $inv-SET_{N1}(mk-SET_{N1}(m, t))(fm) \underline{\Delta}$
 .1 $((t \in \underline{dom} \ fm)$
 .2 $\wedge (\forall (f, r) \in \underline{dom} \ m)((f \in \underline{dom} \ fm)$
 .3 $\wedge (r \in fm(f))$
 .4 $\wedge (m(f, r) \subseteq fm(t))))$
- 34.0 $inv-SET_{NM1}(mk-SET_{NM1}(m))(fm) \underline{\Delta}$
 .1 $(\forall (f, r) \in \underline{dom} \ m)((f \in \underline{dom} \ fm)$
 .2 $\wedge (r \in fm(r))$
 .3 $\wedge (\underline{let} \ (t, rs) = m(f, r) \ \underline{in}$
 .4 $((t \in \underline{dom} \ fm)$
 .5 $\wedge (rs \subseteq fm(t))))$
- 35.0 $inv-SET_{NMK}(mk-SET_{NMK}(m))(fm) \underline{\Delta}$
 .1 $(\forall (f, r) \in \underline{dom} \ m)((f \in \underline{dom} \ fm)$
 .2 $\wedge (r \in fm(f))$
 .3 $\wedge (\underline{let} \ m' = m(f, r) \ \underline{in}$
 .4 $(\forall t \in \underline{dom} \ m')((t \in \underline{dom} \ fm)$
 .5 $\wedge (m'(t) \subseteq fm(t))))$

[Reviewing formulae (24.-29.) we observe how one could "almost" derive formulae by simple syntactic manipulations, or by manipulations, of an algebraic nature, which ascribe particular Domain operations (\times , $\#$, $|$, $-set$, ..., including grouping $()$).]

12.3.2 The Operations

We shall illustrate operations only on the simplest form of data aggregates, that is involving only simple arrow relations (*SET11*) as defined by (10.).

Three kinds of operations will be investigated: operations on files, relations and entire data structure diagrams. To the first group, belong the operations of writing [, updating, reading] and deleting records; to the second, those of connecting and disconnecting records to, respectively from relations; and to the third group, the retrieve operation of finding desired records. We shall illustrate variations of the non-bracketed ([...]) operations.

File Operations

- 36. $FCmd = Write \mid Delete$
- 37. $Write :: Fid \ R$
- 38. $Delete :: Fid \ R$

Writing a record to a file does not interfere with relations involving that file ("insertion manual"):

- 39. type: $Write: DSD \rightarrow DSD$
- 39.0 $Int-Write[mk-Write(f,r)](mk-DSD(fm,sm)) \triangleq$
 - .1 if $(f \in \text{dom } fm) \wedge (r \notin fm(f))$
 - .2 then $mk-DSD(fm + [f \mapsto fm(f) \cup \{r\}], sm)$
 - .3 else undefined

At least two kinds of semantics can be ascribed to the delete operation. Either we can only delete records which are not in any relation involving the file:

- 40. type: $Delete: DSD \rightarrow DSD$
- 40.0 $Int-Delete[mk-Delete(f,r)](mk-DSD(fm,sm)) \triangleq$
 - .1 if $((f \in \text{dom } fm) \wedge (r \in fm(f)))$
 - .2 $\wedge (\forall ((f,), m) \in \text{rng } sm)(r \notin \text{dom } m)$
 - .3 $\wedge (\forall ((, f), m) \in \text{rng } sm)(r \notin \text{union } \text{rng } m)$
 - .4 then $mk-DSD(fm + [f \mapsto fm(f) \setminus \{r\}], sm)$
 - .5 else undefined

Or deletion propagates to all such relations, that is "triggers" corresponding "disconnect" operations:

```

41.0  Int-Delete[mk-Delete(f,r)](mk-DSD(fm,sm)) Δ
      .1    (let fm' = fm + [f ↦ fm(f) \ {r}],
      .2      sm' = [s ↦ ftr
      .3        | s ∈ dom sm
      .4          ∧ ftr = (let ((fr,to),rel) = sm(f) in
      .5            f ∈ {fr,to}
      .6              → ((fr,to),[r' ↦ rs | r' ∈ dom rel \ {r}
      .7                ∧ rs = rel(r') \ {r}]),
      .8          T → sm(f))] in
      .9    mk-DSD(fm',sm')

```

Lines (41.6-7.) express the disconnection of r from the domain (41.6) and range (41.7) of all those relations (rel) which involve the file f either as source ($f = fr$) or target ($f = to$), that is either as owner or member. All other relations are unaffected (41.8). Lines (41.6-7.) express disconnection rather "generously" in that no question is asked whether r actually is involved in rel !

'Set' Operations

```

42.  Connect  :: Sid   (R × R-set)
43.  DisConn  :: Sid   (R × R-set)

```

The *Connect* operation $mk-Connect(s, (r, rs))$ intuitively inserts the 'relation': $[r ↦ rs]$ as part of the denotation of s :

```

44.0  Int-Connect[mk-Connect(s, (r, rs))](mk-DSD(fm,sm)) Δ
      .1    if s ∉ dom sm
      .2      then undefined
      .3      else (let ((f,t),rel) = sm(s) in
      .4        if r ∉ fm(f) ∧ rs ∉ fm(t)
      .5          then undefined
      .6          else (let rel' = if r ∈ dom rel
      .7                    then rel + [r ↦ rel(r) ∪ rs]
      .8                    else rel ∪ [r ↦ rs] in
      .9          let sm' = sm + [s ↦ ((f,t),rel')] in
      .10     mk-DSD(fm,sm'))
44.  type: Connect → (DSD ⇝ DSD)

```

No check is made for records of rs already in the denotation of s under r . The *Disconnect* operation "undoes" what the connect operation is doing:

```

45.0  Int-DisConn[mk-DisConn(s, (r, rs))](mk-DSD(fm, sm))  $\underline{\Delta}$ 
.1    if  $s \notin \text{dom } sm$ 
.2    then undefined
.3    else (let  $((f, t), rel) = sm(s)$  in
.4    if  $r \notin fm(f) \wedge rs \not\subseteq fm(t)$ 
.5     $\wedge r \notin \text{dom } rel \wedge rs \not\subseteq rel(r)$ 
.6    then undefined
.7    else (let  $rel' =$  if  $rs = rel(r)$ 
.8    then  $rel \setminus \{r\}$ 
.9    else  $rel + [r \mapsto rel(r) \setminus rs]$  in
.10   let  $sm' = sm + [s \mapsto ((f, t), rel')]$  in
.11    $mk-DSD(fm, sm'))$ 
45.   type:  $DisConn \rightarrow (DSD \rightarrow DSD)$ 

```

Data Structure Diagram "Navigations"

-- Paths

A sequence, $\langle s_1, s_2, \dots, s_n \rangle$, of arrow ('set' or relation) names may determine a 'path' through a data structure diagram $mk-DSD(fm, sm)$, as follows: each s_i , if actually the name of some arrow in sm , determines a triple: $sm(s_i) = ((f_i, t_i), rel_i)$, thus the arrow name sequence above determines a sequence of from-to file names: $\langle (f_1, t_1), (f_2, t_2), \dots, (f_{i-1}, t_{i-1}), (f_i, t_i), (f_{i+1}, t_{i+1}), \dots, (f_n, t_n) \rangle$. If for all appropriate i we have: $t_{i-1} = f_i$ then we say that $\langle s_1, s_2, \dots, s_n \rangle$ is forwardwell-formed. If for all appropriate i , either $t_i = t_{i+1}$ (that is $f_i = f_{i+1}$) or $t_{i-1} = f_i$, then we say that it is un-directed well-formed. A forward-well-formed path is an ordinary path in the directed graph determined by any data structure diagram. A well-formed path is a path in the corresponding un-directed graph. Well-formed paths may thus embed arrows in opposing directions.

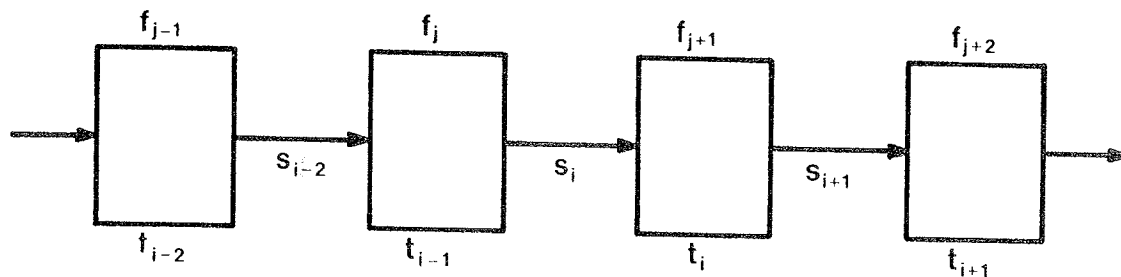


Fig. 16 Forward well-formed Path

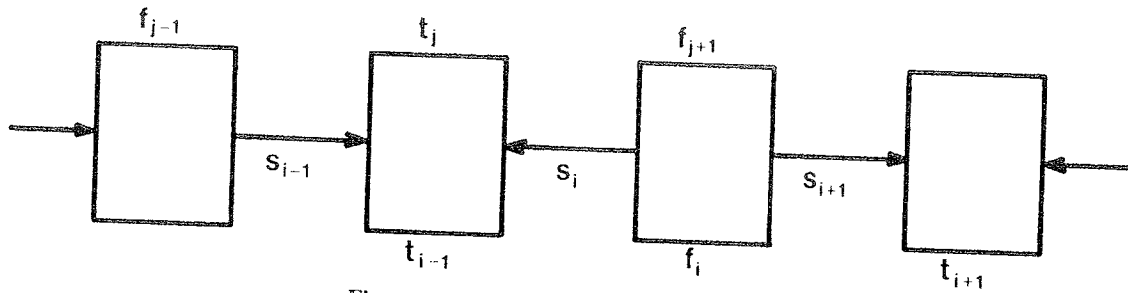


Fig. 17 Undirected well-formed Path

-- Images and Inverse Images

We are given an arrow, s , and a set of records, rs , of either the from f or the to t file of the arrow, and are asked to compute the image, respectively the inverse image, of rs "under the arrow". A figure and a formula for each of the two situations should suffice:

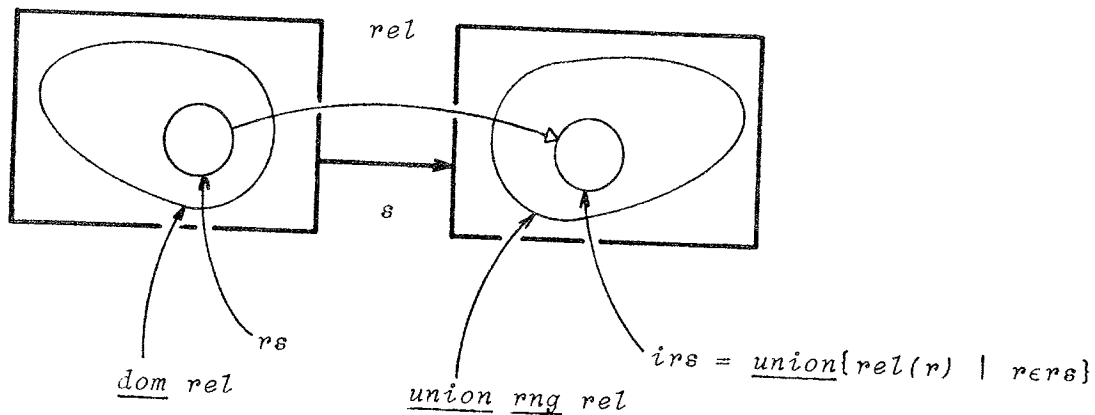


Fig. 18

The inverse image is the set of all those records of the domain of rel which, in rd , map into set of records properly overlapping with rs :

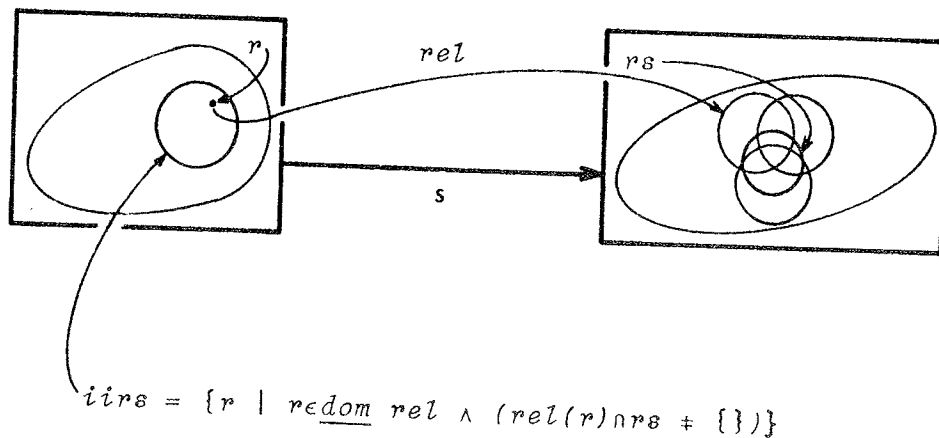


Fig. 19

-- Navigational Retrieval

Given a starting file, f , a set of records, rs , and a well-formed path $sl = \langle s_1, s_2, \dots, s_n \rangle$ we wish to find the set of records rs' which is the combined image/inverse-image of rs under sl :

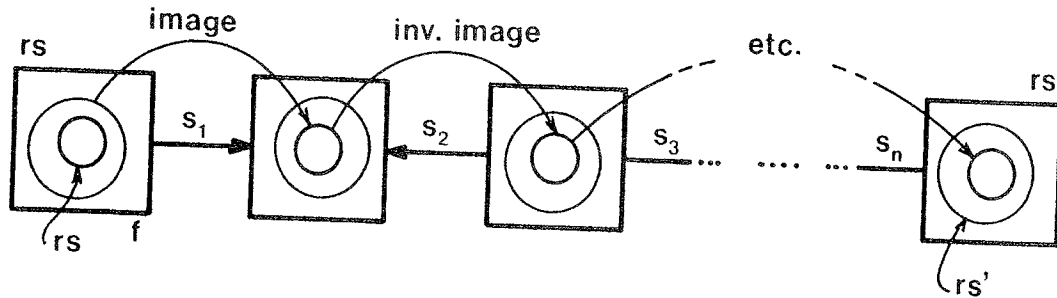


Fig. 20

To effect the retrieval of rs' we specify a command, its syntax and semantics:

46. $Find :: Fid \quad Sid^+ \quad R\text{-set}$
- 47.0 $Val\text{-}Find[mk\text{-}Find(f, sl, rs)](mk\text{-}DSD(fm, sm)) \triangleq$
 .1 $V(f, sl, rs)(sm)$
47. type: $Find \rightarrow *DSD \rightarrow DSD$
- 48.0 $V[f, sl, rs](sm) \triangleq$
 .1 if $sl = \langle \rangle$
 .2 then rs
 .3 else (let $((fr, to), rel) = sm(hd \ sl)$ in
 .4 cases f :
 .5 $(fr \rightarrow V[to, tl \ sl, union\{rel(r) \mid r \in rs\}](sm),$
 .6 $to \rightarrow V[fr, tl \ sl, \{r \mid r \in dom \ rel \wedge rel(r) \cap rs \neq \{\}\}](sm),$
 .7 $T \rightarrow \text{undefined})$
48. type: $Fid \times Sid^* \times R\text{-set} \rightarrow ((Sid \rightarrow SET_{11}) \rightarrow R\text{-set})$

Other forms of navigational retrieval can be defined. One immediate variant on the above is to include, for each arrow specification, a predicate function which "filters" only such records in the image, or inverse image, which satisfy some property:

49. $p \in P = R \rightarrow Bool$
50. $Find :: Fid \quad (Sid \times P)^+ \quad R\text{-set}$

```

51.0  $V[f, spl, rs](sm) \triangleq$ 
.1   if  $spl = \langle \rangle$ 
.2     the  $rs$ 
.3   else (let  $(s, p) = hd\ spl$  in
.4     if  $s \in dom\ sm$ 
.5       then (let  $((fr, to), rel) = sm(s)$  in
.6         cases  $f$ :
.7            $(fr \rightarrow (\text{let } rs'' = \text{union}\{rel(r) \mid r \in rs\} \text{ in}$ 
.8              $\text{let } rs' = \{r \mid rs'' \wedge p(r)\} \text{ in}$ 
.9              $V[to, tl\ spl, rs'](sm)),$ 
.10           $to \rightarrow (\text{let } rs'' = \{r \mid r \in dom\ rel \wedge rel(r) \cap rs \neq \{\}\} \text{ in}$ 
.11             $\text{let } rs' = \{r \mid r \in rs'' \wedge p(r)\} \text{ in}$ 
.12             $V[fr, tl\ spl, rs'](sm)),$ 
.13           $T \rightarrow \text{undefined}))$ 
.14     else undefined)

```

51. type: $Fid \times (Sid \times P)^* \times R\text{-set} \rightarrow ((Sid \nrightarrow SET_{11}) \rightarrow R\text{-set})$

This last definition, incidentally, checked for existence of arrows before checking for undirected path well-formedness -- something the previous definition assumed, but did not check!