MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Computation Structures Group Memo 72

An Approach to Proving the Correctness of Data Base Operations

by

Igor T. Hawryszkiewycz
Jack B. Dennis

This paper was prepared for presentation at the ACM SIGFIDET Workshop on Data Description, Access, and Control, Denver, Colorado, November 29 - December 1, 1972.

October 1972

The work herein was carried out at Project MAC, MIT, and was supported in part by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr N00014-70-A-0362, and in part by the Postmaster-General's Department, Australia.

An Approach to Proving the Correctness of Data Base Operations

I. T. Hawryszkiewycz

J. B. Dennis

ABSTRACT

Work on the development of a precise semantic model for data base systems is presented. Our model is intended to provide a formal basis for investigating suitable semantic principles for data base systems where many users may be performing tasks concurrently, and where data bases are shared among users. The application programs of system users are assumed to be expressed in terms of a modified relational data base model. The semantics of primitive data base operations are expressed in terms of semantic routines that specify transformations of abstract data structures which represent states of the data base system. Our approach to establishing correctness of semantic routines in the presence of concurrent system activities is described.

The work herein was carried out at Project MAC, MIT, and was supported in part by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr N00014-70-A-0362, and in part by the Postmaster-General's Department, Australia.

1. Introduction

We are working toward defining a precise semantic model for data base systems where many users may be performing tasks concurrently, and where data bases may be shared among users. Our approach is illustrated in Figure 1, and makes use of several notions discussed previously by Dennis [1]. We suppose that data bases take the form of collections of domains and relations according to the relational model of Codd [4]. Operations on data bases will be modelled in terms of transformations of abstract data structures by sequences of abstract operations. The objective is to extend the semantics of the relational model to situations in which concurrency and sharing occur, and to specify these semantics in terms of a lower level abstract language. One advantage of this approach is that correct implementation of the semantic specification will automatically ensure correct operation of a data base system.

Codd's work in [2] has shown that the relational model can represent the type of structures outlined in the CODASYL Systems Committee Report [3], and he claims [4] that this model exhibits a greater degree of data independence than the group, tree or plex structures described in the CODASYL Report. Thus our work is applicable to a wide class of data bases.

We assume that the reader is familiar with the relational model of Codd [2]. In this paper we first describe our abstract objects and basic operations. This is followed by a discussion of our representation of domains and relations, with some examples demonstrating our method of semantic definition.

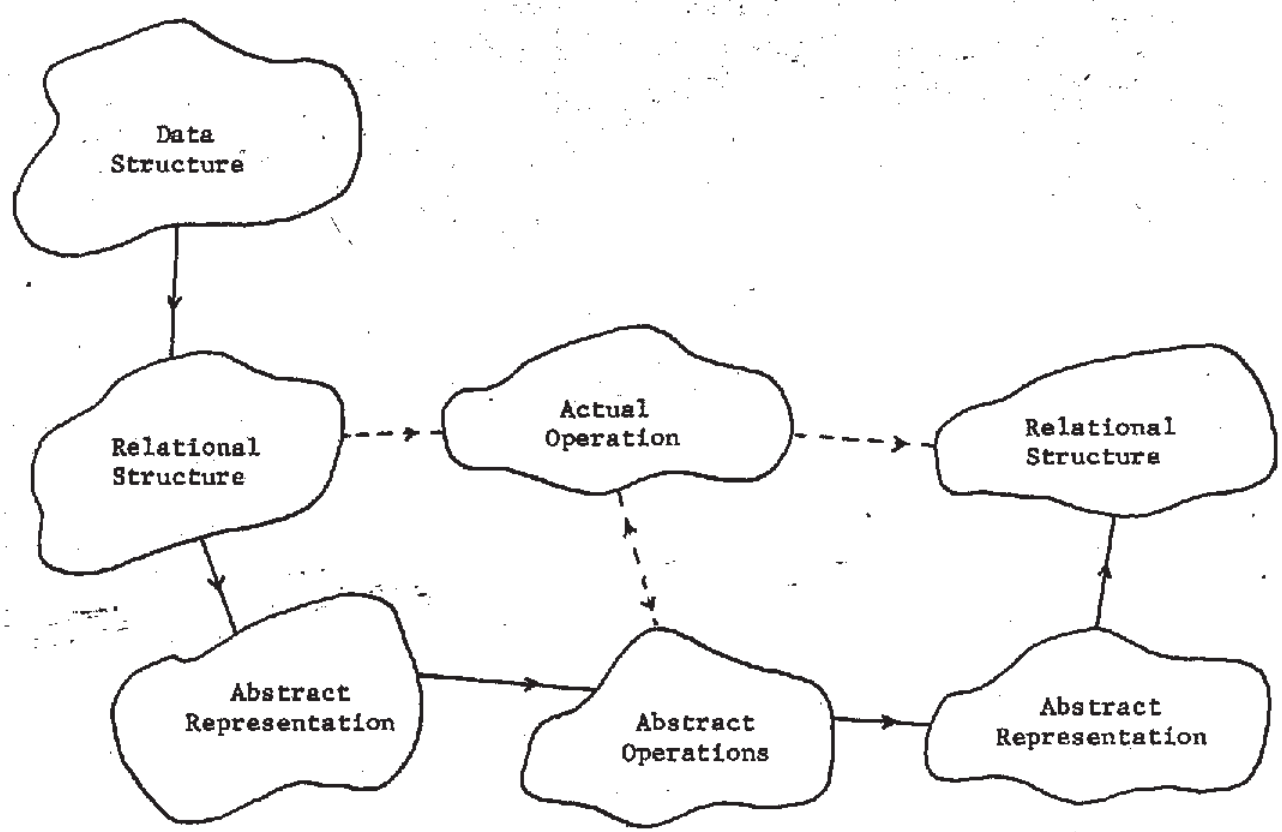


Figure 1. Conceptual Model.

2. Abstract Objects and Operations on Abstract Objects

Our abstract objects are directed acyclic graphs. The arcs between the nodes of the graph are called branches. A special class of nodes that may not have branches emanating from them are known as elementary objects. Each elementary object has a value which may be (for illustration) an integer, a representation of a real number, or a string. Each branch of an object is labelled with a string or an integer.

In our semantic definitions for primitive data base operations, we use two classes of variables -- structure variables and selector variables. The value of a structure variable ranges over the nodes of abstract objects. The value of a selector variable ranges over the set of possible branch labels. By a selector expression we mean either an identifier of a selector variable or an actual branch label denoted by a string enclosed in single quotes. The value of a selector expression is a branch label. If P is an identifier of a structure variable and $x_1, \ldots, x_k$ are selector expressions, then

$$P.\ x_1 . x_2 . \ \ldots \ . x_k$$

is a structure expression. The value of a structure expression is the data base node reached from node P by tracing the path with successive branches labelled with the values of $x_1, \ldots, x_k$. The value of a structure expression is undefined if no such path exists. If E is a structure expression and x is a selector expression, then

$$E(x)$$

is a branch expression, the value of which is the branch of the object that emanates from thenode identified by the value of E, and has the value of x as its label.

To illustrate this notation, consider the abstract object shown in Figure 2. Suppose node n1 is the value of the structure variable P. Then P.'a' will refer to node n2. If 'b' is the value of selector variable x then both P.'a'.'b' and P.'a'.x refer to node n3. P.'a'('b') and

P.'a'(x) will both refer to the branch labelled with the string 'b' and emanating from node n2.

To refer to the value of an elementary object we use the function val. Application of val to a structure expression that refers to an elementary object gives the value of the elementary object. If the structure expression does not refer to an elementary object then val is undefined. For example, in Fig. 2 val(P.'a'.'c') = 3, but val(P.'a') is undefined.

The operations and tests we shall use to build the routines that define data base operations in terms of abstract objects are explained in Table 1. Two Boolean variables, called lock and block, are associated with each node of an abstract object. The lock variable for a node n is used to exclude concurrent processes from simultaneous access to the object having n as its root node. The block variable will be used in the semantic routines to limit alteration of objects that represent parts of data bases shared among system users. The lock and block variables of a node are referred to by writing lock(P) or block (P) where P is a structure expression that refers to the node.

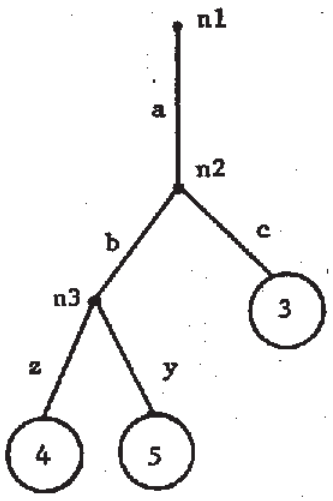


Figure 2. An abstract object.

Table 1. Abstract Functions

selt(P,x)	= T, if there is a branch labelled with the value of selector expression x, and emanating from the node to which P refers = F, otherwise
empty(P)	= T, if there are no branches emanating from the node to which P refers. = F, otherwise
del(P(x))	The branch labelled with the value of the selector expression x and emanating from the node to which P refers is deleted.
append(P,x)	A branch labelled with the value of selector expression x is attached to the node to which P refers.
create(P)	A new node is created and assigned to P. The block and lock variables are set to F.
link(P,Q,x)	A branch labelled with the value of x is created. It emanates from the node to which P refers and terminates on the node to which Q refers.
lock(P)	If the lock variable associated with P is T, then the process waits until it becomes F. It is then set to T and the process continues. If it is F then the process sets it to T and continues.
unlock(P)	The lock variable associated with P is set to F.
principal(p)	= T, if the program that invoked this command belongs to user p. = F, otherwise.
block(P)	The block variable associated with P is set to T.
isblock(P)	= T, if the block variable associated with P is T. = F, otherwise.

3. Representation of Domains

We propose a model for the semantics of a data base system through which a number of users gain access to and manipulate relational data bases by running application programs expressed in terms of primitive data base operations. A state of the data base system consists of the current contents of a collection of relational data bases, the user application programs, and information about the state of execution of each application program.

In our model we represent the data base component of the system state as an abstract object and define a set of primitive data base operations by routines that perform transformations of the abstract state.

In any state of the data base system, the data bases in existence are defined with respect to a collection of domains and a collection of relations that involve these domains. Here we discuss the representation of domains, access to domains, and the formation and sharing of domains. In subsequent paragraphs we discuss the representation of domain values and of relations as abstract objects.

The value of a domain is a set of strings. Domain values are subject to modification by the action of application programs. If D is the set of domain variables that exist at some moment in operation of the data base system, the collection of domain values may be represented by a value function

$$V_D : D \to \wp(S)$$

where S is the set of all strings useable as names of domain elements and $\wp(S)$ is the power set of S.

Let U be the set of users of the data base system. We model the ability of users to gain access to domains by a set N of access pairs:

$$N \subseteq \{\langle u, n\rangle \mid u \in U \text{ and } n \in S\}$$

For an access pair $\langle u, n\rangle$, n is the name by which user u refers to some domain variable $d \in D$. The correspondence of access pairs to domain variables is represented by the access function

$$A_D : N \to D$$

which associates a domain variable with each access pair for which A_D is defined.

We now propose an abstract representation for the domain structure. This is shown in Fig. 3. Here the access function is expressed by the paths between the user nodes and domain variable nodes. The value function is expressed by the paths from the domain variable nodes to the value objects. Thus in Fig. 3 user u_1 refers to domain variable d2 using the name n1, whereas user u_2 refers to the same domain variable by the name n4.

Let us now discuss the semantics of operations on the domains. There are two classes of operations -- those concerned with the access function and those concerned with the value function.

In this paper we will only discuss some of the primitives concerned with the domain access function. These primitives allow us to control the sharing of domains and are discussed in the next section.

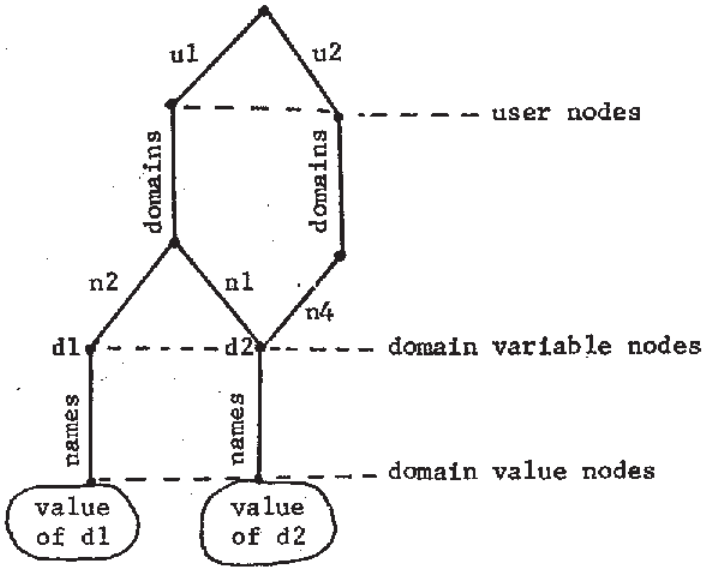


Figure 3. Representation of access to domains.

Table 2. Semantic Definitions of Access

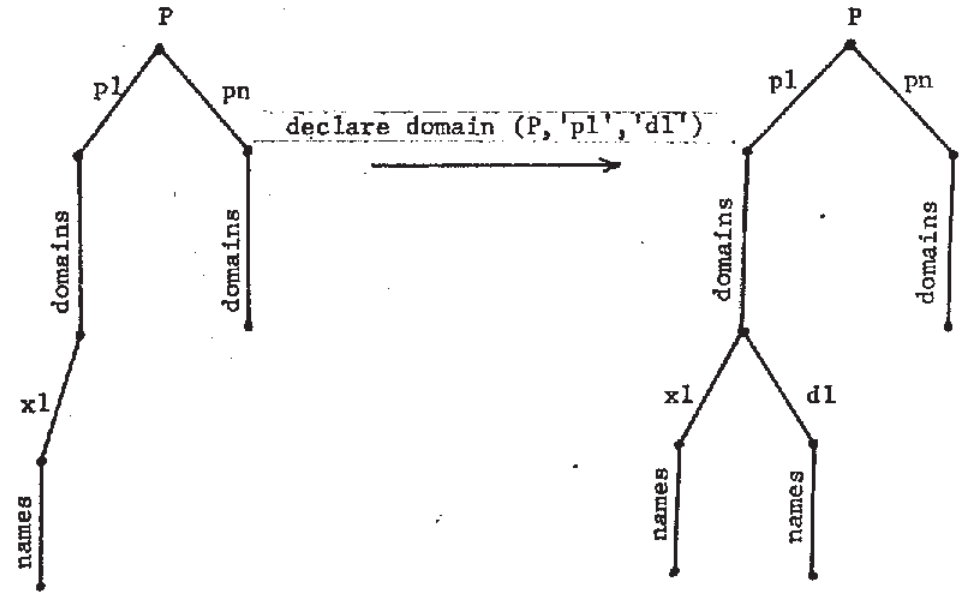
```
declare domain (P,pl,dl)
    if principal(pl)
        then lock(P.pl);
            if selt(P.pl.'domains', dl)
                then unlock(P.pl); return F;
                else append(P.pl.'domains', dl);
                    append(P.pl.'domains'.dl, 'names');
                    unlock(P.pl); return T;
        else return F;
end;

create domain (P,pl,dl,d2)
    if principal(pl)
        then lock(P.pl)
            if selt(P.pl.'domains', dl)
                then if selt(P.pl.'domains', d2)
                    then unlock(P.pl); return F;
                    else block(P.pl.'domains'.dl.'names');
                        append(P.pl.'domains',d2);
                        link(P.pl.'domains'.d2,
                            P.pl.'domains'.dl.'names','names');
                        unlock(P.pl); return T;
                else unlock(P.pl); return F;
        else return F;
end
```

Table 2 (continued)

```
share domain (P,S,p1,d1,p2,d2)
    if principal(p1)
        then lock(P.p1);
            if selt(P.p1.'domains', d1)
                then if selt(S.p1.p2, d2)
                    then unlock(P.p1); return F;
                    else link(S.p1.p2, P.p1.'domains',d1, d2);
                        unlock(p.p1); return T;
                else unlock(P.p1); return F;
    else return F;
end;

borrow domain (P,B,p1,d2,p2,d3)
    if principal(p2)
        then lock(P.p2)
            if selt(B.p2.p1, d2)
                then if selt(P.p2.'domains', d3)
                    then unlock(P.p2); return F;
                    else link(P.p2.'domains',B.p2,p1.d2, d3);
                        unlock(P.p2); return T;
                else unlock(P.p2); return F;
    else return F;
end;
```


P
p1
pn
domains
domains
x1
names
declare domain (P,'p1','d1')
P
p1
pn
domains
domains
x1
d1
names
names

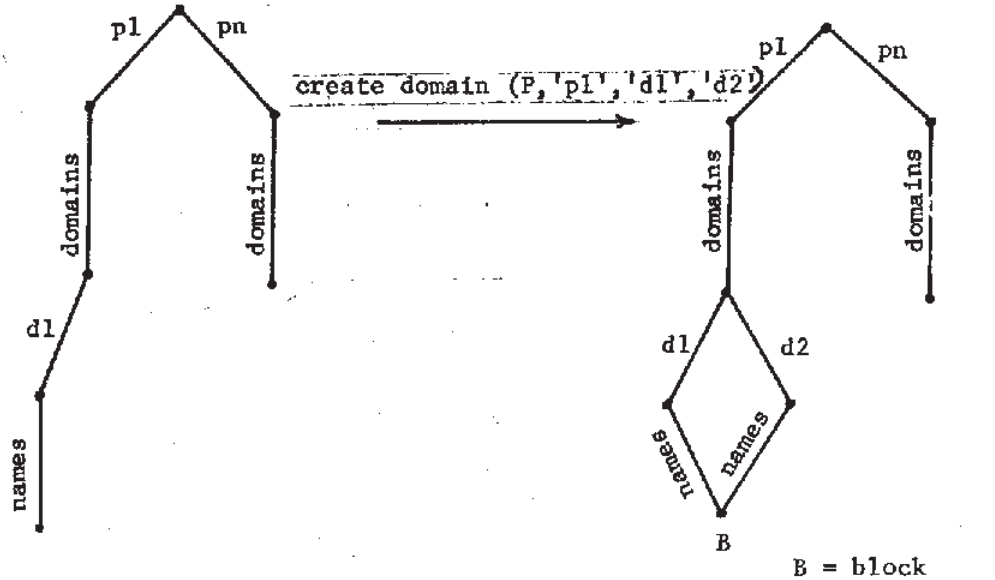

p1
pn
domains
domains
d1
names
create domain (P,'p1','d1','d2')
p1
pn
domains
domains
d1
d2
names
names
B
B = block

Figure 5

3.1 Sharing of Domains

In a shared data base system we must provide a method for making information available to a number of users and yet control this sharing of information in a way that actions of one user do not interfere with the correctness of actions of other users.

One approach that can be used here is to introduce the concept of ownership, which implies that some users may have ownership rights with respect to certain objects in the system which other users do not have.

We may consider the following simple notion of ownership:

(1) If a user owns a variable then he has ownership rights to the variable.

(2) If a user has ownership rights with respect to a variable he may access, or alter the value of the variable, and may give ownership rights with respect to the variable to other users.

This notion of ownership is realized by the primitives declare domain, create domain, share domain and borrow domain. A declare domain operation will create a new domain variable, whose value is initialized to the empty set, and an access pair that defines the name by which this user refers to the domain variable. The create domain primitive creates a new domain variable, whose value is the same as some existing domain variable, and creates a new access pair by which the user refers to the new domain variable.

The declare domain and create domain primitives are defined in terms of abstract operations in Table 2, and their effects on the abstract structure are shown in Fig. 5. The effect of the declare domain needs no explanation, but the create domain needs some elaboration. One of the effects of create domain is that the value structure is now shared by the two domain variables and the block variable of the value node is set to T. The blocking ensures that primitive operations, which subsequently

change the value of a domain variable, do not also change the value of some domain variable that shares the value structure.

For example, in Fig. 4(a) the value structure that is defined by node B is shared by two domain variables. This value may be considered to consist of two parts, S_1 and S_2. Suppose the user who refers to one of the domain variables by the name 'n', changes the part S_1. The resulting structure is shown in Fig. 4(b). The value appearing to the user

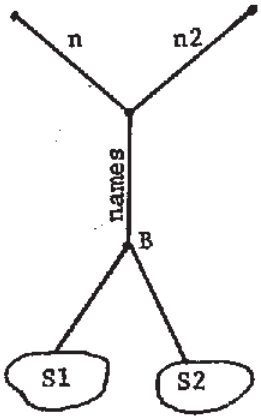


Figure 4(a)

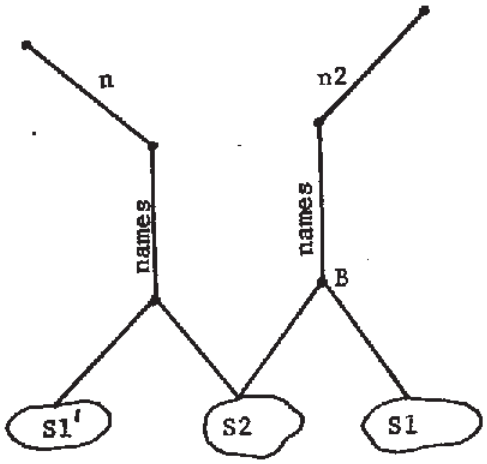


Figure 4(b)

who refers to the other domain variable by the name 'n2' remains unchanged. Thus he retains sole ownership to this domain variable. The node that defines the structure S_2 would now become blocked so that a similar effect results if S_2 is altered.

Sharing of domains is represented in our data base system by the function

$$F_D: U \times U \rightarrow \mathcal{P}(Y)$$

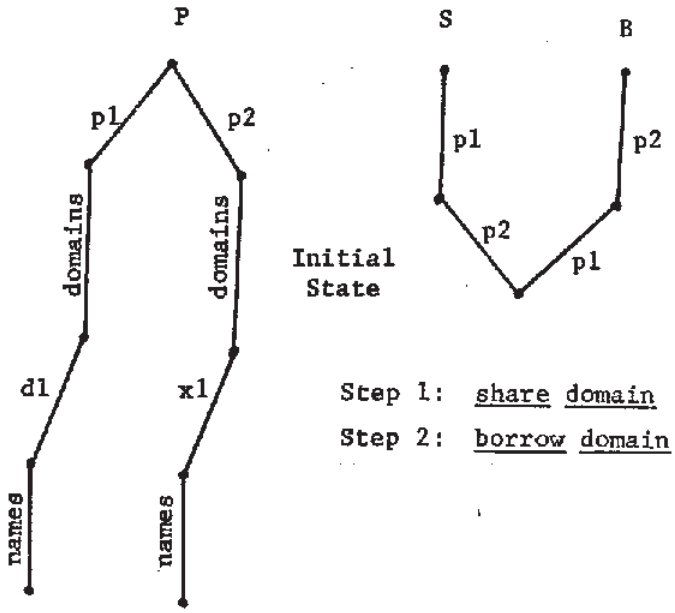
where $Y = S \times S$

where Y is a set of name pairs and $\mathcal{P}(Y)$ is the power set of Y. F_D maps each pair of users $\langle u_1, u_2 \rangle$ to a set of name pairs. Each element $\langle n_1, n_2 \rangle$ of this subset is a name pair which indicates a domain variable that user u_1 wishes to share with user u_2. n_1 is the name by which u_1 refers to this domain variable. The share domain primitive has the effect of adding one pair to the subset defined by some user pair. Each name, n_2, is a name that user, u_2, will use in the borrow domain primitive to access the domain variable that has been made available to him by a share domain primitive. The borrow domain primitive will also create a new access pair $\langle u_2, n_3 \rangle$, which will define the name by which user u_2 wishes to access the shared domain variable, and a reference from this access pair to the domain variable.

Sharing is represented in our abstract state by including a share structure defined by the nodes S and B, and shown in Fig. 6. P in this figure is that part of the state that defines the domains and relations currently accessible to the users in some relational data base. S and B represent that part of the state defined by the relation F_D. A user u_1 makes a domain variable to which he refers by the name n_1 available to user u_2, by linking node S.'u1'.'u2' to the domain variable node with a branch labelled n2. u2 can now gain access to this domain variable by the expression B.'u2'.'u1'.'n2' and create a branch to this variable in the P structure.

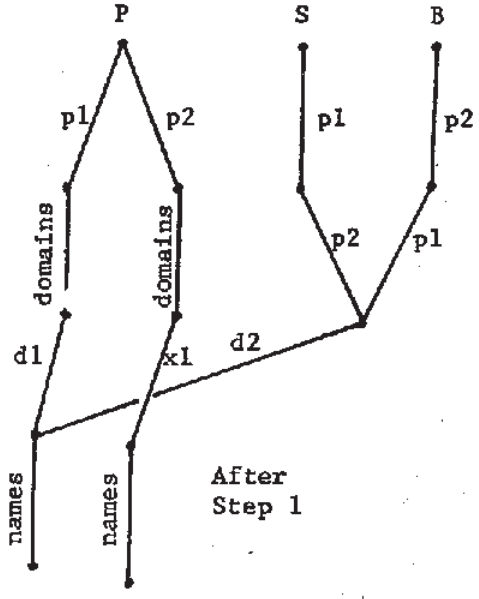
Table 2 defines the primitives share domain and borrow domain, which are used to transfer ownership rights and the effect of them is illustrated in Fig. 6. For convenience in Table 2 we have used the structure variables P, S and B as bound variables. Application programs will be constructed using primitives where these variables are free, and implicitly refer to appropriate parts of the abstract data structure.

We note that given these semantics it is necessary that a user have sole ownership of a domain if we are to ensure that the value of the domain may not be changed by some other user. This is an unsatisfactory



Step 1: share domain (P,S,'p1','d1','p2','d2')

Step 2: borrow domain (P,B,'p1','d2','p2','d3')



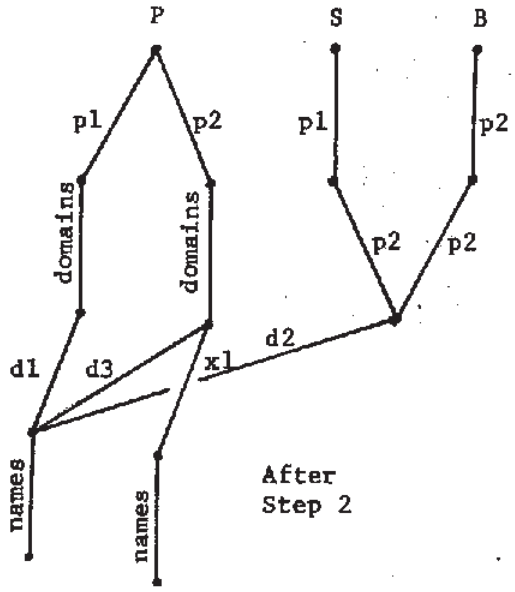


Figure 6

restriction, as a user will often wish to grant other users access to a domain with assurance that the domain value cannot be altered by other users who, however, may have access to the domain. Hence, we require a second kind of ownership, which allows a user to:

(i) access the value of a domain, but not alter it.

(ii) give the same ownership rights to some other user.

Two approaches are possible to the specification of this extended notion of ownership:

(i) Extend our objects so that a variable with value W or R is associated with each node. We can then have the structure shown in Fig. 7. We can define our semantics in such a way that a program accessing the value structure through the branch

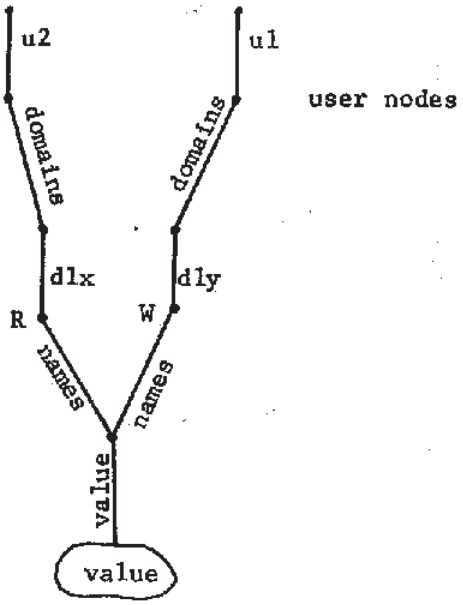


Figure 7

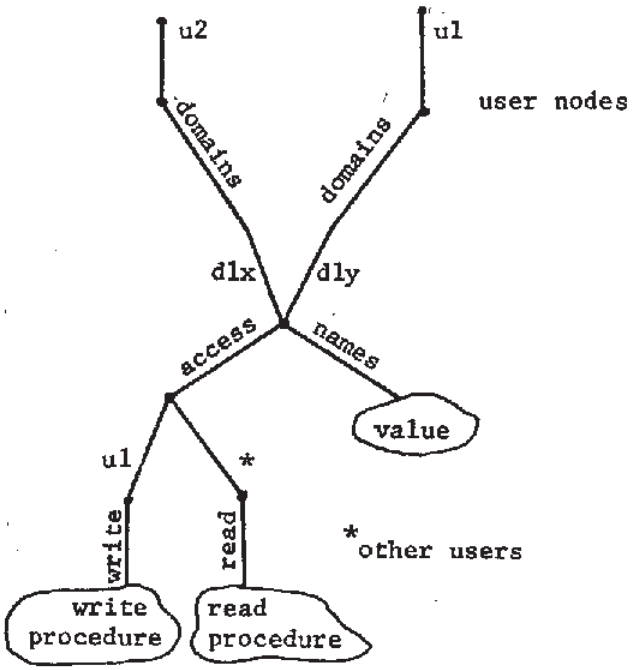


Figure 8

dlx cannot alter the value as R is associated with its domain variable node. A program accessing the value through dly can change the value, as W is associated with its domain variable node.

(ii) We can associate with each domain in the structure, the actual operators that a borrower is allowed to use. This is illustrated in Fig. 8. We associate a domain access structure with the domain variable. User ul is the only user allowed to use the write primitive. All other users can only read the value of the domain variable.

It is not clear at this stage as to which is the more satisfactory alternative. This will become clear as our work proceeds and we develop a complete set of primitives.

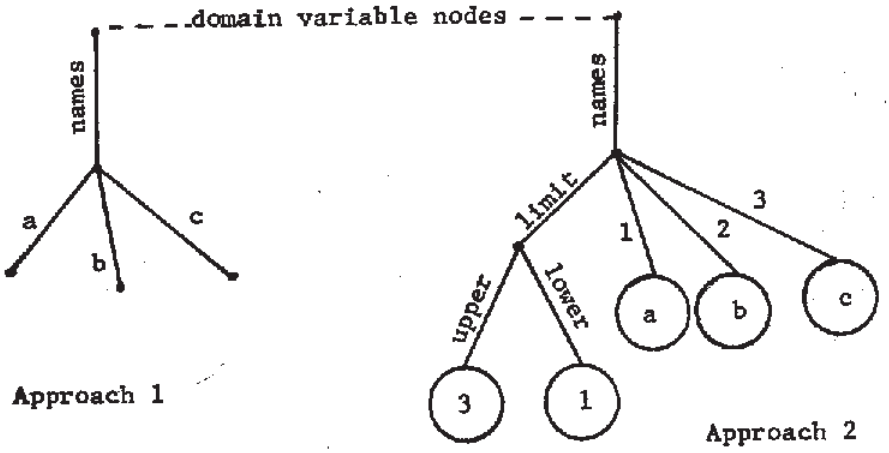


Figure 9. Representation of domain value.

3.2 Representation of Domain Value

Two approaches for the representation of domain values are illustrated in Fig. 9.

Conceptually we must have the ability to obtain any domain element by giving its name, or all domain elements in some conventional order. The first ability -- direct access by name -- is needed, for example, to answer an interrogation as to whether a specified element exists in the domain. The second ability is needed when every domain element must be examined in order to perform the operation requested.

By adopting approach 2 in Figure 9 we can describe the semantics of direct access in terms of a sequential search. Approach 1 is better suited to explain the semantics of direct access but is unsatisfactory for ordered sequential access. We feel that a composite approach is required that allows primitives such as:

"obtain the name of the next element of domain d"

"for all items in the domain d . . ."

This would require the definition of a conventional ordering of the elements of domains which could be the natural order of the elements of S.

4. Representation of Relations

For the representation of the relations of a data base we are again concerned with representation of access to relations and to their values. Access by system users to relations by name is the same as access to domains, and makes use of operations analogous to declare domain, create domain, share domain, and borrow domain. Let R be a set of relation variables, and let the relation access function be

$$A_R : N \rightarrow R$$

New problems arise in representing values of relations, for the elements of a relation may be accessed over various paths and a representation should permit efficient access to relation elements over all access paths that are expected to be heavily used during system operation. Furthermore, it should be possible to easily change the representation to include additional access paths if new uses of the relation are introduced subsequent to data base formation. These requirements are similar to facilities provided in the language LEAP [6].

The relation value function for R is

$$V_R : R \rightarrow \mathcal{R}$$

where $\mathcal{R}$ is the class of all n-ary relations on domains in D where $n \geq 2$:

$$\mathcal{R} = \bigcup \mathcal{R}_n, \quad n = 2, 3, \ldots$$

$$\mathcal{R}_n = \{\langle\langle d_1, \ldots, d_n\rangle, \mathcal{D}\ (\langle d_1, \ldots, d_n\rangle)\rangle \mid d_i \in D\}$$

$$\mathcal{D}\ (\langle d_1, \ldots, d_n\rangle) \subseteq \mathcal{P}(\{\langle e_1, \ldots, e_n\rangle \mid e_i \in V(d_i)\})$$

In the above, $\mathcal{R}_n$ is the set of n-ary relations, where a relation for our purposes is a pair consisting of an ordered n-tuple of domain variables and a set of n-tuples $\mathcal{D}\ (\langle d_1, \ldots, d_n\rangle)$ called relation instances where each component of a relation instance is an element of the value set of the corresponding domain variable.

We have devised an abstract representation for relation values that is able to represent as many potential access paths as appropriate. The representation has three parts as shown in Figure 10. The access structure identifies the domains of the relation and their order. The descrip-

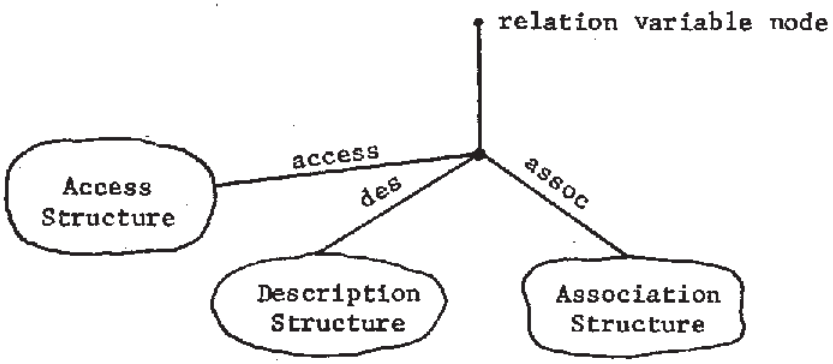
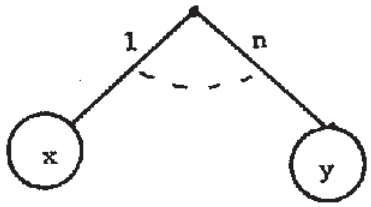
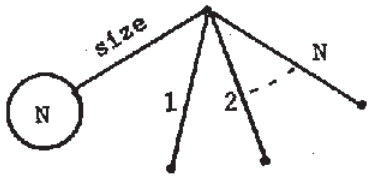


Figure 10

directory object



relation instance object



order object

Figure 11

tion structure contains information about how the set of relation instances is represented in the association structure.

The association structure is composed of the three kinds of objects illustrated in Figure 11:

(i) the directory object: Branches emanating from the root node of this object are labelled with item names. All the item names are elements of the same domain.

(ii) the relation instance object: Each branch is labelled with an integer that identifies a domain and terminates on an elementary object whose value is an element of that domain.

(iii) the order object: A finite number of branches terminate on relation instance objects. The branch labels are consecutive integers 1, ..., N.

Let us now consider how these objects may be used to represent the association structure. Suppose a user can access the relation shown in the following table.

Relation R1

D1	D2	D3
a	n	x
a	m	x
b	n	y

Figure 12 shows one possible representation for this relation. The access structure indicates that the relation has three domains which (in this relation) have the names D1, D2, and D3; the access structure also has links to the domain variable nodes that represent the values of these domains. In Figure 12 the association structure represents the set of relation instances by an order object having one component instance object

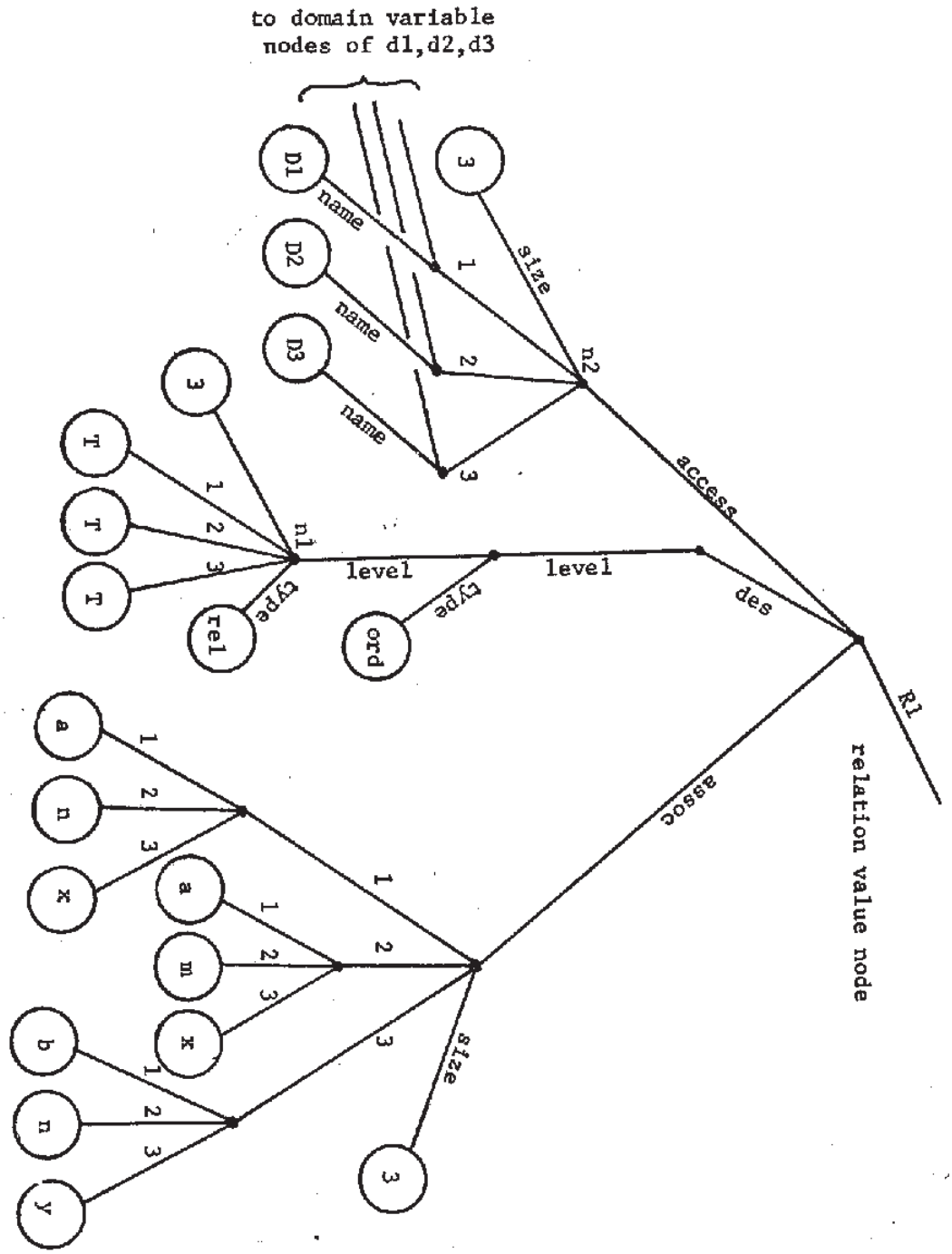

to domain variable
nodes of d1,d2,d3
D1
D2
D3
name
name
name
1
2
3
3
size
n2
access
3
T
T
T
1
2
3
n1
level
level
des
type
type
rel
ord
R1
relation value node
assoc
a
n
x
a
m
x
b
n
y
1
2
3
1
2
3
1
2
3
1
2
3
size
3

Figure 12.

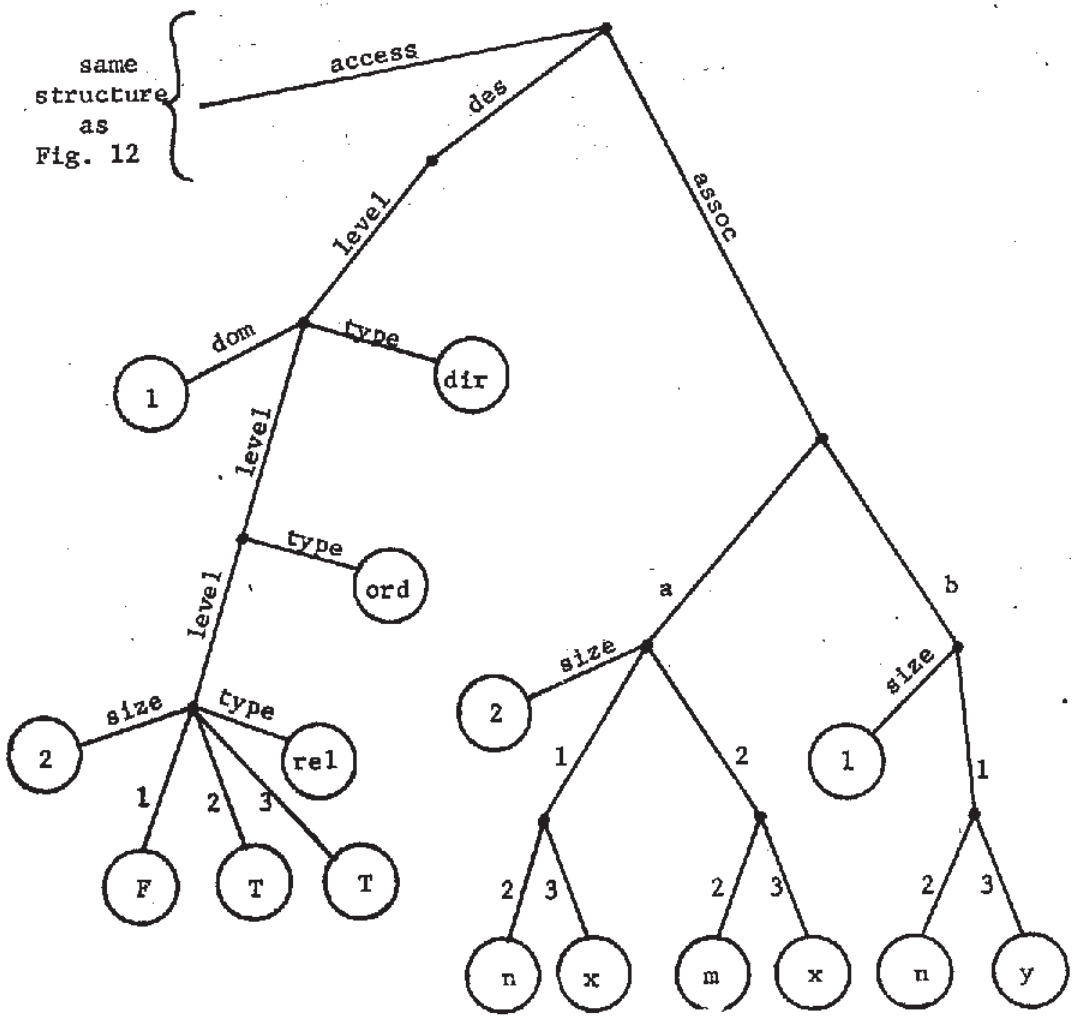

same structure as Fig. 12
access
des
assoc
level
dom
type
1
dir
level
type
ord
level
size
type
2
rel
1
2
3
F
T
T
a
b
size
size
2
1
1
2
1
2
3
2
3
2
3
n
x
m
x
n
y

Figure 13.

for each relation instance of the relation. The description structure states that the two levels of the association structure are composed of an order object (type has the value ord), and relation instance objects (type has the value rel).

The integer-labelled branches emanating from node n1 in the description structure indicate by Boolean values which domains are represented in corresponding relation instance objects of the association structure; in Figure 12 all three domains are represented, as is necessary to completely represent the relation.

The association structure of Figure 12 does not permit easy access to just those relation instances having specified domain elements in certain positions. Efficient access to a relation using elements of one or more domains as keys may be provided by using directory objects at one or more levels in the association structure. In illustration the relation R1 may also be represented as in Figure 13 in which the top level of the association structure is a directory object (type value is dir in the description structure). Since the domain D1 element of a relation instance from the path over which it is reached, the branch labelled 1 of each relation instance object in Figure 12 may be omitted.

5. Correctness of Data Base Systems

The correctness of semantic routines such as those presented in Table 2 for the domain access and sharing primitives of a data base system may be studied as follows: The meaning of each data base primitive is defined by the change it makes in the access functions and value functions for domains and relations. We may consider the access and value functions that apply to some moment in system operation to form a data base state and regard execution of a data base operation as performing a transition between a pair of data base states.

For example, the operation declare domain (P, u, n) can be defined as the following transformation on the data base state.

$D \rightarrow D \cup d$	-- a new domain variable d is created
$N \rightarrow N \cup \langle u, n \rangle$	-- a new access pair is created
$A_D \rightarrow A_D \cup \langle \langle u, n \rangle, d \rangle$	-- the new domain variable is associated with the new access pair
$V_D \rightarrow V_D \cup \langle d, \emptyset \rangle$	-- the initial value of the new domain variable becomes the empty set

R, A_R, F_D and V_R are unchanged.

Given a data base state s_1, execution of a data base operation a results in a new data base state s_2 where

$$s_2 = \tau_a(s_1)$$

and τ_a is the state transformation for operation a. The semantic routine for operation a specifies a sequence

$$z_a = z_{a1}, z_{a2}, \cdots z_{ak}$$

of transformations to be applied to the abstract object t_1 that represents the data base state s_1. Let $\mathcal{O}$ be a relation that contains a pair $\langle s, t \rangle$ if t is an abstract object that represents the data base state s ac-

cording to some scheme such as we have outlined above. Let $\mathcal{D}^{-1}$ be the converse of $\mathcal{D}$. Relation $\mathcal{D}$ is one-too-many because there are many ways in which one relation might be represented depending on the anticipated use.

The semantic routines are correct if two conditions are satisfied:

1) The transformation of t_1 defined by the sequence z_a is consistent with the transformation of s_1 by τ_a; that is

$$s_2 = \tau_a(s_1) = \mathcal{D}^{-1}(z_a(\mathcal{D}(s_1)))$$

for all choices of the initial state s_1. This condition is illustrated by the commutation diagram in Figure 14.

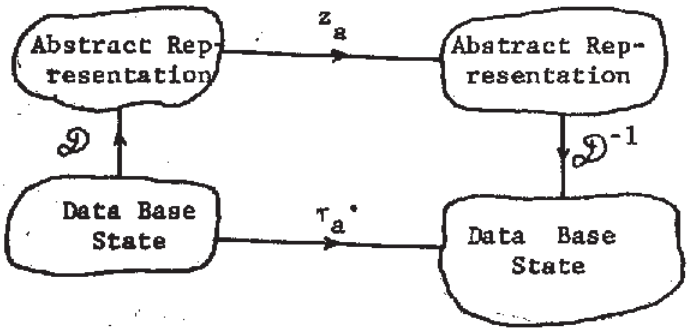


Figure 14

2) Let a and b be data base operations executed concurrently by two users with the data base system in state s_1. There are two "correct" outcomes (which need not be distinct):

$$s_2 = \tau_b(\tau_a(s_1))$$

$$s_3 = \tau_a(\tau_b(s_1))$$

Regardless of how the elements of z_a and z_b are merged to z, the sequence of abstract transformations applied to some object t_1 such that $\langle s_1, t_1 \rangle \in \mathcal{D}$, the result

$$t_2 = z(t_1)$$

is a representation of one of s_2 or s_3; that is, either

$$\langle s_2, t_2 \rangle \in \mathcal{D} \quad \text{or} \quad \langle s_3, t_2 \rangle \in \mathcal{D}$$

We are studying the choice of primitive data base operations suited to data base systems in which there are concurrent activities and sharing of information among users, and the design of semantic routines such that concurrency may be effectively employed while meeting our conditions for correctness.

References

1. J. B. Dennis, On the exchange of information. Proceedings 1970 ACM-SIGFIDET Workshop on Data Description, Access and Control.
2. E. F. Codd, Normalized data base structure: A brief tutorial. Proceedings 1971 ACM-SIGFIDET Workshop on Data Description, Access, and Control.
3. Feature Analysis of Generalized Data Base Management Systems. CODASYL Systems Committee, May 1971 (available from ACM).
4. E. F. Codd, A relational model of data for large shared data banks. Comm. of the ACM, Vol. 13, No. 9 (June 1970).
5. J. B. Dennis, Programming generality, parallelism and computer architecture. Information Processing 68, North-Holland, Amsterdam 1969, pp 484-492.
6. P. D. Rovner and J. A. Feldman, The LEAP Language and Data Structure. Report DS-6898, M.I.T. Lincoln Laboratory, January 1968.