tanulmányok 105/1980

MTA Számitástechnikai és Automatizálási Kutató Intézet Budapest

MAGYAR TUDOMÁNYOS AKADÉMIA
SZÁMITÁSTECHNIKAI ÉS AUTOMATIZÁLÁSI KUTATÓ INTÉZETE

E. Knuth, P. Radó, Á. Tóth

PRELIMINARY DESCRIPTION OF SDLA

25. dec. 1979 II/7

The concepts in this overview, though fruits of lengthy considerations, are yet incomplete in several aspects and demand further affords of deep research. Your remarks will be sincerely welcome too.

This research was supported in part by the National Bureau of Computer Applications and in part by the Hungarian Academy of Sciences.

Tanulmányok 105/1980.

A kiadásért felelős:

DR VÁMOS TIBOR

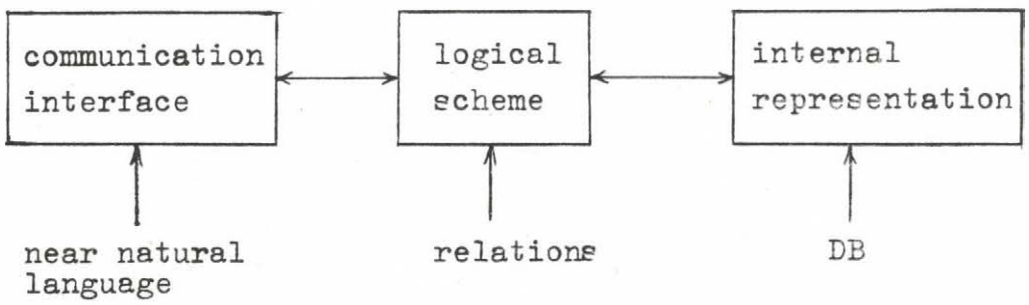
ISBN 963 311 099 8
ISSN 0324-2951

Készült a
KSH Nemzetközi Számítástechnikai Oktató és Tájékoztató
Központ Reprográfiai Üzemében 80/072

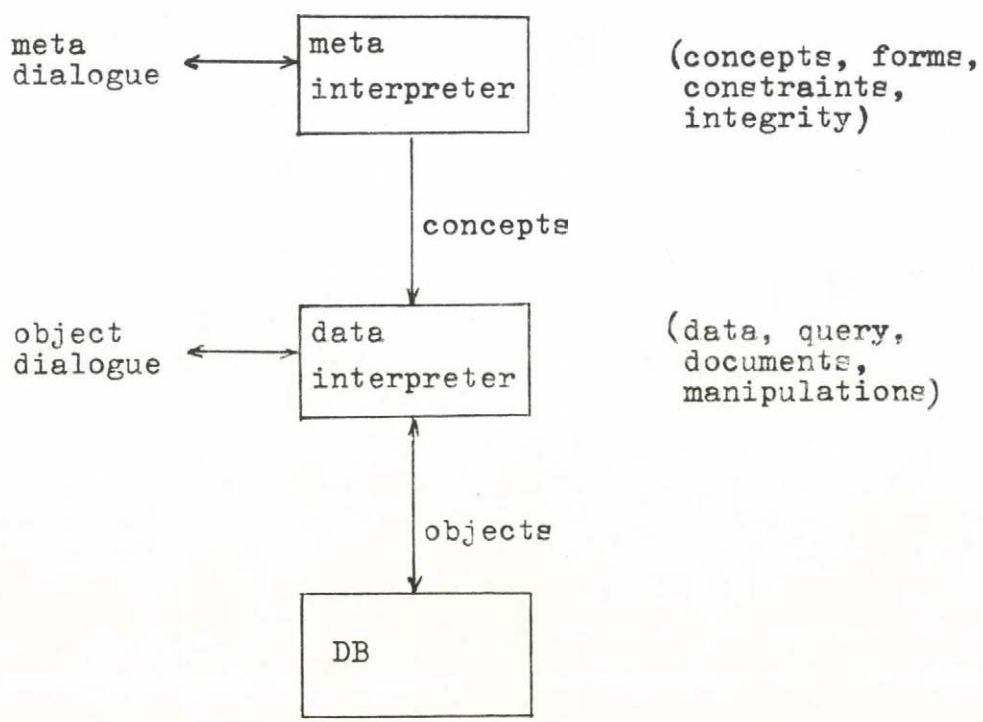
S D L A

Structural Descriptor and
Logical Analyzer

Horizontal view



Vertical view



OVERVIEW OF SDLA

Its purpose

is to aid the survey, recognition, handling, planning, and production of logically complex systems by storing formal descriptions in a data base and facilitating their logical analysis and documentation. This need is widely recognized since it became obvious that the human mind is unable to control all the necessary details of systems consisting of a large number of components and interconnections among them.

Its situation

with respect to similar ones is to form a common base which is unavoidable in any kind of such purpose systems and to provide means for building special-purpose applications.

Several system description languages are known today, both for special targets and for general purposes. An outstanding one of the latter kind is for example the DELTA language, which does not generate an executable code and thus is non-procedural like the SDLA is. In the range of such languages the SDLA stands out by storing the input information in a data base-like form corresponding to a strict logical scheme, thus providing logical tools of a high generality for the information analysis. In the design of the language we still relied on the rich ideas of these languages.

Another kind of the well-known approaches is (in ISDOS, SADT etc.) to represent the information processed in a data base if there is any in the system . These are in general restricted by the fact that (understandably) they are unable to accept concepts but those fixed at the construction of the system and relations between them but of the kinds permitted by their construction. (E.g. SADT uses directed links between the objects, etc) In the SDLA nothing is determined in advance: our aim is to provide the mehanism allowing concept definitions to be the basic facility in the system.

SDLA differs from the other systems in several further points, from which the most essential one is the introduction of the meta relations as constraints and refinements .

Its application

begins with the definition of the concepts needed together with their invariants and integrity constraints. (These can be stored as a library too.) Having determined these we get an interactive data base management system with a high-level user interface to handle it. This differs from the commercial ones in aiming to handle data of a complex logical structure instead of in huge masses, this with a proper consideration of the viewpoints of the planning.

Its Technical means

are

1. A uniform approach and way of handing of both concepts and associations between them; thus taking the latter for a kind of concepts. (This approach refuses Codd's point of view.)

2. The reference attutude delivers the user from the problem if the referenced object is simple or composite and permits to handle recursive structures in an easy way.

3. Semantical constraints: these are constraints which correspond to the meaning of the data types and are defined by the user. They result in the automatic generation of the defined kind of statements.

4. Integrity checking : based on relations and connections between them as declared by the user they are used to check input data descriptions semantically for logical correctness.

5. A concept refinement facility (by the use of types and subtypes) facilitates refined type control.

* * *

The authors express their thanks for their remarks to
G Almássy, I. Bach, A. Benczúr, P. Bernus,
B. Dömölky, E. Farkas, A. Gáspár, I. Gehér, H.J. Genrich,
H. Kangassalo, O. Kiss, L.B. Kovács, A. Márkus, C.A. Petri,
A. Prékopa, Gy. Révész, L. Rónyai, A. Sárközy, P. Szeredi,
J. Szlankó, M. Szokolov, and T. Sztanó.

CONTENT

I. LOGICAL STRUCTURE OF THE DATA REPRESENTED

The logical scheme of the data represented means the aspect of the user towards his data (independently of the physical representation of the data). This is a kind of relational attitude which still differs significantly from that of Codd's both in its form and in its aim.

Communication with the data base is performed on a high level language akin to the way of human thinking which is to be described in Chapter II. This is linked to the relational structure by a well-defined mapping. Even the idea of such an improvement of our system which permits communication without giving thought to the relational substructure is not irrealistic, though it is a dream of the future now.

A basic thought of the logical scheme is the "homomorphism of closed reference fields", see in [6]. In this paper we do not go into these abstract questions, instead we push a move practical, user-related approach. Another basic idea for us is to introduce subordination concepts between relational structures such as refinement and the subordination constraint, which will become a deciding tool for representing realistic connections.

1. Basic scheme

In the data base we store objects, each of which is an instance of an abstract concept.

Objects are described by attributes. An abstract concept is characterized by its associated set of attributes, to which the attributes of its instances correspond in their number and types (like at any conventional parameter transfer).

The actual set of objects as instances to a given concept can always be considered as a relation (i.e. the subset of the Carthesian product of the attribute value ranges). This viewpoint is useful, as it is known, for formalizing operations in a data base.

1.1 Designation of concepts

Defining a concept we give the designation of an object class in advance. This definition takes place on the meta or definitional level. The following formalism will be used for concept definition:

(1) concept conceptname(attributename1:type1,
attributename2:type2,...etc.);

Correspondingly, a concept definition includes the designations of:

a) The concept's name;
b) Its number of attributes (a nonnegative integer);
c) For each attribute its name (selector), and its type.

1.1.1 Attribute types

A type in the above definition may be either a

a) reference type, or a
b) value type.

A reference type can be an arbitrary concept name which is also defined. (Arbitrary in this case means really arbitrary, e.g. itself, as this is just a reference, not a recursion.)

Value types can be the following:

(2) integer ,
real ,
text .

1.1.2 Definition unit

A definition unit is a sequence of concept definitions given at the definition level. It is closed or: self-contained if every attribute type in it is defined within the unit.

We call special attention to the following easy consequences of the above description:

a) (of course) the order of the sequence of definitions has no inportance. A definition unit is correct iff it is closed.

b) The number of attributes is nonnegative, i.e. it can be zero or one. This may seem to be unusual for the traditional concept of relation is usually meant between at least two attributes.

Here we take concepts from a more general point of view, representing them but by their attributes of which there are so many as required at the given abstraction level and context. To have zero attributes is a simple class designation which can have its importance as a classification factor imposed upon objects.

1.1.3 Examples

Each of the given examples is a closed (=self-contained) definition unit.

(3)
```
defunit
    concept object;
endunit
```

As no attribute is given, they are all defined; a class has been defined without further attributes (for the time being , owing to, say, its generality.)

(4)
```
defunit
    concept link ( next:link );
endunit
```

(5)
```
defunit
    concept bintree  element ( leftson:bintree element,
                               rightson: bintree element);
endunit
```

Both these examples are closed (and so correct) definition units. Link defined in (4) permits to define one-directional lists (from which, it goes without saying, cycles and trees in the bottom-up way can be built). Like this, the concept in (5) is for building binary trees.

(6)
```
defunit
    concept man;
    concept woman;
    concept marriage ( husband:man,wife: woman);
endunit
```

(7) defunit

concept man(wife:woman);
concept woman(husband:man);

endunit

These examples show two possible ways for representing heterogeneous pairs. The choice between them should be decided by the problem treated. Both are closed and (7) is an example of an attribute type (e.g. women) which isn't defined until afterwards, as it is permitted to do.

(8) defunit

concept element;
concept ordering (preceded by:element,preceded:element);

endunit

With these concepts total and partial ordering structures (of finite character), lattices etc. can be represented. We have no way to give further information about the semantical contents of the concept such as irreflexivity as yet, but this question will be treated later on.

1.2 Data objects

Suppose a closed definition unit has been given. This permits the description of data objects the following ways:

(9) concept name object name(attr1,...etc.);

and

(10) concept name(attr1,attr2,...etc.);

The "concept name" has to have been defined in the declaration unit. This will be referred to as the qualification or type of the data object now described.

The "object name" in (9) permits the reference of this object by this name. Choosing the form (10) the object cannot be referred by any name, but the process will result in an existing (and with global procedures retrievable) object item.

1.2.1 Type coincidence

Attributes in(9) and(10) e.g. attr1, attr2... must correspond by their types and number to those specified in the declaration. In detail, this means

a) to a value type attribute a corresponding value is to be given;

b) to a reference type attribute the name of an object with a coincident type is to be given;

c) or in both cases it is permitted to leave the attribute's place in the list empty (i.e. not to give any attribute value if not yet known).

1.2.2 Description unit

This is a sequence of data objects (of either one of the forms (9) and (10) each). It is closed if it does not refer to any object as attribute value that is not described in the very unit .

This viewpoint permits to disregard sequential interpretability (which cannot be the case in similar constructs of procedural languages). The gain that lies in data descriptions in arbitrary order will be demonstrated in the reader's efforts if he tries to formulate e.g. (15) say in SIMULA.

1.2.3 Examples

The first of these constructs a list with still no further purpose using (4):

```
(11)  defunit

         concept link element(next:link element);

      endunit;

      dataunit

         link element(A,B);
         link element(B,C);
         link element(C,D);
         link element(D,);

      endunit
```

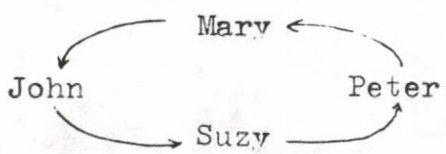
Similarly, a binary tree can be given relying on (5) as

(12)
```
defunit
   concept bintree element(left: bintree element,
                           right: bintree element);
endunit;

dataunit
   bintree element root(A,B);
   bintree element A(C,D);
   bintree element B(E,);
   bintree element A(,F);
   bintree element D(,);
   bintree element E(,);
   bintree element F(,);
endunit
```

The reader will have noticed the last lines of these examples as rather meaningless and necessary but to maintain self-containedness of the descriptions. We remark however, that this applies no more to the user language given in chapter II. which will not require such extra efforts with which we now described a logical scheme.

(13)
```
defunit
   concept man;
   concept woman;
   concept marriage(husband:man,wife:woman);
endunit;

dataunit
   man John;
   man Peter;
   marriage(John,Mary);
   marriage(Peter,Mary);
   woman Mary;
endunit
```

The bigamous ally represented here using (6) cannot be represented using (7) and the unambignity resulting therefrom. On the other hand, this is apt to the representation of a "hopeless love cycle"



(14)
```
defunit
    concept man(sweetheart:woman);
    concept woman(sweetheart:man);
endunit;

dataunit
    man John(Suzy);
    woman Suzy(Peter);
    man Peter(Mary);
    woman Mary(John);
endunit
```

which would not have been permitted by (6). From this example it is evident that proper care must be taken when choosing the system of concepts if an easily handled, safe and adequate representation of data is to be constructed. For this we have other tools to use, too, which we shall describe later on.

(15)
```
dataunit
    man John(Suzy);
    woman Suzy(John);
    man Peter(Mary);
    woman Mary(Peter);
endunit
```

In this example spouses have been defined in an unambiguous way (still referring to the definitions 7 and 14). The desciption is clear and simple, owing to the fact that no sequential interpretability is required.

1.3 Equivalence

A question of grave importance to the basic scheme is when to take two concepts or two data objects for identical. It has been decided with a stroke of the pen, that never. Some comments to this statement follow now.

1.3.1 Equivalence of concepts

Concepts always differ from each other. This is trivially seen unless their attribute lists are identical. Still, concepts serve as means to denote classes (i.e. to define qualities) with the corresponding attribute lists having but secondary importance.

Let e.g.

(16) concept man;
concept woman;

These types have but identical attributes (i.e. none), still the viewpoint to consider these concepts identical is not very yielding. (Yet another question is to define the concept "human" with the above ones being special cases of this in some sense which is detailed at 3.2.)

1.3.2 Equivalence of data

can be considered but in the case of the same type. Still the data are not the same even if all their attributes have the same values. It should be reminded here that these data objects have different names (which may be internal if not defined explicitly). E.g. in the case of

(17) concept worker(works at:dept,year of birth:integer);

the two data objects

(18) worker Ryse(administration,1950);
worker Wright(administration,1950);

are different (according to common sense) and yet have the same contents.

This uniform approach to the problem of equivalence is not the only possibility to adopt. E.g. if the sense a data object makes were dominated by the semantical contents rather than by a name (if any), it would be handsome to handle names as ordinary attributes (of the text type). This problem will, however, be approached in a different way see under the heading "functionality".

1.4 Conclusion

Two kinds of input units, namely definition and data ones have been discussed. For both we demanded to be closed, which property is formal and sufficient for correctness, thus avoiding all considerations with respect to domains.

Owing to this well-chosen formal property and the reference attitude taken the order in the definition/data sequences and the recursion-likeness of the constructions become insignificant.

The two types of units are connected by a homomorphism, the type coincidence.

2. Relations

2.1 The relational view

2.1.1 Representation of concepts in tables

By a relation we shall heuristically mean an empty table which will also be our heuristic interpretation of a concept. The name of the concept will identify the table, which has a fixed number of coloumns corresponding to an attribute each, which in turn identifies that coloumn. Rows in the table are filled in correspondingly to the valid data descriptions, i.e. they correspond to object instances of that concept. For an example consider

(19) concept user's guide(object:program,
environment:subsystem,catalogue no: integer);

which supposes "program" and "subsystem" to be also defined (but, owing to the reference attitude, possibly composite) concepts.

USER'S GUIDES:

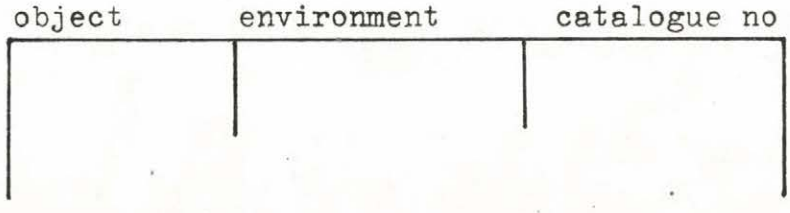


fig. 1.

In the corresponding table the elements to be put in must correspond in their types to those defined previously. In this case

a) the first coloumn should contain reference to a "program" object,

b) the 2 nd a reference to a "subsystem" object, and

c) the 3 rd an integer value.

In accordance with this,

(20) user's guide (payroll processing,finances,1713);

will result in

object	subsystem	catal no
reference to the program "payroll processing"	reference to the subsystem "finances"	1713

fig. 2.

2.1.2 Data names

As we have seen, if we use the formalism (10), that will result in the object instances' all having individual names which permit to refer to them directly. (This possibility is effectuated by an access mechanism by name which is part of the relation's representation.) For example

(21) user's guide BL guide(payroll, finances,1713);

user's guide BL revised guide(payroll, finances,2326);

results in

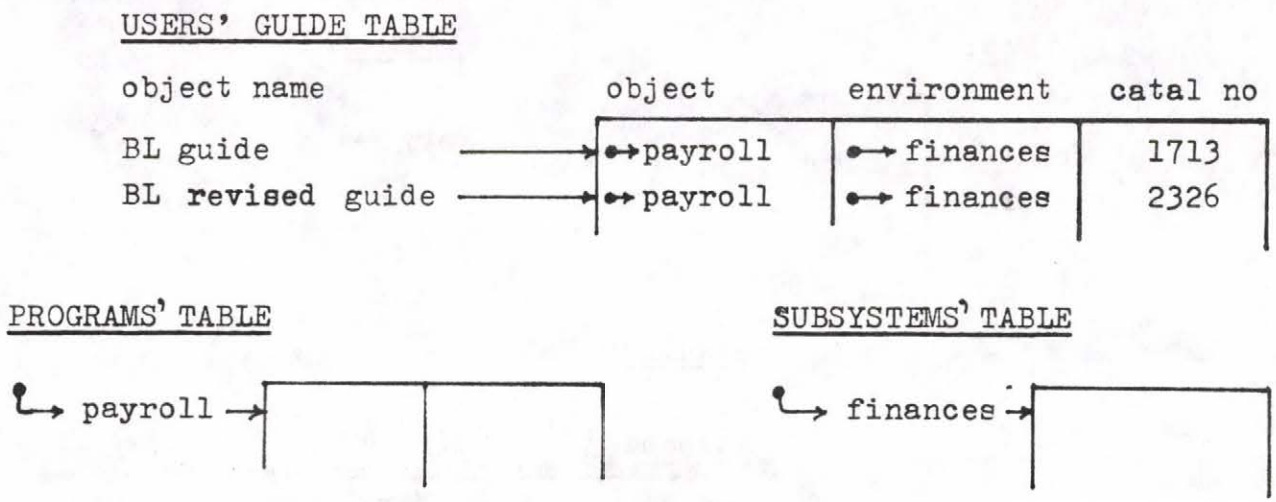


fig. 3.

However, it is important to know that in setting reference values names can be dispensed with. A comfortable way to do this is described in the user's language part. Items with no name are still "citizens with equal rights", as we remarked earlier.

2.1.3 Degenerated relations

Corresponding to the number of a concept's attributes, a table may well have zero or one colonums. This may occur strange at the first sight, but to handle such tables needs no special technique. When the number of attributes is zero objects (rows) in that table carry no information, still they exist as (to an extent abstract) instances and they can be

a) referred to, and

b) wearing a name.

Take for an example the bigamy in ex.(13).

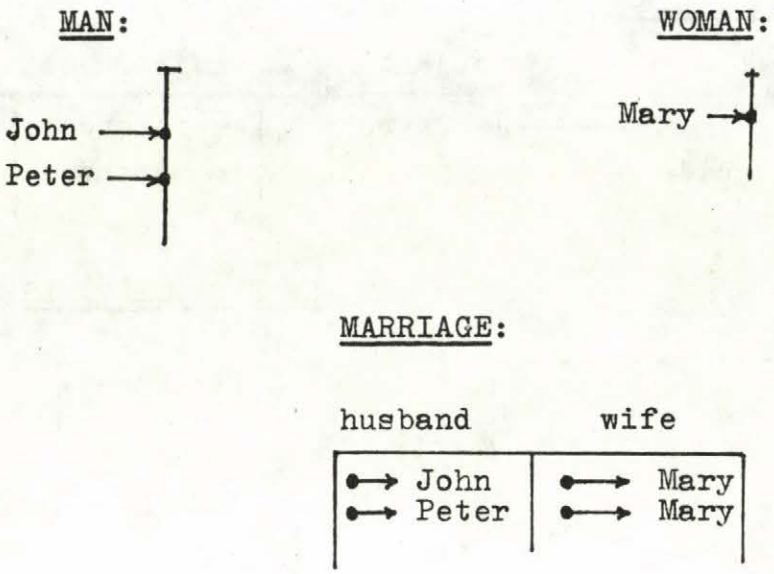


fig. 4.

Or, in the case of ex. (14) we get

fig. 5.

which yields, representing references with arrows

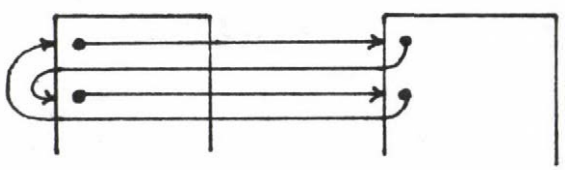


fig. 6.

This figure illustrates the reference scheme to give an exact model of our realistic relations (which are references itself) as opposed with the traditional relational approach which would

not take such information unless coded textually and provided with an indirect mechanism that works on texts for the retrieval.

2.2 Relation operations

As it has been indicated, the relational approach has a key role in the retrieval and parsing of information. Before going into detail in this, we briefly survey the operations used for this. We take as a base the usual relation calculus with the following modifications of the aspect:

a) Using reference type coloumns (attributes) relations among relations can be introduced without claim to deductability to atoms, which profoundly dissatisfies Codd's 1'st normal form. (Moreover, value type references can be omitted altogether in certain models)!

b) Sematical constraints (to be detailed later on) call forth the distingnished treatment of a certain type of projections.

To the concepts hitherto defined we introduced the corresponding basic relations of the system with their types being their concepts. Our basic idea is now the following: using operations on the relations we get new ones, to a fraction of which we shall not attach any type and of which we shall restrict the use. See [6].

2.2.1 "Descent" (Zoom)

This operation can be defined but on reference type relations. Accordingly, it has no corresponding operation in the classical theory of relations,(e.g. at Codd).

It has the following form:

(22) relation.selector

i.e. in a more general form

(23) relational expression.coloumn designator

where the colounm designator is either a cardinal number or a selector name (if the relation is a basic one). In addition, we suppose this colounm to be of reference type. (Take care not to confuse this formalism, with the "remote accessing" of objects to be introduced later.)

Descent means the following. First all but the specified rows are omitted from the relation, then the duplicate elements from this coloumn. The elements we have left in the coloumn are all

of the type determined by the coloumn, say T. Then every element left is substituted by its correspondent from T (as the element was a reference). (The relation thus obtained is a set theoretical subset of T.) The qualification of the new relation is by definition T.

Suppose e.g. we have the following tables:

USE:

process	data
P	A
P7	A3

DATA:

	owner	size
A	T	25
A2	T2	43
A3	T3	12
A4	T3	10

Fig. 7.

Then

(24) use.data i.e. use.2

means the following relation:

	owner	size
A	T	25
A3	T3	12

Fig. 8.

which is a partial relation of the relation "use" and has the type "data". In the next example we do this in a seemingly recursive way. Suppose we have the following relation.

MAN:

	father	wife
Stephen	John	Therese
John	Jack	Martha
Peter	John	Judith
Jack	"unknown"	Esther
Francis	Jack	Mary

Fig. 9.

Then

(25) man.father

results first in "disassembling" the first coloumn as

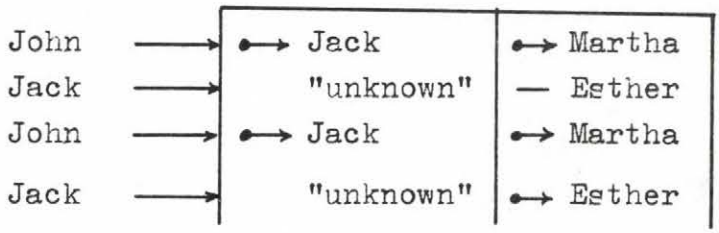


Fig. 10.

then with the omission of the identical lines in

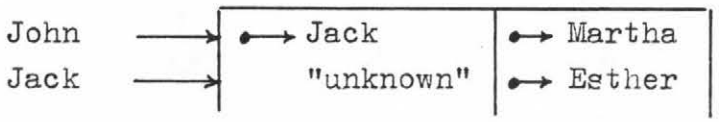


Fig. 11.

which is the set of those men in the original relation who are fathers. As this last relation denoted by

man.father

has the type "man" according to our rules, the expression

(26) man.father.father

is also valid and yields the relation table

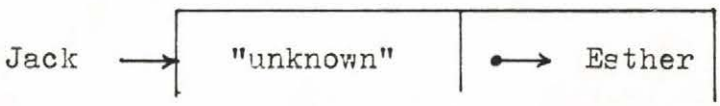


Fig. 12.

2.2.2 Selection (projection + permutation)

It means leaving from a relation some columns, then rearranging the columns into a given order. We note here that in Codd's model redundant rows are omitted at this point which we shall not do as identical rows in our case are not redundant. (see also 3.3.2).(We note here also, that, in the contrary, it was necessary to omit identical rows in the definition of the descent.)

Consider e.g. the relation of

(27) concept update(by:process,updated:data, using:data);

and the table of the "updated" and the "by" columns derived from it. This selection can be denoted as either

(28) (updated,by) update

or

(29) (2,1) update

the first permitted for the basic relations only. The new relation will have by definition no type.

2.2.3 Ascent

This is the inverse in some sense of the descent and used for technical purposes mainly. For any relation T

(30) [T]

the ascent from T is a relation of one column having type T and its elements pointing to the rows of table T.

E.g. for the relation

PROGRAM:

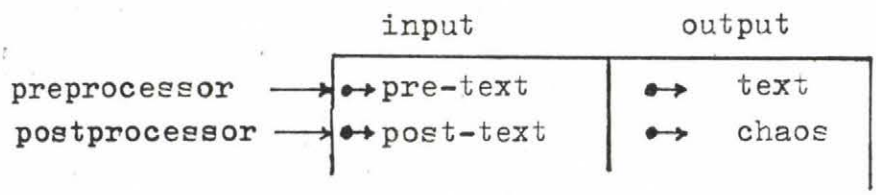


Fig. 13.

[program] means

[PROGRAM]:

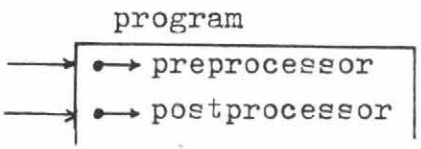


Fig. 14.

Obviously [T].1 always yields T.

2.2.4 Natural join

Let S and T be relations of at least one column and the last column of S have the same type as the first of T. Then

(31) S * T

means the natural join of the realations in the following sense. Having

S:

V	U
C	A
D	A
C	B

T:

	U	Z	T
H	A	E	H
I	B	F	J
J	A	F	J

Fig. 15.

S * T results in

V	U	Z	T
↦ C	↦ A	↦ E	↦ H
↦ C	↦ A	↦ F	↦ J
↦ D	↦ A	↦ E	↦ H
↦ D	↦ A	↦ E	↦ J
↦ C	↦ B	↦ F	↦ J

Fig. 16.

The joined relation will have no type unless one of its components is of one coloumn, when the type of the other is kept for the join (which is a set theoretical part of that one in this case). The case of two one-coloumn relations joined is left to the reader to consider.

Suppose now we have the following concepts:

(32) concept partof(part:element, of :group);

concept usedby (used:process, by:element);

Let now C be a relation of group type. Then

(33) partof* [C]

is a relation holding the rows with elements of groups in C and has type "partof".

(34) (partof*[C]).1

is the set of elements of groups in C and has type "element". At last, obviously

(35) (usedby*[(partof* [C]).1]).1

is the set of process using elements of parts of C. (We note here that these operations are denoted in the user language by a much more comfortable and foolproof formalism).

Consider now the example in 2.2.1. As we have seen

(36) man.father

is a relation of type "man" holding the men who were fathers in the table (i.e. John and Jack). Now

(37) ([man]*man.father).1

contains the rows of those man who had their fathers in the original table,(i.e. of Stephen, John, Peter and Francis).

2.2.5 Set theoretical operations

that are permitted are the union, the intersection and the (non symmetric) difference. Conditions for the executability of the operations are:

a) the types of the columns in the operands must pairwise correspond to each other;

b) the product of the operation will have a type iff the operands have and they are the same (which will be the type of the product, too).

2.3 Analysis

The analysis of the described information is facilitated by the query language, which includes two types of commands those of general and of standard types. The first group of them is performed using operations on the relations, the second uses special commands.

2.3.1 General query

The contents of any relation can be listed using the command

(38) list "relational expression";

in which "relational expression" stands for either the name of a basic relation or a relation derived from these using the permitted operations.

Here we allow one more possibility in building expressions: to fix a column by a given object (corresponding in type). We could not have dealt with this kind of operation in the part 2.2,the operands, now, are not pure relations (but date also occur in them); still this operation results in a relation.

On the list we get it will appear the name of (or the expression resulting in) the relation, the types of all the columns and the contents of the relation in the following form:

a) for value type data their values;
b) for reference type data with own names these names;
c) for unnamed objects internal identifiers (which permit tracing the references through the documentation thus obtained)

are given.

2.3.2 Example

Suppose we have the following description:

(39)

```
defunit
   concept man;
   concept woman;
   concept marriage(husband:man,wife:woman);
   concept cause;
   concept divorce (whose:marriage,because of:cause);

end;

dataunit

    man Romeo;
    woman Juliette
    marriage X1(Romeo,Juliette);
    cause family conflict;
    divorce X2(X1,family conflict);

endunit
```

Here X1 denotes a reference which can be established using the user language without naming it explicitly, so let us consider X1 as an internal name (i.e. a reference to an unnamed object). Then

```
list divorce;
```

induces the table

DIVORCE

name	whose	because of
X2	X1	family conflict

Fig. 17.

and

list marriage;

results in

MARRIAGE:

name	husband	wife
X1	Romeo	Juliette

Fig. 18.

2.3.3 Standardized query

Special types of relations (as binary relations between elements of the same set, trees, lattices etc.) make use of special document formats (as matrices, some sorts of graphic representations etc.). Moreover a demand must be taken into account to list the whole contents of the data base to a variable depth, to its documentation, and to generate reports about its momentary characteristics.

All these details are not parts of this concise description.

3. Meta relations

Up to now we get acquainted with the fundamentals of the logical scheme. This diverged from the usual relational viewpoint first in its reference attitude. Proper description of abstract systems needs even more, and more refined tools. Such will be detailed in this chapter and at the same time they will mark out the basic character of the SDLA.

3.1 Constraint

3.1.1 Example

Suppose we have the following concepts:

(40)
```
concept use(object:data,user:process);
concept generation(result:data,generator:process);
concept use to generate(used:data,
        result:data,producer:process);
```

Now if we have

(41)
```
use to generate (D1,D2,P);
```

we may in the possession of its semantical contents well wish

(42)
```
use(D1,P);
generation(D2,P);
```

both to hold. The validity of these constrained relations can be prescribed in the following ways:

3.1.2 Simple constraint

Without further explanation referring to (40) we permit the constraints of the form

(43)
```
constraint:
  use to generate(1,2,3) ⇒use (1,3);
constraint:
  use to generate(1,2,3) ⇒generation (2,3);
```

in which the numbers in brackets determine to which of the attributes the relation will be constrained.

This statement guarantees the automatic generation of objects of the right hand type whenever an object of the left hand type comes to existence in the system. This object will have no name.

3.1.3 General case

Constraints of the special type are needed in practically every applications. A more general approach is also possible, which permits constraints such as explained by

(44) constraint: "relation expression" ⇒ "relation";

too. E.g. if we have

(45) concept consume(used:data,user:process);
concept produce(by:process,result:data);
concept source(used:data,produced:data);

the constraint

(46) constraint:(consume * produce)(1,2,3)⇒ source(1,3);

can also be prescribed.

However, to permit this or not is disputable, due firstly to its demands in computing capacity. **It is not yet a decided question if such general kind constraints should be permitted.**

A secondary use of constraints is to ensure **efficient access to some** projections by special viewpoints by stating a constraint to the projection with the derived viewpoint at a proper position or introduced into the resulting table for the purpose.

3.2 Subtypes (Type refinement)

The idea is taken from the SIMULA 67 where it first appeared, becoming since an indispensable tool in making well arranged system descriptions. The (meta) relation

"type" - "subtype"

is antisymmetric and means the following:

(47) a) an object belonging to the subtype concept necessarily belongs to the type, too;

b) and possesses all the attributes of the type plus its own subtype-attributes (if any).

3.2.1 The subtype dilemma

It is fashionable to argue about further properties of the metarelation "type-subtype" to require. As the authors haven't finished it either, though they agree that it should be acyclic, we shall just consider the following possibilities:

a) Hierarchical (=tree) structure. This is clear and simple, safe and unambignuous to handle. Some phenomena are more difficult to represent using this one however.

b) Lattice structure; it has a great descriptive power and the drawback of computing demand with its complexity and ambiguity as consequences.

As it is known, SIMULA uses tree structure and provides with much user experience. On the other hand, data base planning tends to aspire handling lattices (even if not as the concepts' structure).

3.2.2 Hierarchic case

could use the following formalism:

(48) concept subtype is type (attributes);

which means the following:

(49) a) this subtype has the attributes given in (48) after the proper attributes of the type;

b) its objects' restrictions to the original non-subtype satisfy the relations the objects of the type do.

c) they satisfy the "generalized type coincidence rule" (see later, at 3.2.5)

It is worth noticing that a subtype definition differs from a type constraint in that it does not involve the generation of new data items, it is just a re-qualification on a new level.

3.2.3 Examples

Introduce the following concepts:

(50)
```
defunit
    concept file(location:device,bloksize:integer);
    concept printfile is file(pagesize:integer);
endunit
```

Then printfile has three attributes in the order "location", "block size", "page size" as in

(51)
```
printfile system output(printer-1,1280,136);
```

Consider now the following concepts:

(52)
```
defunit
    concept human;
    concept man is human;
    concept woman is human;
```

These on the one hand permit the introduction of further concepts based on the general concept "human" as e.g.

(53)
```
concept tax declaration(declarer:human,lies:integer);
```

at which sex is irrelevant (or we suppose so). On the other hand, subtypes can be used as

(54)
```
concept being married(bride:woman);
```

This structure permits a type checking that goes into the very details without too much work. So:

> Subtypes aim not to describe set theoretical classes, where an object of a general class is necessarily involved in some of its subclasses. It may well

represent something that we do not have mere knowledge of, for any of a number of reasons, as e.g. as a description of an abstacted level is needed.

3.2.4 Extreme types

One of them is

(55) concept universal;

which stands for the abstracted root of the tree structure of types and has no attributes. So all the concepts in the scheme will become its subconcepts as

(56) concept concept(attributes);

becomes

(57) concept concept is universal(attributes);

So we get a possibility e.g. to name an object with

(58) concept naming(object:universal,name:text);

where "object" may have any type according to the generalised type correspondence law 3.2.5, to follow .

If we permit a lattice structure of types, we may well meditate about

(59) concept "absolute special";

or "void" or "empty" which has in turn no subtypes, but is subtype of any type. So it is accepted as actual parameter for anything, this for the price of carrying all the attributes ever devised in the system. We may well, however, consider all these attributes to be undefined, and then simply disregard of them.

3.2.5 The general law of type coincidence

(60) If a referenced attribute has type T on definition level, then on the data description level exactly **T and all** its subtypes (transitively) are accepted.

So we can consider a subtype of T to be of type T as well; then all objects are universal as well, and an "absolute special" **object coincides to any type.**

3.2.6 Example

(61)
```
defunit
    concept IO device;
    concept input device is IO device;
    concept printer is output device;
    concept plotter is output device;
    concept opening(periphery:IO device);
    concept clear buffer(periphery:output device);
    concept set origo(periphery:plotter);
endunit
```

and then

(62)
```
IO device fourth channel;
output device system output;
printer matrix printer;
plotter calcomp;
```

Then the instructions

```
opening (fourth channel);
opening (matrix printer);
opening (calcomp);
```

are all acceptable, as "opening" may refer to any "IO device".

clear buffer (system output);
clear buffer (matrix printer);

are also acceptable, as "clear buffer" makes sense for output devices which "system output" and "matrix printer" are. On the other hand,

set origo (matrix printer);

set origo (system output);

are both inacceptable, as the statement doesn't make sense for a printer, and as the system output, though it may well be a plotter, is not defined to be one, so the second statement cannot be accepted either (by 3.2.5).

3.2.7 The empty object

According to our previous statements we have a standard "universal" object type. In a similar way, we introduce an (or more) absolutely special type object nil e.g. as

(63) absolutely special nil;

We agreed previously that actual attribute values may be unknown at the data description. Now we state this so, that as unknown data value the nil accord with the general type coincidence rule (3.2.5).

3.3 Integrity

By the integrity of the data base we mean those of its properties which are to be invariant during its use and specified in advance as such.

Integrity is preserved primarily by the selection of the input data. This means, that input which contradict integrity are rejected. So rejection of a given input may depend on the momentary contents of the data base.

The integrity of relational data bases is usually stated in the form of (functional and multivalued) dependencies. We need

more refined tools then these; the following have been considered from practical points of view:

3.3.1 Simple functionality

The specific kind of relations most in use is functions, i.e. relations where some components' value determines the rest. (The main consequence of using functions for us is not the use of access methods by keys, but the use of functional properties being fulfilled as integrity constraints.) Let e.g.

(64) concept concept(name:text,prefix:concept,rest:attr part);

Then the supplement

(65) function of name;

attached to the definition declares, that a name may be used but for one concept, or else the integrity constraint is contradicted and the second concept of that name will be refused by the data base.

3.3.2 Multivariate functionality

The general form of the supplemental specification that may declare a functionality is the following:

(66) function of "selectors enumerated";

where the rows of the relation are determined but by all of the selectors. When the "key" declared in the specification is all the selectors, the form

(67) function;

is also permitted.

As noticed at 1.3.2, a relation may always contain identical (**except** the object name) rows. This can be disallowed using the declaration function.

3.3.3 General form of functions

In the general case not only basic but also derived relations can be required to be funtions of some key. This general functionality can be declared as:

(68) integrity: relational expression
function of designation of columns;

Finally we remark, that the functionality formalism introduced here (i.e. to claim a set of attributes to be a key) is formally different from the usual concept of functional dependency, (though is its equal in power if the freedom to define and choose concepts is made use of).

3.3.4 Binary properties

are in use widely and many of them are useful to express integrity constraints. E.g. the relation

(69) concept origin(parent:human,child:human);

should obviously be antisymmetrical or else a misconception is to be taken account of. This justifies the constraint

(70) integrity: origin antisymmetric;

A constraint like this has the general form

(71) integrity: "binary rel expr" "property";

where "binary rel expr " is a binary expression of the basic relations with the introduced operations and "property" is one of the constraints e.g.

(72) antisymmetric
irreflexive
hierarchic
precedence
lattice

By "hierarchic" a tree structure is meant, "precedence" is one with a transitive closure that is a partial ordering, and "lattice" is used in the algebraic sense. Clearly

(73) integrity: origin precedence;

holds but

(74) integrity: origin hierarchic;

is not necessarily satisfied if not restricted to the male line. For a more complicated example take (45) and consider

(75) integrity:(1,3)(consume ≠ produce) antisymmetric;

which can be maintained on the ground that why represent somebody's consumption of his own products?(If you want the whole production system be acyclic, use "precedence" instead of "antisymmetric").

3.3.5 Set theoretical relations

as "equivalence" and "containing" can also express integrity constraints, as on (40)

(76) integrity:
(1,3)use for generation ⇒ generation;

does. The difference from (43) now is, that (43) forces the validity of (76) by introducing new instances if necessary, while (76) checks a set theoretical condition and it may refuse data items.

4. Dialogue

Up to now, we considered the logical structure of the informations in the system. Now we shall have a look at how to build such a delicate system.

4.1 The Principle of stepwise construction

The structure of the data base is constructed stepwise, in two disjoint phases: first the logical scheme is obtained in the metadialogue, then the data dialogue fills this scheme with data.

4.1.1 The logical scheme

is built in the metadialogue in steps like in this example. Suppose, first some very abstract and general concepts are worked out for the construction of information systems. These are described as

(77) defunit
endunit
] general concepts for information system design

Then other users may want to introduce special concepts which are genral in e.g. planning steel works' information systems and doing so they improve the system one step further. What they have to add is this:

(78) defunit ... endunit] special concepts for steel industry

At this stage the system can be preserved for future use, or one of, for example, our excellent colleagues Győry György (who translates this paper into English) may decide that he will also add something of his personality to the set of concepts. What they have to satisfy is this:

(79) | the contents of the defunit given the last together with those previously given must be closed but may not be closed in itself .

4.1.2 The data dialogue

is performed in similar steps, after each of which the contents of the data base can be preserved. Similarly to (79) now

(80) | the contents of the data unit given the last together with those previously given must be closed (but may not be closed in itself).

4.2 Process of dialogue revisited

4.2.1 Metadialogue

In the metadialogue

a) concepts,
b) constraints, and
c) integrity rules;

are given in arbitrary order, paying attention to (79) in each step like

(81)
```
defunit

    concept data;
    concept group is data;
    concept element is data;

    concept consist(contained:data,in:group);
    integrity:consist precedence;

    concept relation;
    concept associated data is data
                              (associated to:relation);
    integrity:associated data function of associated to ;

endunit;
```

first and then

```
defunit

    concept informative data is data;
    concept control data is data;

    concept process control(controlled:process,
                            by:control data);
    etc.
endunit
```

4.2.2 The data dialogue

comprises

a) input of data units which

   i) insert data, or

   ii) modify data; and

b) queries in

   i) standardized, or

   ii) general

form. Data units and queries must be separated, but their order is arbitrary, as

```
dataunit

endunit;
list ...
dataunit
etc.
```

4.3 Modification of data

For giving a comfortable way of data modification first the object expressions need to be introduced.

4.3.1 Object expressions

First we define elementary expressions.
An elementary expression is either

a) an ordinary value (with qualification integer, real or text) or

b) a data object name (with its type name as qualification).

For an example consider

(82)
```
defunit

    concept data(owner:process);
    concept process(owner:system);
    concept system(size:integer);

endunit

dataunit
    system S(1000);
    process P(S);
    process R(S);
    data D(P);
    data E(R);
    etc.
```

Now "D" is elementary expression qualified as "data", or "S" is elementary expression qualified as "system".

(83) **An object expression in general is of the form**

K.selector

whenever K is a reference type object expression and it posses the given "selector". The object expression will be qualified as its "selector" is.

(This notation is not to be confused with the similar one used for relation expressions.)

Taking (82) again, as an example of an object expression is

D.owner

which is qualified as "process" and has the reference value "P"; another object expression is

D.owner.owner.size

qualified as integer and with value 1000.

4.3.2 Assignment

makes possible the modification of existing data in the format

(84) K1 assign K2;

(85) where expressions K1, and K2 must have the same type. The value of K2 is assigned to the place determined by K1 as its consequence.

(This is performed in accordance with the principles of ALGOL 68, but we didn't feel it necessary to stress the reference attitude in the formalization.) Taking for our example (82) again,

E.owner assign D.owner;

results in the value of E.owner becoming the value of D.owner (which is P).

4.4 System dynamics

4.4.1 Extensions of concepts

when giving a data description to the system we always have taken the system of concepts for given and fixed. This restriction can be same what alleviated: though obviously no change is permissible that would injure the sense and integrity of the existing data (e.g. to cancel concepts or introduce new integrity constraints), changes that do not violate the correctness of the stored data may be allowed (e.g. to introduce new concepts or to refine existing ones).

Though it is feasible to make such changes accepted in the run-time data description, it is disputable, as at least

a) New concepts' introduction permitted increase the programmer's power and ease: he is not obliged to declare all his concepts for fixed at the start.

b) such changes are acts of grave responsibility and consequences which would allow the users to do dangerous things in the system.

4.4.2 Cancelling data

looks like

(86) cancel object expression;

As a result the referenced object

a) is removed from every relation qualifying it:
b) its name ceases to exists;
c) references to it become void.

Another form of cancelling is that by key and can be used but for relations with "function" integrity. In this case the form

(87) cancel concept name by key attribute values;

can be used with the attribute values corresponding to the selectors given at function respectively.

II. THE USER LANGUAGE

is the tool by means of which the user can interact the system and so his own data. The semantics of this will be defined by a mapping of it onto the logical scheme. The "internal language" described in the previous part could be used for this in principle as well but would be highly inconvenient (it would look somehow like treating data immediately in an intelligent kind of language like SIMULA). The user language has still further purposes than to ensure convenient communication; some important functions of the system are defined at the level of the user language and the mapping to the "internal language".

1. On the transformation of human language

In defining the user language our purpose is to construct one which conforms the rules of a spoken language and is representable by some simple data structures.

1.1 The relational attitude to language

Instead of trying to exhaust a great deal of possible mappings from sentences to data structures we shall restrict ourselves to some characteristic examples now.

1.1.1 Simple qualification

A lot of simple sentences can be represented by a simple qualification of their subject. E.g.

(88) "Mary Brown is blonde"

can be represented introducing the concept

(89) concept blonde;

by

(90) blonde Mary Brown,

or even

(91) "Gustavus drinks during worktime"

may be adequately represented under suitable circumstances by

(92) concept drinks during worktime;
drinks during worktime Gustavus;

1.1.2 Associations represented as attributes

are quite possible if the association is a function in one direction or the other. Another possible way is (without the above restriction) to treat the association as a self-contained concept. Proper decision may or may not be easy and always depends on circumstances like the environment and the use of the system.

Some examples for associations as attributes are

(93) "Peter is John's father"

may be represented as

(94) concept man (father:man);
man John(Peter);

in which representation the concept of fatherhood is not explicitly used. (91) will be possibly coded as

(95) concept co-worker(direction of main activity, during);
co-worker Gustavus(drinking, worktime);

1.1.3 Association versus concepts

Associations as concepts can be reresented without restrictions, but generally in several different ways from which to choose the proper one may be essential. E.g.

(96) "Thomas courts to Martha in order to get acquainted with her friend"

may have the representation

(97) concept courting on purpose(man,woman,purpose);
courting on purpose(Thomas,Martha,to get acquainted with friend);

as adequate or in a different case the adequate one may be

(98) concept courting (man, woman);
concept purpose of courting(courting,purpose);
courting U1(Thomas,Martha);
purpose of courting(U1, to get acquainted with friend);

Which is adequate and which not, depends on the environment and our purpose.

1.2 The equivalence problem

The same statement can usually put from different viewpoints, like

(99) "work causes fatigue" equals
"fatigue is the result of work",

or

(100) "A uses B" equals
"B is used by A".

The equivalence problem now is to map these sentences onto the same logical pattern, i.e. to find them equivalent in parsing. The problem will be solved by using an approach which permits different viewpoints (as cause-result, or uses - used by) to be considered. (see at II.2.1).

1.3 The context problem

In colloquial speech it may **not** be **decidable of a part** which part of the sentence it refers to at least not by simple parsing. What differs colloquial speech from our purposes at this point is that it doesn't matter there. E.g. when parsing

(101) "fist as Paul gave me the hammer the head fell off."

automatically you cannot be quite sure if it was the head of Paul or mine. The problem is encountered also in the case of attributes. It is somewhat like the problem of a multiple "inspect" in SIMULA which is well defined but has no equivalent in human languages and so will lead to catastrophic misinterpretations.

So we need an unambignous system of contexts which is demonstrated so as to be kept in mind. This is aimed by the subordination and juxtaposition structures introduced in II.2.2

2. Means of the user language

2.1 Relative forms

> In our formal language each sentence is put from a definite viewpoint of which we have as many as concepts plus one. Every attribute can be regarded from the viewpoint of any of its attributes.

Consider the following example:

(102) concept use to derive(used:data,user:process,derived:data);

It would come in handy if when speaking of a concrete "data" object we could use statements like

(103) used by P to derive D2;
derived by P using D3;

and speaking of P we could say

(104) uses D1 to derive D2;

This is made possible by permitting to join formally different equivalents of statements to the definitions as e.g.

(105) concept use to derive(used:data,user:process,derived:data);
form used: used by user to derive derived;
form derived: derived by user using used;
etc.

The general form of this is

(106) concept concept name(attributes);
form selector-i: other selectors embedded into text;

The above form declares by an embedding text a description form that is from the viewpoint of the i-th selector and will represent the concept given by the "concept name".

2.2 Subordination and juxtaposition

2.2.1 The general viewpoint

In (102) we saw the relative forms of a statement related to its attributes. There is one more of them, which we shall call the general (or absolute) viewpoint and which has been given as the first definition of the concept (in Chapter I.). E.g. in the case of (102) the embedding text corresponding to the absolute viewpoint would be

(107) use to derive , , .

The absolute viewpoint is the only possible one if there are no attributes .

2.2.2 Absolute statement

means a statement in the general (=absolute) form. Its structure in the user language is defined as

(108) concept name underlined, possibly an object name, attribute list in brackets;

That is, if

(109) concept process;

is a concept, then

(110) process;
process P;

are absolute sentences. Or, in the case of (102)

(111) use to derive (D1,P,D2);
use to derive usage1(D1,P,D2);

are absolute sentences which are mapped to the logical scheme by the identity .

2.2.3 The viewpoint rule

a) after each statement (that may be absolute or relative) its viewpoint becomes valid.

b) Each statement with a valid viewpoint is allowed to be stated.

(i.e. the valid viewpoints form a stack with the "absolute" viewpoint at the bottom.)

c) At the parsing of a statement the highest viewpoint is sought that allows the parsing as the valid viewpoint of the statement.

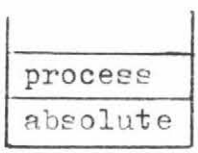
i) This with the others below keep valid,

ii) the rest above it is devalidated.

Example: let

(112) process P;

be parsed with the absolute viewpoint as the only valid one. This results in the "process" viewpoint becoming valid, i.e. in the stack



"viewpoint-stack"

fig. 19.

Parse now

(113) process P;
uses D to derive E;

the result in the valid viewpoints is

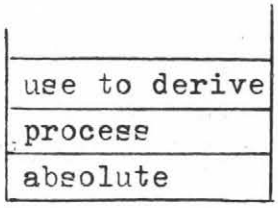


fig. 20.

Now we can put statements from any of the 3 viewpoints. Going on with an absolute statement, the stack becomes

(114) process P;
uses D to derive E;
data F;

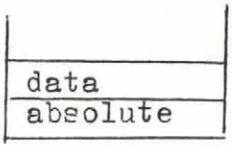


fig. 21.

etc.

(We remark that positioning in the input text lines allows the user an easy parsing of viewpoints, and so his text will be automatically rearranged to this form. This also faciliates easy debugging.)

2.2.4 Relative statements

can be given in the form

(115) possibly object name with a colon, attributes in embedding text;

with the attributes given by expressions as in 4.3.1. (Should an attribute be an object name, its qualification may follow it for better text readability.) For an example consider again

(116) process P;
uses D to derive E;

which are mapped into the logical scheme as

(117) process P;
use to derive(D,P,E);

Now if you want to name the second separately, you should use the form

(118) process P;
U: uses D to derive E;

Finally the form "uses data D to derive data E" is also allowed.

2.3 Technical tools

in this part are intended to enhance the user's comfort.

2.3.1 Type as selector name

is accepted if the attributes are all of different types and so to invent one more name for the selector is superfluous. So e.g. having defined man and woman beforehand and using

(119) concept marriage(man,woman);

will be mapped into the logical scheme as

(120) concept marriage(man:man,woman:woman);

2.3.2 Enumeration

is also permitted, saving much coding work to the user, as e.g.

(121) process X;
uses data D1;
uses data D2;

can be put in the form

(122) process X;
uses data D1,D2;

However, proper care must be taken of the fact that (122) in fact generates two "use" type objects at the same time and statements that follow and do not devaluate the viewpoint "use" will be interpreted as to hold for both e.g. as

(123) process X;
uses data D1, D2;
to maintain relation R;

2.3.3 Macro forms

We give but a notice to the fact that macro forms can be made good use of giving a structured descriptions of similar concept types, like by the macro

(124) macro:
concept subpart(a-concept,a-concept);
for a-concept=process,activity,group;

(We shall not detail this question in this paper.)

2.3.4 Compound relative forms

In cases when a simple constraint is declared for a concept, like at

(125) constraint;
use to derive(1,2,3) ⇒derivation(2,3);

the claim to create the left hand object from the right hand one in this example by the addition of the missing attribute is quite plausible.

This would be fulfilled by permitting relative forms that originate not from the viewpoints of the original selector but from a concept belonging to a set of attributes as e.g.

(126) form derivation: using 1;

Yet the following are to be considered:

a) the mutual correspondences between the attributes have to have been declared explicitely or implicitely somehow.

b) The constraint itself is not to be executed once more.

In order to allow this an alternative form to declare simple constraints in the user language (equivalent in its meaning with (44)) is to be considered.

(127) concept use to derive(used:data,process,derived:data);
implies derivation(process,derived);

The "implies" part is set between the heading of the concept declaration and the enumeration of forms if the declaration of the composite relative form is correct. That is, the whole example should look like this:

(128) concept derivation(process,data);
concept use to derive(used:data,process, derived:data);
implies derivation(process,derivation);
form derivation: using used;

which, according to the above statements, could be used as

(129) derivation(P,D);
using D2;

or even as

(130) data D;
derived by process P;
using data D2;

2.3.5 Embedding absolute sentences

As we have seen, absolute sentences are formally somewhat different from relative ones. For the sake of a uniform description form we shall permit the absolute sentence to be given in an embedding text. The use of this is the same as in the case of the relative forms. To give an example let

(131) concept use to derive(used:data,process,derived:data);
form absolute: process process uses used to derive derived;

Then

(132) U: process P uses D1 to derive D2;

is equivalent with the absolute sentence

(133) use to derive U(P,D1,D2);

and with the same data of the logical scheme .

2.3.6 Relative form as operation

Consider again the concept

(134) concept use to derive(data,process,derived:data);
form process: uses data to derive derived;

Now, as we know already, the relative statement

(135) uses D1 to derive D2;

can be stated with respect to "process" and (like every other statement) it will result in exactly one new data item, (this in our case of the type "use to derive").

For practical purposes related to the user language the same sentence will be permitted in general queries as an operation to establish a relation; its meaning will be the set of procedures that "result" in D2 "using" D1 similarly as in mapping oriented relational database sublanguages .

Notice that this can be denoted using our original operations but in the less selfexplanatory way as

(136) use to derive (D1,,D2).2

In general, substitution into a relative form (taken for an operation) is regulated like restriction by means of an object: any object or relation expression being coincident in type is acceptable. In its content the substitution results in the join with the ascents of the substituted objects and then determining the viewpoint of the form. Its importance lies in its self-explanatory quality for the users as opposed with its compactness of the abstract contents.

Consider for the second example that of I. 2.2.3:

(137) concept use (process,data);
form data: uses data;

concept subpart(data,of:data);
form data: part of of;

Then the set of processes using subparts of a given data can be determined in the original formalism as

(138) (use*⌈subpart(,C).1⌉).1

and using substitution into the relative form as

(139) uses(part of C)

2.4 Alternative of open description

Up to now we expected both the definition and data units to result in closed descriptions if executed consequtively. This is absolutely necessary for the definition part but is not in the data, as undefined values can also be used in the data base. To obtain closedness artificially the following algorithm may be used:

Parsing an (otherwise consistent) description when an undefined object name is found, a new object is generated to it of the possibly most general type (determined by the context) and with undefined attribute values. When it is defined in a later step, its type will possibly be specialized and the attribute values may be filled in.

This algorithm if used would involve the dangers of

a) not having an unambignous declaration of the objects (nor such a list of the declarations).

b) generating a number of guerillas among the objects by a simple slip of the pen.

So if the algorithm is put to use, warnings must under all circumstances be produced of the automatically created objects.

2.5 Some practical problems

2.5.1 Comments

One of the several possibilities for making comments is to introduce the concept

(140) concept comment(universal,text);
form universal: comment text;

with which to any concept comments can be associated. Comments which - in contrast to the above kind - are not stored in the data base are also feasible but not discussed here.

2.5.2 Synonyms

One may easily come to the idea that the handling of the synonyms (abbreviated object names) can be handled like comments. This is just not right, as the object names are just not text attributes and are handled by a suitable internal mechanism (be it a hashing one or something else). The practical solution seems to use the abbreviations as names and to attach the (longer) explanatory terms to them as attributes of type text (which is now justified). A way for this is to give the concept general a text type attribute for detailed explanations as

(141) concept general(explanatory name:text);

of which all our concepts will be subtypes.

2.5.3 Similarity of concepts

Some similar related concepts may well be desirable to be introduced for a number of concepts, as e.g. in

(142) concept subpart(data,of:data);
concept subpart(process,of:process);

This is, however, incorrect, as it doesn't suit the unicity of the concepts, and as the subpart relations in these examples are different.

Still we may want to use the same name for two related concepts (as in colloquial speech) when the different cases can be reduced to the same ancestor concept. This is not the case now, as we don't want to introduce possible process subparts of data and vice versa.

The solution to our case lies in the use of the same relative forms (which is of course allowed) such as:

(143) concept data subpart(data,of:data);
form data: part of of;

concept process subpart(process,of:process);
form process: part of of;

2.5 An example for the use of devices

Suppose we want to store the following description:

(144)
```
process X;
    belongs to system component S;
    uses data D1, D2;
        to derive entity E;
            under condition C;
    uses data D3;
        to update information I;
    uses data D4,D5;
    produces data D6;
data D7;
    is used by process Y;
    produced by process Z;
    etc.
```

a) supposing that parts in the text that are (from the context) obviously equivalent should refer to the same concepts and

b) the subordinations and juxtapositions should be in accord with the positioning of the text.

This can be accomplished by using the following definitions:

(145)
```
concept system component;
concept process;
concept data;
concept entity;
concept condition;
concept information;

concept belong(process,system component);
form process: belongs to system component;

concept use(process,data);
form data: is used by process;

concept produce(process,data);
form process: produces data;
form data: produced by process;
```

```
concept purpose(use);

concept derivation is purpose(entity);
form use: to derive entity;

concept update is purpose(information);
form use: to derive information;

concept derivation condition(derivation,condition);
form derivation: uder condition;
```

and transforming the text into

(146)
```
process X;
system component S;
data D1;
data D2;
data D3;
data D4;
data D5;
data D6;
entity E;
condition C;
information I;

belong(X,S);

use A1(X,D1);
use A2(X,D2);

derivation A3(A1,E);
derivation A4(A2,E);

derivation condition(A3,C);
derivation condition(A4,C);

use A5(X,D3);
update(A5,I);

use(X,D4);
use(X,D5);

produce(X,D6);

data D7;
process Y;
process Z;
```

use(Y,D7);
produce(Z,D7);

(Identifiers in this text beginning with the letter A are "internal" ones that were not in the original text.)

To give the comp de grace, we give the relation tables that are induced by the coded text:

PROCESS:

X
Y
Z

DATA:

D1
D2
D3
D4
D5
D6
D7

ENTITY:

E

CONDITION:

C

INFORMATION:

I

SYSTEM COMPONENT:

S

USE:

A1	X	D1
A2	X	D2
A5	X	D3
	X	D4
	X	D5
	Y	D7

BELONG:

X	S

DERIVATION:

A3	A1	E
A4	A2	E

DERIVATION CONDITION:

A3	C
A4	C

UPDATE:

A5	I

PRODUCE:

X	D6
Z	D7

fig. 22.

3. Some typical application directions

3.1 Causal nets

are structurally simple and still mathematically highly interesting structures. We shall illustrate now with them some typical approaches to representing a given structure.

After some simplifications, a causal net is a directed bipartite graph with one class of the edges having at most one "input" and at most one "output" vertex. It can be interpreted as the data and subprocesses in a bigger process, vertices indicating the use and derivation of data by processes, at most one of each to every data item. A simple example would be

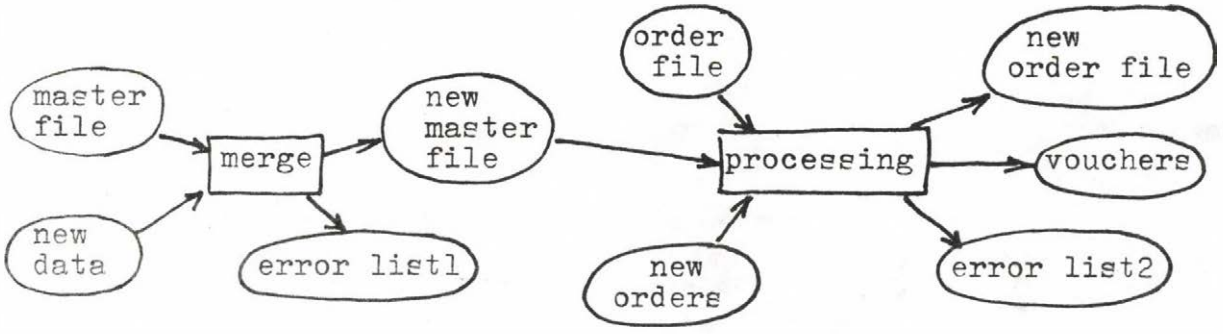


fig. 23.

Consider now some possibilities for the description of such structures. One way to do this is to interpret edges of the above graph as atomic data and vertices as links between them as

(147)
```
concept data;
concept procedure;
concept consume(procedure,data);
   function of data;
   form procedure: consumes data;
concept produce(procedure,data);
   function of data;
   form procedure: produces data;
```

which results in the following description (or in an equivalent of it):

(148) data master file, new data, ... etc. ;

procedure merge;
consumes master file, new data;
produces new master file,error list1;

procedure processing;
consumes new master file, new orders, order file ;
produces vouchers, error list2, new order file;

Another way for the description is to rely on the fact that links to data are unique and so can be handled as attributes. This choice will have influence mainly on the retrieval methods. In this case with

(149) concept procedure;
concept data(producer:procedure,consumer:procedure);

we need the function integrity in (147) no more as it is a consequence of the way of storage. The description using absolute sentences will now look like this:

(150) procedure merge, processing;

data master file(,merge);
data new data(,merge);

data new master file(merge,processing);
data error list1(merge,);

etc.

One more way to perform the same description task is to use

(151) concept data(producer:procedure,consumer:procedure);
form absolute: produced by producer used by consumer;

for writing

(152) master file: produced by nil used by merge;
new data: produced by nil used by merge;
new master file: produced by merge used by processing;

etc.

3.2 SADT

In the case of the SADT [4] model as well, we have several ways to accomplish a structure for both the description language and the data structure. Next we shall give the starting point for one of them. (We shall not deal here with its several possibilities at hand to create sementical integrities and constraints for the lengthyness it would require.) Consider now

(153)
```
concept arrow;
concept box;
comment "arrow" is meant in a general sense as a directed
                                                    network;

concept actionbox is box;
concept actionarrow is arrow;

concept databox is box;
concept dataarrow is arrow;

concept subaction(part:actionbox,of:actionbox);
   function;
   form absolute: subactions of of are part;
   form of: subaction are part;

concept input(arrow,box);
concept output(box,arrow);

concept use is input;
   function;
   form box: uses arrow;

concept control is input;
   function;
   form box: controlled by arrow;

concept mechanism is input;
   function;
   form box: mechanism arrow;

concept generation is output;
   function;
   form box: generates arrow;

etc.
```

which can be used for the slightly simplified description of the example at [7], p. 69:

(154)
```
allocation simulator: action;
   subactions are control, handler, printing;

control: action;
   controlled by handler response;
   generates handling request, print request,
             simulation results;

handler: action;
   controlled by handling request;
   generates handler response, area allocated;
   mechanism allocation algorithm;

printing: action;
   controlled by print request;
   generates allocation structure;
   uses area allocated;

subaction of control are clockhandling, block sizeing,
                         registration, deallocation,
                         condition handling;

clockhandling: action;
   controlled by synchron impulse;
   generates time, virtual time;

etc.
```

3.3 ISDOS

Though it is possible to define the PSL language in its original form as in [3], but we would not do it now. We shall but give a simplified and slightly modified approach to some of its concepts here as

(155)
```
concept information;

concept logical information is information;
concept data is information;

concept set is logical information;
concept entity is logical information;
```

```
concept input is entity;
concept output is entity;

concept group is data;
concept element is data;

concept relation(e1:entity,e2:entity);
form e1: is related to e2;

concept association is relation(associated:data);
form e1: is related to e2;

concept association is relation associated:data ;

concept process;

concept use(process,information);
   form process: uses information;

concept purpose(use, information);

concept pudate is purpose;
   form use: to update information;

concept derivation is purpose;
   form use: to derive information;

etc.
```

A description based on the above approach might look like this:

(156)
```
process X;
   uses group G, input I;
      to update set S;
      to derive entity E;

input I1;
   is related to output P;
      associating data D;

etc.
```

3.4 Data-flow like structures

As the reader will have known it, data flow-like structures are of good use in the traditional architectures of Neumann as well, among others for giving the logical schemes of real-time systems. (See e.g. the system worked out at the Dept. of Continuous Processes at the Computer and Automation Institute Hungarian Academy of Science [8]).

The following ideas can be suggested for such descriptions:

(157)
```
concept data;

concept integer is data;
concept bit is data;

.
.
.
etc.

concept array is data(size:integer);
   comment attention:not underlined integer!;
   form absolute: is of size integer;

concept part(data,of:data);
   form of: consists of data;

concept condition;
concept operation;

concept pre(condition,operation);
   form operation: precondition condition;

concept post(condition,operation);
   form operation: postcondition condition;

concept reception(data,operation);
   form operation data received data;

concept production(data,operation);
   form operation: data produced data;

etc.
```

These can be used e.g. in the following way:

(158)
```
operation X;
   precondition P1, P2;
   postcondition S1, S2, S3;
   data received D1;
   data produced D2, D3;

data D1;
   consists of integer I, array A, bit B;

array A;
   is of size I;
```

APPENDIX

1. Syntax of the logical scheme

here we shall give the most important parts of the language defining the logical scheme, this rather heuristically and not properly formalized for the easier understanding.

logical scheme language=definition part, data part.

1.1 Definition part

definition part=nonempty sequence of def units.

def unit=defunit,declarations separated by semicolons, endunit.

declaration=concept|detached integrity|detached constraints.

1.1.1 Concept

concept=concept,concept name, is, name of main concept, attribute part, constraints and or integrity possibly.

attribute part=void|attribute definitions separated by commas in parentheses possibly.

attribute definition=selector,colon,type.

1.1.2 Type

type=value type|ref type.

value type=integer|real|text.

ref type=declared concept name|universal.

1.1.3 Direct constraint and integrity

constraint=imply,name of another concept,some selectors of the first concept separated by commas in parenteses.

integrity=function,list of selectors possibly.

1.1.4 Relation expression

relexp=concept name|descent|ascent|selection|join|
set operations.

descent=relexp,dot,coloumn designator.

ascent=relexp in *square brackets.*

selection=coloumn identifications separated by commas in parentheses, relexp.

join=relexp,asterisk,relexp.

set operation=relexp,connective,relexp.

connective= $\cup$ | $\cap$ | $\setminus$.

1.1.5 Constraint and integrity

self-contained constraint=constraint:,relational expression, column identifiers in parentheses, arrow, concept name, column substituted.

self-contained integrity=integrity:,integrity description.

interity description=dependence|binary property|set property.

dependence=relational expression,function of, column identifiers separated by commas possibly.

binary property=binary relation expression, property.

property=antisymmetric|irreflexive|identity|hierarchic|precedence|lattice.

set property=relation expression, connective relation expression.

connective

1.2 Data part

data part=sequence of statements.

statement=data unit|modification|query.

1.2.1 Data unit

data unit=dataunit,sequence of data statements separated by semicolons, endunit.

data statement=concept name underlined,object name possibly, sequence of object expressions separated by commas in parentheses possibly.

1.2.2 Object expression

object expression=elementary expression|object expression, dot,selector.

elementary expression=object name|constant.

constant=nil|value.

1.2.3 Modification

modification=assignment|deletion.

assignment=object expression,assign,object expression.

deletion=cancel,object expression.

1.2.4 Query

query=general query|standard query.

general query=list, expression.

expression=relation expression|object expressions substituted into relation expression.

2. User language

Its main alterations from the logical scheme language are:

2.1 Definition part

user concept=concept possibly without is part, relative forms separated by semicolons possibly.

relative form=form,selector name,colon,selectors embedded into underlined text.

2.2 Data part

user data statement=data statement|embedded statement.

embedded statement=object name followed by colon possibly, object statement in underlined embedding text.

user relation expression=relation expression|embedded statement.

References:

1. Holbaek,H.E.,Handlykken,P.,Nygaard,K. System description and the DELTA language. NCC, 1975. Delta Proj.Rep.4.

2. Dahl,O.J.,Myhrhaug,B.,Nygaard,K. SIMULA 67 Common Base Language. NCC 1970. S-22.

3. Hershey,E.A.,Teichroew,D.,Berg,D.L.R.,Winter,E.W.,Bastarache, M.J. Problem Statement Language. ISDOS WP.68. 1974.

4. Ross,D.T. Structured analysis: a language for communicating ideas. IEEE SE 3,1,1977.

5. Kangassalo,H. Logical and semantical properties of environmental situation for stating information requirements. Univ. of Tampere, Dept. Math. Rep. A33. apr. 1979.

6. Knuth,E.,Ronyai,L. Closed reference schemes. MTA SZTAKI Working Paper II 7, jun. 1979.

7. Bartha,J.,Farkas,Zs.,Garami,P.,Keresztély,Zs.,Koppány,L., Kosztolányi,Z.,Németi,T.,Sántáné-Toth,E.,Simor,G. An overview of up-to-date system design concepts (in Hungarian).

8. Almásy,G.,Huppert,A.,Sztanó,T. Private communication.

9. Bernus,P.,Hatvany,J. Computer aids to the design of integrated manufacturing systems.

A **TANULMÁNYOK** sorozatban 1979-ben megjelentek:

88/1979 Renner G. - Gaál B. - Hermann Gy. - Horváth L. - Várady T.: Szoborszerü felületek tervezése és megmunkálása

89/1979 Ruda Mihály: A SIS77 statisztikai információs rendszer /a felhasznált számitástechnikai eszközök, a rendszer szerkezete és programjai/

90/1979 Bányász Cs. - Keviczky L.: Optimum Insensitivity of the Linear-continuous Transformation

91/1979 Téli iskola /Szentendre/

92/1979 Bolla M. - Csáki P. - Fischer J. - Herodek S. - Hoffman Gy. - Kutas T. - Telegdi L. - Wittmann I.: A balatoni ökoszisztéma modellezése

93/1979 Andor László: Kisgépes adatbázis kezelő rendszer

94/1979 Gertler János: Egy statisztikus szürési eljárás számitógépes folyamatirányitásához

95/1979 Báthory M. - Galló V. - Kovács E. - Mérő L. - Siegler A. - Vajta L.: Festőrobot vezérlésére alkalmas alafelsimerési berendezés

96/1979 Mérő László: Konturkeresés zajos digitalizát képekben

97/1979 Pásztorné - Matavovszky T.: Boole-függvény kezelőrendszer

98/1979 Kecskés Zsuzsa: Három dimenziós tárgyak drótvázának ábrázolása vonalrajzoló grafikus berendezésekkel

1980-ban jelent meg:

99/1980 Dokladü szimpoziumov
Szerkesztő: Ivics József

100/1980 IV. Visegrádi Operációs rendszerek elmélete
Téli Iskola

101/1980 Gerencsér László - Hangos Katalin:
Diszkrét lineáris sztochasztikus rendszerek
önhangoló szabályozása.

102/1980 Pásztorné Varga Katalin:
Rekurziv eljárás

103/1980 Gerencsér P. - Szász P. - Zilahi F. - Marton Zs.:
Robotmegfogók adaptivitása I.

104/1980 Knuth Előd - Radó Péter - Tóth Árpád:
Az SDLA előzetes ismertetése