# DATA RELIABILITY

A tutorial paper

C.A.R. Hoare

Department of Computer Science,
The Queen's University of Belfast.

Draft October 1974

---

Summary. This paper surveys the problems of achieving data reliability, and finds them more severe than those of program reliability. It then outlines some of the conceptual and methodological tools which are available for the solution of these problems, including the concept of type, direct product, union, sequence, recursion, and mapping. It touches on the topdown design of data and programs, and argues that references or pointers are to be avoided. It concludes with an annotated bibliography for further reading.

---

To be delivered at the
INTERNATIONAL CONFERENCE ON RELIABLE SOFTWARE,
Los Angeles, April 21-23, 1975.

# Data Reliability

## 1. Flow charts and data diagrams.

The most widely accepted and practised method for the design and documentation of computer programs is the flow chart (Fig.1). But more recently, the disadvantages of flow charts have become apparent. Here are some of them:

(1) They use too much paper: as soon as they overflow a few pages, the extra page turning and cross references form a significant barrier to understanding.

(2) They cannot conveniently be input to a computer, nor output from it.

(3) They do not enable you to understand the whole in terms of its parts, and so they become intellectually unmanageable when applied to large problems.

(4) The slightest fault in an arrow or box has unpredictable and global consequences on the whole program.

(5) These problems are greatly magnified if the structure is changed during program execution, for example, by assigned go to's in FORTRAN, or ALTERs in COBOL.

(6) The physical realisation of an arrow by a jump is surprisingly expensive on modern machines, with cache stores, instruction pipelines and virtual memory.

It is also accepted practice to use diagrams in the design of data (Fig.2). However, these diagrams seem to suffer all the disadvantages of flow charts,but to an even greater degree. For example:

(7) Change in the structure at run-time is now the rule rather than the exception.

(8) Physical realisation of an arrow by a reference or pointer is expensive in storage as well as in time.

(9) The diagram, instead of being an actual representation of the whole program, is only an imagined example of only part of the actual structure at some instant in time.

A simple example with a picture is of great advantage in confirming

our understanding of a complex problem; but as the sole means of developing and communicating such an understanding it is wholly inadequate.

2. Data Reliability.

The problems of program reliability are notorious; but reliability problems which arise with long term data storage are even more severe; and some of the reasons are as follows:

(1) Programs expressed in a suitable high level language can be analysed rigorously by a compiler so that a running program is known to be meaningful, even if it does not do what the programmer wanted. Input data is nothing but an unstructured stream of characters, cards, or bits; it may be meaningless, and even if meaningful, the information conveyed may be false.

(2) If a program behaves incorrectly on a particular run, this does not affect other programs, or even subsequent runs of the same program. However, if a bug is introduced on one run into a data base, it remains there, and can actually propagate itself as later correct programs operate on the data.

(3) If the hardware fails in the middle of a run of a program, it can safely be run again from the beginning, but the same fault in the midst of updating a data base may leave it in a partially or wholly unusable condition.

(4) From bitter experience, users can learn to avoid the bugs in unreliable software. But as soon as bugs appear in a data bank, its users lose confidence, and will withdraw their data for private keeping. Now they no longer have any motive to keep their banked data up-to-date. A run on a data bank is as catastrophic as a run on an ordinary bank which deals merely in money.

This theoretical analysis shows that in the design of data we have been using weaker design methods to tackle problems more serious than those of the design of programs. Practical experience in the design of data bases seems to confirm the theoretical analysis. This paper surveys some of the improved methods for data design and description which are now becoming available, and points out the analogies with some of the improved methods of design and description which are being applied to programs, and which have been packaged and marketed under the brand name "structured programming".

Programmers benefit from these methods largely because they reject flow charts, and deliberately confine themselves to a range of structuring disciplines with the following desirable properties:

(1) They are few in number.
(2) They are logically simple.
(3) They are amenable to simple proof techniques.
(4) They can be applied on a large scale or on a small.
(5) They assist in topdown design or in bottom-up.
(6) They are easy to implement.
(7) The easy implementation is also efficient.

Structuring principles for data should have the same properties.

## 3. Type.

In order to develop the analogy between program and data structuring methods, we must introduce the concept of type. A type determines a class of values, which may be stored in a variable (declared to be of that type), passed as a parameter (specified as of that type), or given as the result of an expression or function (of that type). The primitive types of a programming language, such as the integer of ALGOL 60 and FLOAT of FORTRAN, are obvious examples. But the concept of a type can be usefully generalised to cover structured data, whose pattern can be specified by the programmer himself by means of a type declaration.

A program or procedure may also be regarded as determining a certain class, namely the class of all computations which may result from a particular run of the program, or call of the procedure. All these computations will be similar in their overall structure, even though the exact size and values involved will vary from computation to computation. In the case of a well-structured program, the overall pattern of the class of possible computations is

so clearly revealed in the pattern of the program itself that it is possible to understand and prove the correctness of such a program without even thinking of the large class of potential computations involved. A type declaration should use similar clear patterns to define the structural properties of all possible data instances of that type, independent of their size or the component values involved.

## 4. Direct Product.

The first and simplest structuring method is known by mathematicians as the direct or Cartesian product. For example, a mathematician can define the space of complex numbers as a product of real and real, thus

$$\text{complex} =_{df} \text{real} \times \text{real}.$$

By this definition, he states that each value of the type complex is a structure with exactly two components, a first component which is real, and a second component which is also real; or in other words, an ordered pair of reals. Furthermore, the two reals are entirely independent of each other; their only connection consists in their grouping together in the specified order as a single complex number.

The same structuring method is just as familiar in data processing, where it is known as a record definition. For example, a record which specifies an insertion may be defined:

$$\text{insertion} =_{df} \text{partnumber} \times \text{partdetail};$$

or as it is expressed in a COBOL-like language:

```
01  INSERTION
  02 PARTNUMBER
     03  ....
  02 PARTDETAIL
     03  ....
```

where the items with level number 03 (and higher) indicate the substructure of PARTNUMBER and PARTDETAIL components.

The direct product method of data structuring has close analogies with the method of constructing a program by composition, ie, the compound statement. For example, a particular program or procedure may be composed of two statements:

begin q:=q+1; r:=r-y end

This means that every computation evoked by this compound statement also consists of two disjoint parts; the first adds 1 to q, and the second subtracts y from r. Furthermore, the only connecting relationship between these parts consists in their grouping together in the specified order.

One of the most important advantages of a good structuring method is that it can be applied to components of any size or substructure. The method of program composition can be applied equally to atomic instructions as to complete programs; for example, the following is a very common structure for scientific programs:

begin input; calculation; output end

Similarly, a large database may be described as the direct product of the files which it contains, for example:

database = salesledger × inventory × payroll × catalogue.

5. Discriminated Union.

The next simple structuring method is the discriminated union, which specifies that a choice is to be made from a selection of alternative structures. In the simplest case, the alternatives are just indicators of some condition, for example a possible malfunction of a peripheral device:

exception = parity fault | empty | manual.

This states that the type "exception" consists of exactly three values with the names indicated, and every exception variable has one of these three values. As a more substantial example, consider a transaction-record for a traditional file-updating program. If record definitions have already been made for deletions, insertions, and amendments, a transaction can be defined as being a choice of exactly one of these alternatives:

transaction = insertion | deletion | amendment.

A similar specification can be given in COBOL by multiple record definitions:

```
01  INSERTION
  02  ....
01  DELETION
  02  ....
01  AMENDMENT
  02  ....
```

At lower levels than 01, the REDEFINES clause permits a similar effect.

The discriminated union is closely analogous to the conditional or case construction in programming. A conditional of the form

if B then T else E

evokes computation of exactly one of the alternatives T or E, the selection to be made in accordance with the value of B.

## 6. Sequence.

The third major data structuring method is called the sequence. A sequence consists of none or more components of data, arranged in some meaningful order. In mathematical notation a sequence is frequently denoted by an asterisk, thus

$$\text{string} = \text{character}^*$$

which defines a string as a sequence of none or more characters, for example

```
A
B7Y;
A SLIGHTLY LONGER STRING
```

In data processing, a sequence is usually known as a file, and might be defined, for example:

$$\text{transaction file} = \text{header} \times \text{transaction}^* \times \text{trailer}$$

where "header", "trailer", and "transaction" have previously been given record definitions. A similar, but less precise, specification of a file can be given in the COBOL file description:

```
DATA RECORDS ARE
HEADER, INSERTION, DELETION, AMENDMENT,
   TRAILER;
```

A sequence corresponds to the iterative program structure, using while:

while B do L.

The computation of this structure consists of a sequence of none or more computations of the program component L; the sequence is not bounded in advance, but its length on any given occasion must be finite.

7. Types and recursion.

A procedure declaration in a programming language is a means of packaging a complex program structure, possibly evoking long and elaborate computations, and enabling it to be regarded and used many times by procedure calls, as though it were a single primitive unit of action. Thus the procedure declaration is one of the most powerful tools provided in a high level programming language for mastering the complexity of a large problem.

A similar benefit can be extended to the design and description of data by the type declaration of PASCAL, the mode declaration of ALGOL 68; or perhaps best of all, by the class declaration of SIMULA 67.

A type declaration associates a complex structure description with the type name. This name can be used repeatedly to declare new variables of the type, for example, using PASCAL notation:

```
X, Y:complex;
```

will declare two new variables X and Y of type complex; each of these will display the structure of a complex number, and each will consist of two separate real numbers; but the programmer can regard it as a single unit of data. On a larger scale, two files might be declared:

```
oldmaster, newmaster:masterfile.
```

The analogy with procedure declarations suggests the question, is there any place for recursion in the definition of data structures? Again the answer is yes. The occurrence of a type name inside its own definition denotes an occurrence of a (smaller) instance of a value of that type as a component. This is exactly analogous with the computation of a recursive

procedure, which contains a (smaller) computation of that same recursive procedure in place of each of the recursive calls.

Examples of recursive types occur frequently in programming language definition and processing, and also in general symbol manipulation. For example, the traditional data structure of LISP may be declared:

type list = atom | list × list;

or in other words, a list is either an atom (defined elsewhere) or an ordered pair, whose first and second components are themselves lists.

An example drawn from data processing applications is the catalogue of parts and components which is used in a parts explosion analysis. A part is either bought in (and has no components), or it is an assembly. Each assembly has assembly data, followed by a sequence of the parts from which it is made. These facts can be expressed formally in the type declarations

type part = bought in | assembly
type assembly = assembly data × part*

It is unfortunate that no well-known programming language permits this simple use of recursion in data structuring.

## 8. Operations.

The preceding sections have been based on an over-simplified view of the concept of type, which concentrates solely on their values and their structure. A more balanced view regards a type as a set of values (for variables, functions, and parameters), together with the primitive operations which can be applied to these values. Thus, the concept of the integer type consists not only in a certain range of integer values, but also in the availability of the primitive operators of addition, subtraction, multiplication and division, which are defined on integer arguments and give integer results. It is far better, for conceptual clarity as well as machine independence, to regard the integer type as a set of abstract values on which these operations are defined, rather than as being structured out of its component binary digits, which are used in most machines to represent the integer.

A similar abstraction is highly desirable when the programmer is constructing new types. For example, the complex type too is better regarded as an abstract number space over which certain arithmetic operations are defined, rather than in terms of its representation, say, in polar coordinates. But in this case it is the programmer rather than the hardware designer who must declare the functions or procedures which actually implement these operations. Furthermore, once these operations are implemented, they should be the only means of processing and updating complex numbers; and from then on the programmer should cease to think of them in terms of the pair of real components of which they have been made.

It is this resolve to hide the details of a representation which provides in data structuring the same power of abstraction that is offered by procedures and parameters. The caller of a procedure is encouraged to be unaware of the details of the computation which it invokes, and the nature and names of the local data which are needed temporarily during the computation. The only aspect of the procedure with which he is concerned is its effect on the actual parameters (arguments) which he has passed. Similarly, the user of a type should be concerned only with the effect of the operations which have been made available by the implementor of the type, and not with the details of internal structure.

Methods of operator definition are available in SIMULA 67, and have been developed by Liskov under the name "cluster". A similar idea is incorporated in Parnas' module.

## 9. Mappings.

The survey of data structuring methods given above deals only with the most important structures, which can be easily and efficiently mapped onto machine representations, and it is not intended to be complete. For example, it omits one very useful concept of a finite mapping. A finite mapping, in the mathematician's sense, is a function which is defined only on a finite range of arguments of type A, and which maps each argument onto a value from type B.

In mathematical notation, this may be expressed with an arrow:

$$A \rightarrow B.$$

To the programmer, the most familiar example of a finite mapping is the array, which maps a finite range of integers, say [a...b] onto values of some type, for example, real; Using mathematical notation this may be written:

type vector = [a..b] → real

In the case of a multidimensional array, it is a cartesian product of such ranges over which the array is defined:

type matrix = [a..b]× [c..d] → real.

But the concept of a finite mapping is more general than that; and the range of arguments may be any finite set. For example, a sparse matrix may be regarded as a mapping given only by its (finite number) of non-zero elements, eg

sparse matrix = integer × integer → real

In a more commercial environment, a pricelist may be regarded as a finite mapping which assigns a price to each product

pricelist = product → price.

Similarly, in a compiler, we require a dictionary which maps each identifier onto its decode (type, address, etc.):

symboltable = identifier → decode.

And finally, to illustrate the application of this concept on a large scale, consider a telephone directory. In the simplest view, this is nothing but a finite (but very large) mapping:

telephone directory = subscriber → telephone number.

## 10. Top-down Design.

The concept of mapping offers a very abstract way of looking at a large random-access file, and does not give any help or insight into the way in which the structure is going to be implemented in some combination of storage levels on a large computer. In fact, all the real problems of implementation remain. It is totally unrealistic to suppose that any high level language or automatic process (or even a generalised integrated data base management system),will be able to produce a satisfactory implementation from such an abstract definition.

However, I would suggest that this abstract definition, devoid of all implementation detail, will be helpful to the programmer in the early stages of design of his system. He can design his abstract program as though it operates on the abstract data structure, and can thereby complete a consistent design, without being confused by the details of the representation of his data; then, when he knows more exactly how this data is going to be processed, he may choose the most suitable representation, and implement it by coding the fundamental operations required. In this way, the design and implementation may proceed in an orderly fashion, from the top downwards, or even from the bottom-up, if preferred.

But in practice, progress may not be so orderly. If it appears that there is no acceptably efficient method of representing the data, it may be necessary to reconsider the abstract program. However, the clear separation of decisions and details relevant to the two stages will probably clarify the task of the designer or designers, and will almost certainly help him to communicate and discuss his problems with his colleagues and successors.

## 11. References and Pointers.

One remarkable feature of the structuring methods introduced here is that they make no mention of the reference or pointer, which are traditionally regarded as the prime means of structuring data. The situation seems similar to that of go to statements or jumps, which have traditionally played a

major role in computer programming, and which seem to be going rapidly out of fashion. In fact the analogy goes deeper. In the implementation of data structures use may be made of machine addresses, just as jumps are used in machine code to implement conditionals, and while loops and procedures. The major structuring disadvantage of the jump is that it creates new wide interfaces between distant parts of a program, which look as though they should be separate, and the slightest change to a program can propagate errors rapidly and uncontrollably along these interfaces. I suspect that the same is true of a reference, pointing from one part of a data structure to another distant part, which ought to be disjoint. And I expect that there will be yet another analogy - the recommendation to remove references from data structuring will meet as much controversy as that to remove go to's from programming; and perhaps even more so, because it runs counter to the still prevalent belief in integrated information systems, relational data bases, etc.; and suggests that earlier, simpler, techniques using separate files without cross references may be preferable in many respects. Certainly, in respect of reliability, this theory is confirmed by ample experience.

## Conclusion.

This paper has described a number of simple and efficient methods of structuring data, similar to those recommended for reliable construction for programs; and they seem to share the same desirable properties:

(1) They are few in number.
(2) They are logically simple.
(3) They are amenable to simple proof techniques.
(4) They may be applied on a large scale or on a small.
(5) They assist in top-down design or in bottom-up.
(6) They are easy to implement.
(7) The easy implementation is efficient.

It is for such reasons that I recommend adoption of these data structuring methods, not as an easy road to certain success in complex projects, but as a discipline, possibly harsh and painful, which we adopt willingly to help us in our unending search for simplicity.