Zeitschrift: IABSE reports of the working commissions = Rapports des commissions de travail AIPC = IVBH Berichte der Arbeitskommissionen

Band: 31 (1978)

Artikel: Readability of design programs

Autor: Alcock, Donald

DOI: https://doi.org/10.5169/seals-24921

IABSE
A I P C
I V B H

COLLOQUIUM on:
"INTERFACE BETWEEN COMPUTING AND DESIGN IN STRUCTURAL ENGINEERING"
August 30, 31 - September 1, 1978 - ISMES - BERGAMO (ITALY)

Readability of Design Programs

Lisibilite des programmes de calcul

Ablesung von Planungs Programmen

DONALD ALCOCK

MA, MS, MICE, FIStructE, FBCS

Alcock Shearing & Partners

Redhill, Surrey, England

Summary

Computer programs are being used to determine dimensions of structural members and details of reinforcement in engineering structures. Yet it is seldom possible for a design engineer to discover from a user's manual how such a program reaches these decisions for which he, the engineer, is ultimately responsible. This paper proposes a notation called 3 R for describing computer programs in a way that could make their logic intelligible not only to programmers but also to design engineers less familiar with software.

Résumé

Les programmes d'ordinateur sont utilisés pour déterminer les dimensions des éléments de structure et leurs liaisons. Néanmois, un ingénieur d' études peut rarement découvrir dans un manuel d' utilisateur comment un tel programme aboutit aux décisions pour lesquelles il est lui-même responsable en fin de compte. Ce rapport propose donc la notation 3 R pour décrire des programmes d'ordinateur de façon à en rendre la logique intelligible non seulement aux programmeurs mais aussi aux ingénieurs d'études moins familiarisés avec les programmes.

Zusammenfassung

Computer-Programme werden für die Ermittlung von Dimensionen von Bauteilen und Angaben von Verstärkungen im Maschinenbau gebraucht. Es ist jedoch selten für einen Bauingenieur möglich aus einem Anwendungs-Handbuch herauszufinden wie ein solches Programm Entscheidungen trifft, für welche er als Ingenieur letzten Endes verantwortlich ist. Dieser Bericht schlägt ein System vorgenannt 3 R - welches Computer Programme auf solche Weise beschreibt, dass darin enthaltene Logik nich nur dem Programmierer verständlich ist, sondern auch dem Bauingenieur, der weniger mit Software vertraut ist.

1. INTRODUCTION

If a bridge collapses because it was badly designed the consulting engineer is held responsible - whether the faulty calculations were made by incompetent employees or by computer. The legal problem is to prove the design, not the construction, was to blame - but if proof is possible the consulting engineer's insurance company has to pay up.

That is not necessarily the end of the story. Suppose the consulting engineer had based his bad design on the output of some proprietory program offered by a computer bureau? And if the bureau had offered use of the program on behalf of some other company then the bureau, in turn, would seek recompense from that company. It would be difficult because the author would maintain his program had been misapplied (a factor over which he could have no control) and point to the pile of rubble as evidence. Whatever the financial outcome and legal consequences of such a case, the problem posed here is that of a structural engineer injudiciously using results generated by a "black box".

The blackness of such boxes is examined in this paper, and a notation presented by means of which the logic of design programs could be clearly described - thereby reducing the opacity of potentially dangerous black boxes.

2. DESIGN BY COMPUTER

When computers were first used by structural engineers the only ready-made programs were limited in scope to simple analysis. The structural designer would check his results to ensure, for example, that reactions balanced applied loads; then he would work out areas of reinforcement and devise details of structural connections in the traditional way. The structural designer did not need to know much about the inner workings of the programs he used, but things have happened to change the picture. First the advance in analytical techniques (such as finite-element analysis) has made a computer indispensable and manual checking practically impossible; secondly, the computer is now used to decide the dimensions of structural members, details of connections, and precise sizes and arrangements of reinforcement.

2.1 Using Existing Programs

Despite the dangers of allowing a computer program to take this kind of decision it is inevitable that more and more consulting engineers will be compelled to do so. Design by computer is cheaper than traditional methods; failing to take advantage may mean going out of business. But few structural designers have the time or expertise to write their own design programs so most will have no choice but rely on those written by specialists. There will be ever more specialization because junior designers, being directed by their seniors to use existing design programs, will miss experience that would otherwise give them skill and judgement in the design process, hence the abilty to specify their own design programs.

There would be nothing wrong with a specialist writing a design program for other designers to use if only those designers knew precisely how the program reached its decisions, but the evidence is that they do not.

What information can a structural engineer get about a design program? Usually just its user's manual. This should tell him how to specify a problem by preparing data for punched cards or typing at a terminal of a computer. It should also explain how to interpret results produced by the program, and it should explain clearly what engineering assumptions the program makes and by what logic the program selects sizes and dimensions, but this kind of information is often lacking.

2.2 Experiences with some Design Programs

The Design Office Consortium [1], with the author as consultant, recently evaluated some publicly available programs for the design of reinforced concrete beams according to British Standard Code of Practice CP110. All programs had users' manuals which explained clearly enough how to prepare data, but given an identical design problem they produced amazingly different solutions. In a typical cross section the area of steel considered necessary by one program was several times that specified by another.

Except for one program (in which mistakes were found and subsequently corrected by the program's originators) no program seriously defied Code of Practice CP110; the enormous variance was permissible under the code. Yet from reading the users' manuals there were few clues to suggest the solutions would be different; a designer might reasonably have assumed all seven programs would design much the same beam. In other words the users' manuals lacked fundamental information.

2.3 Inadequacy of Information

It is not unknown for design programs to have no users' manuals at all; the designer gets a few rough notes, or perhaps a demonstration at a terminal to show how the program "asks" for everything it wants. More commonly a user's manual exists, but - like those describing the beam design programs mentioned above - it fails to explain fully how the program reaches decisions. Those secrets are concealed in the programmer's documentation which the user is not allowed to see - or which would be unintelligible if seen. Often there is no programmer's documentation either, the secrets lying buried in the programmer's head. But still such programs are used by designers - and structures built according to their results.

How, then, is a structural engineer to discover what a design program does? One answer may be that he can't. If a programmer does not want anyone else to know how his program works then potential users have little hope of finding out - and had best not use his programs because of the dangers described earlier. But if a programmer does want to communicate ideas to his fellow man he will do so; in words, by flow charts, or other means. On the other hand it is not easy for him to do so because of the gulf of expertise between an engineer who specializes in writing design programs and the practical designer who does not.

The next section of this paper introduces the idea of a notation for describing computer programs and designed to help span the gulf referred to above.

3. BIRTH OF A NOTATION

The author's firm was commissioned by the Design Office Consortium to write a computer program for calculating adjustments to fees payable to building contractors as influenced by certain Indices published monthly by the Department of the Environment. This program is called FORPA [2]. The commission was unusual in that the program was to run with minimal alteration on different makes of computer so that FORPA could be locally maintained wherever installed.

The traditional approach to such a problem would be to publish flow charts, specifications and listings. In this case, however, it was decided to devise a notation by which to describe programs generally - then publish a description of FORPA written in this notation together with a realization of FORPA transcribed from the notation into Fortran. This was done, and an identical Fortran realization runs today on several different makes of computer. Currently an APL realization is being transcribed.

The notation was designed to help in reading programs, writing them, and describing their arithmetic processes. Because reading, writing and arithmetic are called (in colloquial English) "the three R's" the notation has been given the name 3R.

3.1 Development of the Notation

Although 3R was devised with a limited aim - to describe the logic of FORPA to programmers and users alike - the notation was felt to have greater potential. Accordingly the Property Services Agency of the Department of the Environment commissioned the author's firm to assist in preparing a proposal [3] for the further development of 3R in cooperation with members of the C.I.B. working party, W52. The aim would be to refine and develop 3R and use it to describe substantial programs in the field of building design, thereby making the logic of decision processes in those programs intelligible to designers as well as programmers.

Concurrently (and from the point of view of software experts rather than building designers) 3R was presented by its designer, Brian Shearing, at a Seminar at Oxford University under the chairmanship of Prof. C. A. R. Hoare. Although some aspects of 3R were found wanting its reception was enthusiastic.

3.2 Relationship with Programming Languages

It is emphasized that 3R is a notation; not another programming language. It is possible to use 3R to describe a program in enough detail for a programmer to transcribe that program into a programming language, and for a potentail user of that program to comprehend its logic. Nevertheless 3R does have things in common with programming languages and may even be thought of as a "common factor" of common languages. For example, 3R has an assignment statement because most languages have assignment statements; 3R is not recursive because not all common langauges are recursive; and so on. Accordingly there is nothing in 3R to deal with machine-dependent details; where these crop up the 3R description has to break into human language.

4. A BRIEF EXPLANATION OF 3R

The 3R notation is simple. Although space forbids full definition, little detail is omitted in the following explanation of the notation as used to describe FORPA [2].

4.1 Overall Structure

A program described in 3R notation is a sequence of lines of text interspersed with blank lines for clarity. A line starting at the left margin is commentary. An indented line is called a "statement" and forms part of the 3R description, but may still include commentary enclosed in curly brackets.

"Words" of the notation are written in capital letters. "Names" - invented by the person describing a program - are written in small letters, several words being allowed in each name.

Logical flow is generally from one statement to the next until the final one, FINISH. But it is possible to parcel groups of statements into named "blocks" and put these anywhere in the text of a program without altering its logical flow - which simply "passes by" the definition of any block encountered. Definition has the form:

```
LET example block BE
   {sequence of statements}
END OF example block
```

Writing the name of such a block in the main program - as though the name were a statement - is called "invoking" a block. It implies logical replacement of the name by the sequence of statements in the block so named. A block may be invoked not only from the main program but also from within another block, and that from within another, and so on indefinitely - provided that no block is invoked recursively as a result.

Although logical flow may be "nested" to any depth as just described there may be no textual nesting of blocks (with consequent privacy of an enclosed block to its enclosing block); in 3R notation all blocks are at the same level. Likewise there is no nesting of loops or conditional statements - the effect of nesting is achieved by writing one or more "blocklets" within a block as explained later. The structure of programs described in 3R notation is constrained to be simple and linear so that a reader has only one level of thought to contend with at a time.

Communication between a block and the invoking piece of program is by arguments or shared variables or both. This is explained later.

4.2 Variables and Assignments

Variables must be declared before being referred to (not necessarily at the beginning of a program or block) and may have their range specified. In the examples below "colour" may take only three scalar values "red", "white" or "blue"; "number of file" may take any integral value from 1 to 99; "total number of files" only the value 99. Character variables have their limit of length specified as illustrated by "name of file" which may not contain more than six characters.

```
VARIABLE colour IS red OR white OR blue
VARIABLE number of file IS 1..99
INVARIABLE total number of files IS 99
VARIABLE name of file IS CHARACTER*6
```

Conventional real and integer variables may also be declared. And variables may be subscripted, in which case the ranges of subscripts must be specified.

```
VARIABLE stiffness matrix [1..300,1..50] IS REAL
VARIABLE list of six titles [1..6] IS CHARACTER*72
```

Variables declared in the main program are accessible to the main program and every block. Variables declared inside a block are private to that block.

Assignment to a variable is indicated by an equals sign. The expression on the right may involve symbols +,-,*,/,↑ (exponentiate by an integral power) in the conventional way. There may be several assignments separated by semicolons on the same line.

```
stiffness matrix [i,j] = -factor*modulus*inertia/(length↑power)
colour = red;   name of file = colour + "man"
```

The operator, +, in character operations denotes concatenation; the name of file above would become "redman".

There are two special operators, DIV and MOD, for use in non-negative integer expressions. These yeild an integral result, and integral remainder, of a division:

```
integral result = numerator DIV denominator
integral remainder = numerator MOD denominator
```

whereas the operator, /, always yields a real result. Otherwise "mixed mode" is not catered for, but special blocks may be assumed which are capable of converting from one mode to another. An example is:

```
x = real from integer (i)
```

When transcoded from 3R into a programming language such a statement would often become the unadorned statement "X = I". But the description of the program in the 3R notation is explicit and assumes no implicit operations.

4.3 Control Statements and Blocklets

An endless loop is denoted by a sequence of statements sandwiched between the words REPEAT and AGAIN. To leave a loop (transfer to the statement immediately following the word AGAIN) one of the statements in the loop may be the word WHILE or UNTIL followed by a Boolean condition.

```
REPEAT
   { optional sequence of statements}
UNTIL i > j    { or WHILE i <= j }
   {optional sequence of statements}
AGAIN
```

There is only one way to describe a choice of logical pathways in 3R notation - and when specifiying any choice all other possibilities must be explicitly catered for. The statement OTHERWISE FAIL is obligatory. An illustration of a choice between two pathways is:

```
IF x < y
   { statements to apply if x < y }
IF x > y
   { statements to apply if x > y }
OTHERWISE FAIL  { in this example x = y implies failure }
```

Separate pathways join again immediately after OTHERWISE FAIL. The null statement, PASS, is used is cases where no statements are needed on a pathway.

The design of this statement is based on Dijkstra's "guarded commands" [4] and chosen in preference to the unsymmetrical and ubiquitous IF..THEN..ELSE. Although it may seem unnecessarily arduous to enumerate every possible result of every condition, doing so has been found salutory - preventing mistakes that would otherwise have crept into programs. And certainly the person who transcribes from a 3R description enjoys the certainty of all cases having been considered.

Execution of the statement, FAIL, implies the "status" of the program becomes invalid. In every program described in 3R notation lies the concept of its current status being valid or invalid. Status starts as valid, but becomes invalid if the logical flow meets the word FAIL or an inconsistency such as a subscript out of range. The idea behind the concept of status is to provide a tidy mechanism for terminating programs in error. By preceding certain statements with the word TEST, and consulting two special Boolean variables VALID and INVALID, status may be tested and the result acted upon.

```
TEST element = vector [i]
IF VALID
   PASS
IF INVALID { etc.}
```

Testing an invalid status revalidates it. It is possible to induce the status to be invalid again by a FAIL statement. Unfortunately there is not enough space to discuss the mechanism of status more fully.

As stated earlier, loops may not be nested - nor may choice. Within a block, however, the effect of nesting may be achieved by naming - hence invoking - an inner structure as a "blocklet", then defining that blocklet. The first such definition is introduced by the word WHERE; subsequent ones by AND WHERE.

```
        REPEAT
           i = i + 1
        UNTIL i > 10
           inner nest
        AGAIN

     WHERE inner nest IS
        j = 0
        REPEAT
           j = j + 1
        UNTIL j > 10
           innermost loop
        AGAIN

     AND WHERE innermost loop IS { etc. }
```

All blocklets are written before the final END OF statement of their enclosing block. Any blocklet may access the variables declared within its enclosing block (as well as those declared in the main program) so there is no concept of "arguments" to blocklets as there is to blocks, as now explained.

4.4 Arguments, Input & Output

Additional communication is possible by invoking a block with "arguments". These arguments are interspersed among the words of the block's name to help the reader. The following block:

```
        LET stress at face of member BE
        VARIABLE ARGUMENT a IS REAL
        INVARIABLE ARGUMENT s IS top OR bottom
        INVARIABLE ARGUMENT n IS INTEGER
           { sequence of statements }
        END OF stress at face of member
```

could be invoked from the main program, or from another block, as:

```
        f = stress at (top) face of member (6)
```

where the actual arguments f, top, 6 replace dummy arguments a, s, n respectively. All dummy arguments must be declared either VARIABLE or INVARIABLE as shown.

Arguments may be declared not only in blocks but also in the main program. This is the means of communication between the program being described and its environment. The program's input is declared as a set of invariable arguments; its output as a set of variable arguments.

```
        INVARIABLE ARGUMENT keyboard[1..10000] IS CHARACTER*1
        VARIABLE ARGUMENT disk file[1..1000, 1..1000] IS REAL
        INVARIABLE ARGUMENT punched card[1..10000] IS CHARACTER*80
```

5. AN EXAMPLE 3R PROGRAM

The first program to be described in 3R notation was a simple word-processing program. This was subsequently transcribed into BASIC (one day's effort by Brian Shearing) and is the program by which the photographic masters of this paper were produced. Space does not permit reproduction of the word-processing program itself but the following example taken from an earlier paper [3] illustrates its style of documentation.

Following this example there is a reproduction of the statements of the program with commentary removed (automatically by the word-processing system) then a realization of the program in Fortran and another in BASIC.

5.1 A program for searching, described in 3R

The block of program below is designed to search for a given value in a pre-sorted table of values. If a match is found, the position of the value within the table is to be delivered. If no match is found, the block is to fail.

The following example would set the status of execution INVALID if the value were not fould in table[1]..table[n], but would set j if "value" were found in table[j].

```
TEST j = find (value) in first (n) words of (table)
```

The program to achieve the above example is as follows.

```
LET find in first words of BE
VARIABLE ARGUMENT j IS 1..1000
INVARIABLE ARGUMENT value IS INTEGER
INVARIABLE ARGUMENT n IS 1..1000
INVARIABLE ARGUMENT table[1..1000] IS INTEGER
```

Because the values in the table are sorted the method of "binary searching" can be used whereby the range of values considered is repeatedly halved until a match is found. During the search the range of values to be inspected is table[first]..table[last]. The initial range is the full table.

```
VARIABLE first IS 1..1000
   first = 1
VARIABLE last IS 1..1000
   last = n
```

The main part of the block keeps searching until a match is found.

```
REPEAT
   choose a value for j
UNTIL table[j] = value
   adjust the range
AGAIN

WHERE choose a value for j IS
```

Assuming a fairly regular distribution of values in the table, the position to be inspected from the table is chosen to be that in the middle of the current range.

```
   j = ( first + last ) DIV 2
```

```
AND WHERE adjust the range IS
```

If the value just inspected exceeds the given value then the new range is the lower half of the current range.

```
IF table[j] > value
   { first remains unchanged. }
   last = j - 1
```

If the inspected value is less than the given value then the new range is the upper half of the current range.

```
IF table[j] < value
   first = j + 1
   { last remains unchanged. }
```

The blocklet "adjust the range" cannot be entered if table[j] is equal to the value.

```
OTHERWISE FAIL  { computer failure }
```

Before resuming the main loop, "first" is checked not to have overlapped "last" indicating that the value is not in the table (or that the table is not properly sorted!).

```
IF first <= last
   PASS

OTHERWISE FAIL

END of find in first words of
```

5.2 The 3R Code without comments interspersed

```
LET find in first words of BE
VARIABLE ARGUMENT j IS 1..1000
INVARIABLE ARGUMENT value IS INTEGER
INVARIABLE ARGUMENT n IS 1..1000
INVARIABLE ARGUMENT table[1..1000] IS INTEGER

VARIABLE first IS 1..1000
   first = 1
VARIABLE last IS 1..1000
   last = n

   REPEAT
      choose a value for j
   UNTIL table[j] = value
      adjust the range
   AGAIN


WHERE choose a value for j IS

   j = ( first + last ) DIV 2

AND WHERE adjust the range IS
```

```
      IF table[j] > value
         { first remains unchanged. }
         last = j - 1
      IF table[j] < value
         first = j + 1
         { last remains unchanged. }
      OTHERWISE FAIL  { computer failure }

      IF first <= last
         PASS
      OTHERWISE FAIL

   END OF find in first words of
```

5.3 Transcription from 3R into Fortran.....and BASIC

```
      LOGICAL FUNCTION FIND(J,VALUE,N,TABLE)
C
C RETURNS .TRUE. WITH J SET IF AT TABLE(J)
C RETURNS .FALSE. IF NOT FOUND IN TABLE(1..N).
      INTEGER J, VALUE, N, TABLE(1000)
      INTEGER FIRST, LAST
        FIRST = 1
        LAST = N
10      J = (FIRST + LAST)/2
        IF (TABLE(J) .NE. VALUE) GOTO 20
        FIND = .TRUE.
        GOTO 30
20      IF (TABLE(J) .GT. VALUE) LAST=J-1
        IF (TABLE(J) .LT. VALUE) FIRST=J+1
        IF (FIRST .LE. LAST) GOTO 10
        FIND = .FALSE.
30      RETURN
      END
```

```
1000 DIM T(1000)
1010 REM
1020 REM SET N AND V, THEN GOSUB 1050
1030 REM IF V AT T(J) THEN R = 1,
1040 REM IF V NOT IN T(1..N) THEN R=0
1050 LET F = 1
1060 LET L = N
1070 LET J = INT((F+L)/2)
1080 IF T(J) <> V THEN 1110
1090 LET R = 1
1100 GOTO 1170
1110 IF T(J) <= V THEN 1140
1120 LET L = J - 1
1130 GOTO 1150
1140 LET F = J + 1
1150 IF F <= L THEN 1070
1160 LET R = 0
1170 RETURN
```


REFERENCES

1. Computer Programs for Continuous Beams - CP110, Design Office Consortium, Cambridge, 1978
2. FORPA Computer Program, Formula Price Adjustment for Building Contracts, Design Office Consortium, Cambridge, 1978
3. 3R - A Notation for Describing Computer Programs, Property Services Agency, Department of the Environment, London, 1978
4. Dijkstra, E.W., A Discipline of Programming, Prentice-Hall, 1976