An experimental testbed for numerical software

M. A. Hennell
Computational Science Department, University of Liverpool, Liverpool 3

This paper describes an experimental testbed facility designed to examine some of the problems which arise in the implementation of high quality numerical software libraries.

The testbed is used to measure the effectiveness of test programs. Effectiveness here is used in the sense that these test programs should ensure that the routine implementation is error free rather than to examine the numerical properties of the algorithm.

The testbed has been used in extensive investigations of the stringent test programs of the NAG numerical algorithms library (Ford and Hague, 1974) and continuation of this work is seen as a major application for the testbed.

(Received September 1976; revised version received October 1977)

1. Introduction

This paper describes the initial implementation of a software testbed which is ultimately intended to be particularly suitable for testing numerical software. In recent years there has been a great deal of interest in developing techniques for the testing of software up to some standard which is a guide to its quality (ACM, 1975). Most of these techniques are designed to test software in which the control flow is dependent on the values contained by Boolean or integer variables.

Techniques currently in vogue for demonstrating the reliability of computer software fall into three categories.

1. *Formally proving correctness*

This assumes a specification which the program can be proved to satisfy.

2. *Static testing*

These techniques either examine the control flow predicates and seek to demonstrate that there are no impossible paths, unassigned variables or other logical inconsistencies. Alternatively the program can be executed symbolically. This involves putting constraints on the input data and examining the predicates algebraically to determine impossible paths or values of input data to execute particular paths.

3. *Dynamic testing*

This involves selecting input data and executing the program to create an execution history. This enables a determination to be made of statements or branches which are not exercised (by this data). Sometimes the values contained by some variables may be examined. An example of a system of this type can be found in Fairley, 1975.

In the case of numerical software the first two techniques become extremely difficult to employ due to the uncertainty incurred by the use of finite arithmetic and the consequent roundoff and truncation errors. In particular, whilst a program may be proved to be correct, it may still be essentially useless, since it is well known that certain programming constructs lead to disastrous loss of numerical significance. It is also possible that the only specification available is the algorithm itself and that its range of valid input data may be unknown.

The testbed described here is essentially an implementation of the third technique and was designed to examine the quality of the testing procedures for numerical software, particularly those routines which are intended for inclusion in quality numerical algorithms libraries. The problem, therefore, is to study the adequacy of programs which are designed to test the implementation of library routines. This is not the same problem as that of designing test programs to examine the numerical properties of the algorithm (although in practice the latter are frequently used for the former purpose).

The objective of this work is to demonstrate that these test programs really do test the routines in the manner required and to assist in providing improved test programs where necessary. In this paper we distinguish between routines (i.e. routines on test) and test programs which are designed to drive these routines and examine their properties. At present the system is limited to the analysis of FORTRAN IV programs but an extension to ALGOL 68 is under way.

2. System Description

The testbed is implemented on a Computer Technology Modular One minicomputer, partially in machine code and partially in ALGOL 68.

The testbed consists of three phases as illustrated in **Fig. 1**. The first is a static analysis of the specimen routines. Here the structural attributes of the program are recorded and statistics gathered. Attributes of interest are subroutine calls, control jumps, statement types, etc. The second phase is the creation of a runtime history of the routine as it runs in a test program. This history is compiled from a number of program events such as entry to and exit from subroutines, jumps of control, entry and exit from DO loops, the assignment of values to identifiers and the values of predicates in conditional clauses. The system allows any of these events to be selected in a manner described in Section 4.

The third phase is the analysis phase where the data base created in phases 1 and 2 is integrated and analysed, not only to discover deficiencies in the current test data, but also to

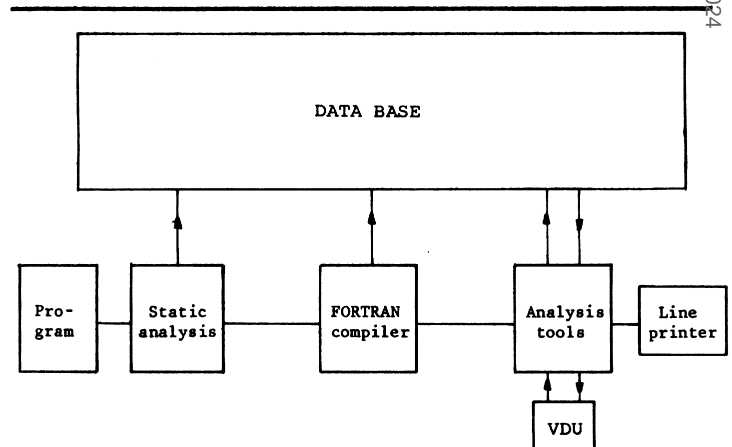

Fig. 1 Block diagram of the components of the testbed

Downloaded from https://academic.oup.com/comjnl/article/21/4/333/356861 by guest on 19 April 2024

attempt to derive new test data which will exercise the routine more rigorously. The system permits more than one execution history to be recorded enabling comparisons of data dependency to be made.

3. Static phase

The purpose of this phase is two-fold. Firstly to obtain data on the structure of the program, such as the possible control jumps. This data can then be compared with the dynamic data to determine which jumps have not been executed by a particular test. The second is that information about programming techniques currently in wide-spread use can greatly assist in the design of systems such as that described here, and also, for compilers and paging systems, etc. For these reasons we collect a large quantity of statistics for each program and accumulate these with the results from previous analyses. Hennell and Prudom (1976) contains an analysis of such statistics gathered from the NAG library. A secondary purpose of this phase is to obtain statistics for various measures of program style and complexity and to attempt to correlate these measures with the difficulty of testing programs (see Hennell, Woodward and Hedley, 1976). To date with only a small sample investigated no clear picture has arisen from this work.

All the routines in this phase are written in ALGOL 68 with a view to their integration into a portable system. Such a system could incorporate a program such as BRNANL (Fosdick, 1974) which instruments the users program with subroutine calls to provide the execution history. The static information from this phase is stored in the data base and consists of the following.

(*a*) a copy of each line of source code and its linenumber

(*b*) its type according to Sale's classification (1971)

(*c*) a table of all jumps, DO loop entry and termination points, subroutine calls and user supplied function calls.

From this information it is possible to obtain all the possible branches in the program.

4. Dynamic phase

The execution history is obtained by running the routine driven by a test program through a specially modified FORTRAN interpreter. This interpreter, originally written as part of a FORTRAN teaching system, provides four tracing facilities. These can be switched on and off within the FORTRAN program by means of embedded comment statements, such as

C-JUMPTRACE-BEGIN
C-JUMPTRACE-END

or, alternatively, the short forms

C-J-B
C-J-E

which switch the jump tracing facility on and off respectively. When the jumptrace is switched on a record of the linenumber at which the jump occurs together with the linenumber to which the jump is made, is stored in the data base.

The other three facilities are:

(*a*) DO-trace (C-D-B, and C-D-E), which records the entry to and exit from DO loops

(*b*) Sub-trace (C-S-B, and C-S-E), which records the entry to and exit from subroutine and function calls

(*c*) All-trace (C-A-B, and C-A-E), which records the values of assignments and the values of predicates in conditionals.

Since these facilities may be switched on and off freely, it is possible to examine in detail the performance of a particular part (or parts) of the routine.

By inserting only comments the routine code is in no way disturbed and possible side effects are avoided (in a system with inserted subroutine calls, conflicts can occur with user supplied routines if the names are identical; also in such a system (see Fairley, 1975) it is necessary to reformat the program to obtain predicates, etc.). Should a system user inadvertently insert such a comment, the only effect would be an increase in execution time of 20%-50%.

The criticism that with the use of a special interpretive system the routine is not being tested in its normal environment is answered by pointing out that the primary interest of the present system is to examine library software which usually has to run on many different compilers. The analogy can be taken with engine testbeds where general performance data can be obtained without the inconveniences of the normal environment. There will of course still remain a whole class of problems associated with specific characteristics of various operating environments. When examining a particular test program's performance all four tracing facilities are switched on at the start of the routine being tested and switched off at its end. It has not been part of the investigation to trace the test program. The reasons for this are outlined in Section 5.

The events selected are then stored in the data base. Pointers are inserted to enable selective analyses to be made of only the subroutine calling sequences or flow of control. The pointers enable a search of the data base to be made in either a forward or backward direction. At the present moment a limited capability for interacting with the executing program is available, but to date experience with this facility has not been wholly encouraging. The principal problem is that the think time exceeds the patience of other potential users of the mini-computer system. When the dynamic phase is completed the static and dynamic data bases are integrated.

5. Analysis phase

Analysis of the data base proceeds in two possible modes, interactively on a visual display unit (VDU) or as batch line-printer output.

All the programs which analyse the data base are written in ALGOL 68. Advantages of this are that the data base structures are simply handled, each word of the standard backing store is addressable and channels can be opened to any VDU console, thereby enabling an interactive analysis to be available in a very simple manner. Portability is assumed since backing store and channel facilities are defined as part of the language.

The contents of the data base may be examined interactively using a simple command language. The user can select a 'window' size which determines how many lines are to be output to the screen in one contiguous block; depressing the blank key gives the next block. To trace line-by-line requires a window size of unity. In response to prompts the user can select a tracing mode, i.e. jump-trace, do-trace, sub-trace or all-trace. Further prompts enable the user to set a particular linenumber and the frequency of occurrence of this line (this enables tracing to commence from within a loop). An analysis routine locates this point in the execution history and tracing may then proceed in either direction. The backward tracing facility has proved to be extremely useful for understanding how a particular data set drives the flow of control along a particular path. It has a lot in common with the technique of searching for a bug using a program trace produced by a compiler when a catastrophic failure occurs. In the latter case the point of the catastrophe determines the starting point in the backward search. For general tracing the user is interested in discovering how the computation reaches a particular basic statement block.

In general this interactive system has proved rather disappointing. The user usually loses track of all the parameters, i.e. loop depth, subroutine call depth, number of cycles per loop and so on. This forces the user to change displays repeatedly to obtain this information. This in turn overwrites much of the

Downloaded from https://academic.oup.com/comjnl/article/21/4/333/356861 by guest on 19 April 2024

```
   *JOB-FTS-3      TRIANGLE PROGRAM                                    1
 0 C-DOTRACE-BEGIN                                                     1
 0 C-JUMPTRACE-BEGIN                                                   1
 0 C-SUBTRACE-BEGIN                                                    1
 1        NR=5                                                         1
 2        NW=5                                                         1
 3        NCASES=4                                                     1
 4        DO 10 INPUT=1,NCASES                                         1 3  6   1
 5        READ(NR,100) I,J,K                                           1 3  6   1
 6        WRITE(NW,100) I,J,K                                          1 3  6   1
 7    100 FORMAT(3I10)                                                 1 3  6   1
 7 C.                                                                  1 3  6   1
 7 C.     CHECK SIDES SATISFY TRIANGLE INEQUALITIES                    1 3  6   1
 7 C.                                                                  1 3  6   1
 8        IF ((I+J.GT.K).AND.(J+K.GT.I).AND.(K+I.GT.J)) GOTO 1         1 3  6   1
 9        WRITE(NW,101)                                                1
 9 C.                                                                  1
 9 C.     TRIANGLE INEQUALITIES NOT SATISFIED - NOT A TRIANGLE         1
 9 C.                                                                  1
10    101 FORMAT(15H NOT A TRIANGLE)                                   1
11        GOTO 10                                                      1
12      1 MATCH=0                                                         4  7   2
12 C.                                                                     4  7   2
12 C.     TRIANGLE INEQUALITIES SATISFIED                                 4  7   2
12 C.     NOW FIND NO OF SIDES EQUAL                                      4  7   2
12 C.                                                                     4  7   2
13        IF (I.EQ.J) MATCH=MATCH+1                                       4  7   2
14        IF (J.EQ.K) MATCH=MATCH+1                                       4  7   2
15        IF (K.EQ.I) MATCH=MATCH+1                                       4  7   2
16        IF (MATCH-1) 2,3,4                                              4  7   2
17      2 WRITE(NW,102)                                                   4
17 C.                                                                     4
17 C.     NO SIDES EQUAL - SCALENE                                        4
17 C.                                                                     4
18    102 FORMAT(17H SCALENE TRIANGLE)                                    4
19        GOTO 10                                                         4
20      3 WRITE(NW,103)                                                       8
20 C.                                                                         8
20 C.     TWO SIDES EQUAL - ISOSCELES                                         8
20 C.                                                                         8
21    103 FORMAT(19H ISOSCELES TRIANGLE)                                      8
22        GOTO 10                                                             8
23      4 WRITE(NW,104)                                                           3
23 C.                                                                             3
23 C.     ALL SIDES EQUAL - EQUILATERAL                                           3
23 C.                                                                             3
24    104 FORMAT(21H EQUILATERAL TRIANGLE)                                        3
24 C.                                                                             3
24 C.     END-DO                                                                  3
25     10 CONTINUE                                                      2  5   9  3
26        STOP                                                                    3
27        END
27 *DATA
28            1         2         3
29            3         4         5
30            1         2         2
31            1         1         1
31 *ENDJOB
```

Fig. 2 A lineprinter listing showing lines of FORTRAN source code down the lefthand side and down the righthand side a display showing the flow of control. Each column shows passage of control flow until a jump is encountered. The succeeding flow of control is then displayed in the adjacent (rightmost) column

contents of the screen. Some work has been performed in Fairley (1975) on a split-screen display in which the upper half keeps an updated display of these parameters while the lower half scrolls the execution history. This improves the system appreciably. Further work with this sort of multiple display may lead to a viable general purpose system.

A second tool which has been found to be useful is a display (on either VDU or lineprinter) showing how the flow of control traverses the program for a particular data set. In **Fig. 2** is an example which illustrates this technique. The dynamic flow of control proceeds sequentially down the code until it encounters a jump to a line which is not the next executable statement. The flow of control is then restarted sequentially from this point. This control flow is represented by a digit in the range 1-9 being printed in a column alongside the source listing, the digit being incremented (modulo 9) and printed in the adjacent rightmost column when a jump occurs, as shown in Fig. 2. In this way the path taken by the control flow is clearly illustrated. It has been interesting to see how even experienced algorithm writers can be astonished at the insight this display gives into the algorithms performance. In terms of information content this display is better than a simple statement execution frequency count since it enables one to examine path frequencies as well. When more than one execution history is available comparison of these displays has been extremely helpful in understanding the data dependency of the routines.

When choosing test data for a particular routine it is desirable to be able to quantify the performance of various test data sets. For this purpose, following the techniques of Brown (1972), we introduce two test effectiveness ratios defined as follows:

$$\text{ter1} = \frac{\text{(no of statements executed at least once)}}{\text{(total number of executable statements)}}$$

$$\text{ter2} = \frac{\text{(no of branches executed at least once)}}{\text{(total number of branches)}}$$

In each case the numerator can be obtained from the execution history, whilst the denominator is obtained from the static analysis of the first phase of the testbed. A suitable criterion for a test-data set is that it should maximise both ter1 and ter2. In practice of course, ter1 tends to unity before ter2. If either quantity is not unity it implies that some code or some branches are not exercised by the current data set. Note also that ter2 = 1 implies ter1 = 1 but not vice versa.

A second possibility is that it may be desired to optimise the program code and for this purpose the 'coverage' (number of times each statement is executed) can be displayed alongside the program listing. This highlights the statements most frequently executed. A similar display can be given for branches which in turn enables conditionals to be reordered so as to minimise the number of tests.

Both ter1 and ter2 for a complete program may never reach unity regardless of the quality of the data. This can happen for instance if the main program branches before the first loop. Since the program starts only once, only one branch can be executed. For a routine however the position is different since it may be re-entered as many times as necessary. For this reason we usually confine tracing to routines.

6. Discussion

This paper has described a software testbed which it is intended will be particularly suitable for examining the quality of testing procedures for numerical algorithms. At present the basic system is completed so that a great deal of useful analysis can be made which it is hoped will lead to improved algorithm tests. So far the bulk of the experience has been gained by analysing a number of routines from the NAG numerical algorithms library (Ford and Hague, 1974) when they are driven by their associated stringent test programs. This has revealed a number of deficiencies in these tests and has helped in the designing of improved tests. It is hoped in time to analyse the whole of the NAG library stringent test programs in this way and in fact an ALGOL 68 version of the testbed is currently being integrated into the NAG ALGOL 68 library coordination process (Hennell and Yates, 1978).

Future developments of the system are following three lines. Firstly there is the ALGOL 68 version mentioned above. As a longer term project new analysis tools are being developed to help answer the problems posed by the current system. For instance, when unexercised code is detected, how can we produce new data which will exercise this code? Current research in the literature usually excludes real numbers due to problems of roundoff and truncation errors.

With the dynamic testing technique the data dependence of the control flow can always be found by executing the routine for a wide range of input data and comparing the runtime execution histories. However in practice this can lead to an unacceptably high number of runs and indeed it is frequently extremely difficult to identify the valid data space for a specific algorithm. It seems obvious from this that static testing techniques and dynamic testing must be integrated to solve this problem, and an investigation is currently in hand.

Finally, a fully parameterised floating point package is being produced for the Modular One computer which together with parameterised system functions will enable an investigation to be carried out into the sensitivity of the control flow to the arithmetic processes.

Downloaded from https://academic.oup.com/comjnl/article/21/4/333/366614 by guest on 09 April 2024

7. Acknowledgements

The author would like to thank Mr A. Prudom for supplying some of the static analysis programs, Mr D. Hedley for assisting with modifications to the FORTRAN interpreter and Dr M. Woodward for helpful discussions. Finally he would like to thank the NAG organisation for making a copy of their numerical algorithms library available, and the SRC for a research grant.

References

ACM (1975). *International Conf. on Reliable Software*, see papers therein, Los Angeles.

Brown, J. R. (1972). Practical applications of automated software tools, TRW report, TRW-ss-72-05, TRW Systems, One Space Park, Redondo Beach, California.

Fairley, R. E. (1975). An experimental program testing facility, *IEEE conference on software engineering*, Washington.

Ford, B. and Hague, S. (1974). The organisation of numerical algorithms libraries, in *Software for numerical mathematics*, ed. D. Evans, Academic Press.

Fosdick, L. D. (1974). BRNANL, a FORTRAN program to identify basic blocks in FORTRAN programs, University of Colorado report cu-cs-040-74.

Hennell, M. A. and Prudom, A. (1976). A static analysis of the NAG Fortran library, Computational Science Dept., University of Liverpool technical report.

Hennell, M. A., Woodward, M. R. and Hedley, D. (1976). On program analysis, *Information Processing Letters*, Vol. 5, pp. 136-140.

Hennell, M. A. and Yates, D. (1978). The ALGOL 68 NAG library coordination support system, submitted for publication in *The Computer Journal.*

Sale, A. H. J. (1971). The classification of Fortran statements, *The Computer Journal*, Vol. 14, No. 1, pp. 10-12.

Book reviews

On-line Data Bases, Infotech State of the Art Report, 1: Analysis and Bibliography, 2: Invited Papers. (*Infotech*, £110)

This is another report in the Infotech Series, in which a series of invited papers and other source material are printed. The convention is that the invited papers are printed in full in the second volume. An editor (in this case C. H. White) presents an analysis of the topic, using quotations drawn from these papers, in volume 1. Additional material from other Infotech sources is included in the analysis where it is relevant. The quotations used are printed in the order relevant to the editorial analysis, not necessarily the order in which the author originally wrote them. The editor decides the structure of his analysis, provides linking text and clarifying comment. The result is that volume 1 gives a logical presentation of the ideas contained in volume 2. For completeness the references given by authors of invited papers are listed both at the end of their papers and in volume 1. From the above it will be apparent that most of the material is printed twice, once in each volume. This approach does, however, give a considerable benefit as the edited analysis gives a broad view of the subject whilst the invited papers have, usually, a practical approach. The reader gets more out of the two books than he would by reading one of them twice!

The authors of the invited papers represent a wide cross-section of data base users and providers. As one might expect, their combined wisdom contains a great deal of commonsense and equally a great deal of valuable information. The editorial analysis of the papers covers aspects of online data base design from design philosophy through implementation, data base systems, performance, reliability and integrity to distributed data bases. The quotations used under each heading are apt and the editorial material helps to make volume 1 very readable. The bibliography is provided by Ian Palmer and contains mostly the references he gave in his own book (1975).

The invited papers contained in the second volume vary in the attention paid to detail from a brief statement of a users current position (Gurr, pp. 117-122) to consideration of the intervals between dumps (Davenport, pp. 65-92) via descriptions of data base systems implemented (e.g. Salter *et al.*, pp. 243-266).

These two books are worth careful study, particularly by someone contemplating the installation of an online data base system. They provide sufficient comment by users on their own experience to enable new installations to avoid some costly mistakes or omissions. Whilst not all of the risk can be removed, at least the ubiquitous wheel will not be re-invented.

R. E. Small (London)

Reference

Palmer, I. M. (1975). *Data Base Systems: A Practical Reference*; CACI.

Computer Methods for Mathematical Computations, by G. Forsythe, M. Malcolm and C. Moler, 1977; 259 pages. (*Prentice/Hall*, £12·80)

As stated in the introduction, 'this book is concerned with solving mathematical problems using automatic digital computers. An important part of the book is a set of FORTRAN subroutines. In fact, the book might well be regarded as an extensive user's guide for the subroutines'.

The subroutines are well documented and compare favourably with available software. It would have been useful to have a few sample outputs. In most cases the mathematical results are stated without proof. This method of presentation will of necessity narrow the appeal of the text.

There are nine chapters, namely, Floating-point computation, Linear systems of equations, Interpolation, Numerical integration, Initial value problems in ordinary differential equations, Solution of non-linear equations, Optimisation, Least squares and the singular value decomposition, Random number generation and Monte Carlo methods. The bibliography and reference are excellent. The problems are well chosen and graded. It seems strange to find no reference to the eigenvalue problem, particularly in this text with its stated goal. The pitfalls of computation are well illustrated by means of examples. The style throughout is lucid, occasionally perhaps over-discursive. At £12·80 it represents reasonable value.

M. P. J. Curran (Galway)

Computer Operating Systems, by D. W. Barron; 1977; 135 pages. (*Chapman and Hall*, £2·95)

A readable paperback but a reprint of a six year old text. This is unimportant because the practice today and history necessary to understand the concepts remain the same.

Firstly a job supervisor is described and then multiprogramming. Processor allocation is explained with an excellent description of activities and semaphores. Store allocation is covered up to early virtual machine and paging mechanisms. The discussion of I/O and filing systems that follows is biased towards multiaccess rather than commercial systems. Finally linked computer systems, JCL and the operator interface are covered. Tailoring of systems at generation time, provision of hardware error diagnostics, real time data acquisition and transaction processing requirements such as airline reservation are omitted.

I recommend this as a first book on operating systems to computer science students or professional programmers.

M. Evans (Cambridge)

Downloaded from https://academic.oup.com/comjnl/article/21/4/333/356861 by guest on 19 April 2024