An experimental testbed for numerical software, Part 2: ALGOL 68

M. A. Hennell and D. Hedley

Computational Science Department, University of Liverpool, Brownlow Hill, Liverpool L69 3BX

In this paper we describe the extension of an existing FORTRAN IV numerical software testbed (Hennell, 1978) to enable ALGOL 68 programs to be investigated. The extensions necessary were two-fold; firstly, a complete rewrite of the first phase, a static analysis in which the source text is reformatted (for reasons stated within this paper), analysed for all possible control jumps and statistics on language constructs are collected.

The second major extension was to incorporate into the second phase an existing ALGOL 68 compiler which after some modification enables dynamic execution histories to be collected in a data base. These modifications to the compiler represent extensions to the language definition which enable user programs to trace themselves. The utilisation of this compiler restricts source code programs to be written in ALGOL 68s, an official ALGOL 68 subset (Hibbard, 1974).

The third, analysis, phase is essentially identical in both the FORTRAN and ALGOL 68 systems.

(Received August 1977)

1. Introduction

In a previous paper (Hennell, 1978) a software testbed was described which is intended to be particularly suitable for the analysis of numerical software. This system analysed the run time execution history of a FORTRAN program or part of a program and compared it with a static analysis, thus enabling us to quantify the quality of program testing since the percentage of statements or branches actually executed can be measured. Program optimisation is also possible since the frequency of execution of individual statements or branches is available. In this paper we describe the extension of the testbed to incorporate the analysis of ALGOL 68 programs.

Basically the original system consists of three phases. In the initial static phase an analysis of the routine is made to determine the statement type, store the source code, determine all possible jumps and gather statistics. The second phase involves running the FORTRAN program with a FORTRAN interpreter and selecting various events for recording in an execution history. The third phase involves analysing the resultant data base using various tools designed to illustrate particular aspects of the program's performance.

To be able to incorporate ALGOL 68 programs the first two phases were revised. In Section 2 we describe how ALGOL 68 programs have been instrumented and a compiler modified to provide a suitable execution history. The acquisition of the execution history from the instrumented program constitutes the second phase of the ALGOL 68 testbed. We also outline the reasons why the first phase performs a reformatting of the source program.

The tracing mechanism involved is both powerful and simple to implement. It is clear that the inclusion of similar tracing facilities in all ALGOL 68 compilers would provide users with an extremely powerful diagnostic facility.

2. Instrumenting technique

To obtain an execution history of a program, three techniques are possible. Firstly the program can be instrumented with calls to various event reporting routines. Systems using these techniques can be found in Fairley (1975) and the references therein. The second technique is to provide monitoring facilities within the compilers (the FORTRAN testbed of Hennell (1978) belongs to this category). The third is to provide a combination of both these techniques. The ALGOL 68 system to be described here falls into the third category.

The tracing of FORTRAN programs is greatly facilitated by the simplicity of the language. In general the major problems arise from the way in which logical IFs are handled. It is not enough, for instance, to record whether the statement has been executed, because clearly the logical expression will always be executed whilst the controlled statement may not. Techniques for coping with this problem can be found in Fosdick (1974).

With a language as powerful as ALGOL 68 where extremely complicated compound statements may be found it is not obvious which events need to be recorded to obtain from the execution history those paths which were elaborated. For instance, with

$$minim\ (x, \textbf{if}\ x > 0\ \textbf{then}\ p\ \textbf{else}\ q\ \textbf{fi}, \textbf{void}: \textbf{goto}\ 1);$$

it is not immediately obvious how one could instrument either the compiler or the source code to trace unambiguously the execution of this construct.

The compiler available was an implementation of ALGOL 68s, a pure subset of ALGOL 68 (Hibbard, 1977), and runs on a CTL Modular One minicomputer in the Computational and Statistical Science Department at the University of Liverpool.

As originally designed this compiler keeps a record of the source code line number in the compiled code, so that when a run time error occurs the user can be given the line number together with a diagnostic message. From experience with the FORTRAN tracing, it was expected that easy access to this line number at certain points in the program would provide a powerful method for analysing the performance of the program. This led to the compiler writer being approached to provide such access within a framework which would enable any ALGOL 68 program to be traced in a meaningful manner.

The mechanism devised was to provide a special variable called trace, declared in the standard prelude with the mode

ref proc (int) void,

Then any routine which is assigned to this variable is called automatically during the running of an ALGOL 68s program at all line number interrogation points. The value of the integer parameter is the source listing line number of the line on which the call of trace has occurred.

The line number interrogation points are:

(*a*) the beginning of every source line

(*b*) the beginning of every routine text

(*c*) the return from every procedure call

(*d*) the entry to any alternative of a choice clause

(*e*) the end of every balance

(*f*) the start of the **while** and the **do** of every loop

(*g*) the end of every loop.

Calls at interrogation points are disabled within the body of the routine text assigned to trace. They are enabled again on normal return from the routine. The only restriction is that no scope checking is performed on assignations to trace. There is also a run time penalty of the order of 20 per cent when a routine is assigned to trace, in addition to the time taken to evaluate the routine.

As in our FORTRAN system, tracing can be freely switched on and off at arbitrary points within the ALGOL 68s source code. The assignation

trace :=(**int** *n*) **void**: **skip**;

disables the tracing.

Consider the unit

x := **if** $a > b$ **then** a **else** b **fi**;

Then line number interrogation points occur at the start of the line, after **then,** after **else** and at **fi**. It now becomes apparent that to gain the maximum amount of information concerning the flow of control through an ALGOL 68 program from interrogating the line number alone, we must reformat the original program so that, for example, components of choice clauses lie on separate lines. In fact our reformatting program performs the following functions

(*a*) closing brackets for choice clauses, labels and the **while** of any loop, are placed on a new line

(*b*) any text following a **goto** statement, or the **in** or **then** of choice clauses, is placed on a new line

(*c*) all the separators of choice clause alternatives and the **do** and **od** of every loop, are placed on a line by themselves.

Having achieved this reformatting the first three categories of line number interrogation points are sufficient to give us the complete path of control flow through the program. It is then useful to define a jump as occurring when the line number does not change by natural selection. More formally we define a jump as occurring in the reformatted program when control may be transferred from line m to line n, where either $m > n$ or $m < n$ and there exists some executable code between these two lines. If at any line where there is a jump there is also the possibility of transfer of control by natural succession to the next line then we add all such control transfers to the set of all jumps and call the result the set of all branches.

An ALGOL 68 program to reformat the test program or routine constitutes part of the first phase of the testbed. To flow trace the reformatted program a macro inserts a suitable prelude, with the trace routine assignment, and postlude and then runs the program on the ALGOL 68 compiler. It should be noted that the execution history is recorded on the standard backing store in which each word is program addressable. At the termination of the program this execution history is retrieved and stored in a permanent file.

In the appendix some examples are given of trace routines to achieve monitoring of various events, such as control flow tracing and assignment monitoring.

Should the user wish to trace only a small segment of a program then the instrumentation has to be carried out manually. In particular all monitoring of variables must at present be hand instrumented.

The last phase continues with a static analysis of the reformatted ALGOL 68 program to determine all the theoretically possible branches.

Jumps in the reformatted program occur at the following points:

(*a*) from every **goto** to its corresponding label

(*b*) from the call of a procedure to the beginning of the procedure text

(*c*) from the end of a procedure text to the calling point

(*d*) from the **in** of a choice clause to the entry of any alternative of a choice clause except the first

(*e*) from the **then** of a choice clause to the end of the alternative immediately following

(*f*) from the end of each alternative of a choice clause except the last to the end of the balance

(*g*) from the **do** to the **od** of any loop except those with fixed bounds which ensure that the loop must be elaborated

(*h*) from the **od** to the **do** of any loop which does not contain a **while**

(*i*) from the **od** to the **while** of a loop.

In cases (*d*) to (*i*) control flow can also carry straight on, thus completing the set of branches.

It must be emphasised that the problems of finding the branches in ALGOL 68 programs is much more difficult than the equivalent in FORTRAN. For example, routine texts may be assigned to a procedure anywhere after it has been declared, and parameters of mode **proc** may themselves be a routine text, rather than a reference to a procedure which has already been declared.

This static analysis is then incorporated into the data base with the execution history.

3. Data base analysis

In the FORTRAN testbed, the FORTRAN interpreter was modified to output event monitoring data to the disc, from which it was analysed by suites of ALGOL 68 programs. One consequence of this technique is that this data was written to that portion of the disc working area reserved for the ALGOL 68 standard backing store (standback). Thus, using the trace routines described in the appendix, which also output to standback, ensures that provided we use a compatible format, the third (analysis) phase of the FORTRAN testbed can be used without modification. This is a particularly satisfactory situation since a great deal of programming effort has been expended on investigating not only analysis techniques but also the presentation of data to the user.

The principle features of the testbed are:

(*a*) a statement execution frequency count (coverage), coupled with the percentage of unexecuted statements and the percentage of unexecuted branches

(*b*) a dynamic display of the program execution as shown in **Fig. 1,**

(*c*) an interactive interrogation of the data base, monitoring the execution of the program in either the forward or backward direction. Details of this facility are described in Hennell (1978) and are identical for both languages.

The principle applications of these tools have been to quantify the testing process (Hennell, Woodward and Hedley, 1976), assist in the derivation of improved test data and improve the quality of the code by carrying out code optimisation using the statement and branch execution frequencies.

The improvement of test data is achieved by choosing a number of data sets and examining the branches executed by each. In this way data which executes paths which have already been executed can be excluded. The final test data set is the collection of these individual data sets which maximises the total number of executed branches. It must be emphasised that there may, of course, be other reasons for including data sets, for instance those which provide difficult tests for the algorithm rather than successful implementation.

```
 # trace header #
 begin
  int from := 1, to := 1;
  reset (standback);
  trace := (int linenumber) void:
  begin
   if linenumber ≠ from + 1 and linenumber ≠ from
   then to := linenumber; put (standback, (from, to) )
   fi;
   from := linenumber
  end;
 # triangle program #
 1begin                                                 1
 2 int cases := 4, i, j, k, match;                      1
 3 to cases                                             1
 4 do                                                   1   3           9           6
 5  read ( (i, j, k) ); print ( (newline, i, j, k) );   1   3           9           6
 6  # check triangle inequalities satisfied #           1   3           9           6
 7  if i + j ≤ k or j + k ≤ i or k + i ≤ j              1   3           9           6
 8  then                                                1   3           9           6
 9   # inequalities not satisfied #                     1
10   print ("not a triangle")                           1
11  else                                                1     4           1           7
12   # triangle inequalities satisfied #                      4           1           7
13   # now find no of sides equal #                           4           1           7
14   match := 0;                                              4           1           7
15   if i = j then                                            4           1           7
16           match + := 1                                                             7
17              fi;                                             5           2         7
18   if j = k then                                              5           2         7
19          match + := 1                                                    2         7
20             fi;                                                6         2         7
21   if k = i then                                                6         2         7
22           match + := 1                                                             7
23              fi;                                                 7         3       7
24   if match = 0                                                   7         3       7
25   then                                                           7         3       7
26     # no sides equal #                                           7
27    print ("scalene triangle")                                    7
28   elif                                                           7           4       8
29      match = 1                                                               4       8
30   then                                                                       4       8
31     # two sides equal #                                                      4
32    print ("isosceles triangle")                                              4
33   else                                                                       4         9
34     # all sides equal #                                                                9
35    print ("equilateral triangle")                                                      9
36   fi                                                           8           5       9
37  fi                                                    2           8           5       9
38 od                                                     2           8           5       9
39end                                                                                     9
   # trace tail #
   ;
  put (standback, (9999,9999) ); close (standback)
 end
```

Fig. 1 **To show flow of control through a program**

4. Conclusions

The system described in this paper is not only suitable for detailed investigations into software quality but by making the facility available to all users of the ALGOL 68s system, including undergraduate students, a considerable amount of experience has already been amassed. In the past a large amount of academic staff time has been expended in convincing students that the behaviour of rogue programs is due to programming errors and not to system errors. With the powerful tracing facilities the students can see for themselves exactly how the program arrives at the particular termination point.

On the other hand convincing the compiler writers of the presence of a system bug is made easier by presenting them with the detailed trace analysis.

The system has been incorporated into the normal coordination and validation process of the NAG ALGOL 68 Numerical Algorithms Library (Hennell and Yates, 1975). Its use in this project has been threefold: to optimise routines (using the instruction count), to detect bugs (its success rate here is very high), and to develop improved stringent test programs. Some stringent tests submitted to the library coordinator by routine implementors have tested less than 70 per cent of the code. The implementors in question are competent numerical analysts who really believed that they had supplied comprehensive tests. Preliminary reports of the use of this testbed for the NAG ALGOL 68 library implementation can be found in Hennell, Hedley and Woodward (1976). In this paper it is demonstrated that a significant proportion of the short paths are not tested and that these short paths are a fertile field for the presence of program bugs.

In a further paper (Hennell, Hedley and Woodward, 1977) we have used the testbed to investigate the difficulty in attaining three separate measures of testing, namely TER1, TER2 and TER3 (see Hennell, Wocdward and Hedley (1976) for definitions). In general, it is shown that despite the fact that for many routines the measure TER1 is near unity (corresponding to all lines of code executed), unity for the higher measures is more difficult to attain.

Some additional aids which apply exclusively to the ALGOL 68 system have already been added; for instance the bracket structure (this is essentially the equivalent of the block structure) can be displayed against a source listing. The user can elect to display either the total bracketing structure or can have the **do-od, if-fi** and **case-esac** structures displayed separately. This facility was originally intended as the first part of a display in which various control structures and control variables will be presented. However, users' demand for this bracketing display to assist in debugging routines which failed compilation has justified it being incorporated into the standard computing system debugging aids.

Finally we remark that it is our intention to continue development of this system to include program analyses beyond the third level described in Hennell, Woodward and Hedley (1976) and to develop tools which will emphasise the facets of ALGOL 68 not available in FORTRAN.

5. Acknowledgements
This work has been carried out in part under SRC Grant B/RG/7747/2. The authors wish to express their gratitude to Dr P. G. Hibbard for the superb tracing facility he implemented for us in his compiler and to Dr M. R. Woodward and Mr J. M. Watt for many helpful discussions.

Appendix

In this appendix we detail some examples of the preludes and postludes currently used in the testbed. The preludes contain the relevant assignments of a routine text to the variable trace, whilst the postludes are added firstly to close the brackets of the prelude and secondly to insert terminators to the data base entries.

1. *Standard prelude for tracing flow of control*

A jump is detected if the line number at a line number interrogation point differs from its previous value by other than unity or zero. The source program must be reformatted for this routine to trace control flow successfully.

```
begin int from := 1, to := 1;
 reset (standback);
 trace := (int linenumber) void:
 begin if linenumber ≠ from + 1 and
  linenumber ≠ from
  then to := linenumber;
   put (standback, (from, to) )
  fi;
  from := linenumber
 end;
```

2. *Standard postlude*

```
  ;
  put (standback, (9999,9999) );
  close (standback)
 end
```

3. *Prelude for printing out the value of a real variable whenever its value changes*

```
 begin ref real trp := nil;
  real trv;
  string trs;
  int trn;
  trace := (int n) void:
  begin if ref real (trp) isnt nil
   then if trp ≠ trv
    then print ( (newline,
     "line", trn, trs, ":=",
     trv := trp) )
   fi
  fi;
  trn := n
 end;
```

Thereafter, assignations such as

$$trv := trp := x;\ trs := \text{“}x\text{”};$$

will cause tracing of x.

References

Fairley, R. E. (1975). An Experimental Program Testing Facility, *IEEE Transactions on Software Engineering*, Vol. 1 No. 4, pp. 350-357.

Fosdick, L. D. (1974). BRNANL, a Fortran Program to Identify Basic Blocks in Fortran Programs, Report CU-CS-040-74, Computer Science Dept., University of Colorado.

Hennell, M. A. (1978). An experimental testbed for numerical software, *The Computer Journal*, Vol. 21 No. 4, pp. 333-336.

Hennell, M. A. and Yates, D. (1975). The Algol 68 NAG Library Coordinator Support System, Report CSS/75/3/1, University of Liverpool, submitted for publication.

Hennell, M. A., Hedley, D., and Woodward, M. R. (1976). Experience with an Algol 68 Numerical Algorithms Testbed, Proc. Poly. Inst. of New York Microwave Research Institute Symposium series, April 1976, Vol. XXIV, Ed. J. Fox, Wiley and Sons, pp. 457-463.

Hennell, M. A., Hedley, D., and Woodward, M. R. (1977). Quantifying the Test Effectiveness of Algol 68 Programs, *Sigplan Notices*, Vol. 12 No. 6, pp. 36-41.

Hennell, M. A., Woodward, M. R., and Hedley, D. (1976). On Program Analysis, *Information Processing Letters*, Vol. 5 No. 5, pp. 136-140.

Hibbard, P. G. (1974). A Sublanguage of Algol 68, *Algol Bulletin*, July 1974 and *Sigplan Notices*, Vol 12 No. 5, 1977, pp. 71-79.