DETECTING ERRORS IN PROGRAMS

by

Lloyd D. Fosdick
Department of Computer Science
University of Colorado
Boulder, Colorado 80309

CU-CS-149-79 February, 1979

To appear in "Proceedings of IFIP Working Conference on Performance Evaluation of Numerical Software", Baden, Austria, December 1978.

DETECTING ERRORS IN PROGRAMS*

Lloyd D. Fosdick
Department of Computer Science
University of Colorado
Boulder, CO 80309, U.S.A.

A review of work on the occurrence and detection of errors in computer programs is presented. This includes: experiments to measure the frequency and distribution of errors; the use of simulation to determine the effect of typing mistakes; data flow analysis for static error detection; and measures to quantify program testing.

1. INTRODUCTION

> "It is natural at first to dismiss mistakes in programming as an inevitable but temporary evil, due to lack of experience, and to assume that if reasonable care is taken to prevent such mistakes occurring no other remedy is necessary".

Stanley Gill wrote these words over twenty-five years ago [1], pointing out that experience refuted this position: experience still refutes it, arguments for proving programs correct notwithstanding. We are human, and being human it is inevitable that we make mistakes which result in errors in computer programs. With improved languages, with careful attention to program design and programming style we can reduce the occurrence of errors. But we cannot eliminate them, so we cannot ignore the problem of error detection or questions concerning the distribution of errors, their types, and their symptoms.

This paper is concerned with the occurrence and detection of errors in computer programs. It is a review intended to show the nature of the research being conducted and the state of our knowledge to those not well acquainted with the subject. It will show that our knowledge of the occurrence of errors in computer programs is scanty and difficult to interpret, that error detection in practice is weaker than it needs to be, that software testing is guided largely by intuition and is lacking a scientific foundation, and finally that there is no shortage of interesting problems in this area.

2. TERMINOLOGY

The terminology used to describe errors in the literature varies in meaning, making it difficult to interpret and compare results. For example, in one article "syntax error" means any error reported by a compiler, in another it means a violation of a grammatical rule. While it might seem easy, anyone who attempts to analyze the problem of error classification will find it very difficult; indeed the notion of error itself is hard to characterize. This is confirmed by the experience of Gerhart [2] who has recently examined this issue in connection with an unsuccessful attempt to build a taxonomy of errors. However we must have a common understanding of the terms used here so some brief definitions suitable for this purpose follow.

Here the word "error" means a construction in the program which violates a language rule, or which may cause a computation that is not intended according to the specification of the program: an error of the first type is called a language error, and an error of the second type is called a specification error.

---

* This work was supported in part by U.S. Army Research Contract No. DAAG29-78-G-0046.

Language errors are divided into two types, syntax errors and semantic errors, according to the following scheme which is similar to that used by compiler writers [3]: the language rules are separated into two groups such that those in one group can be represented by a context free grammar, then a language error is called a syntax error if it is a violation of the rules of the context free grammar, otherwise it is called a semantic error. If a program contains no language errors it is called well-formed, and a well-formed program with no specification errors is called correct. A construction in a well-formed program that is extremely unusual is called an anomaly; it is often a symptom of a specification error.

It will be convenient to use the term "flow graph". This is a directed graph representing the control structure of a program, with the nodes corresponding to statements and the edges to pairs of statements executable in immediate succession. Thus a flow graph is an abstraction of the conventional flow diagram. In speaking of program execution I use a loose but convenient terminology, speaking of executing a node (i.e. a statement) or executing a path (i.e. a sequence of statements).

Two approaches to error detection, called static and dynamic, are distinguished. Static error detection is characterized by the use of techniques that do not require actual execution of the program. The most common form of static error detection occurs during parsing in the compilation of a program. Dynamic error detection, on the other hand, uses information gathered during actual execution of the program to detect errors. The most common form of dynamic error detection occurs during the testing of a program.

Unfortunately language rules and specifications are often incomplete or ambiguous, so the application of these definitions runs into difficulty; therefore in practice the distinction between syntax error and semantic error is often fuzzy, as is the distinction between well-formed programs, and correct and incorrect programs. FORTRAN programs present a particularly difficult problem in this respect. Furthermore the notion of a correct program is idealistic for most practical purposes, especially so when it comes to numerical software, since we lack the mechanisms required for insuring that a program meets its specifications unless the specifications are exceptionally simple. Thus these definitions must be taken as a statement of intent, recognizing that in practice they will be applied imperfectly.

3. OCCURRENCE OF ERRORS

Our concern here is with the frequency of errors in program text and with the types of errors that result from simple mistakes made in the preparation of program text. Data on errors has been obtained by examination of a collection of arbitrary and naturally occurring program text, such as that submitted to a computing facility within a particular period of time [4,5]; by examination of error reports in controlled programming environments [6,7]; and by experiments using deliberately chosen problems, and a controlled group of programmers [8,9,10]. Because it involves human subjects gathering this information is difficult; often students are used as subjects, casting doubt on the generality of the results.

The difficulties of dealing with human subjects can be removed, or at least isolated, by using simulation; an approach which conveniently divides the study of the occurrence of errors into two distinct components which can be studied separately -- the frequency and kinds of mistakes that humans make in the construction of programs, and the effect of these mistakes on program text. The second component can be studied with simulation, thus it can be precisely controlled and large amounts of data can be obtained at low cost.

3.1 Errors produced by humans. For programs in the stage of development when the first compilations are being made, the frequency of errors found in the text ranges from about fifty errors per thousand statements to over one hundred errors per thousand statements.

In discussing their work on DITRAN, a FORTRAN compiler, Moulton and Muller [8] reported on the errors detected in students' programs using it. For errors detected during compilation, which would include all syntax errors and most semantic errors, their data indicate an error frequency of more than forty errors per thousand statements. For errors detected during execution time which would include semantic errors not caught in compilation, their data indicated at least eight errors per thousand statements. Since this data does not appear to include specification errors it seems reasonable to conclude that the frequency of errors in these programs was in excess of fifty errors per thousand statements. These numbers are reasonable but crude because of problems in interpreting the data. They counted diagnostics and one error can cause more than one diagnostic, but they do not distinguish two compilations of the same program from compilation of two different programs. I have assumed one error per diagnostic in estimating frequencies which will tend to make the frequency too high, but failure to distinguish multiple compilations of the same program tends to reduce the frequency.

Youngs [9] has made a careful study of errors in programs and his results are consistent with the error frequencies indicated by the Moulton and Muller data. In Youngs' study the languages ALGOL, BASIC, COBOL, FORTRAN, and PL/1 were used, and programs written by "professionals" and novices were considered. Most of the programs were simple numerical routines of less than eighty statements in length. His data indicates a frequency of more than seventy errors per thousand statements. Moulton's and Muller's data applies largely to beginners, as does part of Youngs' data and one might guess that this would bias the error frequency to be high; however Youngs says "... on first runs of all programs both beginners and advanced programmers had an average of 5.6 errors in their programs ...", and he said that beginners committed 19.4 errors per program compared with 15.1 errors per program for advanced programmers. (There is an inconsistency in these figures that I have not been able to resolve). The significant difference that Youngs did observe between the two groups was in the type of errors committed: for the beginners the error distribution according to type was 12% syntax,41% semantic, 35% "logic" (specification), 5% "clerical" (syntax or semantic),and 12% other; for advanced programmers it was 17% syntax,21% semantic, 51% "logic" (specification), 4% "clerical" (syntax or semantic),and 11% other.

Gannon [10] made a careful study of the occurrence of errors in programs in connection with an investigation of the effect of language design decisions on programming errors. In this study two systems programming languages, TOPPS and TOPPS II, were used. The programmers did not have prior experience with either language but were "reasonably experienced programmers". His data indicated a frequency of 164 errors per thousand statements, including errors made after the first submission to the compiler.

Boies and Gould [5] have measured the frequency of errors which prevent a program from compiling or assembling. The languages used were FORTRAN, PL/1 and assembly language (IBM). All jobs submitted through the IBM TSS/360 computer system at an IBM research center during a 5 day interval were considered. They found that one in five programs contained one or more errors which prevented compilation or assembly. Moulton and Muller in the study mentioned earlier found one in three programs contained such errors. The difference might be due to the fact that the programmers in the Moulton and Muller study were less experienced, but data reported by James and Partridge [4] on errors observed in programs, which included ones written by experienced programmers, also shows that one in three programs "would have been rejected by a conventional computing system". On the other hand results provided by Kulsrud [6] with experienced programmers is somewhat at variance with these results. Kulsrud's results indicate that only about one in eight programs have errors preventing compilation. (I have inferred this from his data, but he did not claim it.) His data also suggests that half of the discovered errors were language errors, and the other half were specification errors. Other data that

seems inconsistent appears in the paper by James and Partridge [4] in which they say they observed three errors per thousand statements. (It is likely they are referring to simple typing mistakes only).

The frequency of residual errors in programs, errors which remain after the program is put into regular service, is difficult to estimate. Reports of incorrect results from programs which have been used successfully over a period of years are not uncommon but they may not reflect program errors: they may be due to changes in use or environment not accounted for in the specifications. But data which is available suggest that a residual error frequency of five errors per thousand statements would not be unusual. In Youngs' study it was found that 27 errors out of an original 383 errors remained in the collection of programs after they had been compiled and corrected ten times; there were about 3500 statements altogether, implying a residual error frequency of about eight errors per thousand statements. In Gannon's study twenty four errors in about 4800 statements were never found implying a residual error frequency of about 0.5 errors per thousand statements. It has been reported [11] that IBM-TSS in its twentieth revision had "12,000 distinct new bugs" implying a residual error frequency of more than four errors per thousand statements. Finally, in a recent study [12] in which a program for the simulation of radar reports was carefully examined, after this program had been used for three years, an average of ten errors per thousand statements was found.

3.2 Errors produced by simulated mistakes. As noted earlier there are advantages in the use of simulation for the study of the nature of errors in program text. The basic idea is to simulate certain types of mistakes made by programmers and then to observe the nature of the errors caused by these mistakes. In experiments of this type we can measure the influence of language design and programming style on the nature of errors caused by human mistakes in programming.

Surprisingly there is almost no work of this type reported in the literature. Some years ago Weinberg and Gressett [13] made a study of the effectiveness of a FORTRAN compiler in detecting errors caused by simulated typing mistakes. However the scope of their experiment was very limited and there was no analysis of errors according to type.

Recently I have conducted experiments to measure the distribution of the kinds of errors caused by simulated typing mistakes in FORTRAN programs [14]. The simulated typing mistakes were: substitution (e.g. DIG instead of DOG); deletion (e.g. OIL instead of FOIL); insertion (e.g. FRIEND instead of FIEND); transposition (e.g. SILT instead of SLIT). Two kinds of substitution mistakes were simulated: nearest-neighbor, in which the substituted character is a nearest-neighbor of the correct character on the keyboard; and random, in which the substituted character is any character. In all cases the only characters considered were the characters in the FORTRAN alphabet. Four algorithms published in the ACM Transactions on Mathematical Software were used as subjects: Algorithm 495, Algorithm 498, Algorithm 505, and Algorithm 513. One thousand samples, each an algorithm with a single simulated typing mistake, were created. This ensemble had the following composition: for each algorithm fifty samples with each kind of mistake, thus 50 x 5 samples of an algorithm were created. For each sample the place where the mistake occured was selected at random, ignoring blanks and comments. Analysis of the one thousand samples yielded the distribution displayed in Fig.1.

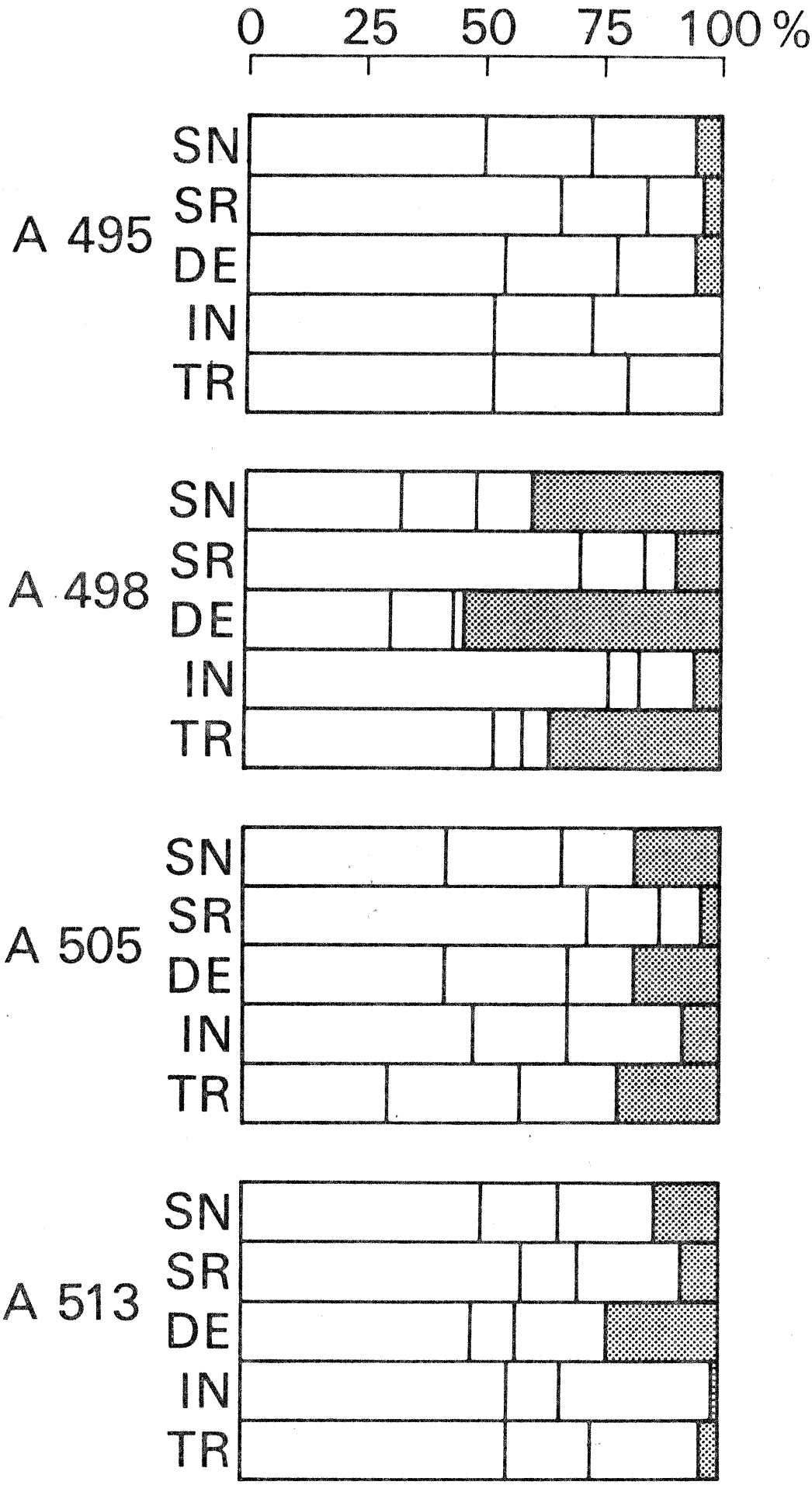
0
25
50
75
100 %
A 495
SN
SR
DE
IN
TR
A 498
SN
SR
DE
IN
TR
A 505
SN
SR
DE
IN
TR
A 513
SN
SR
DE
IN
TR

Figure 1. Distribution of errors caused by typing mistakes.

In this figure the bars are labeled SN for nearest-neighbor substitution mistakes, SR for random substitution mistakes, DE for deletion mistakes, IN for insertion mistakes, and TR for transposition mistakes. The bars of the figure are divided into segments corresponding to the different kinds of errors: the first is for syntax errors; the second is for semantic errors; the third is for simple anomalies, such as an identifier only occuring once; and the fourth, shaded, is for errors which are impossible or very difficult to detect at compile time. With these results as a basis for comparison we could, by performing similar experiments on programs written in other languages, determine whether the use of these other languages increases the likelihood that typing mistakes could be detected at compile time. It is evident that similar experiments in which other kinds of mistakes are simulated could be conducted in order to measure the influence of language design and programming style on the ease of error detection.

4. DETECTION OF ERRORS

This subject has been divided into static and dynamic error detection. A further distinction should be drawn between detecting an error and detecting the error, where the latter implies detection of an error and knowledge of what the construction should be. Detecting the error implies human interaction as normally takes place in proofreading text but this topic is not included here except for the short digression which follows.

Experienced programmers assume too much about the error detecting capability of the systems they use and are inclined to be careless in proofreading the text of their programs. The evidence presented in the last section supports this conjecture. Unformatted program text makes careful proofreading difficult and so an elementary but important tool for error detection is a simple text formatter such as POLISH [15]. One aspect of formatting peculiar to numerical software which is often ignored is the appearance of numerical constants. The designers of mathematical tables have long recognized that digits should be grouped and separated as in

1.32404 74631 77167
1.56663 65641 30231

to avoid transcription errors. This formatting should also be used for constants in computer programs for the very same reason. This point is worth emphasising because the placement of blanks or other separators, such as an underline, within constants is not permitted in all languages and represents a simple language design consideration which can have an important effect on numerical software reliability.

4.1 Static error detection. Compilers normally detect all syntax errors but are weak in detecting semantic errors expecially for programs written in

FORTRAN. Scowen [16] has constructed a small set of erroneous ALGOL and FORTRAN programs and attempted to compile them on various systems. Out of eleven examples in which a semantic error should have been found during compilation the IBM FORTRAN-G compiler did not detect an error in five of the examples.

In the experiment on typing mistakes [14] mentioned earlier, the mutilated programs were submitted to various compilers. All of the compilers considered (MNF, IBM FORTRAN-H, CDC FTN, WATFIV) were weak: MNF missed as many as 20% of the easily detectable errors; FORTRAN-H, FTN, and WATFIV missed as many as 40% of the easily detectable errors. MNF did detect almost all semantic errors but was weak in detecting simple anomalies which clearly signaled the presence of an error; FTN, FORTRAN-H, and WATFIV missed semantic errors and anomalies. Because of the recognized weaknesses of many compilers, and to reduce errors in moving FORTRAN programs from one system to another, a software tool called the PFORT verifier [17] was developed by a group at Bell Telephone Laboratories. It is widely used for static error detection; it is especially useful for detecting errors in the communication between subprograms and for the information it supplies on the utilization of identifiers.

Almost all static error detection systems, including the PFORT verifier, ignore path dependent anomalies or errors such as the statement sequence

```
X=1.0
X=2.0
```

or a statement sequence in which a reference is made to an uninitialized variable as in

```
SUBROUTINE XAMPL (X,Y)
K=1
IF(X .LT. Y) J=J+1
```

The detection of these requires the recognition of certain combinations of events, or their absence, on a path in the flow graph. Many anomalies and errors in this category can be detected statically without excessive cost. The problem is similar to that faced in global optimisation and, like it, can be treated by data flow analysis [18]. Osterweil and I have built a prototype system called DAVE [19] which uses data flow analysis to detect anomalies and errors in FORTRAN programs. In DAVE a program is represented by its flow graph and each node is labelled with information describing the actions taking place on variables at the node. Three actions, reference, define, and undefine, are recognized: "reference" means a value for the variable must be used, as for X in Y = X+1.0; "define" means a value is assigned to the variable, as for Y in Y = X+1.0; and "undefine" means a value for the variable becomes not known, as is the case for the loop control variable upon satisfying a DO loop in FORTRAN and all local variables upon entry to a subprogram. Error detection then depends on recognizing paths in the graph, including those which cross subprogram boundaries, which contain erroneous or anomalous sequences of actions. For example a path in which an "undefine" is followed by a "reference" without an intervening "define" is erroneous, and a path in which a "define" is followed by another "define" without an intervening "reference" is anomalous.

The critical part of data flow analysis is the search technique which is used to detect combinations of events on paths in the flow graph. DAVE uses depth-first search [20]. This has two advantages: the path containing the detected anomaly or error is obtained without extra work as a byproduct of the search and often the search can be terminated before the entire graph has been examined; it has the disadvantage that a separate search is required for each variable. The time for depth-first search is bounded by $K|V|\ (|E|+|N|)$ where K is a constant, $|V|$ is the number of variables, $|E|$ is the number of edges in the flow graph, and $|N|$ is the number of nodes in the flow graph. A different search algorithm described by Hecht and Ullman [21] which is iterative has the advantage that all variables can be treated simultaneously

and, assuming bit-parallel operations take one time unit, has a time bound $KD(|E|+|N|)$ where D, the number of iterations, is close to unity: this scheme is not favored with the two advantages just mentioned for depth-first search but it seems likely that this scheme will give a faster system in practice than one based on depth-first search because usually $|V| >> D$. DAVE executes about 50 times slower than the FTN compiler on a CYBER 175. This is too slow for general use. We are now building a new version, designed for efficiency, which uses the Hecht and Ullman algorithm and are hoping for a factor of ten improvement in speed.

There are several problems, some of which may prove intractable, that limit the effectiveness of data flow analysis. The address of the variable subjected to an action may depend on the computation as with A(J) in the statement A(J) = C. In some cases it may be possible, at least in principle, to determine the value of J from the program itself but in many cases J will depend on data supplied at the time of the computation: we can try to deal with this problem by looking for the possibility of an erroneous sequence of actions for some J but this approach produces false alarms. The practical solution in this case seems to be to issue an alarm if the error or anomaly will arise for every value of J. Another problem concerns flow of information across subprogram boundaries. In FORTRAN, because it does not use recursion, the tracing of data across subprogram boundaries is not too difficult, but in a language supporting recursion tracing data actions across these boundaries is quite difficult and no practical system in which this problem is treated exists. Recent theoretical work on this problem has been done by Barth [22]. A third problem is concerned with the fact that errors or anomalies on paths which are unexecutable cause false alarms unless the unexecutability of the path can be recognized. It should be noted that data flow analysis ignores the predicates associated with nodes in the flow graph and along some paths the predicates may be contradictory (e.g. $X \geq 0$ and $X < 0$) which would mean that the path would never be traversed in any execution of the program. A technique known as symbolic execution has been explored by Clarke [23] to treat this problem; Howden [24], Cheatham, Holloway, and Townley [25] and others have also been exploring this technique.

The essential idea of symbolic execution is to derive symbolic expressions to denote the values of program variables; for instance, from the statement sequence X = 1.0, Y = X+A the expression A+1.0 can be derived to denote the value of Y. Loops and branches cause obvious problems. In some applications the path is specified and these problems can then be avoided; this is the case, for example, in determining whether a particular path is executable. The difficulties of doing symbol manipulation and algebraic reduction on a machine are intrinsic problems of this approach and necessarily cause it to be expensive, but the possibility of being able to make inferences about a large set of computations makes this technique worthy of further study and development.

4.2 Dynamic error detection. In this approach, which takes place during program testing, errors are detected by examination of the results produced from executions of the program assuming an oracle exists for verifying the correctness of these results. An essential difficulty for this approach is that it is restricted to considering the results of a small number of executions because of cost considerations (note that static analysis implicitly takes into consideration many, if not all, executions but the results are less detailed) so it is important to choose the executions carefully in order that any errors which are present are likely to be exposed. Hence work in this area is concerned with measures of effectiveness of executions with respect to the probability of error detection, and with the mechanics of selecting input data sets associated with these measures.

Although numerous papers, e.g. [26,27,28,29], have appeared in recent years discussing measures of test effectiveness, the state of our knowledge of this subject is very unsatisfactory. There has been no significant work done to

relate any proposed measure to a probability of error detection; the argument for accepting a measure is completely intuitive. The simplest measure is the fraction, F_1, of nodes in the flow graph executed. This seems like a reasonable measure since errors at nodes not executed in testing will not be found; therefore we might expect that a higher F_1 would correspond to a lower chance of an undetected error. However this presupposes each node is equally likely to contain an error, and while this might serve as a convenient first approximation to the truth it is certainly of questionable validity. Another measure which has been proposed, and discussed in some detail by Huang [27], is the fraction, F_2, of edges in the flow graph which are executed. It is evident that $F_2=1$ implies $F_1=1$ but the converse is not true, so F_2 can be regarded as a stronger measure than F_1. Again the issue raised above about the relative importance of nodes applies to F_2. It is evident that one can build still stronger measures; for example, Woodward, Hedley, and Hennell [30] have suggested a hierarchy of such measures.

It is an interesting fact that these measures are not used in practice as one might expect. There are at least two situations in which substantial amounts of software are produced, software which is supposed to be reliable, where the value of F_1 at the end of testing is not known. In one situation human life depends on the reliability of the software. The developers of this software rely on their knowledge of the problem handled by the software, the special cases involved, and their experience, to create data for testing the software. In short the reliability provided by a guarantee that $F_1=1$ seems to be not worth the effort. It is easy to dismiss this attitude as foolish, but it would be wiser to examine carefully the reasons for it. Some of these reasons are: confidence based on intimate knowledge of the software that all nodes will be executed; the difficulty of finding test data to achieve $F_1=1$; the difficulty of testing after changes are made to the software to repair errors discovered in testing. Particularly important is the fact that a high level of trust is associated with correct results produced by the software on test problems. It would be worthwhile investigating how well placed this trust is. In a small example studied by Di Millo, Lipton, and Sayward [31] it was found that 8% of the errors deliberately inserted in a short (31 statement) FORTRAN program did not cause erroneous results on tests proposed by experienced programmers.

Two approaches to finding test data to achieve some level of testing, say $F_2=1$, have been discussed in the literature. As might be expected, one of these is based on random selection of test data [32,33]. For large programs this seems impractical unless some carefully designed stratified sampling is used and even then the costs may be too high. The other approach is based on symbolic execution [23,24]. From a theoretical viewpoint this is the more attractive but, as presently proposed, is too expensive to use in practice.

Mills, Gilb, Weinberg and others [11] have discussed seeding programs with known errors and then using the number of errors discovered in testing as a measure of the effectiveness of a test. This is a well-known practice followed by naturalists in studying wild-life populations but no one yet has provided convincing evidence that errors in programs obey the same statistical behaviour as fish in a pond. This approach may be effective but far more needs to be known about the nature of the distribution of errors in programs before it can be applied in a scientific manner. Recently Di Millo, Lipton and Sayward [31] have described a method of test data selection which is related to the error seeding idea. They measure the effectiveness of test data selection by the proportion of errors detected out of a group that has been deliberately inserted in the program.

Another technique advocated [34] for dynamic error detection is the instrumentation of the program with predicates at carefully chosen points in the program. Typically these predicates describe expected values for variables at these points. Like other techniques described here it does not appear to be widely used in practice though forms of this technique have been

used since the earliest days of computing to identify the location of a suspected error by tracing or monitoring an execution.

Finally, it should be mentioned that some efforts have been made to answer the question of whether the correctness of a program can be inferred from a finite set of tests [35,36]. Recently Howden [37] presented a result showing that for a particular class of Lindenmayer grammars it was possible to obtain an interesting correctness result based on a finite number of tests. Work of this kind, while not having any direct application to practice, should deepen our understanding of the testing problem and provide us with information on its limits.

4. CONCLUSION

We must assume that scientific and technological advances will lead to far more powerful computing machines than we have today and therefore that there will be a demand for larger and more complex programs. If this demand is to be met it will be necessary to improve our techniques for developing and maintaining large programs. Error detection is an important part of this and if we are to improve our error detection techniques, then we must know about the distribution of errors and how the distribution is affected by programming language, style, and human psychological factors; we must understand how to analyze programs and to measure their reliability.

As this brief review shows, our knowledge of the occurrence of errors is very limited. The experiments with human subjects are mainly limited to students and so it is questionable whether or not the results could apply to a laboratory or company producing software on a regular basis. Intuition suggests that modular or structured programming will reduce the occurrence of errors and improve our chances of detecting those that do occur. On the other hand there is evidence to indicate a strong correlation between the number of errors in a program and the number of bits required to specify it: this result comes from a theory called "software science" which has not been discussed here but has been reviewed recently by Fitzsimmons and Love [38]. Thus it appears that we still do not fully understand the factors which influence the occurrence of errors in programs.

There is now available a good set of algorithms for performing data flow analysis for non-recursive languages and it seems unlikely that there will be much more improvement in the basic algorithms. Here the work that is necessary for analysis of programs written in these languages is concerned with the implementation of these algorithms, and we need to understand how language features can affect the implementation. There remains a need for theoretical work on the analysis of recursive programs.

It is evident that the quantification of program testing is in a very primitive state. The measures which have been proposed have not been related to the probability of error. Furthermore, in practice the proposed measures are ignored and subjective evaluations are used in their place. There appears to be much room for theoretical and applied work in this area.

5. ACKNOWLEDGEMENT

This paper was written during a visit with the Numerical Algorithms Group Limited in Oxford, England. I thank them for their kind hospitality and I thank Eleanor Capanni of their staff for assistance in preparing the manuscript.

References

1. Gill, S.: The diagnosis of mistakes in programmes on the EDSAC. Proc. Roy. Soc. London 206A (1951), 538-554.
2. Gerhart, S.L.: Development of a methodology for classifying software errors. Final Technical Report (2 July 1976) Computer Science Department, Duke University, Durham NC 27706, U.S.A.
3. Horning, J.J.: What the compiler should tell the user. Lecture Notes in Computer Science, Goos, G. and Hartmamis, J. eds, Vol 21, Ch.5 (1974) Springer-Verlag.
4. James, E.B. and Partridge, D.P.: Tolerance to inaccuracy in computer programs. Comp. J. 19,3 (Aug.1976), 207-212.
5. Boies, S.J. and Gould, J.D.: Syntactic errors in computer programming. Human Factors 16 (1974), 253-257.
6. Kulsrud, H.E.: Some statistics on the reasons for compiler use. Software P.E. 4 (1974), 241-249.
7. Thayer, T.A.; Lipow, M.; and Nelson, E.C.: Software reliability study. TRW-SS-76-03 (March 1976) TRW, Redondo Beach, CA 90278, U.S.A.
8. Moulton, P.G. and Muller, M.E.: DITRAN - A compiler emphasizing diagnostics. Comm. ACM 10,1 (Jan.1967), 45-52.
9. Youngs, E.A.: Human errors in programming. International Journal of Man-Machine Studies 6,3 (May 1974), 361-376.
10. Gannon, J.D.: Language design to enhance programming reliability. Ph.D Thesis, Tech. Rept. CSRG-47, U. of Toronto (Jan.1975).
11. Gilb, T.: Software Metrics. Winthrop Publishers, Inc. (1977).
12. Benson, J.P. and Saib, S.H.: A Software Quality Assurance Experiment. Talk presented at NASA Workshop for Embedded Computing Systems Software, Hampton VA (Nov.1978).
13. Weinberg, G.M. and Gressett, G.L.: An experiment in automatic verification of programs. Comm. ACM 6,10 (Oct.1963), 610-613.
14. Fosdick, L.D.: The effect of typing blunders in FORTRAN programs. Tech. Rept. CU-CS-146-79 (Jan.1979) U. of Colo., Boulder, Colo.
15. Dorrenbacker, J.; Paddock, D.; Wisneski, D; and Fosdick, L.D.: POLISH, A FORTRAN program to edit FORTRAN programs, Tech. Rept. CU-CS-050-74 (July 1974) U. of Colorado, Boulder, CO 80309.
16. Scowen, R.S.: The diagnostic facilities in ALGOL and FORTRAN compilers. Tech. Rept. NAC 81 (July 1977), National Physical Laboratory, Teddington, Middlesex, England.
17. Ryder, B.T.: The PFORT verifier. Software P.E. 4 (1974), 359-378.
18. Fosdick, L.D. and Osterweil, L.J.: Data flow analysis in software reliability. ACM Comp. Surveys 8,3 (Sept.1976), 305-330.
19. Osterweil, L.J. and Fosdick, L.D.: DAVE - a validation, error detection and documentation system for FORTRAN programs. Software P.E. 6 (1976), 473-486.
20. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Computing (Sept.1972), 146-160.
21. Hecht, M.S. and Ullman, J.D.: A simple algorithm for global data flow analysis problems. SIAM J. Computing 4 (Dec.1975), 519-532.
22. Barth, J.M.: A practical interprocedural data-flow analysis algorithm. Comm. ACM 21,9 (Sept.1978), 724-735.
23. Clarke, L.: A system to generate test data and symbolically execute programs. IEEE Trans. on Software Engineering 2,3 (1976), 215-222.
24. Howden, W.E.: DISSECT - A symbolic evaluation and program testing system. IEEE Trans. on Software Engineering 4,1 (Jan.1978), 70-73.
25. Cheatham, T.E. Jr.; Holloway, G.H.; and Townley, J.A.: Symbolic evaluation and the analysis of programs. Tech. Rept. TR-19-78 (Nov.1978), Harvard U., Cambridge MA.
26. Hennell, M.A.; Woodward, M.R.; and Hedley, D.: On program analysis. Information Proc. Letters 5,5 (1976), 136-140.

27. Huang, J.C.: An approach to program testing. ACM Comp. Surveys 7,3 (Sept.1975), 113-128.
28. Pimont, S. and Rault, J.C.: A software reliability assessment based on a structural and behavioral analysis of programs. Proc. 2nd Int. Conf. on Software Engineering (Oct.1976) 486-491. San Francisco CA. IEEE Cat. No. 76CH1125-4C.
29. Brown, J.R.: Practical applications of automated software tools. Tech. Rept. TRW-SS-72-05 (1972), TRW, Redondo Beach, CA 90278.
30. Woodward, M.R.; Hedley, D.; and Hennell, M.: Observations and experience of path analysis and testing of programs. Tech. Rept. (1978), U. of Liverpool, Liverpool, England.
31. Di Millo, R.A.; Lipton, R.J.; and Sayward, F.G.: Hints on test data selection: help for the practicing programmer. Computer 11,4 (April 1978), 34-41.
32. Ramamoorthy, C.V. and Ho, S.F.: On the automated generation of program test data. Proc. 2nd Int. Conf. on Software Engineering (Oct.1976), Supplement 95-102. San Francisco, CA.
33. Hennell, M.A.; Woodward, M.R.; and Hedley, D.: Towards more advanced testing techniques. Tech. Rept. (1978), U. of Liverpool, Liverpool, England.
34. Stucki, L.G. and Foshee, G.L.: New assertion concepts for self-metric software validation. Proc. Int. Conf. on Reliable Software (April 1975), 59-71, Los Angeles CA, U.S.A. IEEE Cat. No. 75CH0940-7CSR.
35. Goodenough, J.B. and Gerhart, S.L.: Towards a theory of test data selection. Proc. Int. Conf. on Reliable Software (April 1975), 493-510, Los Angeles CA, U.S.A. IEEE Cat. No. 75CH0940-7CSR.
36. Howden, W.E.: Elementary algebraic program testing techniques. CSTR 13 (Sept.1976), Dept. Applied Physics and Information Science, UCSD, San Diego, CA.
37. Howden, W.E.: Lindenmayer grammars and symbolic testing. Information Processing Letters 7,1 (Jan.1978), 36-39.
38. Fitzsimmons, Ann and Love, Tom. A review and evaluation of software science. ACM Comp. Surveys 10,1 (March 1978), 2-18.