FORTRAN Error Detection Through Static Analysis

by
Irv K. Wendel and Richard L. Kleir
SWAK
P. O. Box 1440
Oakland, California, 94604

Introduction

As software becomes the major computer expense [1], the consequences of program malfunctions increase dramatically. Software errors can be traced from the original concepts through the program's useful life [2,3,6]. Historical evidence suggests that most programs are never completely debugged since maintenance and extension efforts contribute new problems [3].

Software failures beget both tangible and intangible costs. Direct costs include wasted human and computer time. Indirect costs (consequential damage) such as schedule slippage, unavailable results, lost confidence in the system, and possible damage to customer relationships, frequently exceed the direct costs.

At the present time, there exists no technique which can guarentee total certification of sizable software systems. Techniques have been developed which significantly improve software reliability. These include management methods [9], development disciplines [10,13] and automated tools [4,5,6,7, 8].

Static analysis is an extremely valuable examination tool. By automating many of the time-honored manual inspections, software errors can be discovered or avoided before they become catastrophic program failures. Using static analysis, a small team can effectively deal with large programs at a moderate cost.

Problem Sources in FORTRAN Software

To many, a program error is defined as a mathematical deviation of results from a known equation. With perfect hindsight, the problem is described in terms of how the program should have been created to avoid the problem. While many errors can be circumvented in the design and initial implementation stages, actual error sources are more pervasive; simply employing unusually astute programmers or using a "more powerful" language does not provide a complete solution.

The FORTRAN language. FORTRAN is one of the most widely used and mature programming languages. While some features could surely be improved, the wealth of qualified programmers greatly influences selection of FORTRAN for software implementation. The problem is more often how to implement reliable FORTRAN programs rather than should FORTRAN be used.

A common complaint is, "If you restrict the code to ANSI Standard constructions, the program will be more reliable, transportable, understandable, etc." Why, then, is there such massive deviation from "Standard FORTRAN"? First, the standard document [11] hardly qualifies as light bedtime reading material. The standard is most understandable to the computer population conversant with compilation and language implementation techniques rather than the practicing engineer using FORTRAN for structural analysis of buildings. The pure applications programmer relies upon the compiler to inform him of language deviations and acceptable constructions; his reference is the FORTRAN User's Guide rather than the ANSI Standard.

Most compilers are "compatible with the ANSI Standard." That is, they accept ANSI Standard constructions as well as dialectic extensions. In many cases, extensions provide access to unusual hardware features and compact expression of computational procedures. Is it reasonable to exclude a labor saving extension from the resident staff solely for conformance to the standard? In point of fact, many programmers experience serial monogamy* with their installation; the resident dialect is THE FORTRAN which is acceptable. In numerous cases, compiled code is tested to establish expected behavior. Using these unspecified "curiosities" as features sets the stage for future errors simply by changing compilers [12].

The FORTRAN ANSI Standard is a consensus document prepared to codify common usage and extend the existing definition of FORTRAN constructions. While the current document represents a meritorious effort by numerous contributors, some shortcomings are apparent:

1. The standard specifically addreses the interpretation of FORTRAN constructions and not their implementation [11 Section 1.2]. Issues of compilation and link-loading are not addressed; error potential exists in the processes.
2. The standard does not prescribe "the results when the rules for interpretation fail to establish an interpretation for such a program." That is, where dynamic events in execution violate preconceived conditions, program behavior is "undefined." For example, a file can be marked using an "ENDFILE u" statement; unfortunately, if the endfile records is processed by a READ, "action is undefined" [11 Section 7.1.3.3.3].
3. As the language evolves, new versions of the standard must maintain infrequently used constructions for backward compatibility with existing systems.

---

*Serial monogamy is a form of polygamy in which one is dutiful and faithful to one spouse for a limited period of time. After separation from one spouse, a new spouse is adopted with honorable intentions.

Many programmers live long and happy lives coding FORTRAN without ever using (or understanding) VARIABLE FORMAT or EXTERNAL statements.

It is not the purpose here to vilify the FORTRAN Standard. Rather, we should realize the utility limitations of any such document. A common, unified description of any language is clearly desirable; work in progress on the new draft proposal for FORTRAN is encouraging [15]. We can expect slow steady progress in this area as conflicts arise and are resolved. In the meantime, we will likely gain as little from blindly following "Standard FORTRAN" as from the arbitrary elimination of the GOTO.

Operational Problems. Foibles of FORTRAN coupled with the traditional compile/link/load operations create an environment conducive to undetected program errors. Since the ANSI Standard does not address the issues of compilation or link-loading, the programmer is left to his own wits for a large class of errors located at module interfaces.

FORTRAN requires each module to explicitly specify interface components of COMMON data and passed parameters. Normally, the compiler is structured to process one module at a time; thus, global scope is usually lacking to enforce interface compatability. Although the ANSI Standard specifies that actual and dummy parameter lists must agree in order, number, and type [11 Section 3.4.2], this requirement is rarely enforced by compilation or loading routines. Similarly, the structure and texture of COMMON blocks may be changed from routine to routine. With untested interfaces, the programmer is required to detect and correct these errors from execution malfunction symptoms.

The ANSI Standard specifies 31 Intrinsic Functions and 24 Basic External Functions; additionally, most installations provide a lengthly list of similar library routines available to the programming population. Few programmers are aware of all these names which might have special significance; indeed, for a given application, only a few of these predefined routines are needed or desired. This "convenience," however, is not without complicating side effects.

For example, suppose on a large programming project, two programmers are working in concert. One programmer creates a module called "ZARB" and another wishes to use the library function "ZARB." When their code sections are tested independently, both appear to work correctly. When they are integrated, however, the system fails because the second programmer's references to ZARB are directed to the first programmer's routine ZARB rather than the desired library function. Since this situation will usually not create any diagnostic, the search for the error can be tedious and frustrating; discovery may promote physical violence.

The preceding example is indicative of a larger problem--human communications. Computer programs express both human-machine communications and human-human communications. Unfortunately, programmers are more heavily trained in expressing their desires to the computer than coding for comprehension by another programmer.

Much attention has been directed lately to the problem of specifications. We hear over and over how specifications are incomplete or ambiguous--much of this criticism is justified. Unfortunately, the specification, as a document of functional intent, will always require some level of interpretation by individual programmers to convert the intent into a functional reality. The broader the specification, the more interpretation required; the more detailed the specification, the more program-like it becomes.

While the pursuit of error-proof programming and documentation style is just beginning, it is clear that there is much to be gained through reducing the arbitrary nature of individual style. Poor documentation and tricky coding sequences may not impact the immediate performance of a software system, but they lay the foundation for a host of future errors in maintenance and extension. Techniques are needed to surface obtuse coding patterns before they can affect future reliability.

Temporal Problems. Programs would probably be much more reliable if they could be implemented instantaneously. Unfortunately, any sizable system comes into being over a rather long period of time. Not only must the detailed construction work be done, but rarely does the user population really know what is wanted until some experience is gained in using the system.

During this extended creation period, the programming staff is far from stable. Programmers are promoted or leave for other jobs, errors are uncovered in the original system, specifications are modified, and sections of code are reworked to increase efficiency. Programs which are changed by personnel other then the original authors are subject to "historical legacies" or "software noise." In the simplest form, these are simply residual calculations or code sequences having no current purpose. Maintenance personnel are reluctant to remove these sequences for fear that they might contribute to some unknown mystery functions. For example, one can cease to use the value of a COMMON variable and calculate a new value locally, but rarely will a value assignment to the COMMON variable be deleted for fear it might be used elsewhere. As a result, it is not surprising in a modified system to find many values computed for which there is no current use.

Since these sequences appear to offend little more than program efficiency, what urgency should be attached to their removal? Historical legacies confuse the main issues; confused people make more mistakes. The additional code bulk conceals major program flow patterns. The inability to discover the purpose of a calculation creates doubt in the completeness of the documentation. Questionable documentation is quickly discounted by maintenance personnel and rarely updated. Thus, historical legacies have a detrimental cascade effect.

Clerical Errors. While clerical errors are not as exciting as algorithm failures, they are perhaps more common. While most of the clerical errors are discovered during compilation or unit test, they may creep in after initial program creation. Clerical errors include:

1. Keypunch errors (particularly those which result in valid FORTRAN statements).

2. Medium malfunctions. Grease spots and pin holes in optically read cards or magnetic parity errors, etc.

3. Modification errors. Cards inserted in the wrong location, misdirections to text editors which cause incorrect deletion or insertion in existing modules, etc.

Once a clerical error passes the fine inspection of unit testing, it quickly becomes hidden in the source code bulk.

Efficiency. Ironically, the pursuit of efficiency is a common source of program problems. Efficiency is the sacred goat of the computing profession. Programmers are pounded with the efficiency concept from their first computer exposure. Unfortunately, efficiency is normally expressed as a function of compute time and memory space (about the only two software attributes which can be readily measured); rarely does the efficiency formula include components of expended human labor or cost/benefit trade-offs.

The blind quest for efficiency often leads to convoluted programming practices and overbroad assumptions of the computation environment. Before a program is implemented, we usually cannot predict where most of the computing effort will be centered [17]; often we are surprised to find where the effort is actually expended. Similarly, we are often surprised to find that convoluted routines are sometimes slower than straightforward code [16].

As we can see, the sources of FORTRAN programming errors transcend the language features to include both computer system operations and software project work habits. An effective strategy for early discovery and error prevention must include these considerations. We need to anticipate problems, incorporate past error history, and codify our approach in an evolving fashion. Our purpose then becomes the eradication of costly commonplace errors so that we will have sufficient resources remaining to attack problematic errors for which there are no simple current solutions.

Static Analysis

Definition. Static analysis is the automatic investigation of software without code execution. Static analysis systems are programming aids which produce reports indicating areas for human evaluation.

Static Analysis Systems and Compilers. Static analysis systems and compilers both require interpretation of source code syntax and semantics. The primary purpose of a compiler, however, is the production of efficient object code; other processing is incidental. Since compilation is a very frequent activity, it is desirable to optimize compiler performance. Including static analysis activities in the compiler would substantially slow the compile process. In order to maximize compilation speed, compilers are constructed to "forget" source code details as quickly as possible. Static analysis, on the other hand, consciously records as many source code features as possible to facilitate anomaly discovery.

Diagnostic compilers, such as AID [14], offer a middle ground alternative. These systems can detect many FORTRAN errors as the compilation process proceeds. To accomplish this processing, certain germane source code characteristics are maintainted along with the output code. As references are made to previously encountered modules, the characteristic data base is examined for potential hazards. Unfortunately, most diagnostic compilers require simultaneous access to all program modules for full error detection; they normally do not maintain the diagnostic data base beyond one compilation. Thus, previously checked out routines must constantly be resubmitted for compilation (diagnostic data base creation) to achieve the desired diagnostic effect.

Diagnostic features are usually deeply imbedded in the compilation process. As a result, it is difficult to adapt these checks to a particular situation or incorporate new investigations without fear of disrupting the compile process.

Using a separate static analysis procedure totally divorced from the compile process permits independent and custom controlled interrogations into software structures. Since FORTRAN constructions may be subject to differing interpretations, independent interpretation may suggest possible trouble spots. By maintaining a data base of previously investigated routines, new modules can be compared to existing modules without reparsing.

The most effective system appears to be a combination of compile diagnostics, to the extent they are available from resident compilers, augmented by separate static analysis investigations. Using this approach, many of the existing cracks in the computational environment can be sealed by one system or the other. Duplication of effort is avoided while maintaining diagnostic coverage.

Application of Static Analysis. Static analysis begins by determining if the source code is legal, i.e., compilable. Here, static analysis is quite similar to compilation in purpose and function.

Static analysis tools locate unusual source code constructions. These eccentric language constructions may be manifestations of actual errors or they may merely represent weak programming practices. An example of the latter is an uninitialized variable. In almost all instances, an uninitialized variable is an out-and-out program bug. Yet all storage used by the program may be set initially through job control language, thereby initializing every variable in the program; the variable is no longer uninitialized, and the program may execute smoothly. This is "an accident waiting to happen."

Perhaps the most important capability of static analysis tools is their ability to pinpoint anomalies that require inordinate human effort to uncover. By using static analysis, the debugging phase of program development speeds up greatly, and the specter of schedule slippages and cost overruns is diminished.

Such systems can address a multitude of debugging areas. The arduous human debugging tasks involving path tracing can be reduced. Intermodule boundaries can be effectively examined for anomalies--a difficult and time consuming human task. An example of a boundary problem is a subroutine with four parameters, and a reference to that subroutine with only three. Most compilers and link-loaders will not sound the alarm, and the program may execute to normal termination. By performing

a checklist search for unusual language constructions, static analysis tools easily and cheaply uncover such anomalies.

Another aspect of static analysis is its value during the maintenance phase of a program's life cycle. Modifications often leave historical legacies in their wake, as well as creating the same type of errors produced during the development phase. Examples of historical legacies are variables assigned values but never used, COMMON variables never referenced, unreferenced FORMAT statements, unreferenced statement labels, and unreachable code which has been "short-circuited." Historical legacies, at the very least, cause confusion; confused people make mistakes.

A very important static analysis capability is automatic documentation. Cross-reference maps, calling hierarchy tables, COMMON occurrence lists, etc., facilitate any human interaction in a program's life cycle. Yet computer-generated documentation has its greatest impact during maintenance. Often maintenance personnel are not involved with the development phase and therefore are program novices. The maintenance staff needs help, especially since the maintenance effort often exceeds original program generation by 300%. During maintenance, new and different tools are necessary to quickly determine the ramifications of proposed modifications. For example, in certain instances it would be extremely useful to quickly find all locations a given COMMON variable is used or to know at a glance if a given parameter is an input, output, or input/output parameter. This valuable documentation can be generated by hand, but is more quickly and accurately produced by the computer.

Quality control is another forte of static analysis. As a management tool, static analysis systems can enforce in-house programming standards. A number of software divisions currently use editors to search for outlawed language constructions; unfortunately, editors often cannot achieve the desired granularity available with static analysis.

Finally, the internal mechanisms of a static analysis system can be extended to form an extremely powerful software evaluation system [4], including the base for dynamic analysis.

Processing Sequence.

1. The static analysis system first determines if the examined source code is compilable and flags all non-conforming source.
2. Compilable source code is separated into its component parts which are in turn placed into a data base. The data base contains all relevant components of the source as symbolic entities, recording such vital information as flow of control and intermodule relationships.
3. The static analysis system interrogates the data base, searching for source code anomalies, information for automatic documentation, etc.
4. Information gleaned from the data base is reported to the user.

Desired Traits. Productive static analysis tools share a number of characteristics, without which the utility of such tools may be severely restricted. These traits are:

1. Ability to process standard language features and common extensions, such as the FORTRAN two-way branch logical IF and IMPLICIT typing. Furthermore, processing should not collapse due to any user input. Ideally, a user should be able to randomly select punch cards from a trash can and input them as data without any card or combination of cards forcing processing to cease.
2. Powerful, human-engineered control mechanism. The mechanism should be simple to use and allow the user granularity of control. For example, the user should be able to easily specify whether a given module or an entire program is to be processed, as well as which variable is to be investigated. The user should be able to expressly indicate which anomaly types the static analysis tool is to search for, as well as use a default control mechanism. Moreover, the user should have a measure of control over the report format.
3. Human-engineered report format. A straight forward, concise report is a user's dream; an unintelligible report, an albatross around a user's neck.
4. Expandability. Static analysis tools should be readily modifiable, to react to a changing environment. Examples are the need to search for an unforseen anomaly, to modify the report format, or to process an unusual language extension, such as Univac's FLD function (a FORTRAN bit-manipulating routine).
5. Moderate processing cost. Static analysis systems automate part of the debugging, documentation, and maintenance activities. If the dollar and time benefits of such a system do not outweigh those of strictly manual inspection, then that system is not cost-effective.

Example

One static analysis system currently in use is FACES (FORTRAN Automatic Code Evaluation System) [4]. The initial application of FACES at NASA has unearthed approximately 1 error per 200 FORTRAN statements. The user spent approximately 10 minutes assessing each error.

Conservatively estimating the cost of software construction at $10/statement, a 20,000 statement program would cost $200,000. Projecting the FACES error discovery rate onto a 20,000 statement program, FACES could be expected to reveal 100 errors. Conservatively estimating the direct costs of manually rectifying program bugs at $100/error results in a minimum projected cost of $10,000. Since current FACES processing costs are slightly less than 10¢/statement at a commercial service bureau, the aforementioned program could be examined for approximately $2,000. This is a savings of approximately $8,000 over strictly manual investigation. Indirect costs are generally significantly higher than direct costs and may easily force the actual debugging costs to increase dramatically. Furthermore, there is no way of measuring the effect on intangible assets, such as the loss of customer confidence. The overall impact of FACES can be immense.

Certain types of errors detected by FACES are presented through the following code sequences.

Example 1)

```
      SUBROUTINE SUB ( A, B )
      TILE = A * B
      B = TILL
      RETURN
      END
```

This example displays an uninitialized variable and its counterpart, a variable assigned a value but never used. Such anomalies are often created by keypunch errors, and perpetually occur in software. Of course, keypunch errors lead to other types of debugging errors as well.

Example 2)

```
      SUBROUTINE SUB1
      COMMON /COM/ A(10), B(8), C(10)
         .
         .
      GO TO 200
      J = I + K
         .
         .
      SUBROUTINE SUB2
      COMMON /COM/ A(10), B(8), C(9)
      GO TO 300
100   A(1) = 6.
300   RETURN
      END
```

Certain errors are more likely to crop up during the maintenance phase, where costs can readily outgrow the original construction cost or purchase price. One such habitual error is unreachable code. Another, COMMON Block misalignment, occured in tested code when every COMMON declaration was modified but one.

Example 3)

```
      SUBROUTINE SUB1
      COMMON /COM/ A(2,5), B, C
         .
         .
      SUBROUTINE SUB2
      COMMON /COM/ A(5,2), C, B
         .
         .
```

Transposition errors, so apparent above, can simply disappear in the labyrinth of copious source code.

Example 4)

```
      SUBROUTINE SUB
      IMPLICIT INTEGER (A-Z)
         .
      D = DIST( FOOT, INCH )
         .
         .
      FUNCTION DIST( X, I )
      V = X + 2
         .
         .
```

Type mismatching can be a subtle error, especially when secreted in parameter lists. Rare is the FORTRAN programmer who has not referenced an integer function thinking it had type real or vice versa. This mistake is generally good for a week out of every programmer's life. With a number of programmers assigned to a software system, a lack of perfect communications increases the possibility of the aforementioned malfunction.

Benefits and Comparative Advantages of Static Analysis

Principle Advantages of Static Analysis

Static analysis is a comparatively inexpensive method for the detection of errors. It is well suited to the current working environment and requires no changes to the work habits of professional programmers. Among the advantages of static analysis are:

1. Minimal additional effort. Static analysis can treat software systems without modifying the source code or requiring auxiliary directives. Test data and driver/display code is not needed to obtain results. The programmer is not distracted from his principle duty--examination of the program under investigation.
2. Treatment of unrunnable systems code. Static analysis can be applied to incomplete systems without the need for stubs or dummy programs. A set of programs which utilize common subroutines can be analyzed as a single unit. Errors can be detected in programs which cannot be made to run or hang-up at strategic points.
3. Multiple errors detected in single run. Analysis is not thwarted by the appearance of a single error. The entire software system can be uniformly analyzed in one run. Static analysis users usually receive substancial "food for thought" from one pass.
4. Adaption and extension. Static analysis, at any point in time, includes the investigations known to benefit the inspection of software systems. As new hazards are found, the analysis features can be extended in the investigation repertory. The new hazards can be quickly identified in previously processed software systems.

Disadvantages of Static Analysis. While static analysis is a valuable technique, some problems persist in its application.

1. Limited interpretation of dynamic events. Static analysis may indicate a potential problem where dynamic program events preclude actual errors. Although source code is flagged, the program is constrained from malfunction by dynamics of operation. For example, in Example 5a, a "possible path" is found in which variable A is used but not defined. In the equivalent code of Example 5b, this problem would not arise. Short of simulating program operation or extensive logical analysis, this problem cannot be avoided in static analysis.

Example 5a)

```
      R = 0
      DO 50 I=1,100
C  ON FIRST ITERATION, SET A.
      IF (I.EQ.1 ) A=5
   50 R = A*( I-1 ) + R
         .
         .
```

2. Uncertainty in diagnostics. For maximum utility, static analysis should inform the user when a potential problem is detected. In many cases, there may be substantial question as to whether a hard error exists. Suspicious code is indicated by static analysis; error judgment is left to the numan investigator.

All error detection methods require some level of human evaluation -- static analysis is no exception. The

investigator must decide if the detected event is an error, poor coding, or application of a system nuance. This evaluation is more rapidly achieved when a specific condition is being evaluated and participating source lines clearly identified.

3. Sensitivity to coding style. The quality of results obtained from static analysis is in some measure dependent upon the quality and consistency of programming style. If programmers have used convoluted sequences or misleading FORTRAN techniques, static analysis can be thwarted or only marginal value may be obtained from copious problem indicators.

Comparison to Dynamic Analysis. Since most programmers are familiar with software testing through dynamic test cases, the properties of static analysis are compared to the features of dynamic analysis.

Dynamic analysis [7] is the evaluation of software systems by executing test cases on an instrumented source code version. Results of the test run are returned to the user for evaluation and verification of required program behavior.

Commonly implemented features in a dynamic analysis system include capabilities for:

1. Variable Value Tracing. Normally this is accomplished by monitoring the evaluation accomplished by each line of source text. Typically, bounds are returned to the user indicating the max, min, and average values for the test cases run.

2. Frequency of Execution. The frequency of execution of each program segment is recorded to indicate routines and lines within individual routines which contribute to the execution profile. With this information, the programer can detect source lines not yet tested, areas which will most impact efficiency through faster operation, etc.

3. Flow of Control Monitors. Through monitoring the direction of conditional branches, logic errors and incomplete program testing can be achieved.

4. Conditional Snapshots. The dynamic monitoring facility can be used to install sentries which report conditions when specific events are detected in execution. For example, preestablished bounds on the values assigned to given variables could trigger display of critical values.

Advantages of Dynamic Analysis over Static Analysis. Dyanmic analysis provides results not available through static analysis. These include:

1. Detection of data value dependent problems.

2. Utility for system tuning and efficiency improvement.

3. Determination of test case coverage in software certification process.

4. Automation of recertification after program modification and/or extension.

Disadvantages of Dynamic Analysis. Some disadvantages are:

1. Runnable configuration. Dynamic analysis requires a runnable configuration of software modules.

2. Expense. Run costs of dynamic analysis usually exceed those for static analysis.

3. More evaluation effort. The user spends more time determining if computed values are correct.

4. Influence of instrumentation. The overhead imposed by instrumenting the source code may influence program behavior. If an error source is related to memory allocation, as for an uninitialized variable, the introduction of additonal code may change the error symptoms. The error may in fact disappear entirely, only to reappear when the instrumentation is removed.

A core limited program may expand under instrumentation to the extent that it cannot be loaded. A time critical routine may not have sufficient execution margin to accommadate the data collection process.

5. Maximum benefit on well-behaved programs. Useful results from dynamic analysis may require orderly program termination. Perpetual loops or fatal execution interrupts may result in job termination prior to reporting the statistics collected.

6. Intimate system knowledge required. Since dynamic analysis is essentially a data creation and collection process, human attention is still required to determine the meaning of collected results. Correlation of expectations with experience requires considerable system knowledge.

7. Incomplete tests. Although progress is being rapidly made, dynamic test cases still constitute a collection of necessary but not sufficient conditions for software certificatrion. Clearly, all program paths must be exercised to certify the system; also, the simple execution of all paths does not guarantee proper performance for all data sets. Slight changes in input data can still cause system malfunctions, for example, when an expression used as a divisor evaluates to zero.

Conclusions

Static analysis cannot overcome the myriad problems associated with computer programming. Rather, static analysis is one powerful tool in a toolbox containing other potent programming aids, development disciplines, and management methodologies; these tools can increase productivity and help bring software costs under control. Static analysis is a current, cost-effective solution to current problems in today's programming languages.

We foresee software evaluation systems having major impact within the next few years. We also anticipate static analysis finding extensive use in the near future.

References

1. B. W. Boehm, "Software and Its Impact: A Quantitative Assessment," Datamation, May 1973, pp. 48-59.

2. B. W. Boehm, R. K. McClean, and D. B. Urfrig, "Some Experience with Automated Aids to the Design of Large-Scale Reli-

albe Software," Proceedings of International Conference on Reliable Software, April 1975, pp. 105-113
3. J. C. Dickson, J. L. Hesse, A. C. Kientz, and M. L. Shooman, "Quantitative Analysis of Software Reliability," Proceedings of Annual Reliability Symposium, IFFE, New York, January 1972.
4. C. V. Ramamoorthy and S. F. Ho, "Testing Large Software with Automated Evaluation Systems," Proceedings of International Conference on Reliable Software, April 1975, pp. 382-393.
5. D. J. Reifer, "Automated Aids for Reliable Software," Proceedings of International Conference on Reliable Software, April 1975, pp. 131-142.
6. J. R. Brown and R. H. Hoffman, "Evaluating the Effectiveness of Software Verification--Practical Experience with an Automated Tool," AFIPS Fall Joint Computer Conference, Anaheim, California, December 1972, pp. 181-190.
7. R. G. Stucki, "A Prototype Automatic Program Testing Tool," AFIPS Fall Joint Computer Conference, Anaheim, California, December 1972, pp. 829-836.
8. E. F. Miller and R. A. Melton, "Automated Generation of Testcase Datasets," Proceedings of International Conference on Reliable Software, April 1975, pp. 51-58.
9. F. T. Baker, "Chief Programmer Team Management of Production Programming," IBM Systems Journal, Volume II, No. 1, 1972, pp. 56-73.
10. F. T. Baker, "Structural Programming in a Production Programming Environment," Proceedings of International Conference on Reliable Software, April 1975, pp. 172-183.
11. USA Standard FORTRAN, American National Standards Institute, X3.9-1966, March 1966.
12. F. P. Brooks, The Mythical Man-Month, Addison-Wesley, 1975
13. H. D. Mills, Mathematical Foundations for Structural Programming, Report No. FSC-72-6012, IBM Corporation, Gaithersburg, Maryland, February 1972.
14. G. Dronek and K. Muehring, A Comprehensive Intelligent Debugger (AID), University of California, Berkeley, Report No. L2-CAL-AID, L3-CAL-AID.
15. "Draft Proposal ANS FORTRAN, BSR X3.9, X3J3/76," SIGPLAN Notice, Volume II, No. 3, March 1976
16. B. W. Kernighan and P. J. Plauger, The Elements of Programming Style, McGraw-Hill, 1974.
17. D. E. Knuth, "An Empirical Study of FORTRAN Programs," Software Practice and Experience, Volume I, 1971.