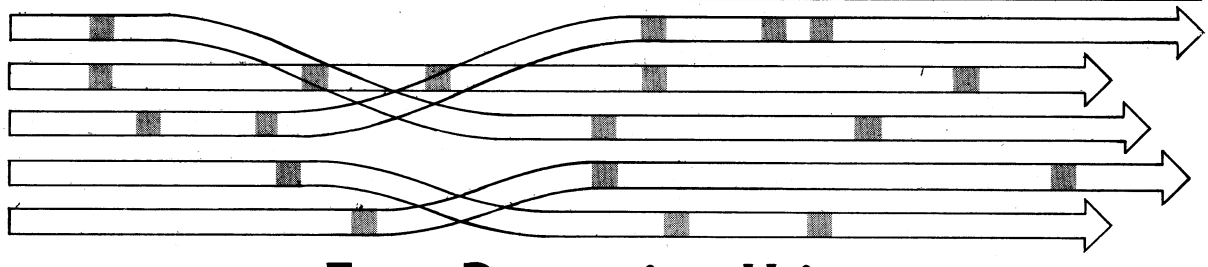
How many types of errors can be detected through static analysis and branch testing? How many man-hours and machine hours do these techniques require? Here are some empirically determined answers.



Error Detection Using Path Testing and Static Analysis

Carolyn Gannon
General Research Corporation

Two software testing techniques—static analysis and dynamic path (branch) testing[1]—are receiving a great deal of attention in the world of software engineering these days. However, empirical evidence of their ability to detect errors is very limited, as is data concerning the resource investment their use requires. Researchers such as Goodenough[2] and Howden[3] have estimated or graded these testing methods, as well as such other techniques as interface consistency, symbolic testing, and special values testing. However, this paper seeks (1) to demonstrate *empirically* the types of errors one can expect to uncover and (2) to measure the engineering and computer time which may be required by the two testing techniques for each class of errors during system-level testing.

The experiment

To provide the material for our testing demonstration, a medium-size, 5000-source-statement Fortran program was seeded with errors one at a time, and analyzed by a testing tool[4] that has both static analysis and dynamic path analysis capabilities.

The test object. The test object program was chosen carefully. In addition to size and functional variety, we needed a program as error-free as possible to eliminate camouflaging the seeded errors. The Fortran program[5] chosen consists of 56 utility routines that compute coordinate translation, integration, flight maneuvers, input and output handling, and data base presetting and definition. The collection of routines has been extensively used in software projects for over 10 years. The main program directs the functional action according to the user's input data. The program is highly computational and has complex control logic. Sample data and expected output were provided with the program's functional description, but specifications and requirements for the program's algorithms were not available. For this experiment, the correct output for the supplied data sets was used as a specification of proper program behavior.

Error seeding. Of the several studies describing error types and frequencies, TRW's[6] is the most relevant. We have used the Project 5 data from that report as the basis for selecting error types and frequencies. Several error categories (namely documentation, problem report, and "other"—which includes software design, compilation errors, and time or storage limitations) which were not relevant to the test object or test environment were omitted. Table 1 shows the types of errors represented by the experiment and summarizes the results. Error frequencies by major category are shown in Figure 1.

There are seven applicable major error categories. Table 2 shows the frequencies of errors by category for the Project 5 data and (adjusted for the omitted "other" category) the frequencies and number of errors used in the experiment. Inasmuch as the difference between columns 5 and 6 affect the evaluation of path testing, they require some explanation.

In addition to generating errors whose type and frequency have their bases in a published study, the location of each error and the program's resulting behavior were also prime concerns in maintaining an objective experiment. In the TRW study, no data linking the error type to software property (e.g., statement type) is presented. Using the error types

0018-9162/79/0800-0026$00.75 © 1979 IEEE

made it necessary to establish correlations between each error type and quantifiable test software properties. Furthermore, since the test object consists primarily of general utility subroutines, many having alternative segments of code whose execution depends upon their input parameter data, we felt that the errors should reside on segments of code that are executed by a thorough (in terms of program function and structure) set of test data, and that the errors should manifest themselves by some deviation in the program's normal output. To generate errors with these properties, the following steps were performed:

(1) The test software was analyzed by the test tool[4] to classify source statements, to obtain software documentation reference material (e.g., symbol set/usage, module interaction hierarchy, location of all invocations), to guide insertion of errors, and to generate an expanded set of test data that provided thorough path coverage. The percentage of path coverage varied from module to module depending upon the main program's application of the utility subroutines.

(2) A matrix showing error types versus statement classification was manually derived.

(3) The information from steps 1 and 2 was combined into a matrix showing potential sites in the software for each error type.

Table 1.
Error detection for each error type.

	Category	Static Analysis	Path Testing Phase	Errors Manifested In Output
A.	COMPUTATIONAL			
A100	Incorrect operand in equation		P,A	2
A200	Incorrect use of parentheses	S	P,0	2
A400	Units or data conversion error		I	1
A800	Missing computation		A,0	2
		1	5	7
B.	LOGIC			
B100	Incorrect operand in logical expression		P*	1
B200	Logic activities out of sequence		P	1
B300	Wrong variable being checked		I,I,P	3
B400	Missing logic or condition tests	S,S	P,A,0	3
B500	Too many/few statements in loop		P,A,A,0	4
B600	Loop interacted incorrect number of times (including endless loop)		A	1
		2	10	13
C/E.	INPUT/OUTPUT			
C200	Input read from incorrect data file		I,I	2
E200	Data written according to the wrong format statement	S	P	1
E300	Data written in wrong format		I	1
E500	Incomplete or missing output		I	1
E600	Output field size too small		I	1
		1	6	6
D.	DATA HANDLING			
D050	Data file not rewound before reading		I	1
D100	Data initialization not done	S	I	1
D200	Data initialization done improperly		I,I	2
D400	Variable referred to by the wrong name	S	A,0	2
D900	Subscripting error		P	1
		2	6	7
F.	INTERFACE			
F200	Call to subroutine not made or made in wrong place		P,I*	2
F700	Software/software interface error	S	I,I	2
		1	3	4
G.	DATA DEFINITION			
G100	Data not properly defined/dimensioned	S	I,I	2
G200	Data referenced out of bounds		A,I	2
		1	4	4
H.	DATA BASE			
H100	Data not initialized in data base		P,I	2
H200	Data initialized to incorrect value		A,I	2
H300	Data units are incorrect		P,I,A*,0	4
		0	6	8
		8	40	49

S = Static Analysis (16% of total errors)
P = Path Testing only (25% of total errors)
A = Inspection aided by Path Testing (20% of total errors)
I = Inspection only (41% of total errors)
0 = Error not detected (16% of total errors)
* = Error site located or improper correction made (counted as not detected)

(4) From the potential site matrix, a list of candidate error sites was randomly generated.
(5) At each site in the list either an error of the designated type was manually inserted or the site was rejected as being unsuitable for the error type.
(6) Errors were eliminated from the error set which caused a compiler or loader diagnostic.
(7) The 86 errors shown in column 5 of Table 2 were selected from the remaining errors using Project 5 error frequency data. Errors from this set were eliminated if they caused no change in the output. Fifteen errors were rejected due to lack of coverage with the test data, and 22 were eliminated for which coverage was achieved without affecting the output. The surviving 49 errors, shown in column 6, were used.

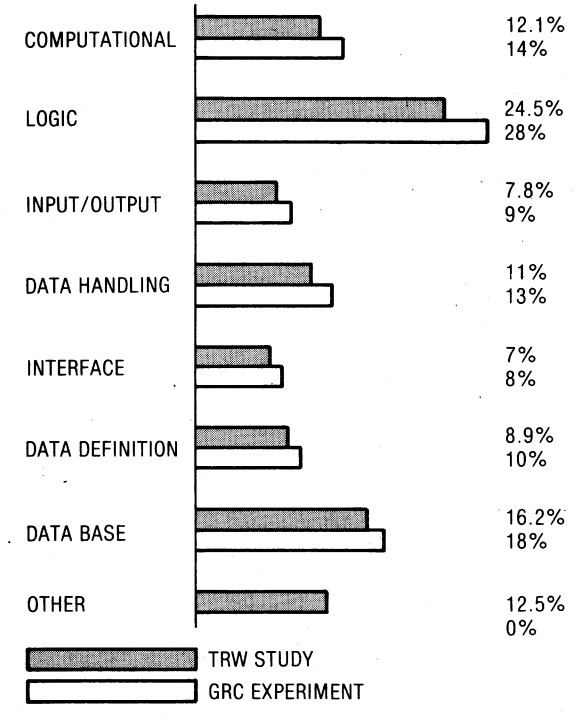


Figure 1. Error frequency in major categories.

Error site execution or reference was verified by an output message placed, for the case of executable statements, at the error site or, for the case of non-executable statements, at the site of reference by some executable statement on a covered path. The impact of this evidence is that path testing with the sole goal of execution coverage is *not* an adequate verification measure. Most software tool developers whose verification tools include a path testing capability advocate their usage with data that demonstate all specific functions of the software. Even then, stress and other performance testing should enter into the total test plan.

The experiment. Errors from the seven major categories were seeded, one at a time, into the Fortran program. The tester (not the error-seeder) was given a compilation and execution listing which gave no clues to the error's location. He was told what was wrong with the output and had, as a specification of proper program performance, a listing of the correct output. The task was to find the error using execution coverage analysis (path testing) or inspection, whichever seemed more appropriate, correct the source, and execute the corrected program to verify the output. Human and computer times were accounted for from the time the tester received the erroneous listing to the time he delivered the corrected listing.

To evaluate the types of errors detected by static analysis, all 49 errors were simultaneously seeded into the program after determining that they did not interfere with each other in the static sense. Only one computer run was required for this evaluation.

Experimental results

Unlike static analysis, which explicitly detects inconsistencies and locates the offending statement(s), path testing is a technique that demands skill to interpret the execution coverage data as well as to recognize improper program performance from the program's output. The analyst must determine, by inspection, which sequences of paths might be infeasible (unexecutable for any data) and which might

Table 2.
Error frequency in major categories.

(1) Project 5 Major Error Categories*		(2) Total Project 5 Errors*	(3) Percent of Errors*	(4) Percent Applicable	(5) Errors Generated	(6) Errors Manifested In Output
Computational	(A)	92	12.1	14	14	7
Logic	(B)	169	24.5	28	25	13
Data Input Data Output	(C) and (E)	55	7.8	9	9	6
Data Handling	(D)	65	11.0	13	10	7
Interface	(F)	48	7.0	8	7	4
Data Definition	(G)	62	8.9	10	8	4
Data Base	(H)	112	16.2	18	13	8
Other	(J)	86	12.5	—	—	—
		689	100.0	100	86	49

*Data derived from Table 4-2 of TRW study.

be critical from a functional point of view. Thus, effective path testing requires considerable knowledge about the program's intended behavior.

For the path testing evaluation phase, we found that the errors were located using three detection methods: path testing alone, inspection aided by path testing, and inspection alone. Some errors were easily detected without the necessity of instrumenting the code to get path coverage. Some errors were found when the path coverage reports narrowed the search to a set of suspicious paths—but then inspection was used to actually determine the error. Other errors were found directly by observing the control path behavior from the coverage reports and the path statement definition listing. In a few cases the wrong "error" was found and only some of the incorrect symptoms disappeared; these are noted in Table 1.

Figure 2 shows the frequencies of error categories detected by the methods described above. The dashed lines show the effect of some degree of path testing coverage by reporting the sum of path testing alone and inspection aided by path testing. As one might expect, logic errors and computation errors (since they often cause a change in control flow) are the best candidates for path testing. Errors in these two categories are often the most difficult to locate, unless a detailed design and specification are also available. Input/output and data definition errors are usually easily detected by inspection alone.

More comprehensive results are shown in Table 1. Note that not all error types were seeded into the program, owing to project limitations. For each error

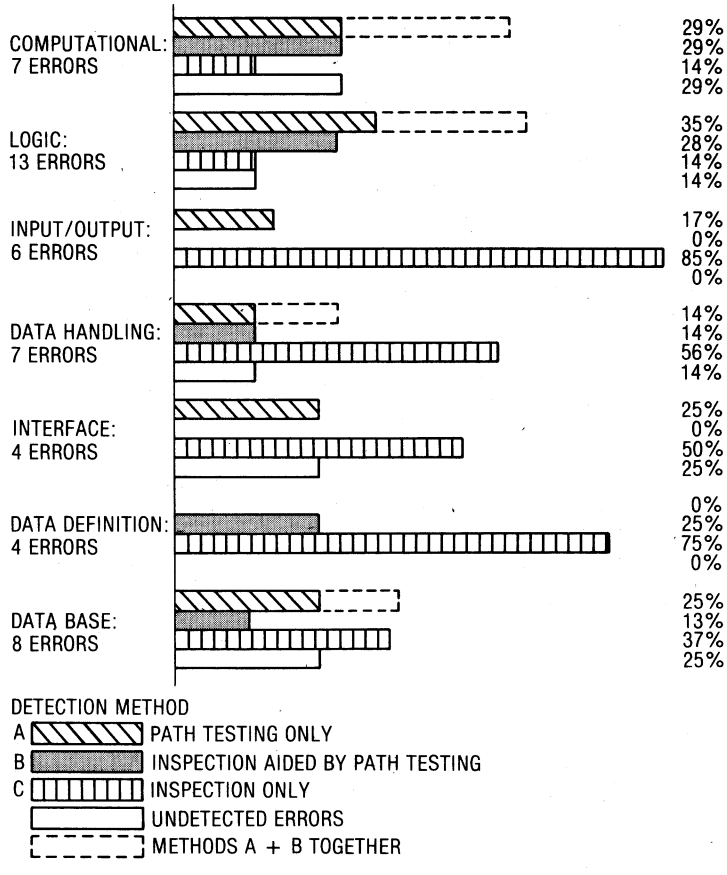


Figure 2. Path testing: frequency of detected errors by category.

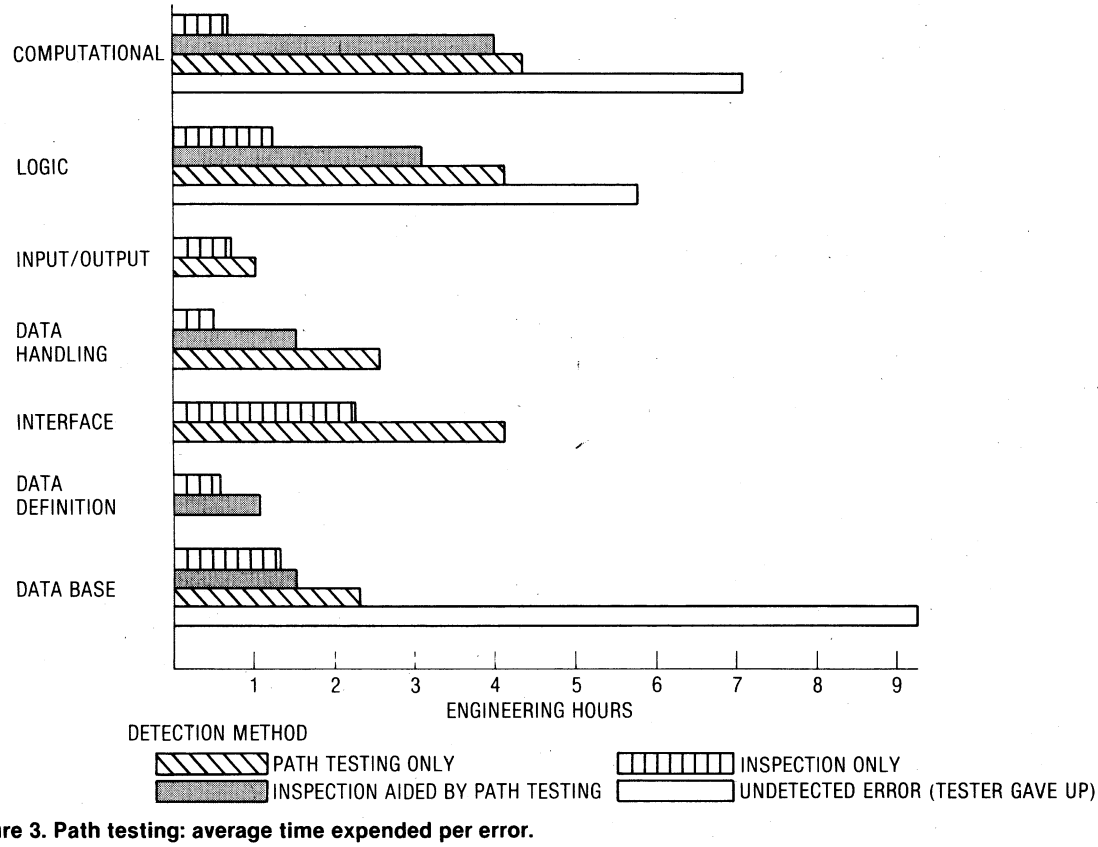


Figure 3. Path testing: average time expended per error.

seeded, Table 1 shows the technique used to detect it. An asterisk next to the technique's indicator signifies that the erroneous statement was located but the "correction" was not the proper one, or that more information (such as a specification) was needed to make the proper changes.

To assess the value of path testing, an account was kept of the resources expended. The average engineering time in hours for finding each error is shown in Figure 3. Most of the errors detected by inspection required only about 1½ hours to find and correct. On the other hand, the more difficult errors requiring path testing took about 4 hours.

Static analysis has capabilities for detecting infinite loops, unreachable code, uninitialized variables, and inconsistencies in variable and parameter mode. Some sophisticated compilers have a few of these capabilities. Until this experiment was conducted, we had no evidence concerning the kinds of errors that static analysis might find in programs of substantial size. As a preliminary task[7] to the experiment, we applied static analysis and path testing to eight small programs from *The Elements of Programming Style.*[8] In that task we found that static analysis detected 38 percent of the total 26 errors, and path testing found 70 percent. However, these eight programs contained numerous uninitialized variables which weighted the measurement in favor of static analysis.

In our experiment, static analysis detected 16 percent (8 errors) of the total 49 seeded errors. We chose not to use the static assertion consistency checking capability which is intended to detect inconsistencies in input/output parameters. When the experiment began, the assertions had to be manually inserted. We also had wanted to use the DAVE[9] static analysis tool, but the test object had too many violations of ANS Fortran to make the effort feasible. Figure 4 shows the frequency of detected errors by major category, and Table 1 lists each error type found by static analysis. One error detected by the graph checking capability of the static analyzer was unreachable code due to a missing IF statement. This error (B400) was not detected by either path testing or inspection. Unreachable code can be very difficult to locate in code filled with statement labels and three-way IF statements, as was the test object for the experiment. Unreachable code may or may not be harmless, but it is always a warning of possible dangers or inefficient use of computer resources.

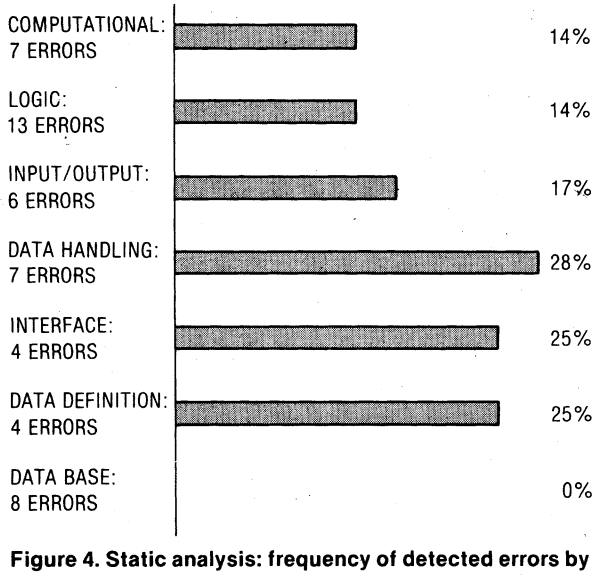


Figure 4. Static analysis: frequency of detected errors by category.

While static analysis did not detect a high percentage of errors, and while most of the errors it did find were also detected by path testing, it has the distinct advantage of being a very economical tool. Only two engineering hours and 24 seconds of CDC 7600 time were required to review the static analysis output and locate the errors. A disadvantage is that if programming practice allows frequent intentional mixed mode constructs or mismatching number of actual and formal parameters, the static analyzer will issue frequent warnings and errors (133 in our experiment) that are harmless to the proper execution of the program.

Conclusions

Both the error-seeding and error-detection activities of the experiment provided concrete data for several conclusions about the two testing techniques. While the experiment was designed and implemented in an objective manner and can be repeated by other interested researchers, it is not our intention to apply a metric or statistical significance to the error detection capabilities of the testing methods. It is our purpose, however, to report the types of errors that can be detected by these techniques. The results of the experiment can also be used as a reference for tool developers seeking to sharpen their tools for more rigorous error detection.

The first major conclusion of the experiment is that one can probably expect as many as 25 percent of the errors in a program of 5000 statements to slip through acceptance testing, if acceptance testing is based primarily on executing all possible paths at least once. It is possible that many of the errors are harmless in one specific application of a general-purpose program (e.g., incorrect computations are never used or are corrected before harm is done). It is more likely, however, that the data generated to satisfy path testing requirements (percentage of coverage) causes control flow to execute *sequences* of paths which do not exhibit the errors. This is one reason why path testing should *always* be coupled with stress or boundary condition testing. Overall path coverage may not be increased, but the *right* sequence of paths may be executed to expose the errors.

Of the two testing techniques, path testing is the most effective at detecting logic, computational, and data base errors. The last two error types are found when their incorrect results affect the control flow. In our experiment, path testing alone detected 25 percent of the seeded errors. This is consistent with the

21 percent (of 28 errors) that Howden stated could be reliably found by path testing.[3] Path testing together with inspection aided by path testing detected 45 percent of the errors seeded in the program. This percentage is lower than the 70-percent path testing error detection rate we encountered in testing of the eight programs from *The Elements of Programming Style.* However, in this experiment 41 percent of the errors were found by inspection, without the need to apply the path testing capability.

In a recent report,[10] Taylor lists useful testing techniques for each of the TRW error types. Our experience demonstrated that path testing is capable of detecting many more error types than he credited it with. However, his evaluation (not based on empirical data) is aimed at *grading* the usefulness of testing techniques. We set out to *measure capability.* The primary disadvange of path testing is that it is a time-consuming, demanding process that (as currently implemented) does not recognize "key" paths or important sequences of paths.

Static analysis is effective at finding data handling, interface, and data definition errors. According to the TRW study, these error categories usually constitute only about 20 percent of the total program errors. However, static analysis is very economical and should be utilized along with program compilation during the software development. ■

Acknowledgment

This experiment was sponsored by the Air Force Office of Scientific Research under contract F49620-78-C-0103. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon. Nancy Brooks performed the research on error type to software linkages and generated the candidate errors. Reg Meeson performed all path and static analysis testing for the 49 individual errors.

References

1. R. E. Fairley, "Tutorial: Static Analysis and Dynamic Testing of Computer Software," *Computer,* Vol. 11, No. 4, Apr. 1978, pp. 14-23.
2. J. B. Goodenough, "Survey of Program Testing Issues," *Proc. of the Conf. on Research Directions in Software Technology,* Oct. 1977.
3. W. E. Howden, "Theoretical and Empirical Studies of Program Testing," *IEEE Trans. Software Eng.,* Vol. SE-4, July 1978, pp. 293-297.
4. D. M. Andrews and J. P. Benson, *Software Quality Laboratory User's Manual,* General Research Corporation CR-4-770, May 1978.
5. T. Plambeck, *The Compleat Traidsman,* General Research Corporation IM-711/2, Sept. 1969.
6. T. A. Thayer, et al., Software Reliability Study, TRW Defense and Space Systems Group, RADC-TR-76-238, Redondo Beach, Calif., 1976.
7. C. Gannon, "Empirical Results for Static Analysis and Path Testing of Small Programs," General Research Corporation RM-2225, Sept. 1978.
8. B. W. Kernighan and P. J. Plauger, *The Elements of Programming Style,* McGraw Hill, 1974.
9. L. D. Fosdick and C. Miesse, *The DAVE System User's Manual,* University of Colorado, CU-CS-106-77, Mar. 1977.
10. R. N. Taylor, *Integrated Testing and Verification System for Research Flight Software— Design Document,* Boeing Computer Services, NASA Contract Report 159008, Feb. 1979.

Carolyn Gannon is a member of the Software Quality Department of General Research Corporation in Santa Barbara. For five years she has been involved in the development of software testing tools for Fortran, Jovial, and Pascal. Her technical interests also include research in software specification, design, and analysis.

Gannon received her BA in mathematics and MSEE in computer science from the University of California at Santa Barbara.