Towards automatic software fault location through decision-to-decision path analysis

by JAMES S. COLLOFELLO
Arizona State University
Tempe, Arizona
and
LARRY COUSINS
GTE Communications Systems
Phoenix, Arizona

ABSTRACT

Software development is a complex and error prone process. As a result of this process, much time is spent debugging software. This debugging process actually consists of two activities, fault localization and repair. For most problems, much of the debugging effort is devoted to fault localization. In this paper, current fault localization techniques are surveyed and a new technique called relational path analysis is proposed. Relational path analysis suggests that there exists information associated with stored execution paths of programs that, when analyzed heuristically, can localize faults with statistical significance. This paper presents a set of candidate heuristics for relational path analysis and the results of an experiment utilizing the heuristics. Conclusions regarding the effectiveness and usability of this technique and future research in this area are also discussed.

INTRODUCTION

Software development is a complex and error prone process. Although hardware costs during the last 30 years have consistently dropped, software costs have continued to climb.[1,2,3,4] One of the significant factors in these increased costs is the expense incurred performing software fault (error) localization and repair. As the complexity and size of software systems continues to increase dramatically, emphasis is needed on developing automated methods to help perform fault localization and repair activities.

It is important to distinguish between the terms fault localization, fault repair, and debugging. Myers defines debugging as:

> the activity that one performs after executing a successful test case [successful in the sense that it found a bug]. Describing it in more concrete terms, debugging is a two-part process; it begins with some indication of the existence of an error (e.g., the results of a successful test case), and it is the activity of (1) determining the exact nature and location of the suspected error within the program and (2) fixing or repairing the error.[5]

Thus, debugging entails both fault localization and repair. In the remainder of this paper, only the fault localization aspect of debugging is addressed.

CURRENT FAULT LOCALIZATION METHODS

Currently, many techniques and tools are used to perform fault localization. These methods can be classified either as knowledge-based or non-knowledge-based. Most of the existing fault localization approaches can be classified as non-knowledge-based. Over 100 such approaches were cited by Myers[5] in 1979. There are two common threads that most of these non-knowledge-based approaches possess. The first is that they usually provide powerful control over the program under test (e.g., symbolic execution of code). The second is that a user is required to provide all the intelligence necessary to guide and especially interpret the testing/debugging session.

Knowledge-based fault localization systems can be identified by their autonomous behavior. The systems themselves interpret the information they generate to localize faults; the information is not passed to a user for interpretation, as is the case in non-knowledge-based systems.

An example of a knowledge-based fault localization system is PROUST.[6] The goal of PROUST's designers was to create a framework sufficient to catch all possible errors in small programs. They also wanted the program to understand the nature of the bugs, state it, and suggest a form of solution. To accomplish these objectives, the system requires that the program be totally and correctly specified. The major practical limitation of this system is that it is extremely difficult to form such specifications even for small programs, and there is no way to guarantee the specifications are correct even after they have been stated.

Another interesting system that approaches debugging through the avenue of a knowledge-base is the Program Testing Assistant developed by Chapman.[7] The unique quality that Chapman's system possesses is that as programs are developed and tested, a user can request that the system automatically store the test cases for future use. When a bug arises in a feature being tested, the system in coordination with the user can request that the appropriate saved test cases be rerun automatically—either before the system has been repaired to aid in identifying the problem or after the system has been repaired to ensure its correctness. In conjunction with this capability, the Programming Testing Assistant heuristically modifies the corresponding test cases when the source code is changed. This preserves the ability of the system to continue to use, if possible, previous test cases to perform a type of automated regression testing of the code.

The major disadvantage of this system is that it only works with LISP code. Given the indistinguishability of LISP code and data, it may in fact only be practical with LISP. The major advantage is the way the system relieves users from having to manually save and execute test cases.

RELATIONAL PATH ANALYSIS

In this section a theory called relational path analysis is described that suggests that there exists information associated with the execution paths of programs which when analyzed heuristically can produce statistically significant fault locations.

The basis of this theory stems from analysis of DD-path (decision-to-decision-path) executions. A DD-path is a section of straight-line code that exists between predicates in a program. The theory suggests that a database of test cases that execute correctly can be utilized to locate DD-paths that contain faults in an incorrect program execution. To accomplish this objective, the database of test cases is supplemented to contain execution path information consisting of the DD-paths traversed for each test case. Various heuristics are then applied comparing the execution path for the incorrect program with those of correct program executions in an attempt to locate DD-paths on the incorrect program path which may be the source of the error.

In the remainder of this section, ten such heuristics based upon various strategies for program debugging are described. Each heuristic examines both the current execution path and the execution path database to identify DD-paths in the program which may contain the error. In the next section, experimental results utilzing these heuristics will be presented.

Heuristic 1

The first heuristic returns all of the DD-paths on the erroneous path that are not executed in any of the correct execution paths in the database. This heuristic is based on the theory that if a DD-path is traversed by an error-producing test case that has never been traversed before, it is likely to contain the error.

Heuristic 2

Heuristic 2 returns all the DD-paths that are elements of the erroneous execution path whose sum of all executions in the correct execution paths is less than or equal to 50. The number 50 was chosen as a possible lower bound for experimental purposes and certainly is not sacred.

The rationale for this heuristic is similar to that for heuristic 1 except that it was thought that the bound would allow flexibility in tuning the heuristic for finding different types of errors. Heuristic 2 recognizes that code paths may be executed several times correctly before some condition occurs that may create an error.

Heuristic 3

Heuristic 3 returns all the DD-paths that are elements of the erroneous execution path and whose sum of all executions, both correct and incorrect, is less than or equal to 50. Again, the number 50 is chosen for experimental reasons. Heuristic 3 is the same as 2 except that the sum of all test case executions is utilized.

The rationale for heuristic 3 is analogous to that for number 2, and is based on the idea that the fewer times a DD-path is executed, the higher the probability that it contains an error.

Heuristic 4

Heuristic 4 returns DD-paths in the erroneous execution path with the maximum ratio of times executed in the erroneous path to total number of executions in the correct execution path database. For example, if one DD-path is executed 500 times by the error producing test case and 100 times by all other test cases, and another DD-path is executed 3 times by the erroneous test case and once by all the others, the ratio values for each would be 5 and 3 respectively. Heuristic 4 would then choose the first as the likely candidate to contain the error.

The rationale for heuristic 4 suggests that DD-paths executed with a higher relative frequency in the erroneous path than in the correct paths may be more likely to contain the error than those with fewer executions.

Heuristic 5

Heuristic 5 returns the DD-path on the erroneous execution path that has been executed a minimum number of times (but non-zero) in the execution path database. Heuristic 5 is based on the theory that the DD-path that has been exercised least by the nonerror producing test cases may be the source of the error.

Heuristic 6

Heuristic 6 returns the DD-path on the erroneous execution path that has the corresponding minimum localized sum. A localized sum for a particular DD-path is defined as the sum of all the execution counts from the execution path database of three contiguous DD-paths which contain the particular DD-path in the center.

This heuristic attempts to locate a localized pocket of minimum contiguous DD-path executions with the assumption that those areas least exercised are more likely to contain the error.

Heuristic 7

This heuristic is analogous to heuristic 5 but it identifes the DD-path with the minimum number of executions by all test cases, including the erroneous path, instead of trying to locate the DD-path in the erroneous execution path with a corresponding minimum number of executions in the execution path data base.

The rationale for including the erroneous execution path in the minimum calculation is that traversals of a DD-path (even on an erroneous execution path) increase the likelihood that the DD-path does not contain the error.

Heuristic 8

This heuristic returns sets of pairs of contiguous DD-paths that have been executed in the erroneous execution path and that have never been executed as a pair in the execution path database. This heuristic is based on the theory that some types of errors are caused by the sequencing of contiguous DD-paths. If a sequence has never been tested, it may be a likely candidate for the error.

Heuristic 9

Heuristic 9 is analogous to heuristic 8 except that this heuristic examines sequences of three contiguous DD-paths instead of examining contiguous sequences of two DD-paths.

A sequence length of three was chosen to increase the accuracy of heuristic 8.

Heuristic 10

This heuristic extends the rationale utilized in heuristics 8 and 9 to examine all contiguous sequences of DD-paths that have been executed in the erroneous execution path that have never been executed as noted in the execution path data base.

RELATIONAL PATH ANALYSIS EXPERIMENT

This section describes the experimentation utilizing relational path analysis. The experimental design is presented first, followed by the actual results and their interpretation.

Experimental Design

The first step of the experiment was to develop a Pascal path analysis tool that could automatically calculate the ten heuristics currently utilized by relational path analysis for a set of sample programs. Ten small Pascal programs were chosen from Jensen and Wirth's *Pascal: User Manual and Report*[8] and Grogono's *Programming in Pascal*[9] to provide a non-biased test set. The programs selected are summarized in Appendix 1.

The next step involved creating test cases for each of the programs using a black box testing approach. The path analysis tool was then invoked for each test case preserving a record of the test case execution. The result of this step is a test suite for each program as well as the information needed for relational path analysis.

To actually determine the effectiveness of relational path analysis, errors then had to be inserted into each program. Ten different error types were chosen from Myers' book *The Art of Software Testing*[5] to seed into the programs. The error types are described in Appendix 2. To simplify testing, each program was associated with only one error type.

After assigning the error types, one error was randomly seeded into the code of each of the ten programs. Then, each program was executed with the previously generated test suite, and an analysis was performed utilizing the relational path analysis heuristics. The results of each error analysis were saved, and the seeding of errors was repeated four more times providing a total of five error analysis results per program.

There were ten null hypotheses to test during the experiment. Each null hypothesis corresponded to a heuristic and could be stated as: "The probability that the heuristic is capable of finding errors is 0." Thus, if a heuristic was capable of finding errors, this experiment would have to reject the null hypothesis at a 90% confidence level. When calculating a heuristic's error localization ability, all ten error types were utilized.

RESULTS

The results of using the ten heuristics on the ten programs are shown in Table I. Each entry in this matrix contains the fraction of time in which the heuristic corresponding to the column found the DD-path(s) that were in error in the program corresponding to the row. The mean at the bottom of each column corresponds to each heuristic's ability to identify errors across all ten programs (i.e., all ten different error types). The standard deviation (SD) and half length (HL) of a 90% confidence interval using the *t* statistic are also shown. Also shown is the average fraction (MDD) of DD-paths returned by each heuristic over the total number of DD-paths. This number provides an indication of the precision of a heuristic. Finally, the mean number of times each error type was found by all of the heuristics is shown in the column labeled "MEAN."

Table II contains the 90% confidence intervals for the ten heuristics tested along with the average percentage of DD-paths returned by each heuristic. The confidence intervals and the percentage of DD-paths returned provide the means of assessing the relative utility of each heuristic.

INTERPRETATION

An analysis of the confidence intervals in Table II shows the null hypotheses for all of the heuristics rejected at the 90% confidence level. Thus, all of the heuristics possess some error localization ability. Table I also illustrates that the heuristics

TABLE I—Results of heuristics

					Heuristic						
	1	2	3	4	5	6	7	8	9	10	MEAN
1	.4	1	.6	.4	.2	.2	.4	.8	.8	.8	.56
P 2	.4	1	.6	0	0	.2	0	.6	.6	.8	.42
R 3	.6	1	1	1	.2	0	.2	.6	.6	.6	.58
O 4	.8	1	1	.4	0	0	0	1	1	.8	.60
G 5	.2	1	1	.8	.4	.2	.4	.6	.6	.6	.58
R 6	.6	1	1	1	.4	0	0	.6	.6	.6	.58
A 7	.6	1	.4	1	.4	0	0	.6	.6	.6	.52
M 8	.4	.8	.8	.8	.2	.2	.2	.4	.4	.4	.46
9	.4	1	1	.8	.2	0	0	.4	.6	.8	.52
10	.2	1	.8	.6	.2	.4	.2	.2	.2	.2	.40
MEAN	.46	.98	.82	.68	.22	.12	.14	.58	.60	.62	
S.D.	.19	.06	.22	.33	.15	.14	.16	.22	.21	.20	
H.L.	.11	.03	.13	.19	.09	.08	.09	.13	.12	.11	
MDD	.15	.95	.90	.09	.09	.09	.09	.45	.48	.42	

TABLE II—Confidence intervals and percent DD-paths returned

Heuristic	C.I	Percent DD-paths
1	(.35, .57)	15%
2	(.95, 1)	95%
3	(.69, .95)	90%
4	(.49, .87)	9%
5	(.13, .31)	9%
6	(.04, .2)	9%
7	(.05, .23)	9%
8	(.45, .71)	45%
9	(.48, .72)	48%
10	(.51, .73)	42%

are not very sensitive to the ten error types (as demonstrated by the mean number of times each error type was found by all of the heuristics). These results suggest there is some basis for relational path analysis as an error localization technique.

The usability of a heuristic requires an examination of both the confidence interval and the percentage of DD-paths returned by the heuristic. Based on the results in Table II, the most usable heuristics appear to be numbers 1 and 4.

CONCLUSION AND FUTURE RESEARCH

In this paper relational path analysis is presented as a new technique for software fault localization. An experiment is also described in which the heuristics comprising relational path analysis were tested and found to localize errors with statistical significance. Although this experiment was limited to small programs and a small number of error types, the results were promising and suggest additional research should be performed. The ultimate goal of this research into relational path analysis should be to develop a powerful fault localization tool. Such a tool could apply sophisticated heuristics to help isolate errors. Although such a tool would be applicable throughout a product's life cycle, its diagnostic capabilities would be most powerful after some systematic testing has been performed. Thus, the fault localization tool would be most beneficial during the later stages of testing and during software maintenance. Considering the high cost of performing maintenance activities and the difficulty of isolating errors during this phase, this tool could be very cost effective.

Several additional research areas must be explored before an effective fault localization tool based on relational path analysis can be developed. First, the current heuristics must be examined and experimented with for additional types of errors and bigger programs. Additional heuristics may also be needed for detecting errors in large programs. Another interesting area to explore is the combination of various heuristics. Although much research remains to be done in this area, the need for automated fault localization is high and the potential benefits are significant.

REFERENCES

1. Boehm, B.W. "Software Engineering." In *Classics in Software Engineering*, E.N. Yourdon (ed.), New York: Yourdon Press, 1979, p. 326.
2. Boehm, B.W. *Software Engineering Economics.* Englewood Cliffs, New Jersey: Prentice-Hall, 1981.
3. Booch, G. *Software Engineering With Ada.* Menlo Park, California: Benjamin/Cummings, 1983.
4. Zelkowitz, M.V. "Large-scale Software Development." In *Principles of Software Engineering and Design,* Englewood Cliffs, New Jersey: Prentice-Hall, 1979, pp. 2–11.
5. Myers, G.J. *The Art of Software Testing.* New York: John Wiley, 1979.
6. Johnson, W.L. and Soloway, E. "PROUST An Automatic Debugger for Pascal Programs." *BYTE* (April 1985), pp. 179–190.
7. Chapman, D. "A Program Testing Assistant." *Communications of the ACM,* 25 (1982) 9, pp. 625–634.
8. Jensen, K. and Wirth, N. *Pascal: User Manual and Report.* New York: Springer-Verlag, 1974.
9. Grogono, P. *Programming in Pascal.* Reading, Massachusetts: Addison-Wesley, 1980.

APPENDIX 1—PASCAL TEST PROGRAMS

1. A simulation program that models passenger buses traveling between stops and shows the passenger throughput.
2. A calculator program that evaluates arithmetic expressions in infix notation.
3. A program that finds the circular radius and center of the circle that intersects three distinct points.
4. A cosine program that arithmetically finds the cosine of a number through iteration.
5. A program that produces a cross-reference for all distinct words contained in a file.
6. A square root program that arithmetically finds the square root of a number through iteration.
7. A program that finds the matrix multiplication of two matrices.
8. A program that changes an infix notation expression to postfix notation.
9. A program that converts regular numbers to Roman numerals.
10. A program that performs a shell sort on a list of numbers.

APPENDIX 2—SEEDED ERROR TYPES

1. Unset or uninitialized data values (value is zero).
2. Wrongly set data (values set randomly).
3. Logic errors; that is, misuse of "and," "or," and "not."
4. Computation errors; that is, incorrect arithmetic precedence, mixed mode problems, and integer division.
5. Incorrect procedure or function output.
6. Does not correctly handle all legal input; that is, boundary conditions not checked, no checks for valid input, and exhaustive decision are not made.
7. Off-by-one arithmetic errors (not loops).
8. Placing of program statements is incorrect; that is, placed external from or internal to the place they should be (e.g., incorrect begin/end grouping).
9. Misuse of comparators; that is, =, <, >, >=, <=, <>.
10. Incorrect looping; for example, wrong assumptions, off-by-one errors, etc.