The design of a prototype mutation system for program testing (PDF)

Article · January 1978

CITATIONS
78

READS
241

4 authors, including:

Richard A. Demillo
Georgia Institute of Technology
117 PUBLICATIONS **8,765** CITATIONS
SEE PROFILE

Fred Sayward
Yale University
48 PUBLICATIONS **3,176** CITATIONS
SEE PROFILE

The design of a prototype mutation system for program testing*

by TIMOTHY A. BUDD, RICHARD J. LIPTON and FREDERICK G. SAYWARD
Yale University
New Haven, Connecticut

and

RICHARD A. DEMILLO
Georgia Institute of Technology
Atlanta, Georgia

INTRODUCTION

When testing software the major question which must always be addressed is "If a program is correct for a finite number of test cases, can we assume it is correct in general." Test data which possess this property is called Adequate test data, and, although adequate test data cannot in general be derived algorithmically,[1] several methods have recently emerged which allow one to gain confidence in one's test data's adequacy.

Program mutation is a radically new approach to determining test data adequacy which holds promise of being a major breakthrough in the field of software testing. The concepts and philosophy of program mutation have been given elsewhere,[2] the following will merely present a brief introduction to the ideas underlying the system.

Unlike previous work, program mutation assumes that competent programmers will produce programs which, if they are not correct, are "almost" correct. That is, if a program is not correct it is a "mutant"—it differs from a correct program by simple errors. Assuming this natural premise, a program P which is correct on test data T is subjected to a series of mutant operators to produce mutant programs which differ from P in very simple ways. The mutants are then executed on T. If all mutants give incorrect results then it is very likely that P is correct (i.e., T is adequate). On the other hand, if some mutants are correct on T then either: (1) the mutants are equivalent to P, or (2) the test data T is inadequate. In the latter case, T must be augmented by examining the non-equivalent mutants which are correct on T: a procedure which forces close examination of P with respect to the mutants.

At first glance it would appear that if T is determined adequate by mutation analysis, then P might still contain some complex errors which are not explicitly mutants of P. To this end there is a COUPLING EFFECT which states that test data on which all simple mutants fail is so sensitive that it is highly likely that all complex mutants must also fail.

Readers wishing a further exposition of the ideas of mutation and substantiation of the assumptions made are referred to References 2 and 10.

THE SYSTEM

A pilot system has been built to implement mutation analysis on programs written in a subset of FORTRAN. The key features of this system are summarized in Figure 1. The system itself consists of 10,000 lines of FORTRAN code, and required six man months to design, implement and debug.

Notice we claim the system is man/machine interactive. In general an attempt is made to assign tasks to both the user and the machine processors which are best suited to using their particular capabilities. One way to see this is to view the system as a sort of "Devils Advocate", which when confronted with a program asks very difficult questions about the motivation behind it ("why did you use this type of statement here, when an alternative statement works just as well?"). The job of the human is then to provide justification (in the form of test data), which will give an answer to such questions.

An overview of the structure of the system is given in Figure 2. We point out that the language FORTRAN was chosen for the first implementation merely as a matter of convenience since it is in common use and there is a large body of software in existence to experiment on. The heart of the system (roughly that shown within the dotted box) is however, language independent, and given a sufficiently general internal form to implement a new language one would merely write a new input/output interface. Projects are currently under way to implement mutation analysis on

* This research was supported in part by NSF Grant MCS76-81486 and U.S. Army Research Grant DAAG-29-76-G-0338.

INTERACTIVE
MACHINE INDEPENDENT
LANGUAGE INDEPENDENT STRUCTURE
MODULAR DESIGN
INTENSIVE MAN/MACHINE INTERACTION

Figure 1—Key features of the pilot mutation system

COBOL and C (an ALGOL like language) using the structure represented by the box contained in the dotted lines.

An attempt was also made to keep the structure of the system largely machine independent. The system was originally programmed to run on a PDP-10 at Yale University. Currently we are in the process of transferring it to a CDC 7600 at the Georgia Institute of Technology.

A single run of a mutation system divides naturally into three phases the RUN PREPARATION phase, in which the necessary variables to send to the mutation executor are defined, the MUTATION phase, in which the actual mutations are produced and executed, and the POST RUN phase, in which results are analyzed and reports are generated. In the following we will describe in more detail the structure and effects of each phase.

The role of the run preparation phase is to initialize the various files and data buffer areas used by the mutation executor. It is characterized by a very interactive nature. The first object the user is requested to supply is the name of the file on which the FORTRAN subroutine resides. Then depending on whether PIMS has been run previously on this routine (in which case the internal form is stored on one of the many files PIMS constructs, see below) the subroutine is parsed into a concise internal format which is subsequently interpreted to simulate execution of the program. A

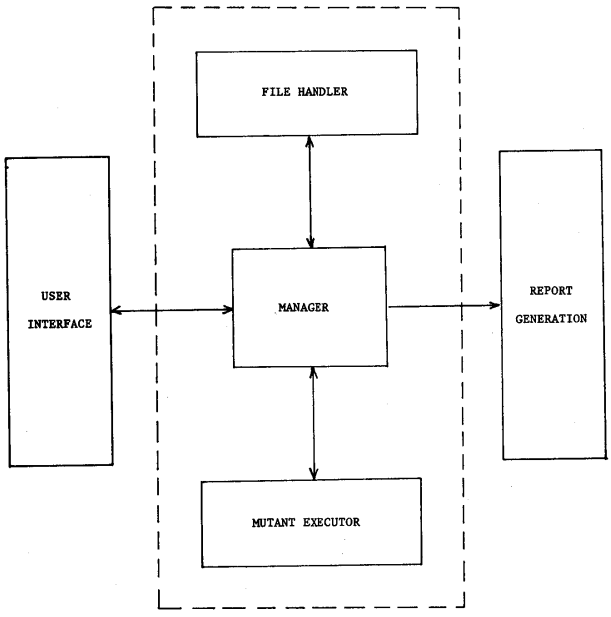

Figure 2

```
IF (A .LT. X(2)) P = 1

[SCALAR. A]
[ARRAY1. X]
[CONSTANT.2]
[AOP.SUBSCRIPT]
[ROP.LT]
[TRF.O]
[SCALAR. p]
[CONSTANT.1]
[ASSIGN.O]
```

Figure 3

fragment of the internal code generated for a given statement is shown in Figure 3.

The user is then interactively prompted for the test data on which the program and mutants are to be tested. After each test case has been specified the original program is executed on the test case and the results displayed so that the user may satisfy himself that the results produced are indeed correct.

After the test data has been entered the user is prompted for a listing of which mutant operators he wishes to enable. At present there are 25 mutant operators. These range from very simple low level ones, such as replacing each data occurrence (where a data occurrence is a scalar, constant or array reference) with all other syntactically correct data occurrences, to very high level mutations, such as deleting statements or altering the control structure of the program. A more detailed description of the mutations performed can be found in Reference 3.

Instead of constructing multiple copies of the program, for each mutant a short (four word) description of the mutation to be performed is kept. Each time the mutant is to be run the original program is then mutated according to the contents of this descriptor.

After the user has specified to the system his program, test data and the mutant operators he wishes applied, the system then enters the MUTATION phase. During this phase there is no user interaction. Mutation descriptor records are read in, one by one, and the mutation is produced. The mutant program is then executed on the test data and marked either "dead," meaning it produced results differing from the original program on at least one test case, or "living." A dynamic record is kept of the number and percentage of living mutants of each mutation type.

When all the mutant programs have been tested the post run phase is entered. In this phase statistics are displayed indicating the results of the mutation run. In addition the user can interactively view descriptions of those mutations which have survived. He can also specify that certain re-

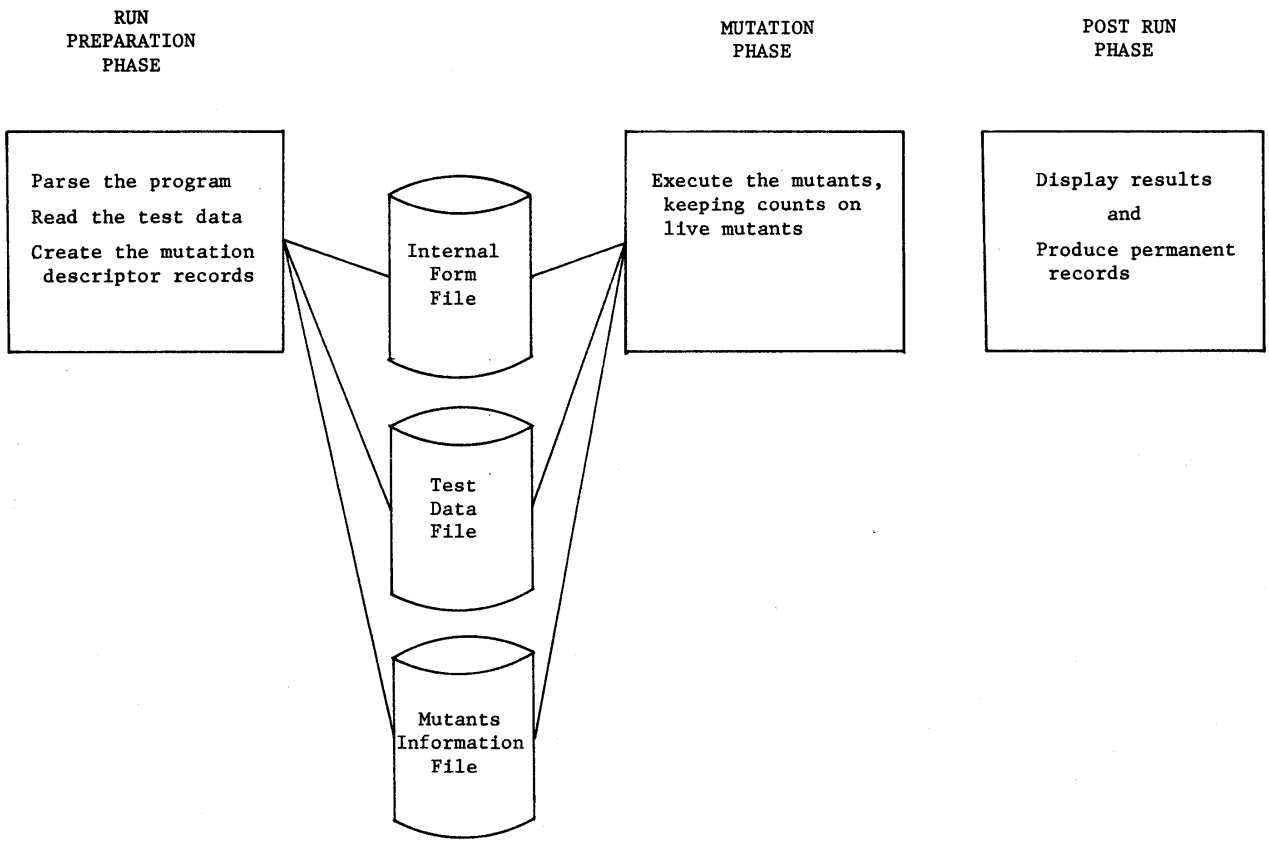

Figure 4

ports be generated in order to provide a detailed permanent record of the mutation run.

At this point, or at a later date, the user can re-run the system and augment his test data in an attempt to make the remaining mutants fail. He may also specify that additional mutant operators be applied to the program. This cycle can continue until the user is satisfied that the current test data adequately tests his program.

There are several files the system produces in order to store information from one run to the next. These are shown in Figure 4, which outlines the major functions of each phase. The internal form file stores the parsed version of the program. The test data file stores for each test case the test data input and the results of execution of that test data. The mutants information file keeps the mutant descriptor records plus various other counts on what types of mutants have been produced.

A COMPARISON OF PIMS TO OTHER DATA TESTING SYSTEMS

Various systems have been discussed in the literature for increasing confidence in the adequacy of test data, as the PIMS system does, or automatically constructing test data which meets some criterion. In this section we will report on experiments which show that the PIMS system is an improvement in this area over other systems which have been proposed.

The most widely known method of constructing test data automatically are those systems which utilize path analysis.[4–7] Essentially, these procedures attempt to construct data which force each statement to be executed at least once, and furthermore which transverse each feasible flow path through the code at least once. In some cases, such as loops, only an approximation to this can be made as the number of flow paths may be infinite. Here it is usual to just construct data which cause the loop to be executed at least twice.

These same objectives are met with mutant analysis in a number of ways, some directly by mutant operators, others indirectly by the coupling effect. There are mutants which cause each statement in the original program to be replaced by a TRAP statement, a special type of statement which if ever executed causes the program to immediately abort. Obviously, then if there is some statement in the program which is never executed, changing that statement to a TRAP statement will not alter the output of the program and hence will easily be detected.

Checking that every decision path is taken is essentially

```
      DO 10 I=1.J
        :
        :
      10 CONTINUE
    ===============
      DO 10 I=1.1
        :
        :
      10 CONTINUE
```

A LIVE MUTATION IF THE LOOP IS ALWAYS EXECUTED ONLY ONCE.

Figure 5

the same as checking that every predicate in the program evaluates at least once to both true and false. If this is not the case, say the predicate always evaluated to TRUE, then we can mutate the predicate in any way we desire as long as it retains this property of always remaining TRUE. These types of mutations are also usually quite obvious and easily detectable.

Mutation analysis can also insure that each loop is traversed at least twice. The only way a loop can be traversed only once (and all loops must be traversed at least once to pass the TRAP statement mutations) is if the terminating condition is the same as the starting condition. But in this case the mutant which replaces the terminating condition by the starting condition will survive (see Figure 5). This is once more easily detected.

With this, mutant analysis possesses all the capabilities of

```
      SUBROUTINE BSERCH(X.Y.N.A.IHIGH.LOW.ERR)
      INTEGER X(N).Y(N).N.A.IHIGH.LOW.ERR.MID
C     BINARY SEARCH PROCEDURE. IF X CONTAINS A ON RETURN
C     X(IHIGH) = A.IHIGH=LOW. IF NOT X(LOW) < A < X(IHIGH).
C     IF A IS OUT OF RANGE ON RETURN ERR CONTAINS 1
      ERR = 0
      IF ((X(1)-A).GT.0) GOTO 11
      IF ((A-X(N).LE.0) GOTO 5
 11   ERR = 1
      RETURN
 5    LOW = 1
      IHIGH = N
 6    IF ((IHIGH-LOW-1).NE.0) GOTO 7
      RETURN
 7    MID = (LOW+IHIGH)/2
      IF ((A-X(MID)).GT.0) GOTO 10
      IHIGH = MID
      GOTO 6
 10   LOW = MID
      GOTO 6
      END
```

Figure 6

```
Replace  IF ((IHIGH-LOW-1).NE.0) GOTO 7
by       IF ((IHIGH-LOW-1).GT.0) GOTO 7

Replace  MID = (LOW+IHIGH)/2
by       MID = (LOW+IHIGH)-2
```

Figure 7

path analysis systems which have been discussed in the literature.

Another class of systems for which extensive claims have been made are those which detect uninitialized variables and dead code.[8] Uninitialized variables are caught as a consequence of the interpretation process in the mutant system. Dead code is easily caught since an assignment made to a dead variable can be mutated in any way whatsoever and the program will remain the same.

A third class of systems for which there has recently been much discussion involves symbolic execution of the program. In one study[9] Howden analyzed 12 programs containing a total of 22 errors. He found that symbolic execution would catch 13 of those errors, while path analysis would discover only nine. In a similar study we estimated that mutation analysis, using only the mutant operators in the present PIMS system, would uncover 18 of the 22 errors. Of the remaining four, three would probably be discovered if we added two new mutant operators which the authors simply had not thought about. Hence, mutation analysis is in certain cases an improvement over symbolic execution.

As an example of the very subtle errors which mutation analysis can discover consider the program to perform binary search shown in Figure 6. If it happens that N=1 when the subroutine is called (i.e., the vector to be searched contains only a single element) then it is not difficult to see that the program will loop indefinitely. It is not clear that either symbolic execution or path testing would be sufficient to discover this error.

When mutant analysis is applied to this program there are two mutants generated (shown in Figure 7) which can only be eliminated by a test case consisting of one element. Hence the error is easily detected using mutant analysis. (There is a second error in this program which is also uncovered by mutant analysis. The discovery of that second error is left to the reader).

FUTURE WORK

There are several directions in which work is currently being pursued with respect to mutation analysis and the pilot mutation system. The most obvious is to show how a similar system might be built around another language, and research is under way to construct systems for COBOL and for C.

Another area of study is the design of an easy to use language for the description of test cases which allows for a variety of features. Test datasets can often be quite lengthy, yet two test cases can be very similar. Also, a user often wishes just to construct a number of random test cases following some specification. (Some of the pitfalls of using random data to test programs are discussed in Reference 10

where it is seen that mutation analysis can help in deriving "good" random test data.) Finding an easy yet powerful method of solving this problem is the goal of one area of research.

Finding a method to detect equivalent mutants is another area currently being pursued. It is often the case that a mutation will not produce a significantly different program (replacing the sequence I=1 J=1 with the sequence I=1 J=I is a trivial example). We have observed that programs tested have between one and two percent equivalent mutants. A method to automatically detect and remove equivalent mutants would allow us to provide even more significant measures of the adequacy of a test data set.

We point out that as a consequence of the modular design of the pilot system either of the above two major extensions can be added without a significant reprogramming effort.

A final area of current interest is the study of mutant operators. Certain operators seem to have a much greater ability to detect errors then others. Analysis of data along these lines would allow us to discover an order of application of mutant operators which would maximize the cost/benefit ratio.

CONCLUSIONS

It has been shown that the ideas of program mutation can be quickly and easily implemented as an interactive system for program testing. The resulting system represents a cost effective engineering approach to testing real world software. Large subroutines (over a hundred statements long) have been analyzed by our system with relative ease.

Mutation is a method of program testing which will significantly raise the level of reliability in both new and existing software, and is a major advance in the area of software testing.

REFERENCES

1. Goodenough, J. B. and S. L. Gerhart, "Towards a Theory of Test Data Selection," *IEEE Tran. Soft. Eng.*, SE-1,2, June 1975, pp. 156-173.
2. DeMillo, R., R. J. Lipton and F. Sayward, "PROGRAM MUTATION—A Method of Determining Test Data Adequacy," in preparation.
3. Budd, T. and F. Sayward, "Users guide to the Pilot Mutation System," Yale University Tech. Rep 114, 1977.
4. Ramamoorthy, C. V., S. F. Ho, and W. T. Chen, "On the Automated Generation of Program Test Data," *IEEE Trans. on Soft. Eng.*, SE-2,4, Dec. 1976, pp. 293-300.
5. Howden, W. E., "Methodology for the Generation of Program Test Data," *IEEE Trans. on Comp.*, C-24,5, May 1975, pp. 554-560.
6. Huang, J. C., "An Approach to Program Testing," *Computing Surveys*, 7,3, Sept. 1975, pp. 113-128.
7. Miller, E. F. and R. A. Melton, "Automated Generation of Testcase Datasets," Proc. 1st Int. Conf. on Reliable Software, *SIGPLAN Notices*, 10,6, June 1975, pp. 51-58.
8. Osterweil, L. J. and L. D. Fosdick, "Some Experience with DAVE—A Fortran Program Analyzer," *AFIPS Conference Proceedings*, Vol. 45, 1976, pp. 909-915.
9. Howden, W. E., "Symbolic Testing and the DISSECT Symbolic Evaluation System," *IEEE Trans. on Soft. Eng.*, SE-3,4, pp. 266-278.
10. DeMillo, R., J. Lipton and F. Sayward, "Hints on Test Data Selection," to appear in *Computer*, April 1978.

A panel session—User experience with new software methods

SESSION CHAIRMAN—VICTOR R. BASILI
University of Maryland

Panel Members

Donald J. Reifer—TRW
Donn Combelic—ITT
J. A. Rader—Hughes Aircraft Company
C. M. Bernstein—Exxon Corporation
F. T. Baker—IBM Federal Systems Division
Susan Voigt—NASA

PANEL OVERVIEW—Victor R. Basili

The development of correct, reliable, less expensive software continues to be a major problem. A great deal has been written and said about various techniques and methodologies for software development and how they are meant to aid in the development process. Unfortunately, most of the available material is by the author of the technique, be it individual or company. This does not always allow the outside user of the technique a fair appraisal or full understanding of how good the technique is, how to use it, and how to adjust it to his environment. This is true for several reasons. First, the author's experience is often limited to a specific application or set of applications and specific environments. There are some genuine questions that arise when taking a technique and moving it to a new application or environment that the developer of the technique had not anticipated. Second, the author does not always tell the prospective user everything he needs to know. Often this is just due to a lack of documentation, or a set of basic assumptions and background that the author did not realize was even necessary. Lastly, one cannot normally expect the author to emphasize the weak points or problems in the methodology. That is just human nature.

The purpose of this panel and the following set of papers is to discuss a set of techniques available in the open literature, some very new, some that have been around for awhile, and ask for an analysis and evaluation by current users. Each of the panelists is not the author of the methodology but a member of a company that is using the methdology or overseeing a contract on the use of the methodology. Some of the users are old hands at the technique; some are novices. I have asked each of the panelists to prepare a short paper covering a brief description of the technique and an evaluation of the technique in a real environment. Suggested ideas to be included in the paper and the oral presentation are given below.

I. THE TECHNIQUE
- The techniques you have been using.
- A description of the project you are using it on.
- The phase of the project in which it is used.
- The phases of the project it affects.
- An overview of the technique.
- Why you chose it.
- The extent to which you are using it.

II. EVALUATION OF THE TECHNIQUE IN A REAL ENVIRONMENT
- What have you felt are its good points and how they have shown to be good.
- What have you felt are the weak points and why.
- How you have adapted it to your environment.
- Would you use it again and if so how would you change or have you already changed the way it should be applied.
- What would you recommend to someone else applying it.

Certainly lots of techniques could have been covered but there was limited time and space available. The techniques chosen were based partially on my own interest and partially on the availability of people willing to discuss specific techniques. Discussants and topics include:

PSL/PSA—Donald J. Reifer,
SADT—Donn Combelic, ITT Telecommunications
Structured Design—Dr. J. A. Rader, Hughes Aircraft
Jackson's Methodology—Clifford M. Bernstein, EXXON Corporation
Boeing's IPAD Methodology—Susan Voigt, NASA Langley Research Center
Correct Program Design—F. Terry Baker, IBM Corporation

The methodologies deal with various phases of the software development process, from requirements to program development, some emphasizing one specific phase and some covering several phases. PSL/PSA and SADT deal predominantly with the requirements phase. Structured De-