Extending Mutation Testing to Find Environmental Bugs*†

Technical Report SERC-TR-21-P

Eugene H. Spafford

Software Engineering Research Center
Department of Computer Sciences
Purdue University
W. Lafayette, IN 47907-1398

spaf@cs.purdue.edu

ABSTRACT

Environmental bugs are bugs caused by limitations of precision or capacity in the environment of a piece of software. These bugs may be difficult to activate and even more difficult to find.

This paper reports on an extension to traditional mutation testing that enables testing specifically for environmental bugs involving integer arithmetic. This method is both simple and effective, and provides some insight into other possible extensions of the mutation testing methodology that can be used to expose environmental bugs.

Keywords: testing, mutation analysis, environmental bugs

The Problem

When translating specifications into algorithms and then into code, programmers may pay insufficient attention to the architecture of their target machine, thus introducing hard-to-find bugs. These bugs are hard to find because they are often highly dependent on the operational environment—the underlying algorithms may be completely correct, but they are not executed on the ‘‘perfect’’ virtual machine the programmer had in mind when developing the code.‡ Typical examples of these *environmental bugs*

* Portions of the work described in this paper were funded under RADC contract F30602-85-c-0255.

† A version of this paper appears in the journal *Software Practice & Experience*, **20**(2), pp. 181-189, Feb 1990.

‡ Interestingly enough, the author has been unable to find any comprehensive published study of these problems, although nearly everyone seems to have favorite anecdotes concerning them.

include:

- memory limitations: arrays cannot be made arbitrarily large and stacks cannot grow without bound, even on machines with virtual memory;
- numeric limitations: machine representations of numbers, floating-point and integer, are bounded in both size and precision;
- the value used to initialize ''uninitialized'' memory pages or segments of virtual memory may introduce unsuspected problems;
- interpretation of supposed constant values may produce faults, such as porting code that dereferences a constant pointer;†
- exception handling and reporting may be different than the programmer believed; and
- system errors: the compiler may produce incorrect code, the hardware may execute instructions differently than expected under some circumstances, and the operating system may introduce intermittent errors not related to the user code.

These bugs may be present in a software system only when that software is run on a particular machine or under particular operating systems; it is precisely these bugs that make porting software to foreign systems so difficult, since they may not be recognized through simple examination of the code, and they may not manifest themselves unless the code is executed with a highly specific set of inputs. The code may be syntactically correct, and the underlying algorithms can even be proven formally correct, yet the program does not function properly on all allowed inputs. It is possible that the code has been thoroughly tested over all possible inputs, only to manifest problems when moved to a different run-time architecture.

As a specific example of this type of bug, consider the Fortran program shown in Figure 1. The user provides three integers as input to this program, known as **tritype** or **triangle**. The program then prints a message indicating if those numbers can be the lengths of the three sides of a triangle, and if so, what form of triangle. The determination of legality is a simple, well-known algorithm: the sum of each pair of sides must be greater than the remaining side, and all sides must have positive length. Versions of this program have been used widely in the literature to illustrate the difficulties in testing code; it is derived from the example presented in reference 17. It is an ideal example because of its simplicity and compactness, and because it takes dozens of distinct test cases to expose all possible coding errors.

For the past few years, researchers working on the Mothra software mutation system [5, 7] have used this code to illustrate the power of mutation analysis as an aid to testers in their development of test data. Although we have long viewed the encoding in Figure 1 as correct, during research leading to this paper a set of non-obvious test

† One example of this error is well known to C programmers porting code to different architectures, and is known as the ''dereferencing NULL pointers'' bug.

cases was discovered that causes this version to print incorrect results.

```
     PROGRAM TRIANG INTEGER SIDE1, SIDE2, SIDE3, TRIANG

       PRINT *, "Input 3 sides:"
       READ *, SIDE1, SIDE2, SIDE3

C      After a quick confirmation that it's a legal triangle
C          detect any sides of equal length

       IF (SIDE1.LE.0.OR.SIDE2.LE.0.OR.SIDE3.LE.0) THEN
       PRINT 10
       STOP
       ENDIF
       TRIANG=0
       IF (SIDE1.EQ.SIDE2) TRIANG=TRIANG+1
       IF (SIDE1.EQ.SIDE3) TRIANG=TRIANG+2
       IF (SIDE2.EQ.SIDE3) TRIANG=TRIANG+3
       IF (TRIANG.EQ.0) THEN

C      Confirm it's a legal triangle before declaring
C      it to be scalene

          IF (SIDE1+SIDE2.LE.SIDE3.OR.SIDE2+SIDE3.LE.SIDE1.OR.
     *     SIDE1+SIDE3.LE.SIDE2) THEN
          PRINT 10
          STOP
       ELSE
             PRINT *, "Sides form a scalene triangle."
          STOP
          ENDIF
       ENDIF

C      Confirm it's a legal triangle before declaring
C      it to be isosceles or equilateral

       IF (TRIANG.GT.3) THEN
          PRINT *, "Sides form an equilateral triangle."
       ELSE IF (TRIANG.EQ.1.AND.SIDE1+SIDE2.GT.SIDE3) THEN
       PRINT 20
       ELSE IF (TRIANG.EQ.2.AND.SIDE1+SIDE3.GT.SIDE2) THEN
       PRINT 20
       ELSE IF (TRIANG.EQ.3.AND.SIDE2+SIDE3.GT.SIDE1) THEN
       PRINT 20
```

```
       ELSE
      PRINT 10
       ENDIF
       STOP

10     FORMAT("Sides do not form a legal triangle.")
20     FORMAT("Sides form an isosceles triangle.")
       END
```

Figure 1

To discover what these values are, consider that our target environment consists of machines using 32-bit, two's-complement integers. Thus, the largest representable integer on these systems is 2,147,483,647. Now consider input to the **triangle** program consisting of the triple (2147483640, 10, 2147483640). The program will label them as not composing the sides of a triangle, even though they form a legal isosceles triangle. This error is caused when any two of the side lengths are summed and a silent overflow (wraparound) occurs, resulting in a machine representation of a negative number (and thus, a value less than the remaining side). This is an example of an environmental bug. Changes to the program near the beginning, illustrated in Figure 2, make the program's behavior correct for any target machine with silent overflow during integer arithmetic.

One approach to detecting bugs such as this is to try a large number of testcases to be sure that the program produces correct results for all input in its defined domain. As experienced software testers know, this is a difficult task—even the best tester may miss critical combinations of input that will expose a bug, as in the example shown above. [6] Testers would benefit from some measure of the adequacy of their testing, and from tools that help generate and select test cases to expose environmental bugs.

Mutation Testing

For over a decade, researchers have been working with *program mutation* as a method of testing software.* A basic goal of program mutation is to provide the user with a measure of test set *adequacy*, by executing that test set against a collection of program *mutations*. Mutations are simple changes introduced one at a time into the code being tested. These changes are derived empirically from studies of errors commonly made by programmers when translating requirements into code, although theoretical justification also can be found for their selection.[10] A mutant is *killed* if the execution of the mutated code against the test set distinguishes the behavior or output of the mutation from the unmutated code. The more mutants killed by a test set,

* Program mutation has been well documented in the literature and will not be described in detail here. The reader unfamiliar with mutation testing is directed to recent references on mutation for detailed descriptions and further references: 4, 5, 7, 13, 15, and 21.

```
     PROGRAM TRIANG INTEGER SIDE1, SIDE2, SIDE3, TRIANG

       PRINT *, "Input 3 sides:"
       READ *, SIDE1, SIDE2, SIDE3

C      After a quick confirmation that it's a legal triangle, detect any sides of equal length

       IF (SIDE1.LE.0.OR.SIDE2.LE.0.OR.SIDE3.LE.0) THEN
        PRINT 10
        STOP
       ELSE IF (SIDE1+SIDE2.LT.0.OR.SIDE1+SIDE3.LT.0.OR.
     *    SIDE2+SIDE3.LT.0) THEN
        PRINT *, "One (or more) side is too long to determine type."
        STOP
       ENDIF
C
       TRIANG=0
```

Figure 2

the better the measured adequacy of the test set. By proper choice of mutant operators, comprehensive testing can be performed,[3] including path coverage[14] and domain analysis.[20] By examination of unkilled mutants, testers can add new test cases to better the adequacy score of the entire test set.

Program mutation is a powerful method for detecting bugs in code, but has no provision (mutant operators) for detecting environmental bugs. Since programs containing environmental bugs may be coded completely correctly, it is only by accident that mutation testing might expose them. However, the general method used in mutation analysis to help testers augment their test cases can be used to help generate data to expose environmental bugs. To test for environmental bugs related to machine representation of numerical quantities, we generate mutants that require all appropriate variables to take on extremal values to be killed. The user must define test cases to kill these mutants, and, in so doing, define test cases where silent wraparound or arithmetic exceptions might occur. Mutants to detect other environmental bugs may also be applied; however, for the remainder of this paper, we will focus on detection of the type of simple integer environmental bug we illustrated in our example.

Consider code for a machine with two's complement integer arithmetic. In every arithmetic expression where an integer variable appears, we would like the tester to provide test cases that cause the expression to both overflow and underflow. If the machine signals integer exceptions, then this will allow us to check the exception

handling mechanism, as well as to exercise intermediate code. If the machine simply does a silent wraparound of the representation, then we will observe the output of the code to determine whether it behaves correctly.

To force the necessary over/underflow, we define two new mutations named IOVFLOW and IUFLOW. Every arithmetic expression in the code containing only integer variables is mutated by each of these mutants. The effect of the mutant is as if the expression were provided to an integer function as an argument. That function returns the value of its argument in every case, except where overflow (for IOVFLOW) or underflow (for IUFLOW) occurs when the expression is evaluated. In those cases, the mutant immediately ''dies,'' signalling that the condition occurred.

It is possible that the expression is incapable of generating the desired condition. As such, we can either attempt to avoid generating that particular mutant, or else we can remove the mutant after it is generated when we recognize that it never will be killed. In mutation terminology, it is considered *equivalent*, since for all input its behavior is exactly equivalent to the original code. Determining equivalency may not be easy to do in all cases, but at least one approach under study by our group shows considerable promise at automatically detecting equivalency.[9, 16]

An example of such a mutant would be:

```
      I = 3
      J = 2
K = IUFLOW(I + J)
```

This is equivalent because there is no way that the variables I and J can ever take on values such that the statement I+J underflows. Dataflow analysis commonly used in compiler optimizers can be used to detect many equivalent cases such as this.

The number of mutants produced by each of these operators will be less than or equal to *Op*, the number of occurences of binary integral arithmetic operators in the program. This can easily be shown using induction; the addition of another operator and variable to an expression results in one more applications of the mutation operator for the augmented expression as a whole. Simple assignment is not mutated, except in cases where implicit type conversion is done (e.g., long integer to short integer). Comparison operators (like `I .LT. J`) would be mutated only if the underlying machine instructions fail to take underflow/overflow into account during the comparison. Thus, the overall complexity of these operations is on the order of the number of occurences of operators. This is similar to the complexity of many other mutation operators; the interested reader should refer to references 1 and 3 for discussions of mutation efficiency and complexity.

An example may help to illustrate the application of these new mutations. In Figure 3 we see the section of code that appears near the end of the original tritype program. To reach this point in the program, all three sides had to be greater than zero, and two of the sides had to be equal to each other. If SIDE1 is equal to SIDE3, then the variable TRIANG is equal to 2 and the middle *if* statement will be executed. The

IUFLOW mutant operator has been applied to the sum expression in that statement. For it to be killed, the tester must devise a test case that has SIDE1 equal to SIDE3, and their sum must wraparound to a negative number—precisely the type of input we showed earlier to produce incorrect results. The triple (2147483640, 10, 2147483640) is just such an input: it kills the mutant, and when the unmutated code is run with it as input, it exposes the environmental bug.

```
      IF (TRIANG.GT.3) THEN
          PRINT *, "Sides form an equilateral triangle."
        ELSE IF (TRIANG.EQ.1.AND.SIDE1+SIDE2.GT.SIDE3) THEN
         PRINT 20
        ELSE IF (TRIANG.EQ.2.AND.IUFLOW(SIDE1+SIDE3).GT.SIDE2) THEN
         PRINT 20
        ELSE IF (TRIANG.EQ.3.AND.SIDE2+SIDE3.GT.SIDE1) THEN
         PRINT 20
        ELSE
         PRINT 10
        ENDIF
        STOP

10      FORMAT("Sides do not form a legal triangle.")
20      FORMAT("Sides form an isosceles triangle.")
        END
```

Figure 3

Further Considerations and Problems

To complete the mutant operators necessary to test for integer arithmetic environmental bugs, we need to know something about the underlying representation of those values. For instance, if the integers are represented in standard two's complement, there will be a representation of a negative value for which there is no corresponding arithmetic inverse (–32768 in 16-bit words, –2,147,483,648 in 32-bit words). Depending on the underlying arithmetic firmware, this value may behave strangely when shifted, negated, or compared. To test for correct behavior of software will require entering test cases such that every variable takes on this singular value. To do this, we can introduce another mutation function, which we can call NZPUSH. The NZPUSH mutant behaves in a manner analogous to IOVFLOW and IUFLOW—it ''dies'' when its argument is this singular value, and otherwise has a value exactly equal to its argument. To kill NZPUSH mutants, input values must be generated to cause appropriate variables to be set equal to this ''negative zero.'' (It is worth noting here that standard

mutation already has a similar mutant operation, ZPUSH, to force variables to take on the value zero. [8])

Note that we do not need to be concerned with an NZPUSH mutation on machines with one's complement arithmetic since that system has defined inverses for every value (there are two representations of zero). Machines with excess notations and signed-magnitude representations would require a mutation similar to NZPUSH, however, since those systems have singular values as well.

To optimize the creation of these mutants, we can make the observation that certain forms of expressions will never overflow/underflow, and thus we do not need to generate the mutations for those expressions. For example, I / J can neither overflow nor underflow for integers of the same precision (except, of course, when J is zero); neither can most expressions involving zero, many expressions with constants, bitwise operations, and some common library functions. A complete list is not provided here because it is dependent on the underlying representation of integers on the target machine, the programming language involved, and the host environment. However, a tester implementing a mutation system should be able to determine a set of expressions that can safely be left unmutated. A conservative approach, at worst, will result in too many equivalent mutants being generated.

Another type of environmental bug may occur if the target machine and source language allow the mixing of integers of different precision. This requires the provision of other mutant operators to expose environmental bugs relating to the unexpected truncation or promotion of integral values to different precisions. For instance, the sequence

```
I = 40000
  J = I
```

will produce potentially unexpected results if I is a 32-bit integer and J is a 16-bit integer (or vice versa). These types of errors should be impossible in a strongly-typed language, and compilers for less strongly-typed languages should generate warnings when encountering such code. Unfortunately, not every language is so strongly-typed (e.g., Fortran, C), and many compiler implementations are less than robust. Again, particular circumstances would dictate the utility and format of mutations to expose these types of bugs.

The whole issue of testing for environmental bugs in floating point code is complicated and highly dependent on the underlying hardware. Floating point values represented according to the IEEE 754 standard, for instance, include denormalized numbers, values for ∞ and $-\infty$, values for *NaN* (Not-a-Number), at least two different precisions, and the potential for expression results to be rounded in several ways.[2] To test for all the possible combinations of underflow, overflow, and loss of precision might be expensive in terms of the number and complexity of mutants involved. In addition, not every machine correctly implements simple floating point arithmetic and standard functions,* and the code under test may have been developed with

* See, for instance, references 18 and 19.

compensation for these known quirks. Taking all these conditions into account, it may not be possible to define a complete, **portable** set of mutation operators to test for environmental bugs. The potential usefulness of such mutations, however, is significant, and the effort spent developing even machine-specific mutant operators would be well-spent. The next generation of our mutation testing environment [5] is being designed to allow the user to specify the nature of some mutations, thus allowing customization for the local machine and language environment. Other environmental bugs, such as sizing dynamic arrays too large, and allocating too much heap, should also be testable with custom mutants.

Conclusions

Environmental bugs can occur in code that is logically correct and can be formally verified. These bugs can be extraordinarily difficult to find without a formalized testing methodology designed to search for them.

This paper has presented an important extension to traditional mutation testing that allows a tester to test methodically for integer arithmetic environmental bugs and provides guidance in the development of test cases to illustrate such bugs. The extensions are simple and easy to understand, both for the tester and for the developer of a mutation-based system—this author was able to add these new mutations to a version of an existing mutation system, Mothra, in a matter of hours.

Experience with this prototype implementation and off-line evaluation of code has, to date, failed to produce an instance where the mutations have not been effective in exposing bugs of this type. In fact, bugs were discovered using this mechanism in code previously thought to be correct and extensively tested. Included in this testing were dozens of programs and routines derived from published texts and papers, including instructional texts. Further experimentation and development is required to extend these results to testing for floating-point environmental bugs, but such efforts should result in a method of considerable power and utility, although their portability may be less than desired for a general testing method. Their efficiency is comparable to other mutation operators, and although mutation analysis, in general, may seem expensive to use, it may often be cheaper than unexpected failure of the software. If the software being tested is for something life-critical, then the investment to customize the mutations and run them should be well worth the effort. Additionally, research is currently under way to use multiprocessor computer systems to speed up mutation testing.[11-13] These techniques will be equally applicable to testing environmental mutants.

This work also hints at some possibilities of using selected mutation techniques to test somewhat different constructs than have traditionally been considered for mutation testing. Testing systems-level code is just one such possibility. Additionally, combining some of these notions with methods used to generate test data for mutation analysis may provide a means of automatically creating test data sets to test for these environmental errors.[9]

More generally, this work suggests some applications of mutation testing technology: to test for environmental bugs and perhaps as a tool to aid in the development of portable code. This work also suggests that the concept of mutation testing is more general and powerful than previously thought.

Acknowledgments

Richard DeMillo helped me clarify my initial thoughts on this topic. The referees made many useful suggestions that were incorporated into the final version of the paper.

References

1. Acree, A. T., ‘‘On Mutation,’’ PHD THESIS, Georgia Institute of Technology, 1980.
2. ANSI/IEEE,, *IEEE Standard for Binary Floating-Point Arithmetic,* IEEE, 1985. (ANSI/IEEE Standard, pp. 754-1985)
3. Budd, T. A., ‘‘Mutation Analysis of Program Test Data,’’ PHD THESIS, Yale University, 1980.
4. Bullard, C. and E. H. Spafford, ‘‘Testing Experience with MOTHRA,’’ *PROCEEDINGS OF 25TH SOUTHEAST ACM CONFERENCE*, Birmingham AL, April 1987.
5. Choi, B. J., R. A. DeMillo, E. W. Krauser, R. J. Martin, A. P. Mathur, A. J. Offutt, H. Pan, and E. H. Spafford, ‘‘The Mothra Tools Set,’’ *PROCEEDINGS OF THE 22ND HAWAII INTERNATIONAL CONFERENCE ON SYSTEMS AND SOFTWARE*, pp. 275-284, Kona, Hawaii, Jan 1989.
6. DeMillo, R. A., R. J. Lipton, and F. G. Sayward, ‘‘Hints on Test Data Selection: Help for the Practicing Programmer,’’ *COMPUTER*, vol. 11, no. 4, pp. 34-41, April 1978.
7. DeMillo, Richard A. and Eugene H. Spafford, ‘‘The MOTHRA Software Testing Environment,’’ *11TH NASA SOFTWARE ENGINEERING LABORATORY WORKSHOP*, Goddard Space Center, December 3, 1986.
8. DeMillo, R. A., D. Guindi, K. N. King, E. W. Krauser, A. J. Offutt, and E. H. Spafford, ‘‘Mothra Internal Documentation, Version 1.0,’’ TECHNICAL REPORT GIT-SERC-87/10, Software Engineering Research Center, Georgia Institute of Technology, 1987.
9. DeMillo, R. A. and A. J. Offutt, ‘‘Experimental Results of Automatically Generated Adequate Test Sets,’’ *PROCEEDINGS OF THE SIXTH ANNUAL PACIFIC NORTHWEST SOFTWARE QUALITY CONFERENCE*, Portland, Oregon, September 1988.
10. Howden, W. E., *Functional Program Testing & Analysis,* McGraw-Hill, 1987.
11. Krauser, E. W. and Aditya P. Mathur, ‘‘Program Testing on a Massively Parallel Transputer Based System,’’ *PROCEEDINGS OF THE ISMM INTERNATIONAL SYMPOSIUM*, pp. 67-71, Austin, Texas, USA, November 10-12 1986.

12. Krauser, E. W., Aditya P. Mathur, and Vernon Rego, ‘‘High Performance Testing on SIMD Machines,’’ *PROCEEDINGS OF THE SECOND WORKSHOP ON SOFTWARE TESTING, VERIFICATION AND ANALYSIS*, Banff, Alberta, Canada, July 19-21, 1988.

13. Mathur, Aditya P. and E. W. Krauser, ‘‘Modeling Mutation On a Vector Processor,’’ *PROCEEDINGS OF THE 10TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING*, Singapore, April 1988. (Previously released as Technical Report GIT-SERC-87/07, Georgia Institute of Technology, 1987)

14. Myers, G., *The Art of Software Testing*, John Wiley & Sons, New York, NY, 1979.

15. Offutt, A. J. and K. N. King, ‘‘A Fortran 77 Interpreter for Mutation Analysis,’’ *1987 SYMPOSIUM ON INTERPRETERS AND INTERPRETIVE TECHNIQUES*, pp. 177-188, ACM SIGPLAN, St. Paul, MN, June 1987.

16. Offutt, A. J., ‘‘Automatic Test Data Generation,’’ PHD DISSERTATION, Department of Information and Computer Science, Georgia Institute of Technology, 1988.

17. Ramamoorthy, C. V., S. F. Ho, and W. T. Chen, ‘‘On the Automated Generation of Program Test Data,’’ *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. SE-2, no. 4, IEEE, December 1976.

18. Schryer, N. L., ‘‘A Test of A Computer’s Floating-Point Arithmetic Unit,’’ COMPUTER SCIENCE TECHNICAL REPORT, AT&T Bell Laboratories, 1981.

19. Spafford, E. H. and J. C. Flaspohler, ‘‘A Report on the Accuracy of Some Floating Point Math Functions on Selected Computers,’’ *THE USENIX ASSOCIATION NEWSLETTER*, vol. 11, no. 2.

20. White, L. J., E. I. Cohen, and B. Chandrasekaran, ‘‘A Domain Strategy for Computer Program Testing,’’ TECHNICAL REPORT OSU-CISRC-TR-78-4, Ohio State University, 1978.

21. Woodward, M. R. and K. Halewood, ‘‘From Weak to Strong: Dead or Alive? An Analysis of Some Mutation Testing Issues,’’ *PROCEEDINGS OF THE SECOND WORKSHOP ON SOFTWARE TESTING, VERIFICATION AND ANALYSIS*, pp. 152-158, Banff, Alberta, Canada, July 19-21, 1988.