A Semantic Model of Program Faults

A. Jefferson Offutt
George Mason University

J. Huffman Hayes
Science Applications International Corp.

Abstract

Program faults are artifacts that are widely studied, but there are many aspects of faults that we still do not understand. In addition to the simple fact that one important goal during testing is to cause failures and thereby detect faults, a full understanding of the characteristics of faults is crucial to several research areas in testing. These include fault-based testing, testability, mutation testing, and the comparative evaluation of testing strategies. In this workshop paper, we explore the fundamental nature of faults by looking at the differences between a syntactic and semantic characterization of faults. We offer definitions of these characteristics and explore the differentiation. Specifically, we discuss the concept of "size" of program faults – the measurement of size provides interesting and useful distinctions between the syntactic and semantic characterization of faults. We use the fault size observations to make several predictions about testing and present preliminary data that supports this model. We also use the model to offer explanations about several questions that have intrigued testing researchers.

1 INTRODUCTION

Many testing techniques and activities revolve around program faults. We often view testing as a process of trying to cause failures, thereby uncovering faults in the software. Many testing techniques are specifically designed to identify faults, and we often evaluate testing techniques on the basis of their ability to detect faults. Fault-based adequacy criteria [4, 7, 11] measure the quality of a test set according to its effectiveness or ability to detect certain types of faults. Error seeding [8, 10, 16] is a technique for estimating the number of faults in the software. Artificial faults are seeded into the program, and the testing is evaluated based on the number of seeded faults found.

The same criticism has been levied against both fault-based testing and error seeding: the artificial faults are not necessarily representative of natural faults. The accuracy of the evaluation during error seeding and the efficacy of fault-based techniques depend on the actual faults. Although it is possible to avoid this problem by using natural faults, this option is expensive and difficult to automate, and makes it difficult to impose proper empirical controls.

In this paper, we argue that these problems, and others, can be better understood and at least partially solved by looking at a syntactic and semantic characterization of faults. Although this is not the first time faults have been viewed in this way, we present a conceptual model and attempt to explore its ramifications. In the rest of the paper, we develop these ideas, discuss several implications of this characterization, and present data that support the model.

2 THE SEMANTIC AND SYNTACTIC MODEL OF FAULTS

The IEEE standard definition of an *error* is a mistake made by a developer [1]. An error may lead to one or more faults. *Faults* are located in the text of the program. A fault is the difference between the incorrect program and some correct program. Note that a fault may be localized in one statement or may be textually dispersed into several locations in the program. Similarly, a fault may be repairable in many ways – that is, there may be many ways to correct a fault, each one leading to a correct, but different program.

This definition is in terms of the syntactic nature of a fault. If the fault is being inserted into the program, then the syntactic nature of the fault is described by the changes to the program. If the fault occurs naturally in the program, then the syntactic nature of the fault is described by the number of changes needed to correct the program. Examples of syntactic characterizations of faults include using an incorrect variable name, or checking to see if a called function fails. These are often things that programmers do by mistake – typos, or

Supported by the National Science Foundation under grant CCR-93-11967. First author's address: ISSE Department, 4A4, Fairfax, VA 22030-4444, phone: 703-993-1654, email: ofut@isse.gmu.edu. Second author's address: 1213 Jefferson Davis Highway, Suite 1300, Arlington, VA 22202, email: hayes@cg4.saic.com.

reading a design incorrectly.

A fault can also be characterized semantically. Each program P can be viewed as having a specification S that defines sets D (the input domain) and R (the output range), and a mapping from D to R ($D \xrightarrow{S} R$). The program may compute results on a superset of D, or if the input is not in D, P may produce output that is not in R (undefined).

A semantic characterization of a fault views the faulty program as containing a computation that produces incorrect output over some subset of the input domain. That is, the mapping of inputs to outputs ($D \xrightarrow{P} R$) is incorrect ($D \xrightarrow{P} R \neq D \xrightarrow{S} R$) for some subset of D.

What this characterization really does is give us different ways to look at a fault. A given fault has a syntactic aspect and a semantic aspect. This characterization is not particularly new or profound, nor is it particularly useful by itself. But this characterization illuminates some interesting aspects of faults when we consider the size of a fault. Generally, we say the *size* of a fault is the scope of the difference between a correct and incorrect version of the program.

We initially define the *syntactic size* of a fault to be the number of statements or tokens that need to be changed to get a correct program[1]. This definition suffers from the problem that there may be many correct versions of the program; thus we refine the syntactic size to be the *fewest* number of statements or tokens that need to be changed. Note that this does not necessarily provide the "best" fix to the program in any sense. We define the *semantic size* of a fault to be the relative size of the subdomain of D for which the output mapping is incorrect. Although the semantic size would ideally be based on a usage distribution that assigns a non-uniform probability to each input, it could be approximated by considering a uniform domain with each input having an equal probability of occurring, or by using some set of inputs generated for testing purposes. When we consider size, the syntactic and semantic characterizations of faults are very different.

Consider very "small" faults. For a syntactically small fault, one token or one statement may be incorrect. For a semantically small fault, P's behavior on a very small portion of D is incorrect. Clearly, a fault that is syntactically small can result in a fault that is very large semantically – that is, the syntactic fault can affect arbitrarily many inputs. Also, a major syntactic fault in P may affect only a few inputs. And of course, there is some intersection where small semantic faults can be modeled as small syntactic faults, and small syntactic faults can result in small semantic faults.

[1]Note that there are many ways to define the syntactic size, many based on the program representation. We choose this definition because it is reasonable and useful – we do not claim that it is the "best" way to define syntactic size.

As an example, consider the program fragment:

```
for i := 1 to n do
    B[i] := B[i]-1;
```

Suppose this program fragment contains the very small syntactic fault that the subtraction operator should have been addition. Although this is small syntactically, the fault will not only affect every input to the program, it will affect every element of `B` for every input, thus the fault is semantically very large.

As another example, consider the calculation of the mean versus the median of a list of numbers. Although the computation for the mean is very different syntactically from the computation of the median, there are many lists of numbers for which the mean and median are the same. Thus, making the wrong calculation would be syntactically large, but semantically small.

Finally, consider a program containing the very small syntactic fault that the `>=` operator should have been strictly `>`. This small syntactic fault will result in a fault that is also semantically small.

3 IMPLICATIONS

By considering this semantic model, and specifically the semantic size of faults, we can gain insight into a number of testing issues. The central issue is that much of the fault-based research has focused on faults that are small syntactically, without consideration of the semantic size. In the following subsections, we discuss how this model relates to several issues in testing.

3.1 Fault Seeding

Fault seeding refers to artificially introducing faults into programs, usually to measure the quality of testing or to empirically compare testing strategies. When we seed faults into programs we often seed syntactically small faults – they are simpler to define and manage. A more effective consideration would be the semantic size. If we insert a large semantic fault, then the fault is easy to detect via testing. Thus, if we are measuring the quality of testing, we are biasing our results toward the testing strategy, and if we are comparing two testing strategies, we will be less likely to detect any difference.

On the other hand, faults that are too small semantically will have the opposite effect. If we are measuring the quality of testing, we are biasing our results against the testing strategy, and if we are comparing two testing strategies, neither will be likely to work very well.

Hamlet [6] pointed out that most empirical comparisons of testing techniques have two problems: a particular collection of programs must be used; and a particular set of test data must be created. Both of these are examples of internal controls on the empirical process

and are the sorts of problems that are always present in any experiment. Internal control problems mean that the results of the experiment may not scale up and be true in all situations. We suggest that another potential problem is that if the techniques are compared based on the faults they find, a particular collection of faults must be used.

Studies using fault seeding have been questioned on the basis of whether the faults were "realistic" or "representative". Unfortunately, we do not know what a realistic, artificial fault is. Although a few studies have successfully used naturally occurring faults, this necessitates an expensive case study approach that is difficult to control scientifically. Based on the size model presented in this paper, we can reasonably consider a collection of faults to be "realistic" if they have a distribution of semantic sizes that is similar to that of real faults. Of course, we have little data about distributions of semantic fault sizes for naturally occurring faults, but this is something that can be measured relatively easily. A collection of artificially seeded faults can then be validated by measuring their semantic size. We suggest three ways to approximately measure the distribution of semantic sizes of seeded faults.

- Take the test cases that were used in the study and count how many test cases find each fault.
- Generate many random test cases and count how many test cases find each fault.
- Obtain inputs following a usage profile and count how many test cases find each fault.

If this approach succeeds, we could develop reasonable estimations of the semantic size of realistic faults. This could then be used to create a data base of programs that have artificial, but *representative*, faults.

3.2 Mutation Operators

Budd [2] discussed the concept of program neighborhoods. A *neighborhood* of a program P is a set of programs that are "close" to P. The program neighborhood concept was used to present the competent programmer hypothesis [4], which states that competent programmers produce programs that are "close" to being correct. This was in turn used to justify the operators that are used in mutation testing – the operators should create mutants that are in the neighborhood of the original program.

In testing classes, this topic consistently generates the question of whether the neighborhood should be semantically close or syntactically close. Budd's description was in terms of semantic neighborhoods, and DeMillo, Lipton, and Sayward's description was in terms of syntax. Discussions with the original researchers revealed that they really wanted changes in the program's functionality, and settled for changes in the text. Although mutation systems create mutants that are small syntactically, semantically small mutants would be harder to kill, thus have the potential to lead to higher quality tests. Mutants that are small syntactically but large semantically only generate noise; they add difficulty to the mutation process without increasing the testing value of the resulting test cases. This is evidenced by the fact that many mutants are trivially killed by almost any test case that reaches the mutated statement, and less directly because there appears to be a large amount of overlap in the mutants in the sense that a test case that kills one mutant will invariably kill many others.

Of course one of the problems with mutation testing has always been that of equivalent mutants – mutants that have no functional effect on the program and thus cannot be killed. In terms of the semantic characteristic of a fault, an equivalent mutant represents a fault whose semantic size is zero.

Considering the semantic size of mutants also relates to several other questions that have been troubling mutation researchers.

1. Why does there intuitively seem to be a close correlation between killing "hard-to-kill" mutants and detecting equivalent mutants? With the semantic size model, the answer to this question becomes obvious. Hard-to-kill mutants are mutants with very small semantic faults – and equivalent mutants have semantic fault size zero. Thus they are related because their semantic sizes are almost the same. From a testing perspective, it can be argued that semantically small mutants are more desirable – they lead to stronger test cases.

2. Why does selective mutation work? Selective mutation [14, 15] is an approximation technique that selects only mutants that are truly distinct from other mutants. Recent results have shown that of the 22 mutation operators used by the Mothra mutation testing system [3], test data generated to kill mutants produced by five operators are sufficient to kill mutants produced by the other operators.

   By considering the semantic model of faults, we can theorize that selective mutation is trying to use only operators that tend to produce mutants that have semantically small faults. If this theory is correct, the operators that work well empirically should produce mutants that are, on average, small semantically. One easily checked corollary to this theory is that selective mutants should contain a relatively high percentage of equivalent mutants. Of course it is unfortunate that if we

successfully create mutants with smaller semantic size, then we also make the equivalent mutant problem worse.

3. Why does the coupling effect hold? The coupling effect says that complex faults are coupled to simple faults in such a way that test data that detects all simple faults in a program will detect most complex faults [4]. The coupling effect has been supported experimentally in a study that compared test sets generated for mutants that involved changes in two places with test sets generated for single change mutants [12], and shown to hold probabilistically for large classes of programs [18]. Unfortunately, the coupling effect has not been adequately explained on an intuitive basis. Although this is speculative, it seems that the semantic model might at least partially explain the coupling effect. In the empirical study, simple faults were modeled as single change mutants, and complex faults were modeled as multiple change mutants. Thus, we could characterize the multiple change mutants as being syntactically larger than the single change mutants.

   There are two reasonable interpretations of the coupling effect within the semantic fault model. One is that as faults get larger syntactically, there is a tendency for the faults to also get larger semantically. If this is true, then faults that are larger syntactically will tend to be easier to find via testing, because they will fail on larger portions of the input space.

   A second interpretation that we believe is more likely is that there is a true relationship between syntactically small faults and semantically large faults. We can consider the *failure region* for a fault to be the portion of the input space that causes the fault to result in a failure. It might be the case that for every semantically large fault, there are one or more syntactically small faults such that there is a large overlap in the two faults' failure regions. In this case, we can expect the two faults to be coupled in the traditional sense.

This discussion brings out a key difference between fault seeding and mutation analysis. When doing fault seeding, we want faults that approximate natural faults as closely as possible – in our terms, by exhibiting a distribution of semantic fault sizes that matches the distribution of natural faults. Mutation, however, may not want a similar distribution of semantic fault size. Using faults that have smaller semantic size may lead to stronger testing.

3.3 Testability

Testability is a software metric that quantifies how difficult it is to test software. This "level of difficulty" could include the cost to generate test cases, write drivers or stubs, or determine correctness for a specific test case. The PIE assessment method for measuring testability [17] is based on the following testability definition: the probability that faults will result in observable failures for a given input distribution or test scheme. PIE implements this definition by computing three measurements: *execution*, which estimates the probability that a faulty statement will be reached; *infection*, which predicts the probability that a fault on a given statement will cause the dynamic data state of the program to become corrupted; and *propagation*, which predicts the probability that a corrupted data state will propagate through the execution stream to cause a corrupted output. These three probability estimates can be combined to gain overall predictions of the testability of statements, units, and programs.

The testability of a program is closely related to the semantic model. If a statement in a program has very low testability, we expect that faults associated with that statement will be small semantically. Likewise, if "existing" faults associated with a given statement are semantically large, we expect the statement to exhibit high testability.

True testability depends on faults, the code, and the test distribution. Our predictions of this unknown entity depend partially on simulated faults; the infection probability is measured by using mutation-like changes. In-depth understanding of representative faults could improve the infection probability estimate, thus increasing the validity of the testability estimates. This knowledge could also give us a more accurate knowledge of the benefit of testing, leading to more effective use of the measurements to complement software testing.

3.4 Impact Analysis

Goradia [5] has suggested a technique called impact analysis that estimates, for a given test case and statement, the "impact" that statement has on the output of the program. We suggest that this is related to the semantic model in the following sense. If a statement has a large impact on the program's output when averaged over a number of test cases, then faults that appear on or partially on that statement will tend to have a large semantic size.

4 EVALUATION

The model we are presenting in this paper is a conceptual tool and does not provide new testing techniques or directly provide new results about existing techniques.

Rather, it is a way of thinking about problems of existing testing techniques that may lead to new insights and solutions to those problems. In this section, we provide data from preliminary investigations into some of those problems, using the semantic model as a basis.

4.1 Equivalent Mutant Clusterings

As stated in Section 3.2, selective mutation tries to select only mutants that are truly distinct from other mutants. Results [14] have indicated that not all of the 22 mutation operators used by the Mothra mutation system are necessary; it has been found that test data that kill all mutants created by only five of the operators will usually kill almost all of the mutants created by all 22 operators. In terms of the semantic fault model, selective m

# of Faults	% of Test Cases
15	80 – 100%
14	20 – 52 %
26	.9% – 10%

Table 1: **Summary of Fault Size Data**

References

[1] *IEEE Standard Glossary of Software Engineering Terminology*. ANSI/IEEE Std 729-1983, 1983.

[2] T. A. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, November 1982.

[3] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. An extended overview of the Mothra software testing environment. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 142–151, Banff Alberta, July 1988. IEEE Computer Society Press.

[4] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[5] T. Goradia. Dynamic impact analysis: A cost-effective technique to enforce error-propagation. In *Proceedings of the 1993 International Symposium on Software Testing, and Analysis*, pages 171–181, Cambridge MA, June 1993.

[6] D. Hamlet. Theoretical comparisons of testing methods. In *Proceedings of the Third Symposium on Software Testing, Analysis, and Verification*, pages 28–37, Key West Florida, December 1989. ACM SIGSOFT 89.

[7] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4), July 1977.

[8] J. Knight and P. Ammann. An experimental evaluation of simple methods for seeding program errors. In *Proceedings of the Eighth International Conference on Software Engineering*, pages 337–342, London UK, August 1985. IEEE Computer Society.

[9] S. Meyers. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley Publishing Company Inc., 1992.

[10] H. D. Mills. On the statistical validation of computer programs. Technical report FSC 72-6015, IBM, 1972.

[11] L. J. Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, August 1990.

[12] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology*, 1(1):3–18, January 1992.

[13] A. J. Offutt and A. Irvine. Testing object-oriented software using the category-partition method. In *Seventeenth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA '95)*, pages 293–303, Santa Barbara, CA, August 1995.

[14] A. J. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf. Interprocedural static analysis of sequencing constraints. *ACM Transactions on Software Engineering Methodology*. to appear.

[15] A. J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *Proceedings of the Fifteenth International Conference on Software Engineering*, pages 100–107, Baltimore, MD, May 1993. IEEE Computer Society Press.

[16] K. Tewary and M. J. Harrold. Fault modeling using the program dependence graph. In *Proceedings of the Fifth International Symposium on Software Reliability Engineering*, pages 126–135, Monterey CA, November 1994.

[17] J. M. Voas, L. Morell, and K. W. Miller. Predicting where faults can hide from testing. *IEEE Software*, 8(2), March 1991.

[18] K. S. H. T. Wah. Fault coupling in finite bijective functions. *The Journal of Software Testing, Verification, and Reliability*, 5(1):3–47, March 1995.