

# Faults' Context Matters

Jaymie Strecker  
University of Maryland  
College Park, MD, USA  
strecker@cs.umd.edu

Atif M Memon  
University of Maryland  
College Park, MD, USA  
atif@cs.umd.edu

## ABSTRACT

When choosing a testing technique, practitioners want to know which one will detect the faults that matter most to them in the programs that they plan to test. Do empirical evaluations of testing techniques provide this information? More often than not, they report how many faults in a carefully chosen “representative” sample the evaluated techniques detect. But the population of faults that such a sample would represent depends heavily on the faults' context or environment—as does the cost of failing to detect those faults. If empirical studies are to provide information that a practitioner can apply outside the context of the study, they must characterize the faults studied in a way that translates across contexts. A testing technique's fault-detecting abilities could then be interpreted relative to the fault characterization. In this paper, we present a list of criteria that a fault characterization must meet in order to be fit for this task, and we evaluate several well-known fault characterizations against the criteria. Two families of characterizations are found to satisfy the criteria: those based on graph models of programs and those based on faults' detection by testing techniques.

## 1. INTRODUCTION

Among software testing researchers, a popular way to evaluate a testing technique is to find how many faults, out of a carefully chosen “representative” sample, the technique detects. In response to such evaluations, testing practitioners have been known to ask, “But does the technique detect the most *severe* faults?” These two points of view, the researcher's and the practitioner's, are at odds; neither can be inferred from the other.

It is not that researchers do not know or do not care about fault severity [4]. Rather, because severity ratings are highly subjective, it is impossible for a researcher to answer the practitioner's question without individually studying each of the programs and associated sets of severity-rated faults that the practitioner has in mind. Obviously, this point-wise

approach is impractical. But, fortunately, the dichotomy here between research and practice is an illusion. By taking into account the *context* in which faults occur, the researcher and the practitioner can reconcile their points of view.

The context of a fault determines how likely it is that the fault would occur and how important it is to eliminate the fault. Some kinds of faults are only possible in certain contexts (*e.g.*, in event-driven systems, mishandling events that occur in rapid succession). Sometimes the context even decides what counts as a fault. (“It's not a bug; it's a feature.”) Many factors can contribute to a fault's context: the requirements and specifications of the program containing the fault; the nature of the program, including its application domain, its design, and its connections to other programs; the way the program is developed, used, and maintained; characteristics of the organization developing the program; and more. Although the notion of fault context as presented here is fuzzy, it should be clear that every fault has a context, whether explicitly stated or not.

Each of the two supposedly irreconcilable points of view described earlier implies a context. The researcher's “representative” sample of faults assumes a knowledge of the kinds of faults that occur, and in what proportions, for the program under test [5, 10]. Whether an evaluation of a testing technique considers all faults to be equally important or “severe” faults to be more important, some system of importance ratings is implied. The second case is especially context-sensitive because the definition of “severe” depends, among other factors, on the program's users and developers. The perspectives of a researcher and a practitioner—or two researchers, or two practitioners—come into conflict when each assumes a different context.

How can we avoid this conflict? Among researchers, a partial solution would be for everyone to study testing techniques within the same context, *i.e.*, a set of benchmark programs with known faults, like Space and the Siemens programs [1], and a common set of assumptions about fault severity, user behavior, and so on. But foremost among the many problems with this approach is the practitioner's still unanswered question: *How will the technique perform in my context?* A better solution, which addresses this question, is to characterize the faults that a testing technique detects in a way that meets the following criteria:

1. provides useful information about the prevention, detection, repair, or cost of the faults,
2. is consistent and objective,
3. can be automated, and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SOQUA '07, September 3-4, 2007, Dubrovnik, Croatia  
Copyright 2007 ACM 978-1-59593-724-7/07/09 ...\$5.00.

4. does not rely on formal specifications or other information that may not be available.

The following example, based on an actual study [2], demonstrates how a fault characterization that meets the above criteria could be used to translate between disparate contexts. An empirical study finds that techniques  $T_1$  and  $T_2$  each detect, on average, 54%-55% of a sample of faults. A breakdown of the results by fault class shows that  $T_1$  detects fewer faults of class  $C_1$  than  $T_2$  does (42.8% *vs.* 66.7%), but more faults of class  $C_2$  (46.7% *vs.* 30.7%). Suppose that a tester wants to know which of  $T_1$  and  $T_2$  would likely reveal more faults in a certain program. In previous versions of the program, faults of class  $C_1$  caused twice as many user-reported failures as faults of class  $C_2$ . Considering these facts, the tester opts for  $T_2$ . What if the tester had not understood how her context and the study’s context differed? Under the mistaken assumption that the two techniques would perform equally well, the tester could easily miss out on the advantages of  $T_2$ .

In the rest of this paper, we evaluate a variety of ways of characterizing faults against the list of criteria above. This paper aims to convince the reader of the following points:

- Empirical results regarding fault detection in testing can be applied in other contexts only if the faults in the study are characterized following the criteria above.
- Several well-known ways to characterize faults—including semantic size, syntactic characteristics, and taxonomies like the IEEE standard classification—fail to meet all four criteria.
- Faults can be characterized with respect to their detection by well-defined testing techniques or their effect on certain program models, in ways that satisfy the criteria.

## 2. VIEWING FAULTS SYNTACTICALLY AND SEMANTICALLY

Offutt and Hayes [10] distinguish between *syntactic* and *semantic* characterizations of faults. Syntactically, a fault is a difference in source code between a correct and an incorrect version of a program. The fault’s *syntactic size* is, roughly speaking, the size of the textual difference in source code. The categories of mutation faults are a prime example of a fault characterization based on a syntactic perspective. Semantically, a fault is a difference in the input-output mappings that the correct and incorrect program versions induce, and its *semantic size* is, again, the size of the difference.

As Offutt and Hayes [10] point out, of the two views, the semantic view is more closely related to program execution and, hence, to testing. Yet the syntactic view persists in the testing literature, perhaps because it aligns with a programmer’s perspective during debugging. The trouble with syntactic characterizations of faults is that their relation to fault detection does not translate across contexts, or even across different parts of the same program—failing to meet Criterion 1. Figure 1 demonstrates this.

As a schema for characterizing faults, semantic size also presents problems. In an empirical study, Offutt and Hayes [10] measure the semantic size of faults by executing the correct and faulty programs with a random sample of inputs. The

```
1 x = null;
2 if (false) {
3     if (x == null) // if (x != null)
4         x.foo();
5 }
6 if (x == null) // if (x != null)
7     x.foo();
```

Figure 1: In lines 3 and 5, the same mutation operator has been applied, changing `!=` to `==`. The program never fails at line 3 but always fails at line 5.

```
1 print("Pick a number between 1 and 100.");
2 read(number);
3 if (number == 4058295011)
4     launchMissiles(); // Oops
```

Figure 2: The fault in line 4 is semantically small but potentially costly.

main problem with this measure is that it is poorly defined, in that it does not specify how many inputs should be sampled. A secondary problem is that a fault’s semantic size is not necessarily related to its elusiveness, assuming one is using a technique more sophisticated than random or weighted random testing. Nor is semantic size necessarily related to cost, as Figure 2 shows. This is true even if the random sampling of inputs is based on operational profiles. To summarize, semantic size fails to meet Criterion 2 and, arguably, Criterion 1.

## 3. VIEWING FAULTS THROUGH PROGRAM MODELS

Along the spectrum from syntactic to semantic views of faults lie interpretations of faults based on program models. Like the syntactic perspective, these interpretations are based on static properties of a faulty program, but, like the semantic perspective, the properties of interest are closely related to the program’s dynamic behavior. Harrold *et al.* [5] offer a prime example of such an interpretation. Refining a fault model by Howden [6], which classifies faults by their effect on the paths followed and the function computed by the program, they further classify faults as either *structural* or *statement-level* for the purpose of fault seeding. The latter classification is based on the way a fault affects the program dependence graph, a representation of the control and data dependencies in the program: structural faults alter the graph, while statement-level faults leave it unchanged.

It should be clear that the approach of Harrold *et al.* meets Criteria 2, 3, and 4. What about Criterion 1? This approach may indeed provide useful information about the cost of detecting and repairing a fault. For example, an abundance of faults of class *Incorrect Expression in Predicate* in a program’s latest release may persuade its testers to use a more rigorous predicate-coverage criterion for the next version. As for the cost of repairing a fault, the program dependence graph provides the information needed to perform a change-impact analysis of the faulty node(s).

Faults may be characterized in terms of other graph models as well. For example, Dinh-Trong *et al.* [3] propose a taxonomy of mutation faults for UML class diagrams. One

can imagine a rudimentary fault taxonomy that would apply to any kind of graph model, with categories like *Extra Node (and Edge)*, *Missing Edge*, and *Intra-Node Fault*. Whether such a taxonomy would meet our four criteria depends in part on the taxonomy’s relationship to development or maintenance tasks (Criterion 1) and the graph model’s ability to be generated automatically from the program (Criterion 3).

## 4. VIEWING FAULTS THROUGH TESTING TECHNIQUES

One problem with characterizing faults by their semantic size, as Section 2 noted, is that its empirical measurement is loosely specified and, hence, prone to inconsistency. A simple improvement would be to specify how many random inputs should be tried—say, enough to cover all functions in the program at least once. This suggests a more general way to characterize faults: by the testing techniques that detect them.

By *testing technique*, we mean any process that yields a test suite (an ordered list of test cases) and a test oracle. Faults may be characterized in terms of their detection by testing techniques, either in an absolute sense (*e.g.*, technique  $T$  can never detect fault  $F$ ) or a probabilistic sense (*e.g.*, 60% of a large sample of test suites produced by  $T$  detect  $F$ ). The first case can arise in GUI testing, for example, in which the test oracle looks only at the state of the GUI [13]. For some programs, such an oracle would not be able to observe failures that corrupt the file system or an external database.

More refined fault characterizations can be created by considering several testing techniques at once. Fault  $F_1$  may always be detected by technique  $T_1$  but never by technique  $T_2$ . Techniques  $T_3$ ,  $T_4$ , and  $T_5$  may detect fault  $F_2$  20%, 50%, and 80% of the time, respectively. This perspective on faults is reminiscent of relations like *More Powerful*, *Better*, and *Probbetter*, which compare two coverage criteria by comparing their likelihood of including failure-causing inputs [12].

At first, the idea of using testing techniques to characterize faults in studies of testing techniques may seem like circular reasoning. However, consider how such a characterization could be used in comparing a new testing technique,  $T_1$ , to two established techniques,  $T_2$  and  $T_3$ . Let us say, arbitrarily, that a testing technique *consistently* detects a fault if at least 80% of a sample of test suites produced by the technique detect the fault. Suppose that  $T_1$  consistently detects 90% of the faults that  $T_2$  consistently detects but only 30% of those that  $T_3$  consistently detects, and  $T_1$  is cheaper to apply than the combination of  $T_2$  and  $T_3$ . A tester who has been using  $T_2$  and  $T_3$  in tandem to test a program would have more reason to switch to  $T_1$  if  $T_3$  has revealed only a few faults than if  $T_3$  has revealed many faults that  $T_2$  has not.

Any characterization of faults in terms of their detection by certain testing techniques should satisfy all of our criteria, assuming that the testing techniques themselves meet Criteria 2–4 and are commonly used in practice (Criterion 1). This family of characterizations could be further generalized by allowing validation and verification techniques other than testing.

## 5. OTHER WAYS TO CHARACTERIZE FAULTS

In an empirical comparison of structural testing, functional testing, and code reading (upon which the example in Section 1 was based), Basili and Selby [2] classify the faults studied using two orthogonal schemes. One scheme has the categories *Omissive* and *Commissive*; the other, categories *Initialization*, *Control*, *Data*, *Computation*, *Interface*, and *Cosmetic*.

Both schemes miss some of our criteria. To decide whether to label a fault *Omissive* or *Commissive*, one has two options: analyze the fault’s relation to the program’s specification, either by obtaining it or by inferring it (violating Criteria 4 or 2, respectively), or look at syntactic characteristics of the fault, *i.e.*, whether code was added or modified to fix it (violating Criterion 1). (The second option is reminiscent of Munson’s and Nikora’s [8] method for counting faults.) In the second fault-classification scheme, the boundaries between some of the six categories are fuzzy, leaving them open to interpretation (violating Criterion 2). For example, passing an array of uninitialized values to a function that assumes the array is initialized with zeros could be either an *Initialization* fault or an *Interface* fault. Basili and Selby [2] themselves note that their classification schemes are somewhat subjective.

Parts of IEEE Standard 1044-1993 (“IEEE Standard Classification for Software Anomalies”) suffer from similar problems. Of the high-level categories in this classification, those that pertain to faults are *Logic*, *Computation*, *Interface/Timing*, *Data Handling*, and *Data*. Each category is further divided into sub-categories. The illustration above of the ambiguity between the *Initialization* and *Interface* categories applies here as well (violating Criterion 2), since *Data Handling* and *Interface/Timing* have equivalent sub-categories. Some other sub-categories could not always be identified objectively and automatically (violating Criteria 2 and 3), at least not without detailed change logs (violating Criterion 4). These sub-categories (and their parent categories) include *Misinterpretation (Logic)*, *I/O Timing Incorrect (Interface/Timing)*, *Data Referenced Out of Bounds (Data Handling)*, and *Output Data Incorrect or Missing (Data)*.

In practice, faults are often classified by their severity. Unfortunately, as noted in Section 1, the notion of severity translates poorly from one context to another, as it depends on factors such as who is using the system and how. Operational profile testing [9] may detect the faults that users are most likely to encounter, but even these faults are not necessarily the most severe (*cf.* Figure 2). Thus, fault severity fails to satisfy Criterion 4.

In an interesting inversion of this paper’s point of view, Andrews *et al.* [1] present evidence that, for the purpose of testing, faults made accidentally by programmers and those generated by mutation operators do *not* significantly differ.

## 6. CONCLUSIONS

In evaluating software testing techniques, researchers are, it seems, charged with the infeasible task of choosing a sample of faults that reflects what other researchers or practitioners might encounter in a generic “real world”. What makes the task infeasible is that the kinds of faults that arise and the kinds of faults that testers want to discover are determined by a complex set of contextual factors that vary from case to case. Researchers can sidestep the infeasible task, however, by characterizing the faults they study

in ways that make sense across contexts. In doing so, they would reduce threats to validity in their own studies, and they would enable practitioners and fellow researchers to take results from the study's context and interpret them in their own contexts.

For software-testing studies, some ways of characterizing faults are more fit than others. This paper presented a list of criteria (Section 1) that we believe to be necessary, though perhaps not sufficient, conditions for an effective fault characterization. Criterion 1 is clearly vital; otherwise, characterizing faults would add no value to software-testing studies. Without Criteria 2 and 3, which are closely linked, researchers and practitioners would likely balk at the cost of integrating fault characterizations into their work. Criterion 4 ensures that a fault characterization can be applied to the large population of programs, including open-source software from sites such as <http://sourceforge.net>, that lack formal specifications and other artifacts from the development process.

Using these four criteria, this paper evaluated several well-known ways of characterizing faults against the criteria. Two families of characterizations were found to satisfy all of the criteria: those based on graph models of programs (Section 3) and those based on faults' detection by testing techniques (Section 4). Both families of fault characterizations, we believe, address the issue raised by Harrold *et al.* [5]: "Although there have been studies of fault categories... there is no established correlation between categories of faults and testing techniques that expose those faults."

## 7. FUTURE DIRECTIONS

To date, the challenge of understanding how one fault differs from another, or why one technique detects a fault that another misses, has perhaps received less attention than it deserves. As a result, many areas for future work lie open. These include developing new fault characterizations relevant to testing, building tools to characterize faults automatically, and, of course, making it standard practice to characterize the samples of faults used in software-testing studies.

As Section 3 argued, one promising basis for fault characterizations is program models such as a program dependence graphs. To facilitate such fault characterizations, tools that can efficiently seed (*cf.* [5]) and analyze faults with respect to the program dependence graph need to be developed for C and Java programs and made publicly available. Other graph-based fault characterizations would require similar tools. Besides the program models suggested here, others (including program invariants and other non-graph-based abstractions) may prove useful in characterizing faults.

Section 4 pointed to testing techniques as another potentially profitable basis for new fault characterizations. A logical first step in this direction would be to revisit earlier empirical comparisons of testing techniques and investigate the overlap in faults detected by different techniques. Data repositories such as those of Rothermel *et al.* [11] and Memon [7] would aid this research.

Any proposed scheme for characterizing faults would have to be evaluated empirically to show that it is related to testing outcomes for real programs. Lessons learned from these evaluations would help us understand whether the criteria listed in this paper are sufficient and, if not, how they should evolve.

## 8. ACKNOWLEDGEMENTS

This work was partially supported by the US National Science Foundation under NSF grant CCF-0447864 and the Office of Naval Research grant N00014-05-1-0421.

## 9. REFERENCES

- [1] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 402–411, 2005.
- [2] V. R. Basili and R. W. Selby. Comparing the effectiveness of software testing strategies. *IEEE Trans. Softw. Eng.*, 13(12):1278–1296, 1987.
- [3] T. Dinh-Trong, S. Ghosh, R. France, B. Baudry, and F. Fleury. A taxonomy of faults for UML designs. In *2nd MoDeVa workshop - Model design and validation*, 2005.
- [4] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 329–338, Washington, DC, USA, 2001. IEEE Computer Society.
- [5] M. J. Harrold, A. J. Offutt, and K. Tewary. An approach to fault modeling and fault seeding using the program dependence graph. *J. Syst. Softw.*, 36(3):273–295, 1997.
- [6] W. E. Howden. Reliability of the path analysis testing strategy. *IEEE Trans. Softw. Eng.*, SE-2(3):208–215, 1976.
- [7] A. M. Memon. Software-testing benchmarks. <http://www.cs.umd.edu/~atif/Benchmarks/>.
- [8] J. C. Munson and A. P. Nikora. Toward a quantifiable definition of software faults. In *ISSRE '02: Proceedings of the 13th International Symposium on Software Reliability Engineering*, pages 388–395, Washington, DC, USA, 2002. IEEE Computer Society.
- [9] J. D. Musa. Operational profiles in software-reliability engineering. *IEEE Softw.*, 10(2):14–32, 1993.
- [10] A. J. Offutt and J. H. Hayes. A semantic model of program faults. In *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, pages 195–200, New York, NY, USA, 1996. ACM Press.
- [11] G. Rothermel, S. Elbaum, A. Kinneer, and H. Do. Software-artifact infrastructure repository. <http://sir.unl.edu/portal/>.
- [12] E. J. Weyuker. Can we measure software testing effectiveness? In *Proceedings of the 1st International Software Metrics Symposium*, pages 100–107, 1993.
- [13] Q. Xie and A. M. Memon. Designing and comparing automated test oracles for GUI-based software applications. *ACM Trans. Softw. Eng. Methodol.*, 16(1):4, 2007.