International Journal of Advances in Engineering Research http://www.ijaer.com

(IJAER) 2013, Vol. No. 5, Issue No. I, January **ISSN: 2231-5152**

STUDY ON TEST-SUITE AND FAULT CHARACTERIZATION

Sanjay Bharwaj

Abstract:

Since there is currently no standard way to choose these characteristics, the selection was necessarily somewhat improvised but was driven by earlier research. Although the literature does not directly suggest viable fault characteristics, it does clearly point to certain test-suite characteristics. Because the test suites and faults in this experiment were generated automatically, characteristics related to human factors did not need to be considered. The independent variables in this experiment are characteristics of faults and test suites hypothesized to affect the probability of fault detection. To make this experiment practical to perform, replicate, and apply in practice, only characteristics that can be measured objectively and automatically were considered.

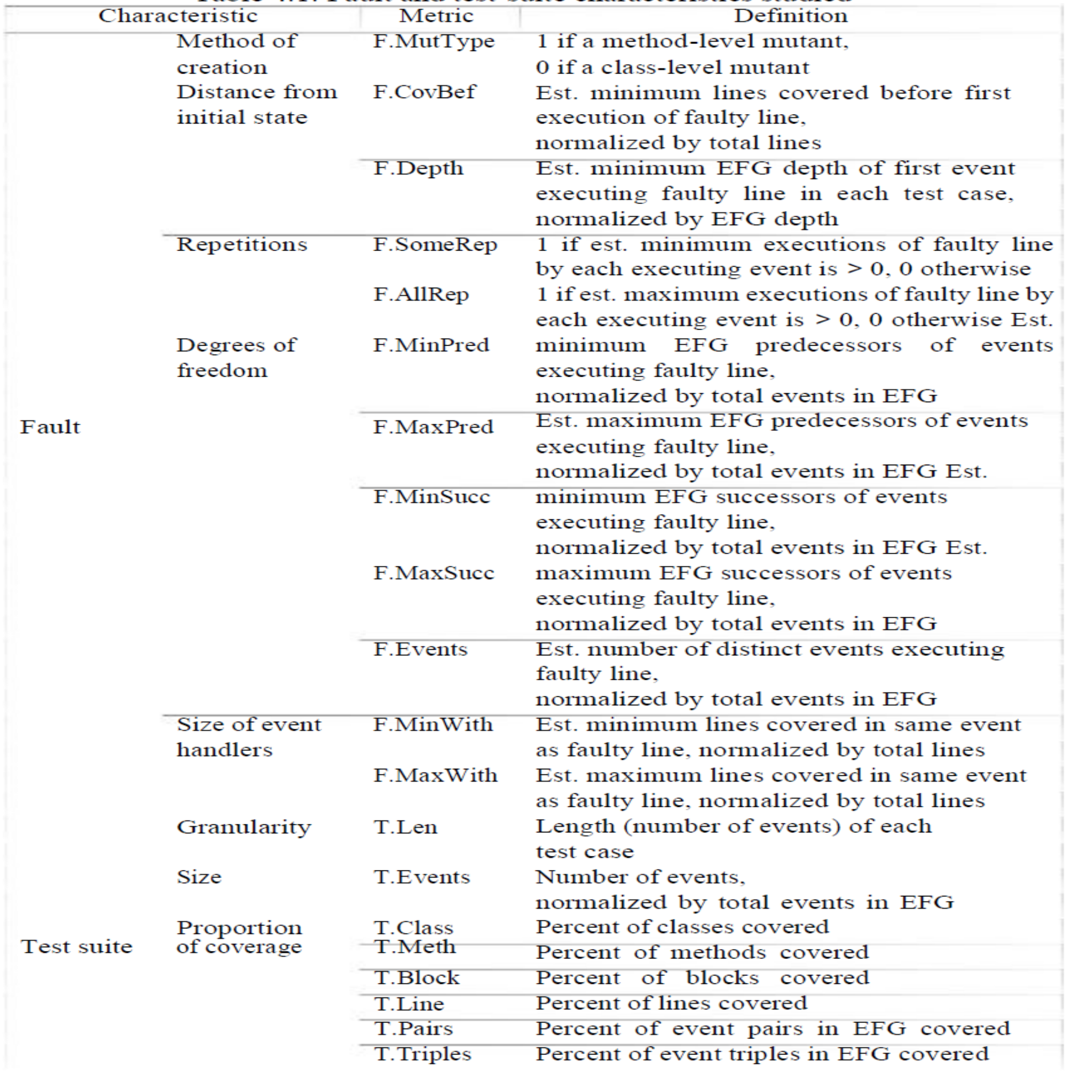
Key words: fault characteristics, test-suite characteristics, probability of fault detection.

Introduction:

It should be noted that, while truly objective measures of faults—measures of intrinsic properties, independent of any test set used to measure them—would be derived from static analysis, static analysis of GUI-based applications is still under development. In this experiment, most characteristic measures were derived from execution data from the *test pool*, the set of all test cases used in the experiment. This is not entirely objective because a different test pool would result in different measurements. Also, it threatens the assumption of independent sampling of data points—an issue examined in Section 4.4.6. However, measures that are closely tied to the test pool (e.g., those averaged across the test cases in the pool) were avoided. The test pool can be seen as an instrument used to estimate the true values of the measures (e.g., the minimum number of GUI events that must be executed to reach the faulty code). As the test pool size increases, the estimates converge to the true values.

Each characteristic describes some property of a fault or a test suite, such as the degrees of freedom in execution of the faulty code or the proportion of the application covered by the test suite. There are often multiple metrics to measure a characteristic, and prior to analyzing the data it is not clear which best predicts Cov or Det. The rest of this section lists the fault and test-suite metrics explored in this experiment, organized by the characteristic they are intended to measure. These are summarized in Table 1.

Table 4.1: Fault and test-suite characteristics studied



Characteristic		Metric	Definition
Fault	Method of creation	F.MutType	1 if a method-level mutant, 0 if a class-level mutant
	Distance from initial state	F.CovBef	Est. minimum lines covered before first execution of faulty line, normalized by total lines
		F.Depth	Est. minimum EFG depth of first event executing faulty line in each test case, normalized by EFG depth
	Repetitions	F.SomeRep	1 if est. minimum executions of faulty line by each executing event is > 0, 0 otherwise
		F.AllRep	1 if est. maximum executions of faulty line by each executing event is > 0, 0 otherwise Est.
	Degrees of freedom	F.MinPred	minimum EFG predecessors of events executing faulty line, normalized by total events in EFG
		F.MaxPred	Est. maximum EFG predecessors of events executing faulty line, normalized by total events in EFG Est.
		F.MinSucc	minimum EFG successors of events executing faulty line, normalized by total events in EFG Est.
		F.MaxSucc	maximum EFG successors of events executing faulty line, normalized by total events in EFG
		F.Events	Est. number of distinct events executing faulty line, normalized by total events in EFG
	Size of event handlers	F.MinWith	Est. minimum lines covered in same event as faulty line, normalized by total lines
		F.MaxWith	Est. maximum lines covered in same event as faulty line, normalized by total lines
Test suite	Granularity	T.Len	Length (number of events) of each test case
	Size	T.Events	Number of events, normalized by total events in EFG
	Proportion of coverage	T.Class	Percent of classes covered
		T.Meth	Percent of methods covered
		T.Block	Percent of blocks covered
		T.Line	Percent of lines covered
		T.Pairs	Percent of event pairs in EFG covered
		T.Triples	Percent of event triples in EFG covered

Review of literature

One fault characteristic studied is the method of creation, which for this experiment is some form of mutation. Although there are too many mutation operators to study each individually, they fall into two categories: class-level (e.g., changing the type of a data member) and method-level (e.g., inserting a decrement operator at a variable use). Class-level and method-level mutations were previously studied by Strecker and Memon [58, 60], whose results were inconclusive but suggested that class-level and method-level faults may be differently susceptible to detection. The label for the metric of mutation type is F.MutType.

Another fault characteristic is the distance of faulty code from the initial state. Faults residing in code that is "closer", in some sense, to the beginning of the program are probably easier to cover and may be easier to detect. One metric measuring this is the minimum number of source-code lines that must be covered before the faulty line is executed for the first time (F.CovBef). This can be estimated by running the test pool with program instrumentation to collect coverage data.

In a GUI-based application, a faulty line may lie in the event handler of one or more events. These events can be associated with the line by collecting coverage data for each event in each test case of the test pool. The minimum EFG depth of the events associated with a faulty line (F.Depth) is another way to measure the distance of the line from the initial state.

The repetitions in which the faulty code is executed may affect fault detection. Faults that lie in code that, when executed, tends to be executed multiple times by iteration or recursion may be easier to detect. Since, for the applications studied, the exact number of times a line is executed depends closely on the test case, two binomial metrics are studied instead. One is whether the the line is *ever* executed more than once by an event handler (F.SomeRep). The other is whether the line is *always* executed more than once by an event handler (F.AllRep).

Another fault characteristic that may affect fault detection is the degrees of freedom in execution of the faulty code. In GUI-based applications, an event handler can typically be executed just after any of several other event handlers.

Faulty code executed by an event that can be preceded or succeeded by many other events may be easier to cover, and it is not clear whether it would be more or less susceptible to detection. The minimum or maximum number of event predecessors or successors associated with a faulty line (F.MinPred, F.MaxPred, F.MinSucc, F.MaxSucc) can be estimated by associating coverage data from the test pool with the EFG. Faulty code executed by more events may also be easier to cover and either more or less susceptible to detection. The number of events executing the faulty code (F.Events), too, can be estimated with coverage data from the test pool.

Morgan et al. [39] report that program size affects fault detection in testing, so the size of the event handler(s) that execute a faulty line may similarly have an effect. Event-handler size can be measured as the minimum or maximum number of lines covered by each event handler that executes the faulty line (F.MinWith, F.MaxWith).

Material and method

Test-suite characteristics

For test suites, one interesting characteristic is the granularity of test cases— the amount of input provided by each test case. In GUI testing, granularity can easily be measured by the length (number of events) of a test case (T.Len). In this experiment, the length of the test cases in a test suite could be measured either by taking the average of different-length test cases in a suite or by constructing each suite such that its test cases have a uniform length. The latter approach is chosen because it has a precedent in previous work and because the assumption of uniform-length test cases, though unnecessary, is not unrealistic for model-based GUI testing. Suites made up of longer test cases may reach “deeper” program states, enabling them to cover and detect more faults.

Clearly, the characteristic of test-suite size can affect fault detection: larger test suites are likely to cover and detect more faults. An important question studied in this experiment is whether they do so when other factors, such as the suite’s coverage level, are controlled for. In some studies, test-suite size is measured as the number of test cases. But for this experiment, since different suites have different test-case lengths, a more meaningful metric is the total number of events in the suite, which is the product of the test-case length and the number of test cases (T.Events).

Another test-suite characteristic that can affect fault detection is the proportion of the application covered. Obviously, the more of an application's code a test suite covers, the more likely it is to cover a specific line, faulty or not. It may also be likely to detect more faults. The proportion of coverage may be measured by any of the myriad coverage metrics proposed over the years. This experiment considers several structural metrics—class (T.Class), method (T.Meth), block (T.Block), and line coverage (T.Line)—because of their popularity and tool support. For GUI-based applications, additional coverage metrics based on the event-flow graph (EFG) are available. Event coverage (coverage of nodes in the EFG) turns out *not* to be a useful metric for this experiment because each suite is made to cover all events. However, coverage of event pairs (EFG edges or length-2 event sequences; T.Pairs) and event triples (length-3 event sequences; T.Triples) is considered. (Longer event sequences could have been considered as well, but length 3 seemed a reasonable stopping point for this experiment. Since this experiment does show coverage of length-2 and length-3 sequences to be influential variables, future experiments can study coverage of longer sequences.)

The first stage of the experiment involves building and collecting data from a sample of test suites and a sample of faults. One of the contributions of this work is to make data and artifacts from this experiment—products of thousands of computation-hours and hundreds of person-hours—available to other researchers as a software-testing benchmark[1]. The artifacts—including source code and configuration files for the applications under test, GUI models created by GUITAR and tailored by the author and scripts

Table 4.2: Applications under test

	Lines	Classes	Events	EFG edges	EFG depth	Data points
CrosswordSage 0.3.5	2171	36	98	950	6	2230
FreeMind 0.7.1	9382	211	224	30146	3	970

-

used by the author—describe the experiment setup more precisely than prose ever could.

GUI testing of the applications was performed with tools in the GUITAR suite. To make the applications more amenable to these tools, a few modifications were made to the applications' source code and configuration files (e.g., to make file choosers open to a Furthermore, the test suites, faults, and data about them provided in the benchmark can be reused by researchers to perform new studies.

Conclusion

Not only should the sample of test suites be large and replicable, but it should also be an independent sample. In other words, the test suites in different ⟨*test suite, fault*⟩ pairs should not be related to one another. For this reason, a unique set of tests was generated for each ⟨*test suite, fault*⟩ pair. (The alternative would be to form each test suite by selecting, with replacement, a subset of a large pool of test cases, as was done in the preliminary study in Appendix A.)

Each test suite satisfies two requirements. First, it covers every event in the application's EFG at least once. This is to avoid obvious conclusions—i.e., that faults in code only executed by one event cannot be covered or detected if the event is not executed. Second, its test cases are all the same length. This is so that test-case length can be studied as an independent variable. The length must be greater than or equal to the depth of the EFG to ensure that all events can be covered. The maximum test-case length studied in this experiment is 20.

Model-based GUI testing has the advantage of being automated, but this is tempered by the fact that existing tools for generating and executing GUI test cases are immature. Also, the EFG is only an approximation of actual GUI behavior; because of enabling/disabling of events and other complex behavior in the actual GUI, not every test case generated from the EFG model is executable. For these reasons, each test suite must be generated carefully to ensure that every test case runs properly.

Bibliography

[1] *First International Workshop on Testing Techniques and Experimentation Benchmarks for Event-Driven Software* (Apr. 2009).

[2] Agrawal, H., Horgan, J. R., Krauser, E. W., and London, S. Incremental regression testing. In *Proceedings of ICSM '93* (Washington, DC, USA, 1993), IEEE Computer Society, pp. 348-357.

[3] Agresti, A. *An introduction to categorical data analysis*, second ed. John Wiley & Sons, 2007.

[4] Andrews, J. H., Briand, L. C., and Labiche, Y. Is mutation an appropriate tool for testing experiments? In *Proceedings of ICSE '05* (2005), pp. 402-411.

[5] Andrews, J. H., Briand, L. C., Labiche, Y., and Namin, A. S. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Trans. Softw. Eng. 32*, 8 (2006), 608-624.

[6] Basili, V. R. Software development: a paradigm for the future. In *Proceedings of COMPSAC '89* (1989), IEEE Computer Society, pp. 471-485.

[7] Basili, V. R., and Perricone, B. T. Software errors and complexity: an empirical investigation. *Commun. ACM 27*, 1 (1984), 42-52.

[8] Basili, V. R., and Selby, R. W. Comparing the effectiveness of software testing strategies. *IEEE Trans. Softw. Eng. 13*, 12 (1987), 1278-1296.

[9] Basili, V. R., Shull, F., and Lanubile, F. Building knowledge through families of experiments. *IEEE Trans. Softw. Eng. 25*, 4 (1999), 456-473.

[10] Briand, L. C., Melo, W. L., and Wst, J. Assessing the applicability of fault-proneness models across object-oriented software projects. *IEEE Trans. Softw. Eng. 28*, 7 (2002), 706-720.

[11] Brooks, F. P. *The Mythical Man-Month: Essays on Software Engineering*, anniversary ed. Addison-Wesley Pub. Co., 1995.

[12] Cai, K.-Y., Li, Y.-C., and Liu, K. Optimal and adaptive testing for software reliability assessment. *Information and Software Technology 46*, 15 (2004), 989-1000.

[13] Clark, B., Devnani-Chulani, S., and Boehm, B. Calibrating the COCOMO II post-architecture model. In *Proceedings of ICSE '98* (Washington, DC, USA, 1998), IEEE Computer Society, pp. 477-480.

[14] Clarke, L. A. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng. 2*, 3 (1976), 215-222.

[15] Davis, M. D., and Weyuker, E. J. Pseudo-oracles for non-testable programs. In *Proceedings of the ACM '81 conference* (New York, NY, USA, 1981), ACM, pp. 254-257.

[16] Do, H., and Rothermel, G. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Trans. Softw. Eng. 32*, 9 (2006), 733-752.