An Empirical Evaluation of Test Adequacy Criteria for Event-Driven Programs

Jaymie Strecker
Department of Computer Science
University of Maryland
College Park, MD 20742

November 30, 2006

Abstract

In model-based testing, particularly of event-driven programs, the problem of balancing reliability with cost challenges testers. This study provides empirical data to help testers decide how best to achieve this balance. We empirically evaluate the size and fault-detection ability of test suites that satisfy various adequacy criteria, some specific to event-driven software. Our results, based on one subject program, show that test suites that exercise interactions between event handlers detect substantially more faults, but are substantially larger, than test suites that only guarantee coverage of statements or individual event handlers.

1 Introduction

A major problem in model-based testing is deciding when to stop. By *model-based testing*, we mean generating (or, equivalently, selecting) test inputs from equivalence classes of inputs defined by an abstraction of the program under test. Different classes of inputs cause the program to behave differently. For most programs there are too many input classes—in other words, too many behaviors, too many paths—to test all of them within a reasonable amount of time. But the more we test, the more program faults we are likely to expose. We must compromise, then, by testing the few inputs, relative to the huge input space, that are most likely to expose faults.

This compromise is especially difficult to make when the program under test is event-driven. *Event-driven* programs (also called *reactive* programs) are made up of many smaller programs called *event handlers* that execute in response to signals called *events*. An obvious way to apply model-based testing to event-driven programs is to consider each event or sequence of events that the user could activate to be its own input class. In many event-driven programs, however, events can fire in almost any order. Different permutations of event sequences can drive the program into different states. But, since the number of unique length-n event sequences grows exponentially with n, we can only hope to test a few.

How many event sequences, then, should we test? An *adequacy criterion* tells us when we have tested enough of the program. ("Enough" depends on how reliable the program must be.) A *coverage metric* measures how much of the application has been tested so far. For example, if the coverage metric is the percentage of source code statements exercised, our adequacy criterion might be to cover 80% of statements. Many different coverage metrics have been proposed—statement, branch, and data-flow, to name a few—and each induces a family of adequacy criteria. Memon et al (Memon et al., 2001) have observed that existing criteria may adapt poorly to the unique structure of event-driven programs, and they propose several new criteria for graphical-user-interface-based (GUI-based) programs. The purpose of this work is to empirically compare the fault-detection ability and cost of applying event-aware adequacy criteria relative and some popular event-ignorant criteria.

Research contributions of this work include:

- An empirical evaluation of adequacy criteria based on statement, event, and event-interaction coverage for one moderate-sized subject program.
- An analysis of the faults that different adequacy criteria tend to expose.

The next section reviews adequacy criteria proposed in previous work. Section 3 describes the design of our empirical study and presents the study results. In Section 4 we interpret the results and point to opportunities for future work.

2 Background and Related Work

Of the multitude of adequacy criteria available today, those based on program statements, blocks, or branches are among the simplest and most popular. For event-driven programs, we hypothesize that these criteria split program inputs into input classes that are too coarse. We can test every program statement but still miss obvious faults—faults that hinder most of the program's users.

Figure 1 illustrates how this can happen. The figure shows a snapshot of TerpCalc, a calculator program in the TerpOffice 2.0 suite[1] written in Java by undergraduates at the University of Maryland. The sequence of circles and arrows shows a sequence

[1]`http://www.cs.umd.edu/users/atif/newsite/terpoffice.htm`

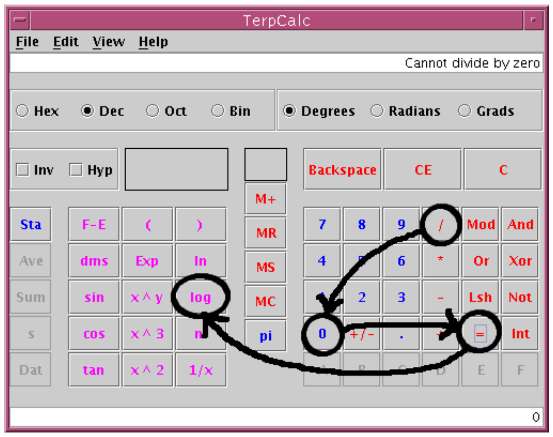

Figure 1: A sequence of events that causes TerpCalc to throw an uncaught exception.

of events that the user can perform. Initially, the upper text field reads “0”. The sequence of button clicks ⟨“/”, “0”, “=”⟩ changes the text field’s value to “Cannot divide by zero”. Clicking on the “log” button immediately after this sequence causes the program to throw an uncaught `NumberFormatException`. The event handler for “log” has tried to take the logarithm of a `String` object (“Cannot divide by zero”).

Faults like this, which arise only when event handlers interact, can elude test suites that satisfy 100% statement (or, equivalently, 100% block or branch) coverage. Consider two test suites for TerpCalc, one that includes the input sequence ⟨”/”, “0”, “=”, “log”⟩ and another that replaces this sequence with the pair of sequences ⟨”/”, “0”, “=”⟩ and ⟨”log”⟩. The two test suites cover the same set of statements, blocks, and branches, but only the first detects the fault in the event handler for “log”.

Some coverage metrics would in fact flag the two test suites above as distinct, and the adequacy criteria they induce would tend to catch event-interaction faults like the one above. Two such families of criteria are based on data-flow through programs and state models of programs. These criteria are more general than the event-aware criteria introduced later in this section; they apply to any sequential program, not just to event-driven programs. Applying data-flow and state-model adequacy criteria, however, presents some well-known obstacles.

Data-flow coverage is concerned with the definitions (*defs*) and uses of variables in programs. Frankl and Weyuker (Frankl and Weyuker, 1988) have proposed several varieties of data-flow adequacy criteria, including all-defs, all-computation-uses, all-predicate-uses, and all-def-use-paths. Because the number of defs and uses in a moderate-sized program is enormous, and available tool support is limited (Harrold and Rothermel, 1997), generating test cases that satisfy data-flow adequacy criteria is often not practical.

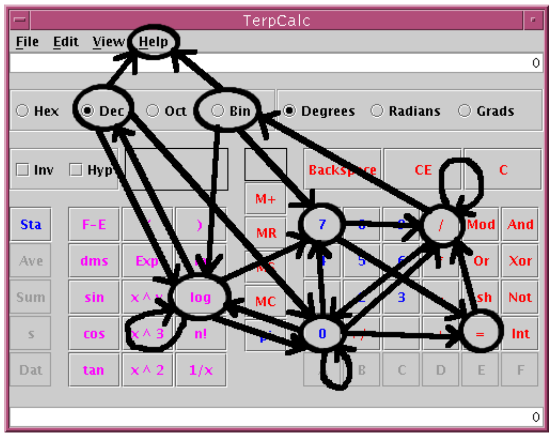

Figure 2: Part of TerpCalc's EFG.

An alternative family of adequacy criteria is based on a state model of the program under test. Tracking the states and state transitions covered during testing, these criteria include all-transitions, all-transition-pairs, and all-round-trip-paths (Briand et al., 2004). In an event-driven program, each event handler can send the program to a new state, and at each state there are typically many event handlers to choose from. Creating such a large state model and generating enough test cases to cover it can also be impractical.

Since event-ignorant adequacy criteria, like those based on statement coverage, may miss event-interaction faults, and since criteria more sensitive to event interactions, like those based on data-flow coverage, may be too expensive, Memon et al (Memon et al., 2001) have proposed new adequacy criteria for one class of event-driven programs. This class is GUI-based programs—programs with which the user interacts through a GUI. The new adequacy criteria follow the structure of event handlers and their potential interactions. A directed graph called an *event flow graph (EFG)* encapsulates this structure.

Figure 2 shows part of the EFG for TerpCalc. In the EFG, nodes represent events, and an edge from node e_1 to e_2 signifies that execution of event e_2 can immediately follow e_1. An *n-way event interaction (EI)* is represented by a length-$(n-1)$ (i.e. n-node) path through the EFG. The *n-way-EI coverage* metric measures the proportion of all feasible n-way EIs that a test suite exercises.

3 Empirical Study

We hypothesize that, for event-driven programs, n-way-EI adequacy criteria for small n tend to expose faults more efficiently than other criteria do. In this study, we compare 1-way-EI and 2-way-EI criteria to statement criteria. We ask the following research questions:

- **Q1.** How many faults do statement, 1-way-EI, and 2-way-EI adequacy criteria tend to expose, relative to each other, at various levels of coverage (e.g. 50%)?
- **Q2.** How many test cases do statement, 1-way-EI, and 2-way-EI adequacy criteria tend to require, relative to each other, at various levels of coverage?

3.1 Procedure

The subject program, TerpSpreadSheet (Figure 3, belongs to the TerpOffice 3.0 suite [2], an open-source office suite developed in Java by University of Maryland undergraduates. TerpSpreadSheet, a spreadsheet program similar in intent to Microsoft Excel, comprises about 15,000 lines of source code. Along with the program, we obtained a set of 5106 test cases (the *test pool*); a statement coverage report for each test case (reported by a coverage instrumenter called Instr[3]); a set of 234 alternate versions of TerpSpreadSheet, each seeded with one fault (the *fault-seeded versions*); and a listing of which faults each test case detects (the *fault matrix*). The test pool covers, with little redundancy, all 2-way EIs in the program. Each test case in the pool exercises a length-2 sequence of events. Each seeded fault is syntactically small, usually touching just one line of source code, and mimics actual faults reported and fixed by the program's developers.

We defined fifteen adequacy criteria by pairing each of the three coverage metrics (statement, 1-way-EI, and 2-way-EI) with each of five coverage levels (50%, 70%, 90%, 95%, and 100%). For each criterion, we constructed a number of test suites by selecting test cases from the pool, following a procedure similar to the one used by Graves et al (Graves et al., 2001). Starting with an empty test suite, we built the test suite one test case at a time by randomly selecting, first, a program unit (e.g. statement) that had not yet been covered and, second, a test case that covered the program unit, until the test suite satisfied the adequacy criterion. (To avoid dealing with unreachable program statements, we calculated statement coverage percentages with respect to the union of all statements covered by the test pool. Thus, our results may over-estimate statement-coverage percentages.) For each 1-way-EI and 2-way-EI criterion, we constructed 1000 test suites, and for each statement criterion, we constructed 150 test suites. Since we already knew which statements and events each test case covers and which faults each test case detects, we could calculate the coverage percentage and the faults detected by each test suite without actually running it.

[2]`http://www.cs.umd.edu/users/atif/newsite/terpoffice.htm`

[3]`http://www.glenmccl.com/instr/instr.htm`

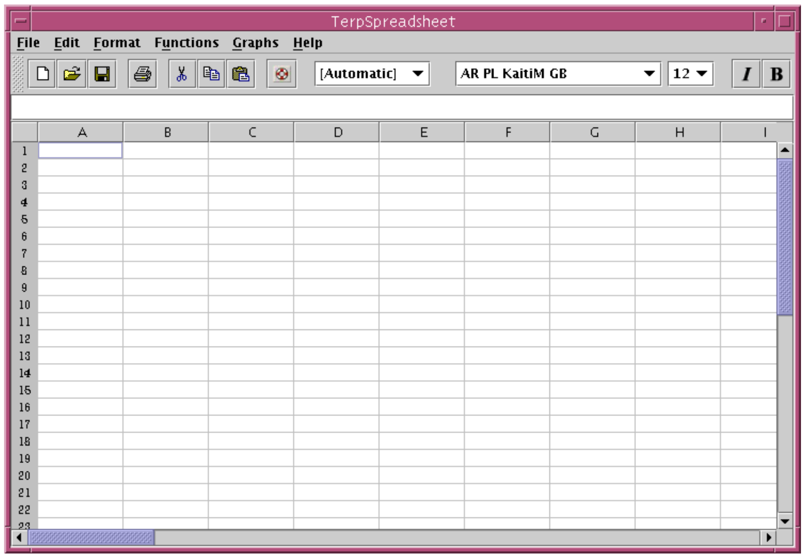

Figure 3: Screenshot of TerpSpreadSheet.

3.2 Results

Table 1 shows the average size of test suites for each criterion. The size increases by several orders of magnitude between the statement criteria and the 2-way-EI criteria. The 1-way-EI criteria fall in between, though much nearer the sizes of the statement criteria.

Table 1: Average test suite size (number of test cases)

	50%	70%	90%	95%	100%
Statement	3.6	9.5	22.6	30.0	55.0
1-way EI	43.6	68.1	96.3	104.4	112.4
2-way EI	1687.6	2417.4	3148.4	3331.3	3514.3

The box plots in Figures 4 and 5 depict, respectively, the number of faults detected and the fault-detection rate with each criterion; each data point represents one test suite. The 2-way-EI criteria tend to detect more faults, but at a much higher cost (number of test cases), than the other criteria.

The differences in fault detection led us to wonder how much the faults detected with different criteria overlap. The Venn diagram in Figure 6 addresses this question. In this diagram, a fault contributes to the count in a coverage metric's circle if a test suite satisfying 100% coverage for that metric has an empirical probability no less than 0.5 of detecting the fault. The statement and 2-way-EI criteria each detected

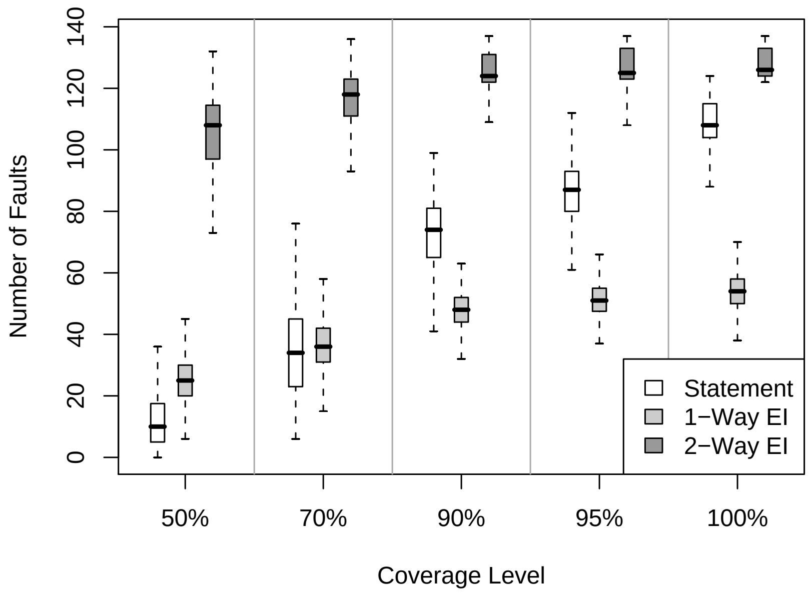

Figure 4: Number of faults detected by each test suite

several faults that no other criterion detected at least half of the time, while the 1-way-EI criterion detected no such unique faults.

4 Conclusions and Future Work

This study found that, at any given level of coverage, 2-way-EI adequacy criteria exposed more faults than statement and 1-way-EI criteria but required much larger test suites. This result is consistent with the result of Memon et al (Memon et al., 2001), which showed that statement coverage often lumps distinct event interactions into the same input class. It is not clear, however, how much the sheer size of 2-way-EI test suites contributed to fault detection; in a future study, we will control for test suite size by matching each 2-way-EI suite with a same-sized test suite randomly selected from the test pool.

Statement adequacy criteria fared surprisingly well in this study, mainly because so few test cases from the pool are needed to satisfy the criteria. Part of the cause may be the way we computed statement coverage 3.1, which can over-estimate the percentage of statements covered. The decision to use a test pool made up of mostly-orthogonal,

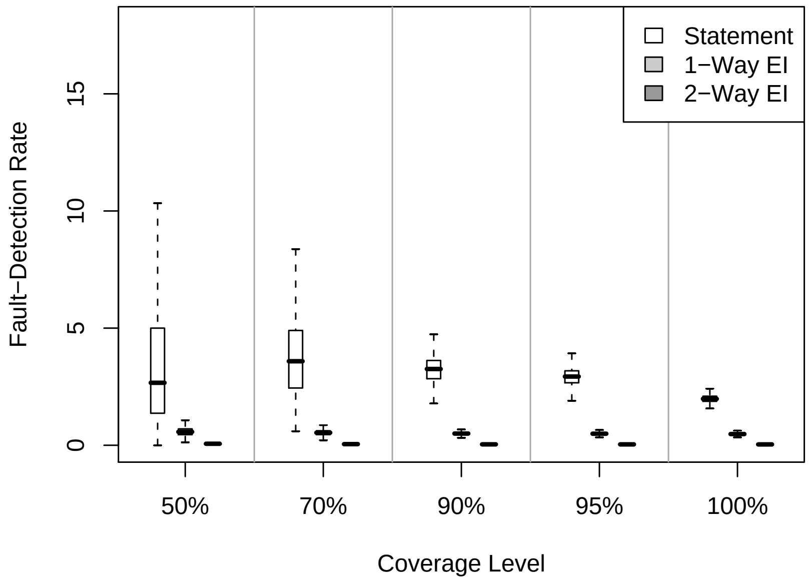

Figure 5: Fault-detection rate (number of faults detected / test suite size) for each test suite

length-2 event sequences may also have contributed, making it easier to choose small statement-adequate test suites by chance. But the strongest uncontrolled influence on the results, we suspect, is the program under test. With TerpSpreadSheet, just three or four length-2 test cases usually suffice to cover 50% of program statements. This implies that most of TerpSpreadSheet's event handlers own only a few statements unique to them; most of the event-handling code is shared across multiple event handlers.

The structure of the program under test may also have contributed to the wide gap in fault-detection ability between 1-way-EI and 2-way-EI adequacy criteria. The more tightly coupled the program's event handlers, the more likely seeded faults are to fall in code that affects event interactions. For a program whose event handlers are more loosely coupled than TerpSpreadSheet's, 1-way-EI criteria may approach the fault-detection ability of 2-way-EI criteria.

The results of this study bring us to an inconclusive conclusion: which adequacy criterion best suits a testing situation depends on what program we are testing, how reliable the program must be, and how many test cases we can afford to generate and run. Under tight time constraints, for programs like TerpSpreadSheet, statement

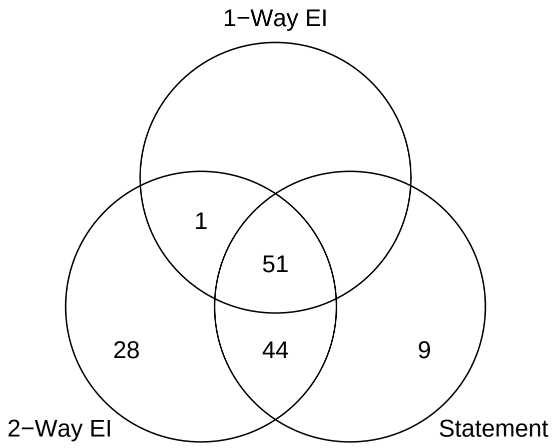

Figure 6: Number of faults detected by at least 50% of test suites for at least one coverage metric at the 100% level

adequacy criteria tend to expose some faults at little cost. When the goal is to find as many faults as possible—again, for programs like TerpSpreadSheet—testers should opt for 2-way-EI criteria or some combination of 2-way-EI and statement criteria.

Future work will examine characteristics of faults which either statement or 2-way-EI criteria, but not both, tend to expose. A study of such faults leads naturally to an examination of event handlers, both faulty and correct, and the adequacy criteria that best suit them. Future studies will investigate additional adequacy criteria, such as data-flow criteria and n-way-EI criteria for $n > 2$.

References

Briand, L. C., Penta, M. D., and Labiche, Y. (2004). Assessing and improving state-based class testing: A series of experiments. *IEEE Trans. Softw. Eng.*, 30(11):770–793.

Frankl, P. G. and Weyuker, E. J. (1988). An applicable family of data flow testing criteria. *IEEE Trans. Softw. Eng.*, 14(10):1483–1498.

Graves, T. L., Harrold, M. J., Kim, J.-M., Porter, A., and Rothermel, G. (2001). An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):184–208.

Harrold, M. J. and Rothermel, G. (1997). Aristotle: A system for research on and development of program-analysis-based tools. Technical Report OSU-CISRC-3/97-TR17, The Ohio State University.

Memon, A. M., Soffa, M. L., and Pollack, M. E. (2001). Coverage criteria for GUI testing. In *Proceedings of the 8th European Software Engineering Conference*, pages 256–267. ACM Press.