Nancy Eickelmann ■ Motorola ■ nancy.eickelmann@motorola.com

Putting Your Best Tests Forward

Gregg Rothermel and Sebastian Elbaum

Successful software evolves. Evolution helps software accommodate new technologies and user needs but can also affect its quality. So, when software engineers modify software, they *regression test* it, rerunning existing tests to verify that existing functionality hasn't been harmed and creating new tests to validate new functionality.

Regression testing is one of the most widely used testing techniques[1] but can be expensive. For example, one company we work with has a regression test suite for a system of only 20,000 lines of code that takes seven weeks and costs several hundred thousand dollars to execute. A second company runs their regression test suite continuously, cycling through the tests over a four-week period as engineers apply changes and then begin the cycle again.

Test case prioritization helps with this because it orders tests so that they help you meet your testing goals earlier during regression testing. Prioritization techniques can, for example, order tests to achieve coverage at the fastest rate possible, exercise features in order of expected frequency of use, or reveal faults as early as possible.

We focus on the last goal, which we describe as "increasing a test suite's rate of fault detection" or the speed with which the test suite reveals faults. A faster fault detection rate during regression testing provides earlier feedback on a system under test, supporting earlier strategic decisions about release schedules and letting engineers begin debugging sooner. Also, if testing time is limited or unexpectedly reduced, prioritization increases the chance that testing resources will have been spent as cost effectively as possible in the available time.

Evidence exists that you can perform prioritization efficiently—Microsoft has applied it to multimillion-line programs.[2] Strong evidence also exists that prioritization can improve test suites' fault-detection rate.[3,4] The factors affecting prioritization's success and governing choices of techniques, however, are complex. In this article, we discuss several practical prioritization techniques, describe factors affecting prioritization, and suggest how to prioritize cost effectively.

Test case prioritization techniques

Researchers have developed many prioritization techniques.[2–5] Most of the techniques rely on test coverage data from instrumenting the program under test. For example, *total statement coverage* prioritization orders tests in terms of the number of statements they execute. The intuition here is that the more code a test

 0740-7459/03/$17.00 © 2003 IEEE

executes, the more likely it will reveal faults.

Additional statement coverage prioritization iteratively seeks the test that covers the most statements not yet covered until it has covered all the statements at least once, then repeats this process until it has ordered all tests. The intuition is that it's better to focus on statements not yet exercised than to simply cover large numbers of statements. Essentially, this technique incorporates feedback into the first technique, using information about tests prioritized so far to prioritize subsequent tests.

These two simple techniques can be adapted in many ways. Some of the more important adaptations involve

- *Different types of coverage.* Prioritization can focus on branches (decision outcomes), basic blocks (single-entry, single-exit sequences of statements), functions, methods, or—at the noncode level—requirements exercised.
- *Change cognizance.* Prioritization can favor modified functions, methods, or covered items over unmodified ones (for example, using modification information provided by configuration management systems or differencing tools).
- *Cost cognizance.* Prioritization can incorporate test cost estimates (for example, execution time) or test importance (for example, how important tested operations are to the user).
- *History cognizance.* Prioritization can incorporate information on test history, such as the length of time since a test was last run or the test's earlier ability to detect faults.

Those who design prioritization techniques can incorporate these and other adaptations singly or in various combinations and with or without using feedback during prioritization.

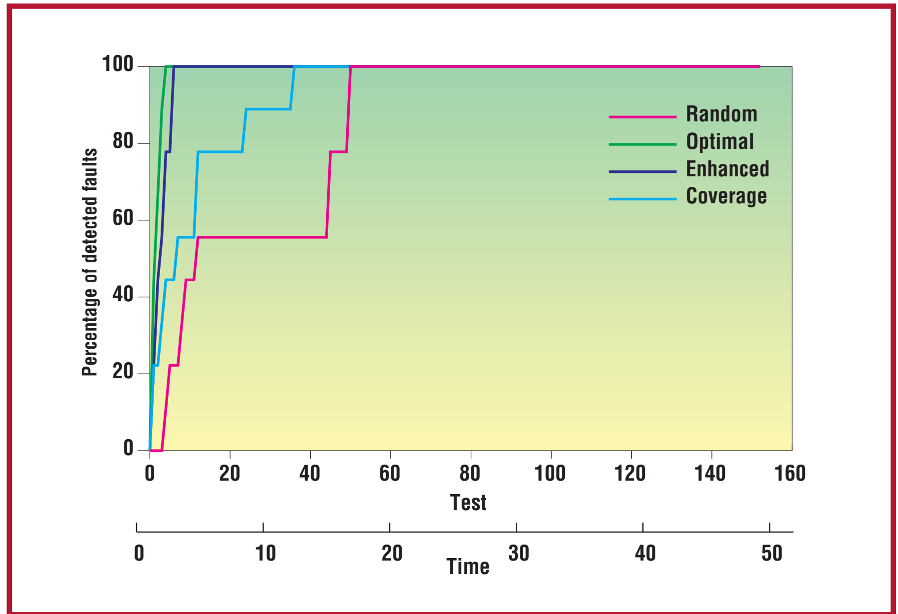


Figure 1. Fault detection rates for four prioritized test suites.

How much can prioritization help?

We have conducted dozens of controlled experiments and case studies of prioritization techniques.[3–5] The most important single result of these studies is this: every prioritization technique we've studied has consistently outperformed unprioritized test suites in terms of increasing their fault detection rates. (See the related sidebar on how these rates are measured.)

Consider a typical example. Figure 1 shows the results of applying four prioritization techniques to a test suite for an antenna-array program from the European Space Agency. The program's developers found several faults in it. The prioritization techniques we used were called *random* (unprioritized), *coverage* (measured in terms of total number of functions), *enhanced* (measured in terms of additional modified functions covered), and *optimal* (not a practical technique, but calculated after the fact to give an upper bound on prioritization effectiveness). For each of the four prioritized suites, the graph plots the percentage of detected faults against the number of executed tests.

After we had run only four tests (3 percent of the test suite), the optimal ordering had revealed all faults that the test suite could reveal, whereas the random ordering had revealed only 11 percent of those faults. The coverage ordering had, in this time, revealed 44 percent of the faults, and the enhanced ordering had revealed 78 percent. After we had run six of the tests (4 percent of the test suite), both the optimal and enhanced orderings had revealed all faults, the coverage ordering had revealed 44 percent, and the random ordering had revealed 22 percent. The coverage and random orderings didn't reveal the last faults until we had executed 24 percent and 33 percent of the tests, respectively.

Of course, such differences in fault detection rates aren't necessarily of practical significance, which depends on the costs associated with your testing activities. For example, if the time scale in Figure 1 denotes minutes, these differences in detection might not matter to you. If it denotes days, they might. You'll have to assess prioritization's cost benefits relative to your own testing processes and cost factors.

Cost-effectiveness factors

Although prioritized test suites regularly outperform unprioritized suites, some prioritization techniques work better than others. In the example just presented, the enhanced technique outperformed the simple coverage technique. In our wider studies, however, prioritization techniques' relative per-

formance varied with characteristics of test suites, programs, and faults. For example, in several cases (and contrary to our intuitions), techniques that incorporated feedback performed worse than those that didn't. Also, in many cases, simple coverage techniques outperformed techniques utilizing modification information.

In studying the factors that affect prioritization cost-effectiveness,[6,7] the most prominent factors we've identified so far are

- *Test suite granularity.* Test suites organized into many smaller tests can be more effectively prioritized than test suites organized into fewer larger tests, but you must balance this against the costs of executing and managing test suites.
- *Modification location.* When modifications and regression faults are confined largely to *core code* (code executed by most tests), techniques using no feedback tend to outperform tests using feedback. Conversely, when modifications and regression faults occur largely in non-core code or are highly distributed across the code, feedback is useful.
- *Level of analysis.* Fine-grained (for example, statement level) analyses provide more effective prioritization than coarse-grained (function level, for example) analyses, but function-level analyses are cheaper.

You can learn several lessons from this. For one thing, putting some thought into your test design can be important for all future regression test runs. Also, tracking where your software has been modified can provide useful data for choosing a prioritization technique. Finally, it's important to consider cost-benefit trade-offs before adopting an approach.

Choosing the best technique for you

Differences in technique performance across different workloads naturally lead to the question, "What technique should I choose?" One dri-

Measuring Fault Detection Rates

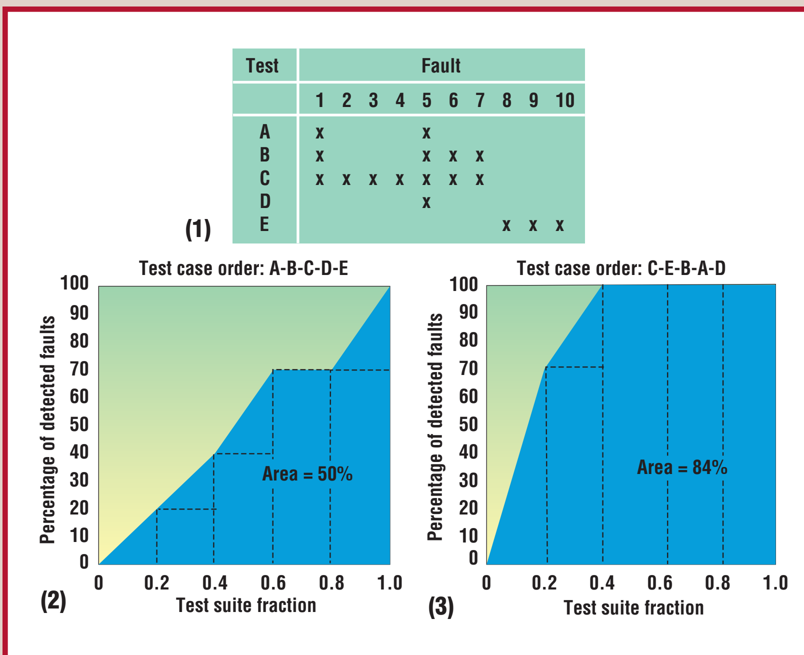
Assessing prioritization techniques' effectiveness is important. To support assessments, we created a metric, *APFD* (average percentage of faults detected), which tracks how rapidly a prioritized test suite detects faults.[1] APFD values range from 0 to 100. Higher APFD numbers mean faster and better fault detection rates.

Suppose you have a program containing 10 faults and a test suite of five tests, A through E, with the fault-detecting abilities shown in Figure A1. Suppose you place the tests in the order A-B-C-D-E. Figure A2 plots the percentage of detected faults versus the fraction of the test suite used under this test order. The area under the curve represents the weighted average of the percentage of faults detected over the test suite's life and is the prioritized test suite's APFD measure—50 percent in this example. Figure A3 reflects what happens when you change the test order to C-E-B-A-D, resulting in the earliest detection of the most faults (84 percent APFD).

In this example, we treat tests and faults as having equal costs and severities, but elsewhere we show how to extend the metric to account for variance in costs and severities by letting the axes denote total test cost and total fault severity and letting tests and faults occupy percentages of these scales.[2]

References

1. G. Rothermel et al., "Test Case Prioritization," *IEEE Trans. Software Eng.*, vol. 27, no. 10, Oct. 2001, pp. 929–948.
2. S. Elbaum, A.G. Malishevsky, and G. Rothermel, "Incorporating Varying Test Costs and Fault Severities into Test Case Prioritization," *Proc. 23rd Int'l Conf. Software Eng.* (ICSE 2001), IEEE CS Press, 2001, pp. 329–338.


(1)

Test	Fault 1	2	3	4	5	6	7	8	9	10
A	x				x					
B	x				x	x	x			
C	x	x	x	x	x	x	x			
D					x					
E								x	x	x

Figure A. Average percentage of faults detected measurements for a test suite with two different prioritizations. A test suite with (1) demonstrated fault-detecting abilities has (2) 50 percent APFD in one order and (3) 84 percent APFD in another.

ver in this decision is simply the data types you can cost-effectively collect for your system, such as coverage, change, and test cost information.

Having determined which approaches you could use, you could try these on historical data for your system and measure their relative effectiveness in the past. Often, relative technique effectiveness remains stable across future releases of given systems, so this can give you a first cut.

Elsewhere we present an approach for further refining technique selection.[8] First, we gather several specific metrics about programs and test coverage over past releases and data on past fault detection results. Using these metrics and an analysis based on classification trees, we then construct decision rules that improve the chances of predicting the best technique correctly. Using this approach, we've improved our chances of selecting the most appropriate technique for a given program and test coverage pattern by as much as 45 percent.

Test case prioritization is not a panacea. If your test runs are fully automated and relatively short-lived, prioritization might make no difference in your processes at all. If your tests have dependencies on one another (that is, your later tests depend on setup that earlier tests perform), your ability to reorder tests might be limited. When regression testing, there is also no excuse for failing to create new tests where they're needed; prioritization targets only reuse of existing tests.

With that said, we believe that prioritization can be cost-effective for many practitioners. If your regression testing takes significant time or resources to complete, requires expensive human intervention (such as in checking correctness), or is performed under an uncertain time allocation, prioritization will likely help. In fact, the worst thing you can do in such cases—if you care about your test suite's fault detection rate—is to not prioritize at all.

One particularly interesting problem prioritization research is just starting to address involves differences in testing processes and how they relate to using prioritization. For example, in *batch* regression testing, a long period of modifications is followed by a usually inadequate period of regression testing and then a system release. In *continuous* regression testing, changes are made to the code base each day and you run as many regression tests as possible each night. These two approaches differ substantially. We are certain that process differences such as this hold important implications for prioritization.

Through further research on this and other prioritization problems, we hope to be able to help practitioners truly put their best tests forward. For further information on test case prioritization and access to interactive tools implementing and assessing several prioritization techniques, see http://mapstext.unl.edu/public/prioritization.

References

1. K. Onoma et al., "Regression Testing in an Industrial Environment," *Comm. ACM*, vol. 41, no. 5, May 1998, pp. 81–86.
2. A. Srivastava and J. Thiagarajan, "Effectively Prioritizing Tests in Development Environment," *Proc. Int'l Symp. Software Testing and Analysis* (ISSTA 2002), ACM Press, pp. 97–106.
3. S. Elbaum, A.G. Malishevsky, and G. Rothermel, "Test Case Prioritization: A Family of Empirical Studies," *IEEE Trans. Software Eng.*, vol. 28, no. 2, Feb. 2002, pp. 159–182.
4. G. Rothermel et al., "Test Case Prioritization," *IEEE Trans. Software Eng.*, vol. 27, no. 10, Oct. 2001, pp. 929–948.
5. S. Elbaum, A.G. Malishevsky, and G. Rothermel, "Incorporating Varying Test Costs and Fault Severities into Test Case Prioritization," *Proc. 23rd Int'l Conf. Software Eng.* (ICSE 2001), IEEE CS Press, 2001, pp. 329–338.
6. S. Elbaum et al., "Understanding the Effects of the Changes on the Cost-Effectiveness of Regression Testing Techniques," *J. Software Testing, Verification, and Reliability*, vol. 13, no. 2, June 2003, pp. 65–83.
7. G. Rothermel et al., "The Impact of Test Suite Granularity on the Cost-Effectiveness of Regression Testing," *Proc. 24th Int'l Conf. Software Eng.* (ICSE 2002), IEEE CS Press, pp. 230–240.
8. S. Elbaum et al., *Selecting a Cost-Effective Test Case Prioritization Technique*, tech. report 03-01-01, Univ. Nebraska-Lincoln, 2003.

Gregg Rothermel is an associate professor of computer science at Oregon State University. Contact him at grother@cs.orst.edu.

Sebastian Elbaum is an assistant professor of computer science at the University of Nebraska-Lincoln. Contact him at elbaum@cse.unl.edu.

Fill here?