

[data management](#)
Sponsored by
[Sleepycat Software](#)

[debugging/qa](#)
Sponsored by
[Security Innovation](#)

[developer tools](#)
Sponsored by
[IBM](#)

[open source](#)

[security](#)

[virtual machines](#)

[Development Tools Directory](#)

Not Found

The requested URL /adi/N339.queue.com/B1532539.3;sz=728x90;ord=72101713 was not found on this server.

Too Darned Big to Test

[Comprehensive application testing and debugging with fault-simulation.](#)
Sponsored by [Security Innovation](#)

From [Quality Assurance](#)

Vol. 3, No. 1 - February 2005

by **KEITH STOBIE, MICROSOFT**

The increasing size and complexity of software, coupled with concurrency and distributed systems, has made apparent the ineffectiveness of using only handcrafted tests. The misuse of code coverage and avoidance of random testing has exacerbated the problem. We must start again, beginning with good design (including dependency analysis), good static checking (including model property checking), and good unit testing (including good input selection). Code coverage can help select and prioritize tests to make you more efficient, as can the all-pairs technique for controlling the number of configurations. Finally, testers can use models to generate test coverage and good stochastic tests, and to act as test oracles.

HANDCRAFTED TESTS OUTPACED BY HARDWARE AND SOFTWARE

Hardware advances have followed Moore's law for years, giving the capability for running ever-more complex software on the same cost platform. Developers have taken advantage of this by raising their level of abstraction from assembly to first-generation compiled languages to managed code such as C# and Java, and rapid application development environments such as Visual Basic. Although the number of lines of code per day per programmer appears to remain relatively fixed, the power of each line of code has increased, allowing complex systems to be built. Moore's law provides double the computing power every 18 months and software code size tends to double every seven years, but software testing does not appear to be keeping pace.

Unfortunately, the increased power has not, in general, made testing easier. While in a few cases the more powerful hardware means we can do more complete testing, in general the testing problem is getting worse. You can test square root for all the 32-bit values in a reasonable time,[1,2] but we are now moving to 64-bit machines with 64-bit values (and even longer for floating point). Assuming a nanosecond per test case, it would take 584 years to test all the values. Sure, you could scale to, say, 1,000 processors, but you would still need six months for just this one test case.

Two other issues of complexity to consider are: the number of different execution states software can go through, and concurrency. User interfaces were originally very rigid, letting users progress through a single fixed set of operations—for example, a single set of hierarchical menus and prompts. Now, good user design is event driven with the user in control of the sequence of actions. Further, many actions can be accomplished in multiple manners (e.g., closing a window via the menu [File/Close], a shortcut key [ALT-F4], or the mouse [click on the Close icon]). Multiple representations of items (e.g., file names) also generally mean multiple execution paths. A manifestation of this is the security issue that occurs when an application makes wrong decisions based on a noncanonical

Subscribe to *Queue*
in print. It's FREE!



First Name

Last Name

Company

Street

City

State/Province

Zip/Postal Code

Email

Continue

representation of a name.[3] These explosive combinations make it virtually impossible for a tester to both conceive of and then test a reasonable sampling of all the combinations.



[One giant leap for developers](#)
[Register now for the Oracle Space Sweepstakes for a chance to win a free ride on a Space Adventures suborbital space flight.](#)

Concurrency is becoming more prevalent at many levels. Beyond multiple processes, we see increasing lower-level concurrency, especially as a result of multicore chipsets that encourage parallelism via threading and hyperthreading; soon, everyone will have a dual-processor machine. Machine clustering and Web services are increasing the concurrency across applications.

All of this creates greater possibilities for more insidious bugs to be introduced, while making them harder to detect.

MISUNDERSTANDING CODE COVERAGE AND STOCHASTIC TESTING

Before discussing possible approaches to these problems, let's clear up some pervasive misconceptions.

Myth: Code coverage means quality. Tools for measuring structural code coverage have become increasingly easy to use and deploy and have thus deservedly gained greater acceptance in the testing arsenal. A fundamental flaw made by many organizations (especially by management, which measures by numbers) is to presume that because low code-coverage measures indicate poor testing, or that because good sets of tests have high coverage, high coverage therefore implies good testing (see Logical Fallacies sidebar). Code coverage merely measures that a statement, block, or branch has been exercised. It gives no measure of whether the exercised code behaved correctly. As code gets too big to easily roll up qualitative assessments, people start summarizing code coverage instead of test information. Thus, we know low code coverage is bad, because if we have never exercised the code, it appears impossible to say we have "tested" it. However, while having something that just executes the code may consequently reveal a few possible flaws (e.g., a complete system failure), it doesn't really indicate if the code has been what most people consider tested.

Tested indicates you have verified the results of the code execution in useful ways. The fallacy of high code coverage will become more evident with the next generation of automatic unit testing tools (discussed later) that can generate high code density coverage with no results checking. Further, results of real software comparing code coverage and defect density (defects per kilo-lines of code) show that using coverage measures alone as predictors of defect density (software quality/reliability) is not accurate.

Logical Fallacies

Denying the Antecedent:

If A, then B; not A; therefore, not B

If low coverage, then poor tests; not low coverage; therefore, not poor tests

Asserting the Consequent:

If p, then q; q; therefore, p

If good tests, then high coverage; high coverage; therefore, good tests

Myth: Random testing is bad. One of the big debates in testing is partitioned (typically handcrafted) test design versus operational, profile-based stochastic testing (a method of random testing). Whereas many organizations today still perform testing (such as choosing the partitions) as a craft (either manually or semi-automatically), it is unclear that this is the most effective method, especially with the increasing complexity of software. Current evidence indicates that unless you have reliable knowledge about areas of increased fault likelihood, then random testing can do as well as handcrafted tests.[4,5]

For example, a recent academic study with fault seeding showed that under some circumstance the all-pairs testing technique (see "Choose configuration interactions with all-pairs" later in this article) applied to function

parameters was no better than random testing at detecting faults.[6]

The real difficulty in doing random testing (like the problem with coverage) is verifying the result. How, for random inputs, can we judge the output? It is easier to have static tests with precomputed outputs. In testing, being able to verify the output is called the test oracle problem.[7,8] There are numerous current areas of work on this problem.

Another problem is that very low probability sub-domains are likely to be disregarded by random testing, but if the cost of failures in those subdomains is very high, they could have a large impact. This is a major flaw of random testing and a good reason for using it as a complementary rather than the sole testing strategy.

Part of the testing discipline is to understand where we have reliable knowledge of increased fault likelihood. The classic boundary value analysis technique succeeds because off-by-one errors increase the fault likelihood at boundaries. Many other analysis techniques have a less tried-and-true underlying fault model to reliably indicate increased fault likelihood. For example, partitioning based on code coverage doesn't guarantee finding the fault, as the right input must be used to stimulate the fault and sufficient verification must be done to realize the fault has produced a failure.

TEST ISSUES: DO OVER

It's clear that increasing structural code coverage doesn't solve the testing problem and that stochastic testing is frequently as good as or better than handcrafted test cases. Given this state of affairs, how do we address the issue of 'it's too darned big to test'? There are several approaches to consider: unit testing, design, static checking, and concurrency testing.

Good unit testing (including good input selection). Getting high-quality units before integration is the first key to making big system testing tractable. The simple way to do that is to follow the decades-old recommendation of starting with good unit testing. The IEEE Standard for Software Unit Testing has been around for years [Std. 1008-1987] and requires 100 percent statement coverage. This has been nicely reincarnated via the test-driven development movement, which also espouses the previously seldom-adhered-to idea of writing the tests before the code. Testing a big system is especially difficult when the underlying components and units are of low quality.

Why isn't good unit testing prevalent? In many shops the problem is partly cultural. Software developers presumed someone else was responsible for testing besides them. Some development groups have spent more effort building out stubbed components and test harnesses than on building the actual components. Standard unit testing harnesses now exist for many environments, and the creation of mock objects is becoming semi-automated. Another reason for poor unit testing is that software was once simpler and reliability expectations lower. In today's world insecure systems arising from simple problems (for example, buffer overruns) that good unit testing could have caught have really motivated management to drive for better unit testing.

A variety of tools exist in this arena. Unit testing frameworks include the XUnit family of tools, Framework for Integrated Test (Ward Cunningham's Fit), Team Share, etc. Further, IDEs (integrated development environments) such as Eclipse and Rational can generate unit test outlines. Any development shop should be able to find and use applicable ones.

In general, today's unit testing requires developers to handcraft each test case by defining the input and the expected output. In the future, more intelligent tools will be able to create good input coverage test cases and some expected output. We may see 100 percent code coverage, with no results checking. We will need to become more aware of the thoroughness of results checking instead of just code coverage. How to measure results-checking thoroughness hasn't really been addressed.

Current automatic unit-generation test techniques work from the simple known failure modes (such as null parameters as inputs) to analysis of the data structures used. Data-structure generation can be based on constraint solvers, models, and symbolic execution.

Other tools can enhance an existing set of tests. For example, Eclat creates an approximate test oracle from a test suite, then uses that oracle to aid in generating test inputs that are likely to reveal bugs or expose new behavior while ignoring those that are invalid inputs or that exercise already-tested functionality.

Even without tools, just training people in simple functional test techniques can help them choose better inputs for their unit testing. You can also start with generic checklists of things to test and then make them your own by recording your organization's history of what makes interesting input. You can find industry checklists in several

books under different names such as the following: “Common Software Errors” (Kaner, C., Falk, J. and Nguyen, H.Q. 1993. Testing Computer Software. Van Nostrand Reinhold); “Test Catalog” (Marick, B. 1994. Craft of Software Testing. Prentice Hall PTR. www.testing.com/writings/short-catalog.pdf); or “Attacks” (Whittaker, J.A. 2002. How to Break Software: A Practical Guide to Testing. Addison Wesley); as well as on the Web (for example, <http://blogs.msdn.com/jledgard/archive/2003/11/03/53722.aspx>).

Good design (including dependency analysis). Many books and papers explain how to do good up-front design (the best kind), but what if you already have the code?

Dependency analysis can be used to refactor code (rewriting to improve code readability or structure; see <http://www.refactoring.com/>). Many IDEs are incorporating features to make code refactoring easier. Good unit tests, as agile methods advocate, provide more confidence that refactoring hasn’t broken or regressed the system.

Dependency analysis allows testers to choose only those test cases that target a change. Simplistic analysis can be done by looking at structural code dependencies, and it can be very effective. More sophisticated data-flow analysis allows more accurate dependency determination, but at a greater cost. Either technique can be a big improvement over just running everything all the time. A test design implication of this is to create relatively small test cases to reduce extraneous testing or factor big tests into little ones.[9]

Good static checking (including model property checking). The convergence of static analysis tools with formal methods is now providing powerful tools for ensuring high-quality units and to some extent their integration. For example, PREFIX is a simulation tool for C/C++ that simulates the program execution state along a selected set of program paths and queries the execution state to identify programming errors.[10] The issue creating the greatest resistance to adoption for most static checking remains the false positive rate. Some tools can only make reliable guesses and sometimes guess wrong. Developers get frustrated if they can’t turn off the tool for the places they know it is wrong. Sometimes developers get carried away and turn off too much, then miss problems the tool could have caught. Some developers don’t believe the issues found are really bugs even when they are.

Many of the more advanced checks require additional annotations to describe the intent. You can emulate some of the stricter checks in any language just using compiler or language constructs such as `assert()`. It is still a challenge to convince many developers that the extra annotations will pay off in higher-quality, more robust, and more maintainable code. Others, however, who have used it and seen nasty issues that would have taken hours, days, or weeks to debug being found effortlessly up front, would never go back.

All of this data can also help independent testers later because defects found through static analysis are early indicators of pre-release system defect density. Static analysis defect density can be used to discriminate between components of high and low quality (fault-prone and non-fault-prone components).

Concurrency testing. Concurrency issues can be detected both statically (formal method properties such as liveness, livelock/deadlock, etc.) and dynamically (automatic race lockset tracking). Static detection frequently requires extensive annotation of the source code with additional information to allow the necessary reasoning to be done about the code. The advantage of static analysis is its potential to uncover all race conditions.

Dynamic race detectors, such as code coverage tools, require a good set of test cases as input because they look only at what the tests force to occur. They can, however, determine race conditions even though the test itself didn’t force it. As long as the test executes the code of the overlapping locks, it is likely to be detected. Several tools, such as Java PathExplorer (JPAX), have been created that extend the Eraser lockset algorithm.[11] Besides no guarantee of completeness without complete test cases, the other issues still being resolved with dynamic race detectors include their performance and the number of false positives they generate.

More traditional methods of concurrency testing involve controlling the thread scheduler, if possible, to create unusual timings. Fault injection tools, such as Security Innovation’s Holodeck, can be used to create artificial delays and increase the likelihood of race conditions. As a last resort, the old standby of just running lots of activity concurrently under varying conditions has been fruitful in finding defects (although perhaps not as efficiently as many of the newer methods).

PRIORITIZE TESTING

The previous section helps set the stage at a basic level with units of code and analysis, but most test organizations deal with testing the integrated whole. Testers have to be creative in addressing the overwhelming size of today’s software systems. When it’s too darned big to test, you must be selective in what you test, and use more powerful automation such as modeling to help you test. Fault injection tools again come in handy as a way to tweak things

when reaching through the overall big system becomes daunting (which it does quickly!).

Use code coverage to help select and prioritize tests. Prioritizing test cases has become increasingly practical in the past two decades. If you know the coverage of each test case, you can prioritize the tests such that you run tests in the least amount of time to get the highest coverage. The major problem here is getting the coverage data and keeping it up to date. Not all tools let you merge coverage data from different builds, and running coverage all the time can slow down testing. The time to calculate the minimization can also be a deterrent.

For many collections of test cases, running the minimal set of test cases that give the same coverage as all of the test cases typically finds almost all of the bugs that running all of the test cases would find.

Industrial experience at several companies appears to confirm this. For example, for an industrial program of 20,929 blocks, choosing tests only by function call coverage required only 10 percent of the tests while providing 99.2 percent of the same coverage. Reducing by block coverage meant that 34 percent of the tests provided the same block coverage and 99.99 percent of the same branch (segment) coverage. Further, there were no field-reported defects found that would have been caught by the full set of tests but missed by the coverage-reduced set of tests.

Best of all, anyone can easily run the experimental comparison. First run the minimal set of tests providing the same coverage as all of the tests, and then run the remaining tests to see how many additional defects are revealed.

You can also combine this with dependency analysis to first target only the changed code. Depending on the sophistication of your dependency analysis, you may not have to do any further testing.

Use customer usage data. Another great way to target testing is based on actual customer usage. Customers perceive the reliability of a product relative to their usage of it. A feature used by only a few customers on rare occasions will have less impact on customer satisfaction than bread-and-butter features used by virtually all customers all of the time. Although development teams can always make conjectures about product usage, which makes a useful start, actual data improves the targeting.

Ideally, you can automatically record and have actual customer usage sent to you. This approach has several problems, including privacy, performance, and security. Each of these can be dealt with, but it is nontrivial. Alternatively, you may have only a select few customers reporting data, you may do random sampling, or you may fall back on case studies or usability studies. You can also work with your customers for cooperative early testing either through beta programs or even earlier pre-release disclosures.[12]

Customer usage data is especially important in reliability testing where an operational profile improves the meaningfulness of the reliability data. Customer usage data is also critical to prioritizing the configurations that you test. Setting up environments similar to those of your customers helps you find interactions that you might otherwise miss.

Choose configuration interactions with all-pairs. A major problem that makes the testing task too darned big is worrying about all the possible configurations. Configurations cover things such as different hardware, different software (operating system, Web browser, application, etc.) or software versions, configuration values (such as network speed or amount of memory), etc. If you understand which interactions of your configurations cause problems, you should explicitly test for them. But when all the software “should” be independent of the configuration, and experience has shown you it isn’t, then you get worried. Setting up a configuration to test is frequently expensive, and running a full set of tests typically isn’t cheap, either.

If you are doing performance testing, you need the more formal design-of-experiments techniques associated with robust testing and requiring orthogonal arrays. For simple verification, however, another easy, cheap technique used by many test shops is all-pairs. The most common defect doesn’t require any interaction. Just triggering the defect under any configuration will cause a failure about 80 percent of the time. After basic testing eliminates those defects, the next most common defects turn out to require the interaction of just two aspects of the configuration. Which two aspects? That’s the trick! Using all-pairs, you can cheaply verify any pair of interactions in your configuration. Public domain tools exist (<http://tejasconsulting.com/open-testware/feature/allpairs.html>), and it takes less than a half hour to download the tool and create a description of your configuration issues. You’ll then get a very quick answer.

As a simple example of this, I consulted with a test group that had tested 100 different configurations, but were still getting errors from the field. I taught them the technique and tool, and in less than an hour they had a result showing that only 60 configurations were needed to cover all pairs of interactions, and the interaction most recently reported from the field was one of them. That is, if they tested with fewer configurations, they could have found the bug

before shipping.

All-pairs testing may be useful in other domains, but the studies are not yet conclusive.

USE OF MODELS FOR STOCHASTIC TESTS

One way toward stochastic testing is via the use of models. You can begin with the simplest of models, such as “dumb monkey” test tools or “fuzz” testers. You can enhance them toward “smart monkey” tools and formalize them using a variety of modeling languages, such as finite state machines, UML2 (Unified Modeling Language, version 2), and abstract state machines. Testers’ reluctance in the past to embrace formal methods came principally from the state explosion problem that appeared to relegate most formal methods to “toy” problems instead of the big system-level problems that many testers have to deal with.

Recent advances in many fields have helped, but especially recent breakthroughs that make SAT (propositional satisfiability) solvers highly efficient and able to handle more than 1 million variables and operations. Models can be used to generate all relevant variations for limited sizes of data structures.[13,14] You can also use a stochastic model that defines the structure of how the target system is stimulated by its environment.[15] This stochastic testing takes a different approach to sampling than partition testing and simple random testing.

Even nonautomated models can provide useful insights for test design. Simple finite state machine models can show states the designers hadn’t thought about or anticipated. They also help clarify the expected behavior for the tester. You can build your own finite state machines in almost any language. A simple example for C# is goldilocks.[16]

The major issue with model-based testing is changing testers’ mind-sets. Many testers haven’t been trained to think abstractly about what they are testing, and modeling requires the ability to abstract away some detail while concentrating on specific aspects. Modeling can be especially difficult for ad hoc testers not used to approaching a problem systematically. Another reason modeling fails is because people expect it to solve all their problems. Frequently it is better just to add a few manually designed test cases than to extend the model to cover all the details for all cases.

ECONOMY IN TESTING

The size and complexity of today’s software requires that testers be economical in their test methods. Testers need a good understanding of the fault models of their techniques and when and how they apply in order to make them better than stochastic testing. Code coverage should be used to make testing more efficient in selecting and prioritizing tests, but not necessarily in judging the tests. Data on customer usage is also paramount in selecting tests and configurations with constrained resources. Pairwise testing can be used to control the growth of testing different configurations.

Attempting to black-box test integrations of poor-quality components (the traditional “big bang” technique) has always been ineffective, but large systems make it exponentially worse. Test groups must require and product developers must embrace thorough unit testing and preferably tests before code (test-driven development). Dependencies among units must be controlled to make integration quality truly feasible.

REFERENCES

1. Kaner, C. 2000. Architectures of test automation. <http://www.kaner.com/testarch.html>.
2. Hoffman, D. So little time, so many cases. www.cs.bsu.edu/homepages/dmz/cs639/So%20little%20time,%20so%20many%20cases.ppt.
3. Howard, M., and LeBlanc, D. 2002. Writing secure code, 2nd edition. Microsoft Press.
4. Nair, V. N., et al. 1998. A statistical assessment of some software testing strategies and application of experimental design techniques. *Statistica Sinica* 8: 165-184.
5. Ntafos, S. C. 2001. On comparisons of random, partition, and proportional partition testing. *IEEE Transactions on Software Engineering* 27(10).
6. Bach, J., and Schroeder, P. 2004. Pairwise testing: A best practice that isn’t. 22nd Annual Pacific Northwest Software Quality Conference, Portland, OR. <http://www.pnsgc.org/proceedings/pnsgc2004.pdf>.
7. Hoffman, D. 1999. Heuristic test oracles. *Software Testing and Quality Engineering* (March/April). <http://softwarequalitymethods.com/SQM/Papers/HeuristicPaper.pdf>.

8. Hoffman, D. 1998. A taxonomy for test oracles. Quality Week '98.
<http://softwarequalitymethods.com/SQM/Slides/ATaxonomyslide.pdf>.
9. Saff, D., and Ernst, M. 2004. Automatic mock object creation for test factoring. ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'04), Washington, DC (June 7-8): 49-51.
10. Larus, J., et al. 2004. Righting Software. IEEE Software 21(3): 92-100.
11. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. 1997. Eraser: A dynamic data race detector for multithreaded programs. ACM Transactions on Computer Systems 15(4): 391-411.
12. Tate, A. 2003. The best testers are free. Software Testing Analysis and Review (STAR West).
13. Marinov, D., and Khurshid, S. 2001. TestEra: A novel framework for automated testing of Java programs. Proceedings of the 16th IEEE Conference on Automated Software Engineering (November): 22-31.
14. Ball, T., et al. 2000. State generation and automated class testing. Software Testing, Verification and Reliability 10(3): 149-170.
15. Whittaker, J. A. 1997. Stochastic software testing. Annals of Software Engineering 4: 115-131.
http://www.geocities.com/harry_robinson_testing/whittaker97.doc.
16. Robinson, H., and Corning, M. Model-based testing in the key of C#.
http://www.qasig.org/presentations/QASIG_Goldilocks2.pdf.

KEITH STOBIE is a test architect in Microsoft's XML Web Services group (Indigo), where he directs and instructs in QA and test process and strategy. He also plans, designs, and reviews software architecture and tests. With 20 years of distributed systems testing experience, Stobie's interests are in testing methodology, tools technology, and quality process. He is also active in the Web Services Interoperability organization's (WS-I.org) Test Working Group creating test tools for analysis and conformance of WS-I profiles. He has a B.S. in computer science from Cornell University.

[Back to Too Darned Big to Test](#)

