

Automated Testing with an Operational Profile

Robert V. Binder
*m*Verify Corporation

Overview

Although the rationale for testing with an operational profile is compelling, realizing its promise presents significant practical problems. Ideally, we'd like a fresh test suite every time we test. Because critical failures are often a result of high load in combination with unusual input sequences, fresh test suites that can vary load and achieve stress are highly effective. The article presents a case study where these problems were solved and discusses application to testing mobile systems.

A Compelling Approach

Software testing presents two fundamentally hard problems: choosing tests and evaluating test results. Choosing tests is hard because there is an astronomical number of possible test inputs and sequences, yet only a few of those will succeed in revealing a failure. Evaluation requires generating an expected result for each test input and comparing this expected result to the actual output of a test run.

Even with rigorous software engineering, it is a near certainty that undiscovered bugs are present, especially in complex distributed systems. Testing can reveal bugs, but is expensive. We must therefore marshal available testing resources to achieve the greatest possible improvement in reliability of a system under test. But how?

Suppose we have achieved adequate component testing and demonstrated minimal end-to-end stability. Now what? How can we best spend scarce testing resources? The logic of testing with an *operational profile* is compelling. "Testing driven by an operational profile is very efficient because it identifies failures (and hence the faults causing them), on the average, in the order of how often they occur. This approach rapidly reduces failure intensity as test proceeds, and the faults that cause frequent failures are found and removed first. Users will also detect failures on average in order of their frequency, if they have not already been found in test." [1] When we want to maximize the reliability of a fielded system (and thereby maximally enhance the user/customer experience) and have no better information about how to find bugs, any other allocation of testing resources is sub-optimal. Testing in accordance with an operational profile is therefore a key part of my test automation strategy.

This article summarizes how my automated model-based testing approach has evolved over the last ten years and how it continues to evolve to meet the challenges of 21st century information technology.

Automation Challenges

Realizing the promise of testing with an operational profile posed significant practical problems. The test automation systems I built were designed to support rapid incremental development of complex distributed applications. Requirements, interfaces, and components were fluid. We had to be able to revise the profile, regenerate and then run test suites within fixed windows. Simply re-running the same profile-based test suite was unacceptable.

Unless a test configuration is unstable or otherwise dependent on external sources of variation, repeating a test suite will only drive the system under test through the same states and execution sequence. Even if the system under test has been changed and a test suite is an accurate sample from an operational profile, the bug-revealing ability of an unchanged test suite diminishes as it is repeated. Although test suite repetition is useful for regression testing, re-running a test suite on the same code and configuration will not reveal new failures. Worse, this can lead to false confidence.

Fresh Test Suites

Ideally, we'd like a *fresh* test suite every time we're ready to exercise the system under test. A fresh test suite is a unique, but statistically accurate sample from an operational profile. Hand-crafted production of fresh test suites for rapid test cycles (for example, hourly) is impractical for even a system with tens of operations. The complex distributed systems I worked on required tens of thousands of tests cases to achieve a minimal approximation of actual usage and were constantly being debugged, extended, and revised.

Although it isn't hard to write software that automatically generates "random" test inputs, this is not the same as generating a test suite that is an accurate sample from an operational profile. To generate truly fresh test data values, we also found it was necessary to sample from the domains of the system state space and input variables. Established techniques from discrete event simulation and AI-based meta-programming were adapted for this requirement. The end result (of two years of development) was that it took only seconds to set model parameters for a test run, which then drove automatic generation of hundreds of thousands of fresh test cases in hours. Each generated test suite was unique, but still conformed to the statistical distribution specified by our operational profile model.

But then, our fresh test suites posed a new problem. High volume automated testing is not very meaningful if the output of the system under test is not checked. Output checking requires development and evaluation of expected results. In testing jargon, an *oracle* produces the expected result for any test input. A perfect oracle is therefore an error-free realization of the same requirements as the system under test. Developing a perfect oracle is nearly always impractical or infeasible. However, we

extended our models generate some expected results along with inputs, and developed an executable model of the core system logic. So, along with fresh test inputs, we either produced fresh expected results or were able to automatically evaluate test run output. We achieved enough of an automated oracle to use high-volume input generation with confidence.

Including a Load Profile

The traditional approach to “system” and “stress” testing is to first develop test suites to evaluate functionality, then repeat the same test inputs in greater quantity and/or speed to reveal capacity-related problems. However, critical failures are often a result of high load in combination with unusual input sequences. The more-of-the-same approach wastes an opportunity to test under high stress with realistic variation. This is often effective in revealing failures that occur when a system is under heavy utilization. Heavy utilization often occurs when a failure can have the worst possible effect: for example, peak traffic in transaction processing or during a rapid high-stress maneuver that relies on automated motion control.

We designed our test generation technology to dial-in the total quantity of input and input rate as a function of time. Thus, generating 100,000 total test cases to be submitted at the same average rate was no harder than generating 1,000,000 test cases to be submitted at varying rates corresponding to typical peak loading patterns. Both generated fresh and realistic samples drawn from the same operational profile. (Technically, this had the effect of creating a different operational profile for each combination of the load and behavioral models). This improved test effectiveness. We found many bugs with this strategy and eliminated the excess time and cost of separate stress testing.

Results

Our most extensive application of this strategy supported development of system for real time quote and order processing at a leading financial exchange, which was designed for a peak of several million transactions in a six hour trading session. Every daily test run (about 200) used a fresh test suite which covered the entire daily trading cycle.

Daily use of the test process and automated environment revealed some limitations with our approach. We discovered that our AI-based meta programming architecture did not scale well, imposed a high maintenance cost, and was subsequently an obstacle to technology transfer. We relied on off-the-shelf GUI test agents to apply inputs to some system interfaces. These agents were designed to support interactive testing with hard-coded scripting. Although serviceable, they did not support coordination of distributed test suites and lacked features to achieve full control and monitoring of the interface. These problems were not corrected due to time and budget

constraints. We augmented automated test-generation with manually coded scripts for special cases, and did manual testing for some GUI interfaces that were too complex or unstable.

Despite these limitations, our automated operational profile testing approach was highly effective in revealing bugs and in evaluating pre-release system stability. The test process revealed about 1,500 bugs over two years. Project management and developers decided that all but a few of these bugs were worth fixing and verifying the fix. About 5% of the bugs were showstoppers. We used the same test generation approach to verify the bug fixes. Our five person team was very productive, in terms of number of tests run per total effort and reliability improvement per total cost. The last pre-release test run applied 500,000 fresh test cases in two hours, with no failures detected. Post-release, no failures were reported in the first six months of operation.

21st Century Challenges

I believe that information technology will undergo radical change in the next five to ten years. This is an inevitable result of steady improvement in mobile computing platforms, innovation in human-computer interfaces, and increasing wireless network capacity. These advances will have far-reaching social consequences. Soon, most everyday things will have built-in mobile information technology, including clothing, jewelry, vehicles, structures, and open spaces. With ubiquity, high reliability is more important than ever.

Four years ago, I began work to adapt automated profile-based testing to the unique challenges of testing mobile and wireless applications. Mobile applications add unstable wireless data channels and real-time, location-based functionality to ever-more complex architectures for distributed systems. Along with new high-value capabilities, mobile technology also presents new failure modes. Mobility-specific factors must be included in the operational profile to generate realistic test suites for mobile application systems. I believe that at least ten times more testing will be needed to achieve reliability comparable to what can be obtained for wired, static systems. Mobile application developers will face many new design and implementation challenges. With the increased scope and inherent variation of mobile information technology, I see that realistic profile-based testing holds the only hope of achieving high reliability.

Expanding and automating the profile to generate time-varying load as well as functionality allowed us to achieve high reliability for complex distributed applications. In a similar manner, expanding and automating the operational profile to include the key dimensions of mobility will achieve high reliability in complex, distributed, and mobile applications. For the same reasons that merging functionality and stress testing improved both efficiency and effectiveness, we expect to be able to do a lot more for much less, despite the new challenges of mobility.

Sampling the Operational Profile

An operational profile is the estimated relative frequency (i.e., probability) for each “operation” that a system under test supports. Operations map easily to use cases, hence operational profiles and object-oriented development work well together. However, the Unified Modeling Language (UML) standard for Use Cases does not provide sufficient information for automated testing. Extended Use Cases [2] are consistent with the UML, and include domain and probability information necessary for automated testing.

A code sketch follows that shows how test cases can be generated for an operational profile for some Extended Use Cases. We’ll use the ATM example presented in *Software Reliability Engineering - An Overview*. Each operation corresponds to an Extended Use Case. The following model is simplified, as it does not represent sequential and conditional constraints. We’d need to further subdivide each Extended Use Case (operation) into variants to allow realistic sampling from the input data domains and to generate the expected results.

Many deterministic algorithms have been developed to choose operation sequences with constraints. However, even a simple system of constraints can require a very large number of sequences to achieve a coverage goal of interest. Discrete Event Simulation strategies (sometimes called “Monte Carlo simulation”) have been long used to provide statistically adequate samples from very large state spaces. Given the astronomical number of input combinations for most software systems, this strategy is well-suited to choosing test sequences from an operational profile. The following Tcl [3] script illustrates the principle using the ATM operational profile example.

```
puts -nonewline "How many test cases? "
gets stdin n

for {set ix 1} {$ix <= $n} {incr ix} {
    set p [expr {rand()}]

    if          {$p <= 0.33200} {puts "Enter Card"}
    } elseif    {$p <= 0.66400} {puts "Verify PIN"}
    } elseif    {$p <= 0.86300} {puts "Withdraw checking"}
    } elseif    {$p <= 0.92900} {puts "Withdraw savings"}
    } elseif    {$p <= 0.96900} {puts "Deposit checking"}
    } elseif    {$p <= 0.98900} {puts "Deposit savings"}
    } elseif    {$p <= 0.99564} {puts "Query status"}
    } elseif    {$p <= 0.99896} {puts "Test terminal"}
    } elseif    {$p <= 0.99954} {puts "Input to stolen cards list"}
    } elseif    {$p <= 1.00000} {puts "Backup files"}
    } else      {puts "Error in $p"}
}
```

To facilitate selection with a pseudo-random number generator, the original probabilities have been transformed into a cumulative distribution. This script will generate a sequence of operations with two properties. First, each operation will occur roughly in proportion to its original probability. For example, out of a 1,000 generated test cases, “Withdraw savings” would appear about 66 times. Second, for any given generated test case, the

operation of the next test case occurs in proportion to its assigned probability. For example, “Withdraw savings” would be followed by “Deposit Savings” for about 2% of the Withdraw Savings test cases.

Of course, developing a complete model-based testing system which respects all constraints and samples variable domains is considerably more complex. For example, the application-specific system discussed in this article comprises about 60,000 lines of Prolog, Java, SQL, Perl, NT command scripts, and 4Test (Segue’s proprietary test scripting language.)

References

[1] John D. Musa. *Software Reliability Engineering: More Reliable Software Faster and Cheaper*. AuthorHouse. 2004.

[2] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 1999.

[3] Tcl (Tool Control Language) is a widely used open source scripting language which is a de facto standard for test automation. To download a Tcl interpreter and more about Tcl, go to <http://www.activestate.com/Products/ActiveTcl>.

About the Author

Robert V. Binder has over 32 years experience as a software entrepreneur and technologist. Before founding *mVerify*, his consulting firm delivered custom automated testing solutions for very high reliability applications. He is internationally recognized as the author of the definitive *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Binder holds an MS in Electrical Engineering and Computer Science from the University of Illinois at Chicago and a MBA from the University of Chicago. He is an IEEE Senior Member and serves on Agitar’s Technical Advisory Board. He can be reached at Bob_Binder@mverify.com.

Copyright © 2004, Robert V. Binder. All rights reserved.