Software Reliability Engineering
Software Reliability Engineering:
More Reliable Software Faster and Cheaper
2nd. Edition

John D. Musa

The classic guide to software reliability engineering (SRE), the time-saving practice for the software professional

ACM Software Engineering Notes: “…an introductory book, a reference, and an application book all compressed in a single volume…The author’s experience in reliability engineering is apparent and his expertise is infused in the text.”

IEEE Computer: “Toward software you can depend on…This book illustrates the entire SRE process…An aid to systems engineers, systems architects, developers, and managers.”

NOW thoroughly rewritten for the latest SRE practice, enlarged 50%, and polished by thousands of practitioners, with:

- Methods to develop and test software more efficiently, making you more competitive in a world of globalization and outsourcing and giving you more time for your personal life
- Workshops for applying what you learn to your project (NEW FEATURE)
- Frequently asked questions (DOUBLED TO MORE THAN 700)
- Step-by-step process summary
- Simple user manual for CASRE software reliability software, downloadable from the internet
- List of articles on user experiences (you can find how others practiced SRE in various settings)
- Glossary
- Basic practice separated from special situations and math background for faster learning
- Exercises to reinforce what you just learned
- A casual, readable style that works well for self learning
- Everything you need to implement SRE in your organization, even showing you how to persuade other people to adopt the practice

All updated, enhanced, and exhaustively indexed to make the perfect deskside companion for the software professional, engineer, or manager

If you liked the First Edition, you’ll love the Second and want to update.


TO BROWSE AND BUY (WITHOUT LEAVING HOME!)

To browse in detail like you would in a bookstore, visit http://members.aol.com/JohnDMusa. Order one or more copies of this Print on Demand (POD) book directly (with more than 21% discount) from AuthorHouse by using the hyperlink there or call 1-888-280-7715 toll free. Your order will be custom printed and shipped to you as fast as you can obtain a traditionally published professional book. This 630 page book comes in two editions:

<table>
<thead>
<tr>
<th>ISBN</th>
<th>Retail</th>
<th>AH price*</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-4184-9388-0</td>
<td>$79.95</td>
<td>$62.95</td>
</tr>
<tr>
<td>1-4184-9387-2</td>
<td>$69.95</td>
<td>$54.95</td>
</tr>
</tbody>
</table>

* plus shipping and handling
Software Reliability Engineering—
An Overview

More Reliable Software Faster and Cheaper

By John D. Musa, Software Reliability Engineering and Testing Courses

Software reliability engineering (SRE) focuses on making practitioners more competitive without working longer. This is valuable in a world of globalization and outsourcing where professionals want more time for their personal lives. Customers demand that software-based products be more reliable, built faster, and built cheaper (in general order of importance). Success in meeting these demands affects the market share and profitability of a product for a company and hence individuals in the company. The demands conflict, causing risk and overwhelming pressure, and hence a strong need for this practice that can help balance them.

SRE is now standard, proven, widespread, and widely applicable. It is very effective and low in cost, and its implementation has virtually no schedule impact. We will describe it and how it works. We will then outline the SRE process to give you a feel for the practice.

What’s SRE Like?

SRE, backed with science and technology, quantitatively plans and guides software development and test, always keeping in mind a sound business perspective. You add and integrate it with other good processes and practices; you do not replace them. You choose the most cost-effective software reliability strategies for your situation.

We deliberately define software reliability in the same way as hardware reliability, so that we can determine system reliability from hardware and software component reliabilities, even though the mechanisms of failure are different.

Why SRE Works

SRE works by quantitatively characterizing two things about the product: the expected relative use of its functions and its required major quality characteristics. The major quality characteristics are reliability, availability, delivery date, and life-cycle cost. You then apply these characteristics in managing development and test.

When you have characterized use, you can substantially increase development efficiency by focusing resources on functions in proportion to use and criticality. You can also maximize test effectiveness by making tests highly representative of use in the field. Increased efficiency increases the effective resource pool available to add customer value.

When you have determined the precise balance of major quality characteristics that meets user needs, you spend your increased resource pool to carefully match them. You choose software reliability strategies to meet the objectives, based on data collected from previous projects. You also track reliability in system test against its objective to adjust your test process and to determine when you can end test. The result is greater efficiency in converting resources to customer value.

A Proven, Standard, Widespread Best Practice

SRE is a proven, standard, widespread best practice. As one example of its proven benefits, AT&T applied SRE to two different releases of a switching system, International Definity PBX. Customer-reported problems decreased by a factor of 10, the system test interval decreased by a factor of 2, and total development time decreased 30%. No serious service outages occurred in 2 years of deployment of thousands of systems in the field.

SRE became an AT&T Best Current Practice in May 1991 after undergoing rigorous scrutiny as to its cost-effectiveness by a large number of managers. McGraw-Hill recognized SRE as a standard practice by publishing an SRE handbook in 1996. The IEEE and the American Institute of Aeronautics and Astronautics, among others, have developed standards.

Users have published almost 70 articles about their successful application of SRE, and the number continues to grow [1,2]. Since practitioners generally publish infrequently, the total number of successful applications is probably many times greater. We have picked examples of major users for this special issue.

SRE is widely applicable. Technically, you can apply SRE to any software-based product, starting at the beginning of any release cycle. Economically, you can apply SRE to any software-based product, except for very small components (perhaps those involving a total effort of less than 2 staff months). However, if you use a very small component for several products, then it probably will be feasible to use SRE. If not, it still may be worthwhile to implement SRE in abbreviated form.

SRE is independent of development technology and platform. It
requires no changes in architecture, design, or code, but it may suggest changes that would be useful. You can deploy it in one step or in stages.

SRE is very customer-oriented: it involves frequent direct close interaction with customers. This enhances a supplier’s image and improves customer satisfaction. SRE is highly correlated with attaining Levels 3 through 5 of the Software Engineering Institute Capability Maturity Model.

Despite the word “software,” SRE deals with the entire product, although it focuses on the software part. It takes a full-life-cycle view, involving system engineers, system architects, developers, users, and managers in a collaborative relationship.

The cost of implementing SRE is small. There is an investment cost of not more than 3 equivalent staff days per person in an organization, which includes a 2-day course for everyone and planning with a much smaller number. The operating cost over the project life cycle typically varies from 0.1 to 3 percent of total project cost, dropping rapidly as project size increases. The largest cost component is the cost of developing the operational profile.

The schedule impact of SRE is minimal. Most SRE activities involve only a small effort that can parallel other software development work. The only significant critical path activity is 2 days of training.

**SRE Process**

The SRE process has six principal activities, as shown in Figure 1. We show the software development process below and in parallel with it timewise. Both processes follow spiral models, but we don’t show the feedback paths for simplicity. In the field, we collect certain data and use it to improve the SRE process for succeeding releases.

---

### Define Product

The first activity is to define the product. This involves first determining the supplier and the customer, which is not a trivial activity in these days of products built at multiple levels and across corporate boundaries. Then you list all the systems associated with the product that you must test independently. An example is variations, which are versions of the base product that you design for different environments. For example, you may design a product for both Windows and Macintosh platforms.

### Implement Operational Profiles

Operational profiles quantify how software is used. An operation is a major system logical task, which returns control to the system when complete. An operational profile is a complete set of operations with their probabilities of occurrence. Table 1 shows an illustration of an operational profile.

To develop an operational profile, we first list the operations. We then determine the operation occurrence rates and find the occurrence probabilities by dividing the occurrence rates by the total occurrence rate.

When implementing SRE for the first time, some software practitioners are initially concerned about possible difficulties in determining occurrence rates. Experience shows them that this is usually not a difficult problem. Use data often exists on the business side of the house that they are unaware of. Companies usually approve new products for development only after making a business case study. Such a study must typically estimate occurrence rates for the use of the planned functions to demonstrate profitability.
In addition, data is often available or can be derived from a previous release or similar system. One can collect data from the field, and if all else fails, one can usually make reasonable estimates of expected use. In any case, even if there are errors in estimating occurrence rates, the advantage of having an operational profile far outweighs not having one at all.

Once you have developed the operational profile, you can employ it, along with criticality information, in many different ways to more effectively distribute your development and test resources. In addition, you can:

1. Identify operations whose usage does not justify their cost and remove them or handle them in other ways (Reduced Operation Software or ROS)

2. Plan a more competitive release strategy using operational development. With operational development, development proceeds operation by operation, ordered by the operational profile. This makes it possible to deliver the most used, most critical capabilities to customers earlier than scheduled because the less used, less critical capabilities are delivered later.

### Engineer “Just Right” Reliability

A failure is any departure of system behavior in execution from user needs. It differs from a fault, which is a defect in system implementation that causes the failure when executed. Failure intensity is simply the number of failures per time unit. It is an alternative way of expressing reliability.

To engineer the “just right” level of reliability for a product, you must first define “failure” for the product. You then analyze user needs and set the product’s failure intensity objective (FIO). You compute the FIO for the software you are developing by subtracting the total of the expected failure intensities of all hardware and acquired software components from the product FIO. You track reliability growth during system test of all systems you are developing with the failure intensity to failure intensity objective (FI/FIO) ratios. You also apply the developed software FIOs in choosing the mix of software reliability strategies that meet reliability, schedule, and product cost objectives with the lowest development cost.

### Prepare For Test

The Prepare for Test activity uses the operational profiles you have developed to guide the preparation of test cases and test procedures. You distribute test cases across operations in accordance with the operational profile and criticality information. For example, if there are 500 test cases to distribute and operation A has an occurrence probability of 0.17 and ordinary criticality, it will get that fraction of them, or 85.

The test procedure is the controller that invokes test cases during execution. It uses the operational profile, modified to account for critical operations and for reused operations from previous releases, to determine the proportions of invocation.

### Execute Test

In the Execute Test activity, you will first allocate test time among the associated systems and then types of test (feature, load, and regression). You then distribute test time among operations in accordance with the operational profile. Identify failures, along with when they occur, for use in Guide Test.

### Guide Test

For software that you develop, track reliability growth as you attempt to remove faults. Input failure data that you collect in Execute Test to a reliability estimation program such as CASRE [2]. Execute this program periodically and plot the FI/FIO ratio as test proceeds as shown in Figure 2. If you observe a significant upward trend in this ratio, you should determine and correct the causes. The most common causes are system evolution, which may indicate poor change control, and changes in test selection probability with time, which may indicate a poor test process.

If you find you are close to your scheduled test completion date but have an FI/FIO ratio substantially greater than 0.5, you have three feasible options: defer some features or operations, rebalance your major quality characteristic objectives, or increase work hours for your organization. When the FI/FIO ratio reaches 0.5,
you should consider release as long as essential documentation is complete and you have resolved outstanding high severity failures (you have removed the faults causing them).

If you expect that customers will acceptance test the software you are delivering, you should do so first with a procedure called certification test. Certification test involves plotting the time of each failure as it occurs in normalized units (MTTFs) on a reliability demonstration chart as shown in Figure 3. In this case, the first two failures fall in the Continue region. This means that there is not enough data to reach an accept or reject decision. The third failure falls in the Accept region, which indicates that you can accept the software, subject to the levels of risk associated with the chart you are using. If these levels of risk are unacceptable, you construct another chart with the levels you desire [1] and replot the data.

**Collect Field Data**

After you ship a product, you collect certain field data to use in engineering succeeding releases and other products, often building recording and reporting routines into the product. Collect data on failure intensity and on customer satisfaction with the major quality characteristics and use this information in setting the failure intensity objective for the next release. Measure operational profiles in the field and use this information to correct the operational profiles you estimated. Finally, collect information that will let you refine the process of choosing reliability strategies in future projects.

**Conclusion**

If you apply SRE in all the software-based products you develop, you can be confident that you have maximized your efficiency in balancing your customers’ needs for reliability and availability, time of delivery, and cost. Being able to do this is a vital skill to possess if you are to be competitive in today’s marketplace.

**To Explore Further**


2. Software Reliability Engineering website. An essential guide to keeping current in software reliability. Includes short and long overviews; book, course, and consulting information; a bibliography of articles by SRE users; deployment advice; the Question of the Month; links to standards, data, and the downloadable CASRE program: http://members.aol.com/JohnDMusa/

continues on page 7

**Figure 2. Plot of FI/FIO Ratio**

**Figure 3. Reliability Demonstration Chart**
Automated Testing with an Operational Profile

Robert V. Binder, mVerify Corporation

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.

Software Reliability

Continued from page 6.

John D. Musa is one of the founders of SRE, an IEEE Fellow, and Engineer of the Year 2004. In Who’s Who in America since 1990, he is a prolific researcher, international consultant and teacher, and experienced and practical software developer and manager.

Widely recognized as the leader in reducing SRE to practice, he spearheaded the effort that convinced AT&T to make it a “Best Current Practice” in 1991. He has actively implemented and disseminated worldwide his vision of applying SRE to guiding business, engineering, and management decisions in the development, testing, acquisition, and use of software-based systems.


Author Contact Information:

Copyright John D. Musa 2004. Permission is granted to reproduce or distribute this article in its entirety with proper credits included, provided it is not sold or otherwise commercially exploited.
Automated Testing
Continued from page 7.

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 test 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 to 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 a 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 continues on page 9
Automated Testing
Continued from page 8.

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, page 4 of this issue. Each operation corresponds to an Extended Use Case. The following model, shown in Figure 1 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.

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 1,000 generated test cases, “Withdraw savings” would appear continues on page 10
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

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    {$p <=  0.99564} {puts "Deposit savings"}
    elseif    {$p <=  0.99896} {puts "Query status"}
    elseif    {$p <=  1.00000} {puts "Test terminal"}
    elseif    {$p <=  0.99954} {puts "Input to stolen cards list"}
    elseif    {$p <=  1.00000} {puts "Backup files"}
    else                       {puts "Error in $p"}
}

Figure 1. ATM Example

About the Author
Mr. Binder is founder and CEO of mVerify Corporation (www.mverify.com), a startup developing the next generation of testing software for mobile applications: “A Million Users in a Box™”. He has over 29 years of software and systems engineering experience and is an internationally recognized expert in software testing. 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
Applications of SRE in the Security Domain

William W. Everett, James C. Widmaier

Introduction

There has been interest at the National Security Agency (NSA) in applying Software Reliability Engineering (SRE) practices in the security domain. Nine years ago, the second author spent a sabbatical with John Musa in developing methods for Software Reliability Engineering Testing (Musa&Widmaier1996) which he took back to NSA. This article describes some of the initiatives that both authors have been involved with in applying and extending these methods at NSA.

Using Reliability as a Yardstick in Comparing Software Engineering Methodologies

The first initiative taken by NSA was to use reliability as a yardstick to compare two different Software Engineering methodologies (Widmaier1999). The initiative was aimed at answering the question: “Where should a software manager invest time and resources in building higher quality products?” Reliability provides a quantified measure of customer perception of product quality. Two independent software development organizations (contractors) were given identical functional and reliability requirements for a Personnel Access Control System (PACS) and the same resources to build a pilot PACS application. One organization had a mature software development process verified at SEI/CMM level 4. The second organization used formal methods to develop the pilot application. A third organization was engaged to do Reliability Demonstration testing of the completed pilot applications to certify the functional and reliability requirements were met. Interestingly, neither application met the reliability requirements although the application developed using formal methods came closest. This approach in using SRE provided a quantitative way of comparing two different approaches to building a software application.

Automating SRET

With the success in applying SRE in the first initiative, NSA was interested in seeing how SRE could be applied in-house. The need was recognized for tools to automate and standardize SRET. The next initiative was to build an Automated Reliability Testing (ART) toolset to support SRET and to apply it to an in-house built product.

Some of the needs identified for such a tool were: (1) ability to model sophisticated operational profiles¹, (2) generate test cases from the operational profile, (3) automatically schedule and dispatch test cases, (4) collect failure and execution time data from test runs, (5) format such data so they could be input to and analyzed by software reliability analysis tools (e.g., CASRE, SRE Toolkit, Reliability Demonstration Charts). In addition, a process was needed to define how to use ART in managing the reliability of software.

ART used Teradyne Corporation’s TestMaster® tool to model operational profiles (Everett&Widmaier1999). ART was used in managing the reliability of ART itself. Reliability requirements including failure rate objectives and an operational profile were established for ART. ART was then used to generate test cases from the operational profile, to run the test cases, collect failure data and analyze it. Using ART, we showed that the completed ART toolset satisfied its reliability requirements.

In addition to using ART on itself, we also used ART on an in-house developed application (Widmaier&Everett 2002). The application analyzed C-code files, summarizing size and complexity of the C-code. The application had been in use for several years so a lot of information was available to characterize its usage. Again, reliability requirements were established for the application. Developing an operational profile was a challenge. The profile consisted of a characterization of the C-code files that the application would process. ART was used to generate random test cases from the operational profile which consisted of syntactically correct C-code files with known size and complexity measures. ART was then used to run the test cases. The known values of these measures were compared with values produced by the application within ART to identify failures. In this case, we were able to show the application did not meet the reliability requirements we set for it. More important, we uncovered a number of failure types for the application that were not spotted during the years that the application was in operational use.

Applying SRE to Security-Based Applications

With the success in applying SRET to an in-house developed prod-

continues on page 12
Applications of SRE
Continued from page 11.

uct, the next initiative was to apply it to security-based applications. The applications selected were firewall products (Everett&Widmaier2002).

We developed a set of reliability requirements around a potential use of a firewall product within an in-house organization. We interviewed the organization with regard to their reliability needs. We established quantified reliability objectives and an operational profile under which the objectives were to be met. The operational profile not only characterized the packet traffic and network topology but also included Operations, Administration, Maintenance and Provisioning (OAM&P) activities. In addition to normal packet traffic, we did mimic some covert attack activity. We acquired two, off-the-shelf commercial firewall products to test. We setup a test bed and simulated a year’s worth of operational activity.

Although we encountered no packet traffic failures, we did encounter OAM&P failures. Applying Reliability Demonstration methods, we able to reject that one of the products met our reliability requirements and accept that the other product did.

Existing SRE Across the Life-Cycle

In the next initiative, we were interested in seeing how SRE methods could be applied across the life-cycle. In this initiative, we applied SRE methods to an in-house developed security-based application (Tokeneer). The application used smart cards and biometrics to authenticate individuals before allowing them to enter a secure facility. In addition to defining Reliability Requirements for the application and then using Reliability Demonstration Testing to validate the requirements were met, we also investigated how SRE could be used during design. We reviewed the design of the application and made recommendations for design changes that would enhance the reliability of the application. One recommendation was to isolate the critical security functions to a kernel and focus enhanced development methods in implementing the kernel. Drawing on our experience in working with this application, we developed a Protection Profile (PP) for a family of Token ID Station (TIS) products. Our “reliability perspective” was particularly helpful in developing the “vulnerability” and “security policy” parts of the PP. It also helped in identifying security functions needed to mitigate security failures.

Applying SRE Earlier in the Life-Cycle

Most of our previous initiatives were “after the fact” in that the product was already implemented when we did our work. In our next initiative, we wanted to do the reliability work “up front”. The next initiative was to establish reliability requirements for a high-assurance version of an ASN.1 compiler before it was implemented and then do Reliability Demonstration testing after it was implemented to validate the requirements were met. Again, we interviewed perspective users to characterize failures that they would be concerned about and characterize an operational profile. We defined reliability requirements that included quantified objectives and a corresponding operational profile under which the objectives were to be met. The reliability requirements were established before the application was built. We were not involved in building the product but were brought in to do Reliability Testing of the completed application. As with the C-code analyzer, part of the operational profile included a characterization of ASN.1 specifications. We extended parts of ART to randomly generate ASN.1 data specifications based on our operational profile. We acquired a commercial version of an ASN.1 compiler to test against the in-house developed compiler. We setup a test bed that enabled us to (1) generate random ASN.1 data specifications, (2) compile the specifications using the in-house and commercial compilers into C-code, (3) incorporate the C-code for the data specifications into a dummy application thus creating two versions of the dummy application, (4) compile and run the dummy application to encode values for the data specifications, (5) use each version to decode values encoded by the other version. With this approach, we were able to detect when failures occurred and on further investigation attribute the failure to either the in-house or commercial compiler.

Again, we developed an operational profile based on some potential use of the compiler by interviewing potential customers/users. The reliability objectives were a bit harder to establish.

The results were that a number of shortcomings in meeting explicit and implied functional requirements were uncovered. The success in finding such shortcomings was attributed to the independent perspective that the reliability engineer contributed from the perspective of those involved in designing and implementing the application.

Applying SRE to High Integrity Security Applications

In our latest initiative, we wanted to extend SRE methods to building high integrity security based applica-
Applications of SRE
Continued from page 12.

tions. An organization was contracted to build an EAL 5+ TIS Kernel product based on the architecture and PP we defined in the th initiative. The reliability requirements were to be the same as those we had established for Tokeneer in the 4th initiative. Care was taken to explicitly define the behavior expected of the critical functions. A formal functional and design specification for the product were written in Z7. The code was written in SPARK©8 from the Z design specification. This allowed the expected behavior of the kernel to be formally (mathematically) verified to be correct.

We conducted reliability demonstration testing of the implemented product. Although we encountered operational failures (again, mostly OAM&P related), none of the failures were deemed “in scope” of what was explicitly specified for the TIS Kernel.

The result of this work demonstrated the effectiveness of using formal methods to manage the reliability of the explicitly defined behavior of critical functions. However, SRE continued to add value in managing the reliability due to unexpected behavior of the product, i.e., behavior outside of what was explicitly defined.

Conclusion

Based on the results of the various initiatives on applying SRE in the security domain at NSA, we strongly feel that SRE does indeed add value to the software development process. In addition, the approach we took in the technology transfer of SRE within the security domain is an effective one. We were able to achieve some immediate benefits in applying SRET in the PACS project. We then focused on developing support tools for SRET (ART). Using ART, we extended SRET first to an in-house developed application, then to commercial security-based products. Then we extended the application of SRET to in-house developed, security-based products. In addition, we moved from a strictly “testing” focus to applying SRE to earlier phases of the software lifecycle.

1 Specifically, the tool should be able to represent operational profiles in terms of extended finite state machines.
2 There was a third product which we eliminated right off as it could not be provisioned within our operational profile environment.
3 Protection Profiles define the security and assurance requirements for a family of products. They form an integral part of the Common Criteria for assessing and certifying security-based applications. The Common Criteria is an international standard (ISO International Standard 15408) adopted and followed by the international security community.
4 ASN.1 (Abstract Syntax Notation One) is used to specify the structure of data that is transmitted between applications. An ASN.1 compiler generates code for a particular language (e.g., C, Java) that enables specified data structures to be encoded and decoded in a platform-independent manner so they can be passed between applications.
5 A version incorporating code created from the in-house ASN.1 compiler and the other from the commercial compiler.
6 EAL (Evaluation Assurance Level) specifies the level of assurance that should be met in implementing a security-based application. It is part of the Common Criteria (ISO International Standard 15408). Higher EAL levels require more rigorous assurance methods be applied in implementing the application and outlines what assurance methods should be applied for a particular level.
7 Z (pronounce Zed) is a formal notation with a sound mathematical basis used in developing computer systems.
8 SPARK toolset is a product of Praxis Critical Systems. It is a constrained version of Ada. Software written in SPARK can be formally analyzed as to its correctness using a static analyzer and proof checker that is part of the SPARK toolset.

Acknowledgements

We wish to acknowledge other NSA engineers and Harold Wilson (SPRE, Inc.) for the roles they played in the projects described in this article.

References

(Widmaier&Everett2002)

(Widmaier&Widmaier1999)

(Everett&Widmaier2002)

(Everett&Widmaier1999)
Microsoft has found the approach presented in this article to be effective in assessing the reliability of mass market software products such as Windows, both before and after product release. In addition our approach speeds the failure detection and resolution process to assure that quick fixes are developed and then deployed to the user base resulting in an ever increasing level of reliability.

In today’s environment the reliability of software systems is fundamental not just for corporations running large mission critical servers but also for the knowledge worker at the office and even the consumer who stores their family photos and does their banking on their personal computer. Customers have an ever increasing expectation of the reliability of their products that not only requires a focus on producing reliable products, but also developing the ability to quickly produce fixes that can be delivered to those customers.

Historically the computer industry used reliability modeling to predict future behaviour. Hardware failure rates dominated the overall system reliability and the systems were run and managed in very controlled environments. Through capturing failures occurring during system test and feeding them into these models it was possible to accurately predict (to some degree) a product’s future reliability. As the complexity of the systems and software increased then the causes of system failures altered. The operating system started to have an increasing impact on overall system reliability, as did other factors relating to the complexity of the system, such as Human Computer Interface (HCI) failures [2]. While companies continued to rely on software modeling they also started to measure the system behaviour on the customer sites to gain a wider perspective of the actual failure rate of computers [3]. Over time the customer feedback indicated that measuring reliability in terms of system crashes did not fully match their perception of reliability.

Mass market products, such as the Windows operating system, have a number of unique characteristics. Firstly it is no longer possible to test all possible combinations of user configurations. For Windows XP there are currently 35,000+ available drivers, with each driver having over three different versions in the

---

**Applications of SRE**

Continued from page 13.

**About the Authors**

William W. Everett: Bill has been involved in the field of Software Reliability Engineering (SRE) for 17 years. He is chair of the IEEE CS SRE Committee and chair of the Steering Committee for the International Symposium on SRE (ISSRE). For the past 8 years, he has been Principal Consultant and owner of SPRE, Inc., a firm providing consulting/training support in SRE. Prior to that, he was a Distinguished Member of Technical Staff at AT&T Bell Labs. Bill holds a PhD degree in Applied Mathematics from the California Institute of Technology and an Engineer’s Degree from the Colorado School of Mines. He can be reached via email at wwe@SPRE-Inc.com.

James C. Widmaier: Jim is recently retired from the National Security Agency where he has worked since 1974. He was employed as a physicist from 1968 to 1974 with the Electro-Physics Labs. He has a B.S. degree in Physics from Lafayette College and an M.S. in Physics from Pennsylvania State University where he has also completed work for a PhD. Jim is a member of IEEE and ACM and has published several articles related to software quality assurance. He can be reached via email at jcwidmaier@aol.com.
Software Reliability
Continued from page 14.

field, making the number of possible combinations of hardware and drivers, for all practical purposes, infinite. Additionally it is virtually impossible to capture the usage profile of the product. For instance the system management procedures for Windows vary from zero management in the home all the way to servers in corporate datacenters with strict policies.

Therefore the question that Microsoft faced, in developing new products, is whether to persist with reliability models as a reliability predictor or to move towards a measurement based approach.

Product Reliability - Prediction and Measurements

System reliability is a measure of its ability to provide a failure-free operation. For many practical situations this is represented as the failure rate. If the total number of failures for \( N \) installations in a time period \( T \) is \( F \), then a good estimate for the failure rate of the software is \( \lambda = F / (N * T) \). This approach for measuring failure rates has been widely used [1].

Reliability growth models, using data collected during testing, predict the future reliability of the product assuming:

- Configuration of the test systems are representative of the user environment
- Product usage and management does not impact reliability
- Failures captured during system test are representative of customer failures
- Failures occur once and are then corrected

We can test these assumptions against the two usage extremes of a mass market product such as Windows. Corporate IT departments typically have a standardized server environment, high quality administrators and experienced users. The opposite is a home user running Windows on a machine bought from a white box manufacturer.

Corporate customers traditionally pre-stage their systems. The computers will only use a standard set of components, such as signed drivers. The systems usage profile and management will still affect product reliability [2, 3], but a local call desk will assist users thereby limiting the impact of any failures.

The home PC is increasingly becoming more complex than those found in corporate environments. These systems often consist of state of the art CPU’s, graphical processors, sound cards, modems and networking. Within the home they will be connected to a number of products, of different ages, such as cable modems, printers, cameras, MP3 players and other computers in the house. The home user’s configuration is less likely to have been well tested, as the possible number of configurations is infinite. The quality of some of the drivers may be questionable and these will not have been digitally signed by Microsoft. A signed driver indicates that it has been designed for Microsoft operating systems and has been tested for compatibility with Windows. Users are informed that they are running unsigned drivers but may have no option but to install the driver (for instance if their cable modem driver, provided by the ISP, is unsigned). The user may have little to no computer experience but may still be performing system management tasks, such as creating a network of computers in the home; this will inevitably increase the likelihood of HCI induced failures (e.g. due to users attempting to configure systems in ways they were never designed for).

For systems in controlled corporate environments it is possible to build reliability models that are representative of their usage profiles but this is much more difficult for systems in the home environment. For home systems a measurement based approach is applied.

Problems in areas normally not addressed by reliability models, such as HCI, can be effectively addressed through measurement (identifying areas of complexity which results in incorrect system settings). HCI problems that are identified can be addressed using traditional HCI techniques [6]. To address faults in released products Microsoft has developed Windows Error Reporting (WER). This collects and provides fault resolution to all users. The effectiveness of these processes in addressing overall product reliability is continually measured and improved.

Both modeling and measurements have advantages and disadvantages. Modeling provides a means of prediction, but without complete usage characterization it only addresses a subset of failures. For home users in particular there is a decreasing relationship between actual and predictive failure rates. Applying measurement techniques allows companies to capture a complete picture of reliability, albeit later in the development cycle. In choosing to use a measurement approach the next issue to address is how to interpret the data collected through the measurement programs.

Characterizing Reliability of Software Systems

To measure the reliability of a software product it is essential to determine what constitutes a failure. Reliability is defined as the probability of encountering a failure in a
Software Reliability

Continued from page 15.

specified amount of time t, where a failure can be defined as a departure from requirements. A failure is traditionally viewed as having occurred when a system stops responding/working, e.g., the system crashed or hung. However, as already mentioned, there are many types of failures (departures from requirements) beyond hard failures. Customers may view service disruption, not resulting from a system failure, as a systems reliability event.

Based on extensive customer feedback, Microsoft takes a broad view of customer reliability requirements, whereby systems and software should operate without disruption. As such, system and software reliability also takes into account other types of disruptions including planned events, e.g., system shutdowns required to install an application. Microsoft categorizes disruptions into six different classes of events:

- **Resilience**: The system will continue to provide the user service in the face of internal or external disruptions.
- **Recoverable**: Following disruption the system can be easily restored, through instrumentation and diagnosis, to a previously known state with no data loss.
- **Undisruptable**: System changes and upgrades do not impact the service being provided by the system.
- **Performing**: Provides accurate and timely service whenever needed.
- **Production Ready**: On release the system contains a minimum number of bugs, requiring a limited number of predictable patches/fixes.
- **Predictable**: It works as advertised, and is backward compatible.

As mentioned, a key aspect of mass market software products is the extensive variety of possible usage scenarios. For example, home users want their drivers (all 100,000+ possible versions!) to work with their specific systems but are not overly bothered if any configuration changes require a shutdown as these are frequently occurring events (e.g., turning it off at night). For the data center IT system manager, dependent upon a limited set of well tested drivers running across thousands of servers, any configuration changes requiring a shutdown is very costly. The disruption classification above provides a framework for assigning weights based on specific user needs. Using the proper weights for each class of disruption we can arrive at a meaningful assessment of reliability for specific customer scenario. With these scenario specific weights, the reliability of a software system can then be defined as the 6-tuple

\[ R = \left( R_{\text{Resilient}}, R_{\text{Recoverable}}, R_{\text{Undisruptable}}, R_{\text{Performing}}, R_{\text{Production Ready}}, R_{\text{Predictable}} \right) \]

Where we use common reliability metrics to define each member of the 6-tuple, for example, for the Resilient class of disruptions we define the reliability \( R_{\text{Resilient}} \) as

\[ \text{Mean Time To Resilience Disruption (MTTRD)} = \left( \sum \text{system uptimes} \right) / (\# \text{of resilience disruptions}) \]

Similar definitions can be developed for corresponding Availability and Downtime metrics. Any improvement in individual metrics will result in a corresponding improvement in the reliability experienced by the user. These six measures of reliability can be aggregated into a single overall reliability number using the specific scenario weights. Keeping the measurements separate allows us to identify where improvement might be needed. For instance, if we conclude that Resilient systems should function without crashes or hangs irrespective of hardware and software errors, then the subset of events \( \{ I \} \) for the Resilient class of disruption would be \{ crashes, hangs \} and \( R_{\text{Resilient}} \) is a measure of the time to a crash or hang.

This approach recognizes that a mass market software product like Windows does not have a single reliability figure; product reliability is a function of the customer usage scenario and the level of reliability will likely vary from scenario to scenario. Additionally with hundreds of millions of users it is necessary to develop multiple measurement approaches, dependent upon whether you want to collect usage profiles from a target set of users or failures occurring on all user systems.

Breadth and Depth Measurement Approach

Once reliability goals for specific customer scenarios have been defined it is necessary to implement an effective reliability measurement approach. In the past, failures reported by users to a product support organization have been used [1] to evaluate the product’s reliability. But it is well known that customers do not report all the problems they encounter especially when they solve it themselves. This non-reporting is far more pronounced in mass-market products, compared to software products with on-site service contracts. Reports received are likely to be limited to hard failures (as these may require a support call to resolve).

The best approach for collecting failure data is through automated system reporting based on product...
Software Reliability

Continued from page 16.

instrumentation and triggers. When collecting data from software products, with a user base in the hundreds of millions, scalability becomes a major issue. Our approach is to collect data using a breadth and depth perspective. We take a broad sample of data to assure we are covering the entire population base and the variety of possible issues. This data is not detailed but it does allow us to scale to large numbers of reporting customers. We then focus on getting detailed data on a small subset of those users. The broad data is used to assess cost/probability of disruptions for the user base and the depth data is targeted to identify the root cause of the disruption to be used for product improvement. For privacy reasons, the customer decides if and what data they will share.

For the Windows product we have automated reliability measurement mechanisms including WER that focuses on crashes and hangs, and the Microsoft Reliability Analysis Service (MRAS) [7] focused on reliability and availability tracking of Windows servers, and products running on servers like MS SQL database, MS IIS web server, MS Mail Exchange, and the Windows Active Directory. WER collects high level information on each crash/hang reported to Microsoft, but in general will get a dump from only a very small number of the crashes submitted that is sufficient to identify the cause of the problem. Aside from providing Microsoft with data for product improvement both of these processes also provide customer feedback. Table 1 shows the information collected and customer feedback provided in each case.

WER is available to all Windows XP and Windows Server 2003 users; MRAS has been deployed to over 200 corporate customers and is being used extensively within Microsoft to collect data from thousands of servers. Both WER and MRAS have been instrumental in assessing the reliability of beta versions of Windows Server 2003 at both Microsoft and customer sites and are being used for ongoing reliability tracking and new product version evaluation [5].

Acknowledgement

This paper is derived from a prior paper [5], published at the ISSRE 2004 conference at Saint-Malo. The authors would like to acknowledge the work of Ben Errez and Pankaj Jalote who were joint authors of this prior paper.

Table 1: Data collecting and customer feedback

<table>
<thead>
<tr>
<th>Data Collection</th>
<th>Microsoft Data</th>
<th>Customer Feedback</th>
</tr>
</thead>
<tbody>
<tr>
<td>WER</td>
<td>crash &amp; hang dumps</td>
<td>available fixes for crashes &amp; hangs</td>
</tr>
<tr>
<td>MRAS</td>
<td>disruption times &amp; causes</td>
<td>reliability metrics &amp; disruption reasons</td>
</tr>
</tbody>
</table>

References


6. A general list of usability papers and techniques can be found at http://www.microsoft.com/usability/publications.htm


About the Authors

Mario Garzia is Director of Windows Reliability at Microsoft. Prior to joining Microsoft in 1997, he was a Distinguished Member of Technical Staff at AT&T Bell Laboratories working in the areas of telecommunication and computer system and service performance and reliability. He is the co-author of the book Network Model-
Are you getting **All** you can from your software investment?

**The DACS ROI Dashboard**

**Impact on Quality (% defect reduction)**

75th Percentile: 90

Oklahoma City Air Logistics Center, Test Program Set and Industrial Automation line observed the defect density decrease 91%, from 3.3 to 0.3 Defects per KSLOC, while achieving CMM Level 4.

**Fact Information**

- **CMM Software Process Level**
  - Initial State: CMM Level 1 in 1990
  - Process Improvement
  - CMM Level 3 in 1990
  - CMM Level 4 in 1999
  - ISO 9001 in 1999

**Observed Results:**
- Total development time was 15 months measured before the improvement and 13 months measured after the improvement
- Number of defects observed per unit output was 1.2 defects per KSLOC measured before the improvement and 0.3 defects per KSLOC measured after the improvement
- CMM Level 4 in 1990
- ISO 9001 in 1999

**Graphs Showing Impact Of Software Technologies On:**
- ROI
- Productivity
- Quality

**Summarizes Facts From Open Literature**

Data & Analysis Center for Software
P.O. Box 1400
Rome, NY 13442-1400

http://iac.dtic.mil/dacs

Access the DACS ROI Dashboard!

http://www.thedacs.com/databases/roi/
Introduction

The Space Shuttle avionics software represents a successful integration of many of the computer industry’s most advanced software engineering practices and approaches. Beginning in the late 1970’s this software development and maintenance project has evolved one of the world’s most mature software processes applying the principles of the level 5 of the Carnegie Mellon University Software Engineering Institute’s Capability Maturity Model. This article explores the successful use of extremely detailed fault and failure history, throughout the software life cycle, in the application of software reliability engineering techniques to gain insight into the flight-worthiness of the software.

Using the Shuttle application, we show how Software Reliability Engineering (SRE) can be applied to: interpret software reliability predictions, support verification and validation of the software, assess the risk of deploying the software, and predict the reliability of the software. Predictions are currently used by the software developer to add confidence to the reliability assessments of the Primary Avionics Shuttle Software (PASS) achieved through formal software certification processes.

Interpretation of Software Reliability Predictions

Successful use of statistical modeling in predicting the reliability of a software system requires a thorough understanding of precisely how the resulting predictions are to be interpreted and applied [6]. The PASS (430 KLOC) is frequently modified, at the request of NASA, to add or change capabilities, using a constantly improving process. Each of these successive PASS versions constitutes an upgrade to the preceding software version. Each new version of the PASS (designated as an Operational Increment, OI) contains software code which has been carried forward from each of the previous versions (“previous-version subset”) as well as new code generated for that new version (“new-version subset”). We have found that by applying a reliability model independently to the code subsets according to the following rules, we can obtain satisfactory composite predictions for the total version:

1. all new code developed for a particular version uses the same development process.
2. all code introduced for the first time for a particular version is considered to have the same life and operational execution history
3. once new code is added to reflect new functionality in the PASS, this code is only changed thereafter to correct faults.

Estimating Execution Time

We estimate execution time of segments of the PASS software by analyzing records of test cases in digital simulations of operational flight scenarios as well as records of actual use in Shuttle operations. Test case executions are only counted as “operational execution time” for previous-version subsets of the version being tested if the simulation fidelity very closely matches actual operational conditions. Pre-release test execution time for the new code actually being tested in a version is never counted as operational execution time. We use the failure history and operational execution time history for the new-code subset of each version to generate an individual reliability prediction for that new code in each version by separate applications of the reliability model. This approach places every line of code in the total PASS into one of the subsets of “newly” developed code, whether “new” for the original version or any subsequent version. We then represent the total reliability of the entire software system as that of a composite system of separate components (“new-code subsets”), each having an individual execution history and reliability, connected in series. The developer uses this approach to apply the Schneidewind Model [5, 6] as a means of predicting a “conservative lower bound” for the PASS reliability. This prediction is important because the user can be confident that it is highly likely that the software reliability would be no worse than this bound in operation.

Verification and Validation

Software reliability measurement and prediction are useful approaches to verify and validate software. Measurement refers to collecting and analyzing data about the observed reliability of software, for example, the occurrence of failures during test. Prediction refers to using a model to forecast future software reliability, for example, failure rate during operation. Measurement also provides the failure data that is used to estimate the parameters of reliability models (i.e., make the best fit of the model to the observed failure data). Once the parameters have been estimated, the model is used to predict...
the future reliability of the software. Verification ensures that the software product, as it exists in a given project phase, satisfies the conditions imposed in the preceding phase (e.g., reliability measurements of ultrareliable systems software components obtained during test conform to reliability specifications made during design) [2]. Validation ensures that the software product, as it exists in a given project phase, which could be the end of the project, satisfies requirements (e.g., software reliability predictions obtained during test correspond to the reliability specified in the requirements) [2]. Another way to interpret verification and validation is that it builds confidence that software is ready to be released for operational use. The release decision is crucial for systems in which software failures could endanger the safety of the mission and crew (i.e., ultrareliable systems software). To assist in making an informed decision, we integrate software risk analysis and reliability prediction.

**Risk Assessment**

Safety risk pertains to executing the software of an ultrareliable systems system where there is the chance of injury (e.g., astronaut injury or fatality), damage (e.g., destruction of the Shuttle), or loss (e.g., loss of the mission) if a serious software failure occurs during a mission. In the case of the PASS, where the occurrence of even trivial failures is extremely rare, the fraction of those failures that pose any safety or mission risk is too small to be statistically significant. As a result, for risk assessment to be feasible, all failures (of any severity) over the entire 20-year life of the project have been included in the failure history database for this analysis. Therefore, the risk criterion metrics to be discussed for the Shuttle quantify the degree of risk associated with the occurrence of any software failure, no matter how insignificant it may be. This approach can be applied to assessing safety risk where sufficient data exist.

The prediction methodology [3] provides bounds on total test time, remaining failures, and time to next failure that are necessary to perform the risk assessment. Two criteria for software reliability levels are defined. Then these criteria are applied to the risk analysis of ultrareliable systems software, using the PASS as an example. In the case of the Shuttle example, the “risk” represents the degree to which the occurrence of failures does not meet required reliability levels, regardless of how insignificant the failures may be. Next, selected prediction equations that are used in reliability prediction and risk analysis are defined and derived.

**Criteria for Reliability**

If the reliability goal is the reduction of failures of a specified severity to an acceptable level of risk [4], then for software to be ready to deploy, after having been tested for total time \( t_f \), it must satisfy the following criteria:

1) predicted remaining failures \( r(t_f) < r_c \), (1) where \( r_c \) is a specified critical value, and

2) predicted time to next failure \( T_F(t_f) > t_m \), (2) where \( t_m \) is mission duration.

For systems that are tested and operated continuously like the Shuttle, \( t_f, T_F(t_f), \) and \( t_m \) are measured in execution time. Note that, as with any methodology for assuring software reliability, there is no guarantee that the expected level will be achieved. Rather, with these criteria, the objective is to reduce the risk of deploying the software to a “desired” level.

**Remaining Failures Criterion**

Using the assumption that the faults that cause failures are removed (this is the case for the Shuttle), criterion 1 specifies that the residual failures and faults must be reduced to a level where the risk of operating the software is acceptable. As a practical matter, \( r_c=1 \) is suggested. That is, the goal is to reduce the expected remaining failures of a specified severity to less than one before deploying the software. The assumption for this choice is that one or more remaining failures would constitute an undesirable risk of failures of the specified severity. Thus, one way to specify \( r_c \) is by failure severity level, as classified by the developer, (e.g., include only life threatening failures). Another way, which imposes a more demanding criterion, is to specify that \( r_c \) represents all severity level, as in the Shuttle example. For example, \( r(t_f) < 1 \) would mean that \( r(t_f) \) must be less than one failure, independent of severity level.

If \( r(t_f) > r_c \) is predicted, testing would continue for a total time \( t_f \), that is predicted to achieve \( r(t_f) < r_c \) using the assumption that more failures will be experienced and more faults will be corrected so that the remaining failures will be reduced by the quantity \( r(t_f) \rightarrow r(t_f) \). If the developer does not have the resources to satisfy the criterion or is unable to satisfy the criterion through additional testing, the risk of deploying the software prematurely should be assessed (see the next section). It is known that it is impossible to demonstrate the absence of faults [1]; however, the risk of failures occurring can be reduced to an acceptable level, as represented by \( r_c \). This scenario is shown in Figure 1. In case \( A \), \( r(t_f) < r_c \) is predicted and the mission begins at \( t_f \). In case \( B \), \( r(t_f) > r_c \) is predicted and the mission would be postponed until the software continues on page 21.
is tested for total time \( t_\text{f} \), when \( r (t_\text{f}) < r_c \) is predicted. In both cases, criterion 2 must also be satisfied for the mission to begin. If neither criterion is satisfied, the software is tested for a time which is the greater of \( t_\text{t} \) or \( t_\text{f} \).

Equation (3) is plotted in Figure 3 as a function of \( t \) for \( r_c = 1 \), where positive, zero, and negative values correspond to \( r (t) > r_c \), \( r (t) = r_c \), and \( r (t) < r_c \), respectively. In Figure 3, these values correspond to the following regions: CRITICAL (i.e., above the X-axis predicted remaining failures are greater than the specified value); NEUTRAL (i.e., on the X-axis predicted remaining failures are equal to the specified value); and DESIRED (i.e., below the X-axis predicted remaining failures are less than the specified value, which could represent a “safe” threshold or in the Shuttle example, an “error-free” condition boundary). This graph is for the Shuttle Operational Increment OID (with many years of operation): a software system comprised of modules and configured from a series of builds to meet Shuttle mission functional requirements. In this example, it can be seen that at approximately \( t = 57 \) the risk transitions from the CRITICAL region to the DESIRED region.

**Time to Next Failure Criterion**

Criterion 2 specifies that the software must survive for a time greater than the duration of the mission. If \( T_F (t_i) \leq t_m \) is predicted, the software is tested for a total time \( t_\text{t} > t_i \) that is predicted to achieve \( T_F (t_\text{t}) > t_m \), using the assumption that more failures will be experienced and faults corrected so that the time to next failure will be increased by the quantity \( T_F (t_\text{t}) - T_F (t_i) \). Again, if it is infeasible for the developer to satisfy the criterion for lack of resources or failure to achieve test objectives, the risk of deploying the software prematurely should be assessed. This scenario is shown in Figure 2. In case A \( T_F (t_i) > t_m \) is predicted and the mission begins at \( t_i \). In case B, \( T_F (t_i) \leq t_m \) is predicted and in this case the mission would be postponed until the software is tested for total time \( t_\text{f} \), when \( T_F (t_\text{f}) > t_m \).

**Remaining Failures Metric**

The mean value of the risk criterion metric (RCM) for criterion 1 is formulated as follows:

\[
RCM r (t) = (r (t) - r_c) / r_c = (r (t) / r_c) - 1 \tag{3}
\]

\[21\]

(Continued from page 20.)
**Time to Next Failure Metric**

Similarly, the mean value of the risk criterion metric (RCM) for criterion 2 is formulated as follows:

\[
RCM T_F (t_i) = \frac{(t_m - T_F (t_i))}{t_m} = I - \frac{T_F (t_i)}{t_m}
\]  

(4)

Equation (4) is plotted in Figure 4 as a function of \( t_i \) for \( t_m = 8 \) days (a typical mission duration time for this OI), where positive, zero, and negative risk corresponds to \( T_F (t_i) < t_m \), \( T_F (t_i) = t_m \), and \( T_F (t_i) > t_m \), respectively. In Figure 4, these values correspond to the following regions: CRITICAL (i.e., above the X-axis predicted time to next failure is less than the specified value); NEUTRAL (i.e., on the X-axis predicted time to next failure is equal to the specified value); and DESIRED (i.e., below the X-axis predicted time to next failure is greater than the specified value). This graph is for the Shuttle operational increment OIC. In this example, the RCM is in the DESIRED region at all values of \( t_i \).

**Lessons Learned**

Several important lessons have been learned from our experience of twenty years in developing and maintaining the PASS, which you could consider for adoption in your SRE process:

1) No one SRE process method is the “silver bullet” for achieving high reliability. Various methods, including formal inspections, failure modes analysis, verification and validation, testing, statistical process management, risk analysis, and reliability modeling and prediction must be integrated and applied.

2) The process must be continually improved and upgraded. For example, experiments with software metrics have demonstrated the potential of using metrics as early indicators of future reliability problems. This approach, combined with inspections, allows many reliability problems to be identified and resolved prior to testing.

3) The process must have feedback loops so that information about reliability problems discovered during inspection and testing is fed back not only to requirements analysis and design for the purpose of improving the reliability of future products, but also to the...
requirements analysis, design, inspection and testing processes themselves. In other words, the feedback is designed to improve not only the product but also the processes that produce the product.

4) Given the current state-of-the-practice in software reliability modeling and prediction, practitioners should not view reliability models as having the ability to make highly accurate predictions of future software reliability. Rather, software managers should interpret these predictions in two significant ways: a) providing increased confidence, when used as part of an integrated SRE process, that the software is safe to deploy; and 2) providing bounds on the reliability of the deployed software (e.g., high confidence that in operation the time to next failure will exceed the predicted value and the predicted value will safely exceed the mission duration).

References

About the Author:
Norman F. Schneidewind is Professor of Information Sciences at the Naval Postgraduate School. Dr. Schneidewind was selected for an IEEE USA Congressional Fellowship for 2005 and will work on the Committee on Governmental Affairs in the U.S. Senate.

Dr. Schneidewind is a Fellow of the IEEE, elected in 1992 for contributions to software measurement models in reliability and metrics, and for leadership in advancing the field of software maintenance”. In 2001, he received the IEEE “Reliability Engineer of the Year” award from the IEEE Reliability Society.

In 1993 and 1999, he received awards for Outstanding Research Achievement by the Naval Postgraduate School. He is the developer of the Schneidewind software reliability model that is used by NASA to assist in the prediction of software reliability of the Space Shuttle. This model is one of the models recommended by the American Institute of Aeronautics and Astronautics Recommended Practice for Software Reliability.

nschneid@nps.navy.mil
The DACS Gold Practice Initiative:

- Promotes effective selection/use of software acquisition & development practices
- Defines essential activities/benefits of each practice
- Considers the environment in which each practice is used
- Addresses the timeliness of practice benefits
- Recognizes interrelationships between practices that influence success or failure
- Contains quantitative and qualitative information
- Includes discussion forums, case studies and survey opportunities
- A continually evolving resource for the DoD, Government, Industry and Academia
- Free to use/free to join

Current Gold Practices:

- Architecture-First Approach
- Binary Quality Gates at the Inch-Pebble Level
- Commercial Specifications and Standards/Open Systems
- Ensure Interoperability
- Formal Inspections
- Formal Risk Management
- Integrated Product and Process Development
- Model-Based Testing
- Plan for Technology Insertion
- Requirements Management
- Requirements Trade-Off/Negotiations
- Statistical Process Control
- Track Earned Value

Learn More About the DACS Gold Practice Initiative:
http://www.goldpractices.com

Become a Registered Member!
http://www.goldpractices.com/registration/register_step1.php

Data & Analysis Center for Software
P.O.Box 1400
Rome, NY 13442-1400

http://iac.dtic.mil/dacs
The first 50 people to send in a completed survey will receive a FREE DoD/IT Acronym CD from the DACS.

This valuable CD-ROM contains over 9,000 Department of Defense and Information Technology acronyms. There are hundreds of acronym lists available but none are as well done as this CD AND specifically targeted towards DoD and Information Technology. This unique-shaped CD-ROM plays in your computer’s regular, hub-mounted, CD drive. You’ll use this great resource over and over again. It’s FREE, just for filling out our brief survey on the next page!
Software Tech News Subscriber Survey

1. Which volume of the Software Tech News did you receive?  STN 8:1 Software Reliability Engineering

2. When did you receive the newsletter? (month/year)

3. How satisfied were you with the CONTENT of the newsletter? (Article Quality)
   - Very Satisfied
   - Satisfied
   - Neither Satisfied nor Dissatisfied
   - Dissatisfied
   - Very Dissatisfied

4. How satisfied were you with the APPEARANCE of the newsletter?
   - Very Satisfied
   - Satisfied
   - Neither Satisfied nor Dissatisfied
   - Dissatisfied
   - Very Dissatisfied

5. How satisfied were you with the OVERALL QUALITY of the newsletter?
   - Very Satisfied
   - Satisfied
   - Neither Satisfied nor Dissatisfied
   - Dissatisfied
   - Very Dissatisfied

6. How satisfied were you with the ACCURACY of the address on the newsletter?
   - Very Satisfied
   - Satisfied
   - Neither Satisfied nor Dissatisfied
   - Dissatisfied
   - Very Dissatisfied

7. Approximately how much of the newsletter do you read?
   - The entire issue
   - Most of the content
   - About half the content
   - Briefly Skimmed
   - Didn’t Read

8. Would you read this newsletter in an E-mail newsletter format?
   - Definitely
   - Probably
   - Not Sure
   - Probably Not
   - Definitely Not

9. How did you request the product or service?
   - Phone Call
   - E-mail
   - DACS Website
   - Subscription Form
   - Other ____________________________

10. Would you recommend the DoD Software Tech News to a colleague?
    - Definitely
    - Probably
    - Not Sure
    - Probably Not
    - Definitely Not

11. What topics would you like to see this newsletter devoted to?

    Comments (Optional)

    Contact Information (Optional*)
    Name: Position/Title:
    Organization: Office Symbol:
    Address: 
    City: State: Zip Code:
    Country: E-mail:
    Telephone: Fax:
STN Editorial Board

Philip King  
Editor  
ITT Industries, DACS

Paul Engelhart  
DACS COTR  
Air Force Research Lab (IFEA)

Morton A. Hirschberg  
Editorial Board Chairman  
Army Research Lab (retired)

Ellen Walker  
ITT Industries, DACS

Thomas McGibbon  
DACS Director  
ITT Industries, DACS

David Nicholls  
DACS Deputy Director  
ITT Industries, DACS

Marshall Potter  
Federal Aviation Administration

Richard Turner  
OUSD (AT&L)

About the DoD Software Tech News

Article Reproduction
Images and information presented in these articles may be reproduced as long as the following message is noted:

“This article was originally printed in the DoD Software Tech News, Vol. 8, No. 1. Requests for copies of the referenced newsletter may be submitted to the following address:

Philip King, Editor  
Data & Analysis Center for Software  
P.O. Box 1400  
Rome, NY 13442-1400

Phone: 800-214-7921  
Fax: 315-334-4964  
E-mail: news-editor@dacs.dtic.mil

An archive of past newsletters is available at:  

In addition to this print message, we ask that you send us three copies of any document that references any article appearing in the DoD Software Tech News.

Cover Design by Joe Barbaccia,  
ITT Industries

About This Publication:
The DoD Software Tech News is published quarterly by the Data & Analysis Center for Software (DACS). The DACS is a DoD sponsored Information Analysis Center (IAC), administratively managed by the Defense Technical Information Center (DTIC). The DACS is technically managed by Air Force Research Laboratory, Rome, NY and operated by ITT Industries, Advanced Engineering and Sciences Division.

To Subscribe to this Publication Contact:
Phone: 800-214-7921  
Fax: 315-334-4964  
E-mail: news-editor@dacs.dtic.mil  
Web:  
www.dacs.dtic.mil/forms/userform.shtml

Distribution Statement:  
Unclassified and Unlimited
In This Issue

Software Reliability Engineering-
An Overview ........................................... 3

Automated Testing with an
Operational Profile................................. 7

Applications of SRE in the
Security Domain.................................... 11

Software Reliability Engineering
for Mass Market Products........................ 14

Application of SRE to Ultrareliable
Systems - The Space Shuttle ............... 19

About the DoD Software Tech News......... 27

Advertisement

The DoD Software Tech News is now ac-
cepting advertisements for future newsletters. In
addition to being seen by the thousands of
people who subscribe to the DoD Software
Tech News in paper copy, the Tech News will
also be placed on the Data & Analysis Center
for Software’s website
(http://iac.dtic.mil/dacs/), exposing your prod-
uct, organization, or service to hundreds of
thousands of additional eyes.

Interested in learning more? For rates, lay-
out information, and requirements contact:

Philip King, STN Editor
Data & Analysis Center for Software
P.O. Box 1400
Rome, NY 13442-1400

Phone: (800) 214-7921
Fax: (315) 334-4964
E-mail: news-editor@dacs.dtic.mil