NPS-PM-09-146

ACQUISITION RESEARCH SPONSORED REPORT SERIES

Driving Automated Open-Architecture Testing: An Operational Profile Model-Development Strategy

30 September 2009

by

Dr. Luqi, Professor

Dr. Valdis Berzins, Professor, and

Paul Dailey

Graduate School of Operational & Information Sciences

Naval Postgraduate School

Prepared for: Naval Postgraduate School, Monterey, California 93943

ACQUISITION RESEARCH PROGRAM
GRADUATE SCHOOL OF BUSINESS & PUBLIC POLICY
NAVAL POSTGRADUATE SCHOOL

The research presented in this report was supported by the Acquisition Chair of the Graduate School of Business & Public Policy at the Naval Postgraduate School.

To request Defense Acquisition Research or to become a research sponsor, please contact:

NPS Acquisition Research Program
Attn: James B. Greene, RADM, USN, (Ret)
Acquisition Chair
Graduate School of Business and Public Policy
Naval Postgraduate School
555 Dyer Road, Room 332
Monterey, CA 93943-5103
Tel: (831) 656-2092
Fax: (831) 656-2253
e-mail: jbgreene@nps.edu

Copies of the Acquisition Sponsored Research Reports may be printed from our website www.acquisitionresearch.org

ACQUISITION RESEARCH PROGRAM
GRADUATE SCHOOL OF BUSINESS & PUBLIC POLICY
NAVAL POSTGRADUATE SCHOOL

Abstract

The ability to automate the testing of software is critical in the successful acquisition of future Open Architecture (OA)-based weapon and combat systems. In the OA environment, software modules are reused in different environments, and systems are highly modular in nature. These situations result in increased numbers of potential software configurations. As configurations change and OA software is put in new environments, the potential for finding or creating new bugs increases. The concept of using an operational profile, or environment model that generates inputs to an OA software module, based on probabilistic distributions, to assist in the automated testing process is critical for catching such bugs. This paper describes an ongoing research effort on operational profile models, presents a summary of prior relevant work, and outlines an initial strategy for developing and implementing the operational profile model concept. Following the model acquisition approach, a recommendation for future work is presented.

Keywords: Open Architecture, Automated Testing, Software Testing, Operational Profile

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

This research effort is being conducted under the guidance of Paul Dailey's PhD committee—consisting of Dr. Valdis Berzins, Dr. Luqi, Dr. Ronald Fricker, Dr. Clifford Whitcomb, Dr. Robert Harney and Dr. Peter Musial.

THIS PAGE INTENTIONALLY LEFT BLANK

About the Authors

Dr. Luqi is Professor of Computer Science at NPS. Her research on many aspects of software reuse and computer-aided software development has produced hundreds of research papers in refereed journals, conference proceedings and book chapters. She has served as a PI or co-PI for many research projects funded by the DoD and DoN. She has received the Presidential Young Investigator Award from NSF and the Technical Achievement Award from IEEE.

Valdis Berzins is a Professor of Computer Science at the Naval Postgraduate School. His research interests include software engineering, software architecture, computer-aided design, and theoretical foundations of software maintenance. His work includes papers on software testing, software merging, specification languages, and engineering databases. He received BS, MS, EE, and PhD degrees from MIT and has been on the faculty at the University of Texas and the University of Minnesota. He has developed several specification languages, software tools for computer-aided software design, and a fundamental theory of software merging.

Paul Dailey is a systems engineer at the Naval Postgraduate School and has worked for the Department of the Navy for seven years, including as a test & evaluation engineer for the Naval Surface Warfare Center, Port Hueneme Division, Detachment Louisville from 2002 to 2009. He holds a MS in Systems Engineering from the Naval Postgraduate School and a BS in Electrical Engineering from the University of Louisville. He is currently pursuing a PhD in Software Engineering from the Naval Postgraduate School, focusing his research on the automated testing of software.

THIS PAGE INTENTIONALLY LEFT BLANK

NPS-PM-09-146

ACQUISITION RESEARCH SPONSORED REPORT SERIES

Driving Automated Open-Architecture Testing:
An Operational Profile Model Development Strategy

30 September 2009

by

Dr. Luqi, Professor

Dr. Valdis Berzins, Professor, and

Paul Dailey

Graduate School of Operational & Information Sciences

Naval Postgraduate School

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

Introduction **1**

Ongoing Operational Profile Research Effort **3**

Summary of Prior Relevant Work **7**

Testing in an Open Environment 7

Determining when It Is Safe Not to Retest a Component following Changes in Configuration 8

Cost of Automated Testing 13

Testing with Operational Profiles 16

Operational Profile Model Development Methodology **19**

Process Overview 19

Steps to Develop and Operational Profile Model 19

Recommendations for Future Work **25**

List of References **27**

THIS PAGE INTENTIONALLY LEFT BLANK

Introduction

As more open systems are being fielded by the US Navy and its allies, traditional system boundaries and, subsequently, interfaces become more varied and more challenging to define. The modularity of open systems and software adds significant flexibility in their configuration, enabling components to be added or modified to meet specific requirements with relative ease when compared to non-open systems. With this flexibility comes increased difficulty of conducting testing using traditional testing approaches. New methods are required to effectively test such systems due to the increased potential for configuration changes.

Current US Navy combat and weapon system test procedures require an integration test event with every change to the software or system configuration to certify that the software-intensive system-of-systems is stable and functional. These integration test events are costly, require substantial coordination, and are often only conducted once every 1 to 2 years. As more systems are moving to a modular open architecture (OA), software configurations are changing with increased frequency, requiring more testing—which is expensive and time consuming. As the need for software testing increases, new testing methods are required to keep up. Automated software testing seems like a logical choice to deal with the increased demand. Many people are working on various strategies to implement automated testing in this domain. One idea which could assist in the overall effort is the use of operational profile models. The combination of automated testing (driven by an operational profile model) with other research efforts in this domain (that focus on reducing the amount of testing required during a configuration change) is key to realizing a successful software test strategy for US Navy OA weapon and combat systems (Berzins & Dailey, 2009).

An operational profile model, in terms of this study, is an *a priori* model that provides inputs to an OA software module under test. It does this by sampling from probability density functions (PDFs) that model specific inputs from the actual

environment to the software module. For this research, the operational profile model's purpose is to drive automated testing of OA software aimed at probabilistic reliability assessments supported by statistical confidence levels. Ideally, the automated testing process would utilize the operational profile model for the generation of inputs to the software under test, and analysis could be done on system outputs in an automated black-box approach. This study further focuses on determining how to most effectively develop and implement such models within the US Navy OA weapon system software domain.

Testing driven by an operational profile model is very efficient because it generally identifies failures in the order of how often they occur. This approach, therefore, rapidly reduces failure rates as testing proceeds, and the faults that create frequent failures are generally identified first (Musa, 2004). Using operational profile models for testing of OA systems is also ideal because of the standards put in place to define the architecture. The same standards are used by all software developers who are working to develop similar functioning software and to achieve successful integration within the open system. If the operational profile model is developed with the standards in mind, then the same profile model could be used to test multiple iterations or releases of similar-functioning software. The opposite also applies when testing within the OA domain. If a software module is being reused from a previous application, testing it all over again within its new domain may seem to some as unnecessary because it is being reused to avoid unnecessary development costs. However, it is necessary to retest proven software when its environment changes, and having an operational profile model to drive software testing is an efficient way to model the new environment.

Ongoing Operational Profile Research Effort

When developing an operational profile model to be used for testing OA software, a designer must define the boundaries of the software module to be tested and the interfaces between it and its environment. Such boundaries and interfaces should be definable based on OA standards for the particular type of software. The latest OA standards within the US Navy weapon system community are defined by the Navy Standards Working Group (NSWG).

The main task of this ongoing research effort is to derive a process for developing an operational profile model. Most of the effort is devoted to determining how to use historical and/or real-time source data to derive the PDFs that make up the operational profile model. One of the main technical issues designers must resolve to effectively use historical data and/or real-time data is how to choose a realistic granularity for the profile model that will also result in adequate levels of confidence in the model's accuracy. Either discrete or continuous PDFs from some family of distributions, combined with a small number of parameters that can be estimated from the data are used to make up the modeled inputs for the system under test. The available source data is finite and usually does not provide unlimited or high-levels of resolution, thus requiring some degree of approximation for the construction of the PDFs. Along with the approximations, some degree of statistical uncertainty in the accuracy of the model exists and should be calculated by the profile developer. The broader challenge is determining what methodology of calculation should be used, as well as linking the results of the calculation to confidence levels of the accuracy of the PDFs.

It is important for designers to understand the set of all operational states in which the software module functions. To achieve this understanding, designers could develop multiple profiles to replicate the different states, or designers could manipulate the distribution sampling from one prototype so it would cover all possibilities if the environment behavior is properly characterized.

Other considerations include understanding the dependencies that may exist between different inputs or between inputs and outputs. A good understanding of the relationships—along with capturing the relationships, possibly using conditional statistics—is critical to properly modeling the operational environment.

In summary, the designers' ability to develop a complete operational profile model based on limited historical source data is vital for enabling successful automated testing. This ongoing research is a necessary step in the successful evolution of software testing for US Navy OA combat systems and weapon system software programs.

There are four main goals of this ongoing research effort:

1. Determine an overall methodology for developing an operational profile model to drive automatic OA software testing.
2. Determine how to efficiently use an operational profile model in the US Navy OA weapon system software automated testing domain.
3. Determine how to calculate the reliability of the software component being tested using the operational profile model.
4. Determine how to practically derive and calculate confidence in the accuracy of PDFs that accurately represent messages coming from the actual environment for a software module under test, including existing dependencies and multiple states of operation.

A PDF can either be represented discretely or continuously. When developing a discrete PDF, designers should organize source data points in meaningful ways—like a histogram, for example, which represents the range of possible inputs for the given message or signal. Designers could then analyze and extrapolate it into a complete discrete probability distribution. This completely defined probability distribution could be used empirically, or it could be modified to reflect certain operational conditions prior to sampling during testing. When developing a continuous distribution, designers need to estimate parameters of the input distribution such as distribution type, mean, standard deviation, etc. Just like the discrete distributions described above, designers could leave the derived PDF

alone, or they could modify it to reflect a particular operating condition and make it ready to be sampled from during testing. Depending on the type of software input and available source data, one method may be more useful than the other. Both options are considered during the course of this research.

Modification of the initially calculated PDF may be desired in several cases. Designers could modify the test distribution to overestimate the mass on the tail ends of the distribution. This would ultimately test a wider range of inputs for the OA software and, in some cases, would result in a more robust system under unexpected boundary conditions. Depending on the implementation, testers often desire to develop both normal and worst-case operational profiles so that different operating scenarios can be covered. This approach ensures that the software will perform adequately not only in a normal operational mode, but also when under stress.

Confidence in the accuracy of the derived PDFs can be calculated based on the sample size and expectation in a confidence interval calculation. This type of calculation can be automatic and can be used to assess whether or not PDF is an accurate representation of the actual environment variable within some level of confidence. For discrete PDFs, testers can generate histograms for sampling based on a variable bin size instead of the typical fixed interval that allows for the same number of samples to exist in each bin, thus varying the accuracy of the model but fixing the confidence level.

Once a collection of statistically sound PDFs are developed which cover all inputs for a given software module under test, testers combine them together into an operational profile model that can generate a collection of samples based on the PDFs and other user-specified or programmed criteria—otherwise known as test cases. Certain dependencies could exist between system inputs; thus, an operational-specific program will be generated that defines distributions with the captured dependencies, as well as how sampling should occur when trying to replicate certain operational states. Lastly, there will be some user-interface

between the model and an external automated testing control source that will command the model to run in a particular state and start generating outputs. This operational program, the collection of PDFs, and the interface to the automated testing control program will make up the operational profile model.

When attempting to demonstrate this concept completely, it will also be necessary for the model to perform some automated analysis of the program output. An oracle should be made by profile developers which will flag errors in the software component's behavior during testing. This functionality is not really part of the operational profile of the system, but it is a necessary step of the automated testing process. In addition, it can ideally be implemented as part of the profile model operational program—which, in total, would have the operational profile, the user-interface, and the oracle and possibly other test-related reporting features.

For a US Naval weapon system prototype implementation, NPS researchers will identify specific software modules, along with their interfaces, using input from the NSWG and other system requirements specification sources. The interfaces will be characterized based on a limited amount of real-world data in all modes of operation, and a concept-demonstration prototype operational profile model will be generated by NPS researchers including the predicted confidence levels in the profile. The accuracy of the prototype will be checked by running multiple test cases and comparing the test results to a complete set of actual, real-world manual test results. The goal of this accuracy checking is to determine the minimum acceptable confidence level in the model. This level enables the testing to be effectively and reliably conducted. The designers will standardize all methods, calculations, and algorithms used to create the PDFs based on applicability to particular message or signal types. As part of the operational profile model, the designer will develop and combine a user interface, output checking oracle, and other necessary functionality needed to conduct automatic testing. Then, the designer will use the lessons learned from the implementation of a prototype, combined with other analyses, to

put together an optimized methodology for acquiring operational profiles within the US Navy weapon system domain.

Summary of Prior Relevant Work

Testing in an Open Environment

One of OAs benefits is to reduce the risk of bugs by having defined standards for interfacing components. Testing in an OA environment requires a modified approach when compared to traditional software testing. Testing expenses required for OA systems include specialized testing tools, test preparation and management, unit, functional, integration, regression and user acceptance testing. Benefits provided by OA in the testing domain include the following:

Open Standards can reduce the extent of testing at a component unit level, if open source or vendor implementations are used. Integration testing may be reduced if other components of the system, especially 3rd party components, have been previously been designed and tested to be compatible with the relevant Open Standards. Over time, the use of OSR-conformant components can also reduce the cost of regression testing.

Modularity best supports unit testing when well-defined interfaces encompass the full functionality of the component, and testing can be designed before development is completed. Low dependency metrics in a system reduce the number of integration tests that are needed because there are fewer points of variability at the system level.

Extensibility can directly support testing by making it easy to instrument the component and the system with no impact on the system functionality. Functional extensions to a component are also well-defined; thus, extending unit and systems tests should be relatively straightforward. Even regression testing is unlikely to require extensive modification.

Interoperability requires additional testing when components are created. However, OA returns the investment by facilitating integration in subsequent use of components—thus leveraging prior testing efforts associated with component design and development.

These benefits are nullified when the process is constrained by traditional US Navy combat system and weapon system testing approaches. Current practices require a complete system integration test event to be conducted with every configuration change, often occurring once every two years. This approach was adopted long ago because of the safety and integration concerns that come with software-intensive weapon and combat systems. Complete system integration testing is costly under traditional system architectures and often requires retesting of proven system elements that did not change from one baseline to the next. Testing in this way increased the time it took to field new technologies or capabilities due to the cost-driven, bi-annual schedule. This approach will be even more inefficient if used on OA-based systems. The modularity and scalability benefits of OA allow for quick configuration changes to update and tailor capabilities to mission needs. These benefits will be nullified if the current US Navy test strategy is used, as the cost would grow exponentially with constant complete integration testing; likewise, the needed configuration changes would not be made when needed. Because of this, Test & Evaluation (T&E) could be the Achilles heel for US Navy OA initiatives. New technology, processes and policies are needed to safely reduce this effort and free resources for testing new functionality (Berzins & Dailey, 2009). These new techniques should be capable of determining when it is safe not to retest a component when the overall system configuration changes.

Determining when It Is Safe Not to Retest a Component following Changes in Configuration

In his 2008 text, Berzins explains that if the requirements related to a component have not changed, and the behavior of the components has not changed, then retesting may not be necessary. The range of conditions under which

a component is expected to provide its operational capabilities is particularly relevant to testing and re-testing. The rest of this section addresses how to statically and dynamically check that the behavior of a component has not changed, assuming for the moment that its requirements and range of operating conditions have not changed.

A type of dependency analysis known as program slicing can be used to identify parts of the unchanged code that have the same behavior in the new release as in previous one (Weisser, 1984). A program slice at a given observation point is a self-contained subset of the code in the sense that it contains all of the code that can affect the behavior visible at the observation point. If two different programs have the same slice for a given observation point, then they have the same visible behavior at that point. Consequently, if the new release has the same slice as the old release for a given service, then that service will have exactly the same behavior in the new release as in the old one and, consequently, may not need regression testing (Gallagher, 1991). This fact is useful because program slices can be computed for software systems on practical (large) scales. The testing-reduction method that follows from this observation is to compute the slice of each service with respect to the new release and the old release, and retest only the services for which these slices differ.

In the context of technology-advancement upgrades, the test-reduction method described above must be augmented with focused, automated testing to produce a substantial reduction in retesting. Technology upgrades usually run on a new version of the operating system. If the source code of the operating system is proprietary and, hence, not available for static analysis (commonly true, except for open source systems such as LINUX), then the only safe assumption is that all operating system services have been impacted by the upgrade to the new version. Thus, any service whose slice includes a dependency on a system call would be potentially impacted and would have to be retested, based on the simple slicing approach outlined in the previous paragraph. This is likely to include most of the

application-level modules—severely limiting the amount of savings that can be obtained using slicing alone.

Automated testing, however, can enable larger reductions in retesting if it is focused on the middleware interface to the underlying operating system services. Fortunately, interviews with representative stakeholders confirmed that most Navy systems with open architectures are designed around a middleware interface that encapsulates all operating system calls. Such middleware interfaces are also prevalent in other DoD systems, including the US Army's Future Combat System (FCS). Application architectures are typically designed in this way to ease the job of porting the application to new operating systems, whether they are new releases of the same product or different products. Consequently, each new release of the operating system and the neighboring middleware layer are both designed to preserve the observable behavior of the previously available system calls if at all possible—even if the details of the implementation may vary from one release to the next. If we know that the observable behavior of a given system call is the same in the old and the new version of the operating system, then we can truncate the slice at the middleware layer for that call, and conclude that the behavior of an application service is unaffected by the OS change if its abbreviated slices in the two versions are the same. The proposed enhancement to dependency analysis using program slicing is to check this property for each system call in the middleware layer via automated testing.

This same strategy can also be applied at higher levels of middleware. For example, for the common case of applications that have been developed for the Java or .NET platforms, the interface to operating system resources is the framework runtime, such as the interface to the Java foundation classes. One related viable strategy for reducing testing of unchanged application code is bounding slicing by the interfaces at this level and using automated testing to show equivalent behaviors of the two releases at these interfaces. A related, common pattern of changes that should not affect behavior involves framework evolution, in

which applications are recoded to migrate from “deprecated” (soon to become obsolete) interfaces to the corresponding new versions of the interfaces. Although such changes produce differences in the code, they are intended to preserve behavior, and should be amenable to the automated test strategy. Thus, modules, one level above the framework runtime interfaces, are additional candidates for automated testing and slicing cutoff boundaries.

Automated testing is attractive in these contexts because a simple, reliable implementation of a “test oracle” is possible for the encapsulated operating system’s services. A “test oracle” is a process for automatically determining which test outputs pass and which ones fail. The “unchanged behavior” condition can be easily checked by software for a given set of input data. This is possible since both the old and the new versions of the operating system are available for testing, and test scaffolding software can compare the results of the two versions via equality tests. The existence of such a “test oracle” implies that the OS middleware testing process can be completely automated—enabling economic and practical testing with statistically significant sample sizes that support very high confidence levels, or, in some cases, even exhaustive testing of the operating system interfaces that supports definite conclusions. The proposed automated testing process would, thus, classify all of the services in the middleware interface to the operating system into two groups: those whose behavior is the same in both versions of the operating system (the preserved services), and those whose behavior differs in the two versions (the modified services). We expect the first group to be much larger than the second group.

In such cases, we can cut off slices at the system calls to the preserved services, and conclude that unmodified application components do not have to be retested unless their slices differ or contain system calls that invoke one of the modified services. The operating system interface always needs to be thoroughly retested, but this can be done by the affordable automated process described above.

The above analysis depends on the assumption that we can accept a statistical inference about the unchanged behavior of the operating system's calls, if the statistical confidence level is high enough. Since most military decisions must be based on information that has the same degree of uncertainty, we do not expect lack of certainty to be a problem in principle. We, therefore, consider how to determine what level of confidence would be "high enough," and how many test cases are necessary to reach that level of confidence.

We start with a consideration that should be meaningful to the stakeholders: if the mean time between observations of a behavioral difference in a given operating system's service is substantially (k times) longer than a mission, it is acceptable to ignore risks due to the possibility of such an unexpected difference. The meaning of "substantially" can be expressed as a numerical safety factor k that can be understood and set by system stakeholders based on their tolerance for risk.

Next, we measure the mean number of executions per mission e_s for each service s in the middleware interface to the operating system. The objective of the automated testing for each service s is to ensure the mean number of executions between observed differences in the behavior of service s is at least N_s, where

$$N_s = k\, e_s$$

Theorem 4.3 from Howden (1987) can then be used to determine the required number of test cases T_s for each service:

$$T_s = N_s \log_2 N_s$$

If we run T_s test cases that are independently drawn from the probability distribution characterizing the mission (called the operational profile), the theorem will enable us to conclude that the mean number of executions is at least N_s with a statistical confidence level $(1 - 1/N_s)$; however, this is contingent upon none of the T_s test cases showing any differences in the behavior of the services under the new version of the operating system from those in the previously released version.

The rationale for this choice of confidence level is that it makes the probability of making a false positive conclusion no more than the acceptable frequency of behavioral differences—thus scaling the risk due to random sampling errors to match the specified maximum acceptable failure rate. False positive conclusions correspond to cases in which the frequency of behavioral differences in the new release of the operating system service in question is actually greater than the target bound ($1/N_s$), but the automated testing procedure failed to observe a difference due to random sampling fluctuations that caused conforming results to appear purely by chance. The test set size T_s has been chosen to make the probability of such a chance observation at most ($1/N_s$).

Thorough statistical testing of the operating system interfaces has the additional benefit of increasing the confidence that hardware differences (and possibly different versions of the compilers, linkers and loaders) have not affected the behavior of the applications built using these services.

Cost of Automated Testing

There are several different kinds of automated testing. The most common kind is semi-automated testing. This approach automates the type of testing currently performed manually. It is commonly the first kind of automated testing implemented in an organization because it does not involve any process changes. In this type of approach, the test cases are still developed individually by test engineers, but the test cases are run automatically, and the results are classified into pass or fail categories automatically—often by comparison to previously captured test outputs that were originally individually examined and categorized by people. In this approach, execution and categorization of test results is automated, but the choice of test cases and the initial pass/fail decisions are not. This approach saves appreciable time and effort relative to a completely manual approach, but the human effort required is still proportional to the number of test cases.

Another approach particularly relevant in our context is automated statistical testing. In this approach, the choice of test cases and the initial pass/fail decisions are automated, as well. This makes a great difference because the human effort involved does not increase with the number of test cases to be executed. This enables economical application of the very large test sets needed to achieve the coverage required to support high levels of statistical confidence in the dependability of the software. The high levels of statistical confidence are needed to avoid testing for other unchanged code based on indirect evidence that the behavior of the underlying services on which the unchanged code depends has not changed.

The context identified in the previous section is well suited for automated statistical testing, because the choice of test cases and the initial pass/fail decisions are easily automated in that context: the first can be done by random sampling from the operational profile, and the second by comparison of the results produced by the previous release of the software to those produced by the new release.

The variation in the number of the test cases T_s required as a function of the acceptable risk of false positive conclusions ($1/N_s$) is illustrated in Table 1.

Ns	***C***	***Ts***
10^3	0.999	1.0×10^4
10^4	0.9999	1.3×10^5
10^5	0.99999	1.7×10^6
10^6	0.999999	2.0×10^7
10^7	0.9999999	2.3×10^8
10^8	0.99999999	2.7×10^9
10^9	0.999999999	3.0×10^{10}

Table 1. Number of Test Cases Required for Different Levels of Risk Tolerance

Ns: Desired lower bound on mean number of executions between differences

C: Statistical confidence level

Ts: Number of independent random test cases required

Figure 1 shows how the cost characteristics of the proposed automated testing approach compare to the costs of manual testing. The cost curves are close

to straight lines; the fixed costs of automated testing are larger than for manual testing, and the marginal cost of adding another test case is much smaller for automated testing than for manual testing.

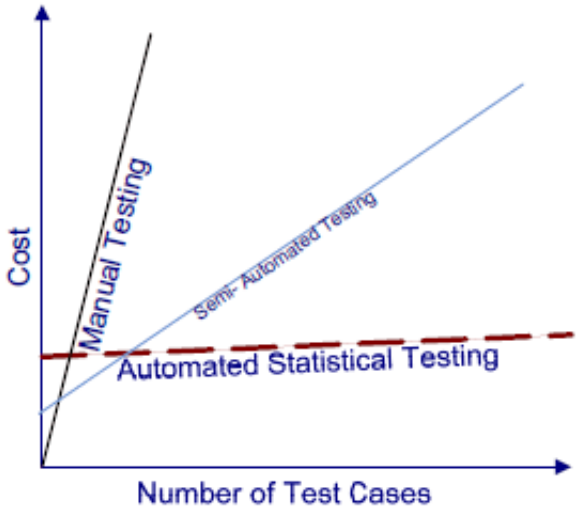

Figure 1. Testing Cost Characteristics

In order to determine the crossover points, we must obtain and analyze experimental data. This process is still ongoing. However, we expect automated testing to be affordable—even for the very large numbers of test cases needed for high confidence in stability of OS services across different releases. We also expect manual and semi-automated approaches not to be affordable when we test to high confidence.

Regarding the time and other resources to perform the proposed automated statistical testing, we can note the following:

1. It typically takes a small amount of time to perform a single system call.
2. Testing using independent, random samples is easily parallelizable and could be effectively spread over large numbers of processors using well-established techniques—such as Google's Map Reduce programming model (Lammel, 2008)—if very high confidence levels are needed.

3. Behavior of operating system calls can be tested independently of other shipboard systems. Such testing does not require interactions with human operators.

Since the testing process is completely automated, the variable cost of these tests is due to computing time and hardware, but not to human effort. The benefit of the automated statistical test approach described here is that there are no variable costs for labor. Since computing resources are currently inexpensive and steadily getting cheaper, even the relatively large numbers of test cases needed for high confidence are likely to be affordable.

This approach does involve some fixed costs for human effort that may be higher than in less-disciplined manual approaches. These costs are due to the need for the following activities:

1. Measurement of operational profiles—i.e., the frequency distributions of operating system calls and their associated input parameters. Instrumented versions of the software can be used during exercises to collect measurements of the operational profiles, or, if the computational overhead of doing this is acceptable, measurements could also be collected during actual operations. Ongoing research is developing systematic, computer-assisted methods for this process.

2. Coding more sophisticated test-driver software that includes code for generating random samples from the measured operational profiles, code that implements test oracles as described above, and code that keeps track of testing statistics and reports them.

Testing with Operational Profiles

An operational profile has been defined as the set of input events that the software will receive during execution, along with the probability that the events will occur and the set of all input events generated by external hardware and software systems that the software is expected to interact with during execution (Voas, 2000).

Taking that definition a bit further with respect to probabilities, an operational profile model, in terms of this study, is a statistical model—developed with some previous knowledge of the population—that provides inputs to an OA software

system module under test based on a sampling from PDFs (which represent the interfaces from the actual environment to the software module). The operational profile model's purpose is to drive automated testing of OA software aimed at probabilistic reliability assessments supported by statistical confidence levels. Ideally, the automated testing process utilizes the operational profile model for the generation of inputs to the software under test and possibly analysis of the system outputs.

Accurate estimates of operational profiles, preferably based on actual measurements, are necessary because in all practical cases, *the reliability of a software system is meaningless without firm knowledge of the operational profile*. This claim is based on the hypothesis that all real systems have at least one input value *x* for which they perform correctly, and at least one other input value *y* for which they do not. If we know *x* and *y*, we can construct a spectrum of possible operational profiles for which the reliability of the same system ranges from 0 to 1 and attains every value in between.

The above line of reasoning shows that the only systems whose reliabilities do not depend on the operational environment are those that fail for all possible inputs (reliability uniformly 0, not interesting), and those that operate correctly for all possible inputs (reliability uniformly 1, not attainable in practice for large systems).

For all other systems, the reliability is determined by the operational profile and can vary widely for different operational contexts. This has serious implications for component reuse, which is a cornerstone of the Navy OA initiative.

Operational profiles have been used by the testing research community for many years and have been applied in many contexts. For example, they have been measured and used to assess the reliability of telephone-switching software.

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 a 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 (Musa, 2004).

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 can, therefore, be a key part of a test automation strategy (Binder, 2004).

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 are consistent with the UML, and include domain and probability information necessary for automated testing (Binder, 1999).

Operational Profile Model Development Methodology

Process Overview

The methodology defined in this section defines how to develop an operational profile model used to test a US Navy OA weapons or combat systems software component under test. This approach is broken down into several areas—including identifying the component, understanding its operation and interfaces, characterizing the inputs statistically, characterizing the outputs (for input dependency analysis and for understanding of how to detect bugs), deriving testing profiles, creating the operational profile user-interface and core functionality, and lastly, developing the supporting documentation.

This approach was developed by NPS as part of the ongoing Operational Profile Model research effort and is in the process of being validated via the development of a concept-demonstration prototype model. There it will be critiqued and optimized as the study continues. Based on the results of the prototype model development and component testing, we will present a refined approach to acquire an operational profile model for testing of US Navy OA weapons system software at the completion of the ongoing research study.

Steps to Develop and Operational Profile Model

1. Identify Component Boundaries
    a. Identify Component
    b. Identify Component's Environment
2. Understand Component Operation
    a. Identify Inputs
    b. Identify Outputs
    c. Identify Controls
    d. Identify Mechanisms

e. Understand Component's Operational Requirements
f. Understand Component's Technical Requirements
g. Identify States of Operation

3. Define Component Interfaces
    a. Define Inputs
    b. Define Input Formats
    c. Define Outputs
    d. Define Output Formats
4. Characterize Inputs (For Operational Profile PDF Derivation)
    a. Gather Available Real-world/Historical Input Data
    b. Organize Real-world/Historical Input Data
    c. Analyze Real-world/Historical Input Data for Each Historical Environment
        i. Relate Operational Data to Software Inputs
        ii. Assess Software Input Dependencies
            1. Assess Dependencies between Inputs
            2. Assess Dependencies with Time
            3. Assess Dependencies between Inputs and Outputs
            4. Assess Dependencies with Operational State
        iii. Derive Operational Probabilistic Characterization for Each Input
            1. Utilize Kernel Density Estimation (KDE) for Continuous PDFs
            2. Utilize Variable-width Histogram Bin Approach for Discrete PDFs
            3. Utilize Bayesian Approach for Probabilistic Dependencies
            4. Utilize COTS Analysis Tools for Calculations & Analysis
                a. S-Plus/R
                b. MATLAB
                c. Oracle Crystal Ball
        iv. Calculate/Determine How to Maximize Confidence in Results

1. Calculate Confidence Based on Sample Size
2. Utilize Variable-width Histogram Bin Approach to Fix Confidence

v. Repeat for Each Historical Environment

d. Assess Gaps in Real-world/Historical Input Data
e. Determine Historical and Current Operational Environment Differences
    i. Analyze Current Environment Changes
    ii. Relate Environment Differences to Affected Input Changes
f. Modify Historical Probabilistic Characterizations to Reflect Current Environment
    i. Analyze Profiles of Historical Environments
        1. Relate Calculations to Historical Environments
        2. Relate Confidence Level to Available Data
    ii. Predict Necessary Changes to Inputs for the Current Environment
        1. Utilize Bayesian Approach for Dependency Analysis
        2. Minimize Use of Subjective Analysis (if possible)
    iii. Acquire Relevant Data for Current Environment Analysis (if possible)
    iv. Analyze Relevant New Environment Data to Compare with Predicted Changes (if possible)
    v. Assess Software Input Dependencies
    vi. Derive Operational Probabilistic Characterization for Each Current Environment Input
    vii. Calculate/Determine How to Maximize Confidence in Current Environment

5. Characterize Outputs (for Oracle Development)
    a. Define Component Operational Failures
        i. Define Failures by Event

ii. Define by Defining Success, and Relate Any Other Behavior as Failure

b. Relate Operational Failures to Specific Observable Output Behavior

c. Determine Necessary Conditions/Analysis Required to Detect Output Failures

6. Identify Testing Profiles
    a. Derive Operational Testing Profile(s) for Operational Test & Evaluation (OT&E)
    b. Derive Stress-testing Profile(s) for Developmental Test & Evaluation (DT&E)
7. Develop Operational Profile Model User Interface
    a. Develop Component Loading Interface
    b. Develop Operational Profile Loading Interface
        i. Include Fields Used to Load Desired Probability Distributions for Each Input
        ii. Include Fields for Selecting Desired Number of Test Cases to be Generated
        iii. Include Fields for Starting/Stopping Test
    c. Develop Output Analyzer
        i. Include Fields Used for Defining Conditions of Failure
        ii. Include Fields for Loading Failure Analysis Scripts
    d. Develop Analytical Interface
        i. Include Fields Identifying Current Progress of Testing
            1. Number/Percentage of Tests Complete
            2. Number/Percentage of Failed Runs
            3. Analysis of Failed Runs (Real Time or Upon Completion)
            4. Calculation of Confidence in Software Reliability
8. Develop Operational Profile Model Core Functionality
    a. Develop/Code Operational Profile Model Input Generator

b. Develop/Code Operational Profile Model Interfaces to User & Component
c. Develop/Code Operational Profile Output Analyzer Oracle & Data Logger

9. Develop Operational Profile Supporting Documentation

THIS PAGE INTENTIONALLY LEFT BLANK

Recommendations for Future Work

To realize the benefits of automated testing driven by an operational profile model, future work will need to be done in support of Operational Profile Models and in other related domains. The next step in this domain will be focused on the implementation of an operational profile model concept demonstration prototype, which will be used to validate and refine the proposed development methodology outlined above. A way to validate both the accuracy of the operational profile model and the reliability of the software being tested needs to be developed and refined. Also, if specific environment inputs exist that may lead to failures with severe consequences and that cannot be confidently estimated using conventional techniques, it may be necessary to develop new techniques for these specific cases. Lastly, research into the possibility of using operational profile models in an "open-loop" testing mode—instead of a black-box mode—would drastically reduce automated testing computing time for the testing of components with multiple states of operation. This open-loop approach would make it possible for any state to be set at the same time as a specific input—without the need of previous input files to get the component to the state of operation.

THIS PAGE INTENTIONALLY LEFT BLANK

List of References

Berzins, V. (2008, April 8). *Which unchanged components to retest after a technology upgrade*. In Proceedings of the 5th Annual Acquisition Research Symposium (pp 142-153). Monterey, CA: Naval Postgraduate School.

Berzins, V., & Dailey, P. (2009, May 14). How to check if it is safe not to retest a component. In *Proceedings of the 6th Annual Acquisition Research Symposium* (pp. 189-200). Monterey, CA: Naval Postgraduate School.

Binder, R.V. (1999). Testing object-oriented systems: Models, patterns, and tools. Boston, MA: Addison-Wesley.

Binder, R.V. (2004, December). mVerify Corporation. Automated testing with an operational profile. *DoD Software Tech News, 8*(1).

Gallagher, K. (1991, August). Using program slicing in software maintenance. *IEEE Transactions on Software Engineering, 17*(8), 751-760.

Howden, W. (1987). *Functional program testing and analysis*. New York: McGraw-Hill.

Lammel, R. (2008, January). Google's map reduce programming model—Revisited. *Science of Computer Programming, 70*(1), 1-30.

Musa, J.D. (2004). Software reliability engineering: More reliable software faster and cheaper. Section 2.1: AuthorHouse.

Voas, J. (2000, March/April). Will the real operational profile please stand up? *IEEE Software, 0740-7459*.

Weisser, M. (1984, July). Program slicing. *IEEE Transactions of Software Engineering, SE-10*(4), 352-357.

THIS PAGE INTENTIONALLY LEFT BLANK

2003 - 2009 Sponsored Research Topics

Acquisition Management

- Acquiring Combat Capability via Public-Private Partnerships (PPPs)
- BCA: Contractor vs. Organic Growth
- Defense Industry Consolidation
- EU-US Defense Industrial Relationships
- Knowledge Value Added (KVA) + Real Options (RO) Applied to Shipyard Planning Processes
- Managing the Services Supply Chain
- MOSA Contracting Implications
- Portfolio Optimization via KVA + RO
- Private Military Sector
- Software Requirements for OA
- Spiral Development
- Strategy for Defense Acquisition Research
- The Software, Hardware Asset Reuse Enterprise (SHARE) repository

Contract Management

- Commodity Sourcing Strategies
- Contracting Government Procurement Functions
- Contractors in 21st-century Combat Zone
- Joint Contingency Contracting
- Model for Optimizing Contingency Contracting, Planning and Execution
- Navy Contract Writing Guide
- Past Performance in Source Selection
- Strategic Contingency Contracting
- Transforming DoD Contract Closeout
- USAF Energy Savings Performance Contracts
- USAF IT Commodity Council
- USMC Contingency Contracting

Financial Management

- Acquisitions via Leasing: MPS case
- Budget Scoring
- Budgeting for Capabilities-based Planning
- Capital Budgeting for the DoD
- Energy Saving Contracts/DoD Mobile Assets
- Financing DoD Budget via PPPs
- Lessons from Private Sector Capital Budgeting for DoD Acquisition Budgeting Reform
- PPPs and Government Financing
- ROI of Information Warfare Systems
- Special Termination Liability in MDAPs
- Strategic Sourcing
- Transaction Cost Economics (TCE) to Improve Cost Estimates

Human Resources

- Indefinite Reenlistment
- Individual Augmentation
- Learning Management Systems
- Moral Conduct Waivers and First-tem Attrition
- Retention
- The Navy's Selective Reenlistment Bonus (SRB) Management System
- Tuition Assistance

Logistics Management

- Analysis of LAV Depot Maintenance
- Army LOG MOD
- ASDS Product Support Analysis
- Cold-chain Logistics
- Contractors Supporting Military Operations
- Diffusion/Variability on Vendor Performance Evaluation
- Evolutionary Acquisition
- Lean Six Sigma to Reduce Costs and Improve Readiness

- Naval Aviation Maintenance and Process Improvement (2)
- Optimizing CIWS Lifecycle Support (LCS)
- Outsourcing the Pearl Harbor MK-48 Intermediate Maintenance Activity
- Pallet Management System
- PBL (4)
- Privatization-NOSL/NAWCI
- RFID (6)
- Risk Analysis for Performance-based Logistics
- R-TOC AEGIS Microwave Power Tubes
- Sense-and-Respond Logistics Network
- Strategic Sourcing

Program Management

- Building Collaborative Capacity
- Business Process Reengineering (BPR) for LCS Mission Module Acquisition
- Collaborative IT Tools Leveraging Competence
- Contractor vs. Organic Support
- Knowledge, Responsibilities and Decision Rights in MDAPs
- KVA Applied to AEGIS and SSDS
- Managing the Service Supply Chain
- Measuring Uncertainty in Earned Value
- Organizational Modeling and Simulation
- Public-Private Partnership
- Terminating Your Own Program
- Utilizing Collaborative and Three-dimensional Imaging Technology

A complete listing and electronic copies of published research are available on our website: www.acquisitionresearch.org

THIS PAGE INTENTIONALLY LEFT BLANK

ACQUISITION RESEARCH PROGRAM
GRADUATE SCHOOL OF BUSINESS & PUBLIC POLICY
NAVAL POSTGRADUATE SCHOOL
555 DYER ROAD, INGERSOLL HALL
MONTEREY, CALIFORNIA 93943

www.acquisitionresearch.org