2009-07-01

Reducing the Cost of Risk-based Testing: Management of Testing Options to Manage Risk in Test and Evaluation

Karl D. Pfeiffer

NPS-AM-09-114

ACQUISITION RESEARCH SPONSORED REPORT SERIES

Reducing the Cost of Risk-based Testing: Management of Testing Options to Manage Risk in Test and Evaluation

15 July 2009

by

Dr. Karl D. Pfeiffer, Assistant Professor

Dr. Valery A. Kanevsky, Research Professor, and

Dr. Thomas J. Housel, Professor

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

In the acquisition or management of complex systems, testing is the means by which we trade budget or schedule for information about the likelihood our system will work correctly under operational load. Branch paths in hardware and software increase as a function of the number of components and interconnections, leading to exponential growth in the number of test cases required for exhaustive examination, or perfect knowledge, of a complex system. In practice, the typical cost for testing in schedule or in budget means that only a small fraction of these paths are investigated. In this work, we develop an abstract model to describe system testing and the information return (or reduction in risk) for the attendant cost in time and mondey. This model is supported by a mathematical analysis suitable for Monte Carlo simulation. The long-term goal of this modeling work is to construct a decision-support tool for the Navy Program Executive Office Integrated Warfare Systems (PEO IWS) offering quantitative information about cost versus diagnostic certainty in system testing.

Keywords: diagnostic testing, regression testing, automated testing, Monte Carlo simulation, sequential Bayesian inference

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgements

The authors would like to thank Mr Mark Wessman for his continued and invaluable assistance with this project. His insights into Fleet operations and the Integrated Warfare enterprise have helped to ground this work in operationally relevant issues, ensuring that we are crafting tools for the most pressing problems.

THIS PAGE INTENTIONALLY LEFT BLANK

About the Authors

Karl D. Pfeiffer is an Assistant Professor of Information Sciences at the Naval Postgraduate School and an active-duty Air Force officer. His current research interests include decision-making under uncertainty, particularly with regard to command and control (C2) systems; stochastic modeling of environmental impacts to weapons and communication systems; and probability modeling and numerical simulation in support of search, identification and pattern recognition applications (e.g., complex system testing, allocation of effort for reconnaissance, etc.).

Karl D. Pfeiffer
Graduate School of Operational and Information Sciences
Naval Postgraduate School
Monterey, CA 93943-5000
Tel: 831-656-3635
Fax: (831) 656-3649
E-mail: kdpfeiff@nps.edu

Valery A. Kanevsky is a Research Professor of Information Sciences at the Naval Postgraduate School. His research interests include probabilistic pattern recognition; inference from randomly distributed inaccurate measurements, with application to mobile communication; patterns and image recognition in biometrics; computational biology algorithms for microarray data analysis; Kolmogorov complexity, with application to value allocation for processes without saleable output; and Monte Carlo methods for branching processes and simulation of random variables with arbitrary distribution functions. Kanevsky's most current work is focused on statistical inference about the state of a system based on distributed binary testing. Another area of interest is in the so-called needle-in-a-haystack problem: searching for multiple dependencies in activities within public communication networks as predictors of external events of significance (e.g., terrorist activities, stock market anomalies).

Thomas J. Housel is a Professor of Information Sciences at the Naval Postgraduate School. He specializes in valuing intellectual capital, knowledge management, telecommunications, information technology, value-based business process re-engineering, and knowledge value measurement in profit and non-profit organizations. His current research focuses on the use of knowledge-value added (KVA) and real options models in identifying, valuing, maintaining, and exercising options in military decision-making. His work on measuring the value of intellectual capital has been featured in a *Fortune* cover story (October 3, 1994) and *Investor's Business Daily*, numerous books, professional periodicals, and academic journals (most recently in the *Journal of Intellectual Capital*, 2005). His latest books include: *Measuring and Managing Knowledge* and *Global Telecommunications Revolution: The Business Perspective* with McGraw-Hill (both in 2001).

NPS-AM-09-114

ACQUISITION RESEARCH SPONSORED REPORT SERIES

Reducing the Cost of Risk-based Testing: Management of Testing Options to Manage Risk in Test and Evaluation

15 July 2009

by

Dr. Karl D. Pfeiffer, Assistant Professor

Dr. Valery A. Kanevsky, Research Professor, and

Dr. Thomas J. Housel, Professor

Graduate School of Operational & Information Sciences

Naval Postgraduate School

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

1. Overview

There were three primary goals for this research:

1. Complete an analysis of current testing strategies within the Integrated Warfare System (IWS) program offices for balancing risk with cost.
2. Develop new testing protocols that include risk parameters that will reduce the cycle-time and cost for IWS system tests.
3. Embed these algorithms in a demonstration-level prototype decision-support system (DSS) to provide a simple tool for the IWS leadership to use in developing a useful requirements analysis.

In the course of this research, we found that some of the preliminary approaches we presumed would help address these basic goals did not pan out. For example, the use of a real options/integrated risk management framework did not provide the level of detail and types of algorithms required to produce a practical, reliable and theoretically defensible approach to the problem. Similarly, a review of general decision support system tools did not provide a ready candidate to address this highly complex problem. These initial explorations led us to develop a truly innovative approach to address the goals of this study.

Because our unique approach required a deep level of abstraction, we needed access to subject-matter experts (SMEs) in the integrated warfare system community to ensure the fidelity and usability of the final model. This access and the required data concerning component structure and test protocols proved to be an enormous challenge, greatly extending our development time. The incremental nature of this work in a model-test-model approach required real-world data for evaluation before moving much beyond the prototype spiral. Although lack of access to SMEs slowed down progress beyond this spiral, we used this time to improve the mathematical rigor and model fundamentals within this work.

In early stages of our review of the problem space, we believed that real-options portfolio management would provide a useful framework for evaluating the trade-offs

between system testing and cost. Real-options portfolio management allows for the comparisons of various options—in this case, testing options—in terms of the returns on investment such options would provide. After considerable debate among the researchers, we decided that while this approach initially seemed promising, it was not specific enough to enable selection of the most promising testing methods—methods that would reduce the attendant risks of neglecting problems in the systems.

There were two features of the distributed test environment (DTE) that made application of the real-options (RO) portfolio optimization approach problematic:

1. RO-based analysis assumes that choice of an option determines the state of a system and, therefore, allows assessment of the risk/loss/benefit of selected options. In the DTE, it was not possible to accurately select an option because the test(s) generally would not identify faulty entities and would only change posterior probabilities of the system entity's states. The difference, therefore, was irremovable uncertainty in the DTE.

The second issue was purely computational:

2. RO required, either explicitly or implicitly, that the number of options be tractable so all relevant risk/value/cost could be evaluated and compared for all options. In the DTE, the number of options is very large, typically many billions. So, heuristics were needed to eliminate most of them and to prove that the remaining ones contained options sufficiently close to optimal.

For these two reasons, the focus was on developing algorithms for option selection rather than for option analysis (as was the case in RO).

We reviewed a variety of existing decision support system (DSS) tools in hopes of finding one that would address the needs of the problem the way we had structured it after some preliminary research. We assumed we would have access to large databases of historical test results. As such, we assumed that a variety of DSS tools might be useful in categorizing and refining our test-versus-risk analysis.

Decision support systems can be categorized as data-driven, model-driven, knowledge-driven and collaboration-driven (Power, 2004). Key technologies in data-driven decision support include data warehouses, on-line application processing (OLAP), and data mining. Data warehouses are databases designed to support managerial decision-making by integrating data from multiple legacy databases. OLAP tools allow users to examine the warehouse data in a cross-tabs format, which allows "drill-down" and "roll-up" operations across user-specified dimensions (Dolk, 2000). Data mining is the application of statistical techniques, such as decision trees and neural networks, to warehouse data to identify spatio-temporal patterns in that data.

Model-driven DSS technologies encompass a wide range of analytical models from operations research and management science disciplines (Kottemann & Dolk, 1993). These include optimization, regression, decision analysis (e.g., analytical hierarchy process), and various forms of simulation—such as Monte Carlo risk analysis, discrete event simulation, system dynamics and agent-based simulation. Prescriptive models such as optimization and decision analysis tell us "what should be done"; descriptive models such as Monte Carlo tell us "what is"; predictive models such as multiple regression tell us "what will be," and constructive models such as agent-based simulation tell us "what could be."

Knowledge-driven decision technologies support the flow of knowledge within and across organizations. One example of a knowledge flow technology is the expert system, which captures rules from one or more experts in a specific domain (e.g., acquisition procedures) in a way that lets the non-expert benefit from this knowledge. Another example is the use of computational organizations for testing the relative performance of different organizational designs (Nissen & Buettner, 2004).

We assumed that we would be able to use these various DSS tool suites to develop appealing and practical user interfaces for analyzing the test-risk-based data. However, it became apparent after an extensive review that such tool suites were not aligned with the unique requirements of the problem and, therefore, could not be used without the aid of the rudimentary tool created in this preliminary study (see Appendix A.

for the coding for the preliminary test-risk DSS tool developed for this research). It is quite likely that some of the tool suites reviewed may prove useful in the next steps of this research as historical test-risk data becomes available.

In this research, we have established the ground work for a decision-support tool for the Navy Program Executive Office Integrated Warfare Systems (PEO IWS). This tool can provide quantitative information about trade-offs among cost, risk, and the degree of system testing conducted. Initially, we sought to answer the question: given a failure in an operational system, what is the best test-risk-cost strategy to locate the failed unit of replacement? Further development of this model will investigate the question: given an engineering upgrade to a module, how much regression testing must we accomplish on the system for a given level of risk?

The scientific contribution of this work lies in a novel, information-driven approach to testing. Having characterized our system in terms of probabilities of failure of individual components, we can assess at any time the information entropy associated with that knowledge and assess the change in entropy possible by applying particular tests from our diagnostic inventory. We can then more readily assess quantitatively the information returned for the cost incurred by a test or battery of tests.

We expect the practical results of this work will be useful throughout the system lifecycle, from acquisition to fielding and maintenance. The decision-support prototype tools delivered should, for example, yield significant insight for decision-makers designing test suites for new weapons systems and improving the use of existing suites in current systems, such as the AEGIS combat system. This work should also be useful for optimizing decisions within the corrective maintenance (courses of action) module within the condition-based maintenance (CBM) and distance support (DS) systems for the Surface Warfare Enterprise.

2. Background

Testing of complex systems is a fundamentally difficult task, whether locating faults (diagnostic testing) or implementing upgrades (regression testing). The number of branch paths through the system typically grows in proportion to the number of components and interconnections, leading to near-exponential growth in test cases for an exhaustive examination. This study examines optimal system testing using classic fault diagnosis scenarios as the basis from which to develop a mathematical model flexible enough to extend to regression testing cases.

Mathematical models of component and system reliability have roots in the work of von Neumann (1952) and Moore and Shannon (1956a; 1956b), as well as the seminal text by Barlow and Proschan (1965). The focus of these early works is generally on assessing the overall system reliability—particularly with regard to the economics of preventative vice reactive maintenance (see, for example, Bovaird, 1961). In the present work, the focus is on efficiently identifying a defective-by-design or failed component in a complex system.

This fault diagnosis is sometimes referred to as the test-sequencing problem and has also been well studied (see, for example, Sobel & Groll, 1966; Garey, 1972; Fishman, 1990; Barford, Kanevsky & Kamas, 2004). In general, these investigators start with a system in a known failed state with the goal of finding the most cost-effective sequence of diagnostics to locate the failed component (or components) under a given set of assumptions.

In contrast to fault diagnosis, the general case of regression testing appears to have received less attention in the open literature, with more specific cases examined in the realm of software engineering (see, for example, Weyuker, 1998; White, 1992; Tsai, 2001; Mao & Lu, 2005; Leung, 1991; Rothermel, 2001). These studies typically start with a fully functioning system undergoing component

modification or upgrade, with the task of establishing that component modifications have not introduced new defects into the system.

In the present study, we treat testing as a unified activity, with risk and cost as the common tension regulating the degree of testing required. From a fault-diagnosis perspective, we consider both the cost of module replacement and the cost of testing. We want to replace the fewest number of components as quickly as possible while ensuring the system is restored to perfect functionality. From a regression testing perspective—particularly with the open architectures employed within the Integrated Warfare System—we need to conduct sufficient testing following a component upgrade to verify that the system remains in perfect function. The element of risk is that costs incurred for perfect knowledge may rapidly approach infinity. From an operational perspective, then, we must accept with some level of confidence (e.g., 99%, 95%) that our diagnosis or prognosis is correct.

3. Model Formulation

The growing use of commercial-off-the-shelf technologies in current weapons systems (Caruso, 1995; Dalcher, 2000), coupled with the complexity of end-to-end systems (Athans, 1987), suggests that we may never have enough information to fully specify our system as a white box—with all software, hardware and communication interfaces perfectly characterized. We thus construct our model with broad parameters that can be constrained as narrowly as available information permits.

We characterize the model system as a collection of modules comprising the system and as a collection of tests used to interrogate the system. When the system is down, we assume that one or modules has failed. We examine the system through this test suite to locate the correct module or modules to replace. We assume that tests return ambiguous information about the state of modules within the system, and that some sequence of tests must typically be applied to arrive at a correct diagnosis. Stochastic simulation of the model system provides a framework in which different strategies may be applied and measured for further insight. Using this Monte Carlo approach, we may also test the bounds of our initial assumptions with additional simulation.

3.1. System and Module Objects

We form the model system **S** as a collection of modules, or units of replacement. Each module M_i represents the smallest diagnostic unit, which does not necessarily correspond to a single physical component in the modeled system. We consider, for example, a computer server comprised of motherboard, hard drive and power supply, each of which may cause the computer server to fail. This would be modeled as a single module labeled *Server* if the standard corrective maintenance action is to replace the entire unit. A fundamental assumption in this abstraction is that the physical system is decomposable into these units of

replacement. We note that even in this example, a separate diagnostic model could be applied to the server, treating each of its subcomponents (motherboard, hard drive, power supply) as replaceable units.

We assume **S** is always in one of two states: fully functioning (UP) or inoperable (DOWN). Each module is similarly assessed as GOOD or BAD. We take **S** as UP if and only if every M_i is GOOD. In practice, this means that if we find **S** inoperable, we may assume that one or more modules have failed. In this event, we seek to replace the fewest number of modules with the least testing and in the shortest time.

Each M_i is modeled as the unit circle A_i. Defects, when present, are assumed uniformly distributed on this circle. We assume that while multiple modules may be defective, only one defect exists per module. A defect in M_i is modeled as a random point on A_i or, equivalently, a random point on the interval [0, 1].

Fundamental to this aspect of the model is a source of failure rate data for the system components. These failure rates become the a priori data in the larger probability model, and so do not necessarily need to be precise to add value to simulation results. The relative rates among the modeled components (e.g., the *Server* module fails about five times as often as the *Router* module) should be close to the observed data in the physical system to provide the most realistic convergence in testing to a correct diagnosis.

3.2. Test Objects

Tests are modeled as system objects which, when executed, provide an ambiguous assessment of one or more modules within **S**. This ambiguity stems from two essential elements that map the tractable model to physical reality.

The first aspect is that any given test likely exercises only a portion of the functionality within a module. Although the module is the unit of replacement, we

parameterize the sub-module details by treating them as a continuous space covered, in part, by a given test.

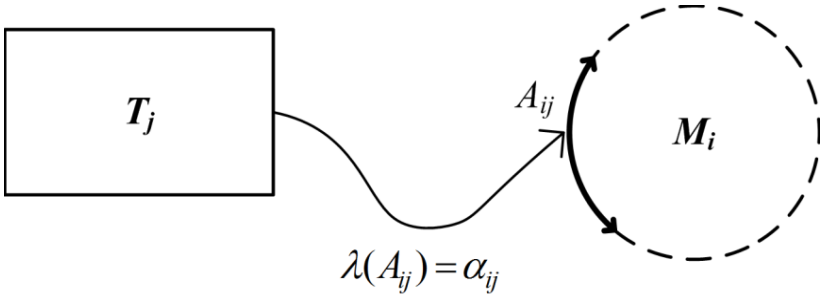

Figure 1. The simple coverage of test T_j on module M_i indicated by the solid arc A_{ij}. The measure of this coverage $\lambda(A_{ij}) = \alpha_{ij}$ represents the fraction of M_i exercised by T_j.

We model the coverage of test T_j on module M_i as the arc A_{ij} (Figure 1). When T_j is executed, or applied to the model system, the arc A_{ij} on M_i is inspected for a defect. Given the assumption that defects appear uniformly on this unit circle, the probability that a defect in M_i will be detected by T_j is the measure of this arc or $\lambda(A_{ij}) = \alpha_{ij}$. The scalar probability of detection by a test is precisely the user-specified functionality exercised by the test. This element of our language of description permits some ambiguity in characterizing the physical system (e.g., built-in self-test 3 exercises about 45% of the functionality of the graphics processing unit) without loss of rigor in modeling these tests and modules. In practice, given a sufficient number of real-world cases from the physical system, this estimate for A_{ij} could be refined through analysis of simulation results.

The second ambiguous aspect is that any given test likely covers multiple modules, such that any test result must be interpreted as applying to *all* modules covered by that test (Figure 2). For example, a positive result (FAIL) from a diagnostic test that covers the modules *Carburetor, Distributor Cap,* and *Spark Plug Wiring* indicates that at least one of these modules contains a defect (has failed), though additional testing would be required to identify which of these modules is the

culprit. Because we expect that a given test exercises multiple modules in the system, we speak more generally of the coverage of T_j on **S** (Figure 2).

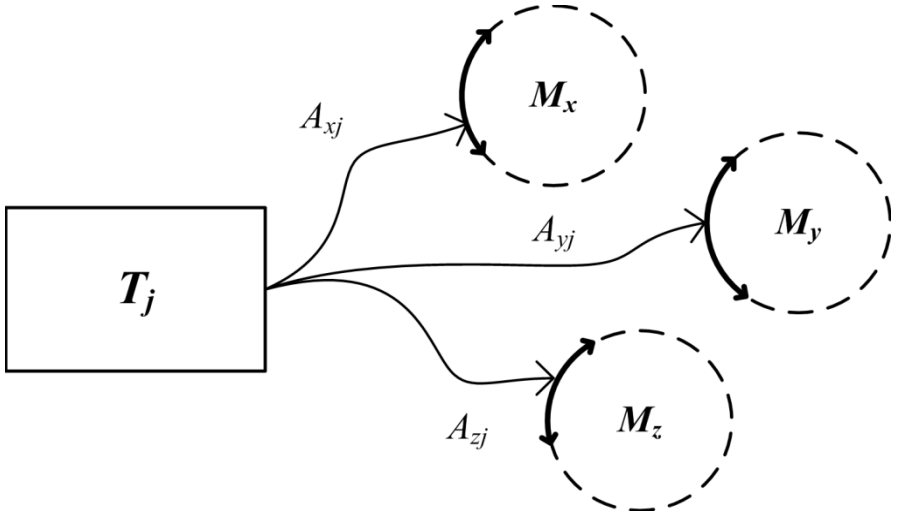

Figure 2. Notional depiction of the coverage of T_j on **S**, with multiple modules exercised upon execution of this test. A FAIL result from T_j indicates that at least one of the subset $\{M_x, M_y, M_z\}$ has failed**.**

Within the model, a test when executed assumes one of two values: PASS or FAIL. A PASS result for a given test T_j indicates that no region covered by this test contains a defect. A FAIL result indicates that at least one of the modules covered by T_j contains a defect, or is BAD in the model definition. While a FAIL result should reduce the set of modules that may need to be replaced, a perfect result—replacing only those modules that have failed—will typically require some sequence of tests. Indeed, for a particular configuration of tests and modules, this perfect result may not be achievable. Analysis of simulation results should help to identify those cases in which further testing will yield no new information.

The use of vector arcs to model the coverage relationship between tests and modules enables precision when specifying the coverage by multiple tests on a single replaceable unit (Figure 3). Although several tests in the system suite may exercise a given module, it is likely in the physical system that these tests overlap

significantly. This language of description, then, permits a user specification of the physical system in broad terms (e.g., the *Remote Control* test and *Obstacle Detection* test both exercise about 70% of the *Garage Door Motor* module, with about 20% overlap between the two tests). Even if these data are estimated from the physical system, existing case data and simulation results could be used to provide better specification of these joint coverages.

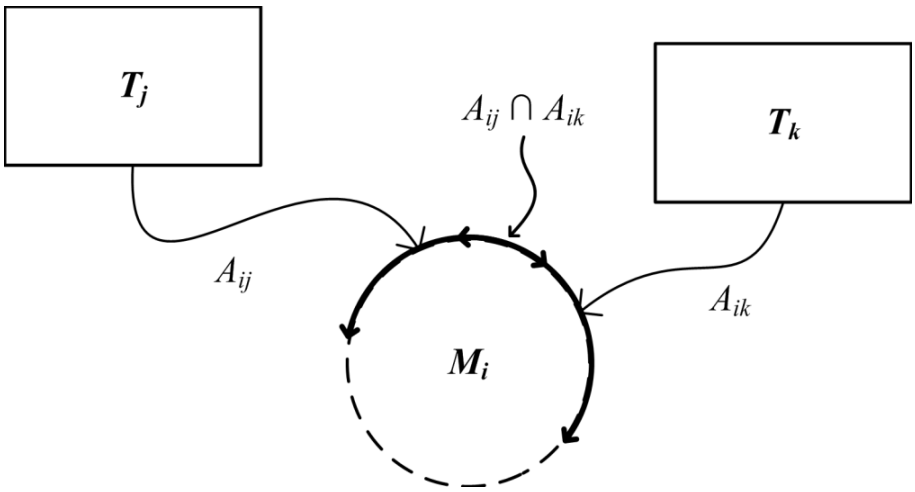

Figure 3. Overlapping coverage between tests T_j and T_k are characterized with the arcs A_{ij} and A_{ik}. The joint coverage is computable as the intersection of these arcs.

3.3. Summary

This conceptual model captures the essential elements of a system with respect to diagnostic testing and module repair or replacement. The physical system is specified in terms of modules, tests, and coverages, with model elements constructed in such a way that imperfect information can still be used as an initial state. Although the model requires that the physical system be decomposable into discrete units of replacement, this does not limit the usefulness of this approach. The fundamental diagnostic techniques could easily be applied at the sub-module level by treating a given module as a system, with sub-components then modeled as modules. In the present study, we limit our investigation to a single-layer model, though future work could investigate multiple diagnostic levels across a complex

system. We next formalize these model elements in mathematical language to construct a suitable computer simulation to investigate these testing strategies.

4. Mathematical Fundamentals

Our goal in system testing is to maximize certainty at minimum cost. In developing a probability framework to model this process, we first form simple objective measures to characterize knowledge of the system state. We next examine a simple, step-wise strategy to predict a test sequence that will maximize or minimize these measures. We then extend these simple strategies by considering a variable cost per test to examine diagnosis under limited resources.

4.1. Objective Measures of the System State

Let B_i be the event that module M_i is BAD, with probability $P(B_i) = b_i$. Given a system **S** comprised of *m* modules, we can characterize our knowledge of the system state as a vector of these probabilities:

$$\mathbf{K}^t = \left\{ b_1^t \ldots b_m^t \right\} \tag{4.1}$$

The index *t* is time-like, indicating the number of tests that have been applied to the system. At *t = 0,* no tests have been applied, and all b_i are set to their initial failure rates. Fundamental to our conceptual model is a source of failure rate data, or an a priori probability that a particular replaceable unit is defective.

We desire a diagnosis in which the components of **K** are only zero or one, meaning that we know with absolute certainty that a particular module is GOOD or BAD. In practice, this ideal diagnosis may be too costly or simply impossible to determine (see, for example, Cover & Thomas, 1991, Ch. 7). Instead, we take a step-wise approach in which we apply tests from our suite of diagnostics to incrementally improve our knowledge of **S**.

One intuitive measure of $\mathbf{K}^t$ is the information entropy (Shannon, 1948). For a single module, we compute the entropy h_i as:

$$h_i = -b_i \log_2 b_i - (1-b_i)\log_2(1-b_i) \tag{4.2}$$

We see that as b_i tends to zero or one, h_i is minimized (Figure 4). By applying tests from our diagnostic suite, we should become more certain about the state of a module (GOOD or BAD). We measure this improvement in certainty as a reduction in the individual module entropy. Across the system, we take the aggregate measure as:

$$H^t = \tfrac{1}{m}\sum_{i=1}^{m} h_i^t = \tfrac{1}{m}\sum_{i=1}^{m} -b_i^t \log_2 b_i^t - (1-b_i^t)\log_2(1-b_i^t) \tag{4.3}$$

Using this measure, we seek an ordering of k tests such that:

$$H^0 \geq H^1 \geq \ldots \geq H^k$$

That is, each test applied should act to modify some subset of module b_i to reduce the entropy of $\mathbf{K}^t$. An optimal step-wise strategy, then, would seek to maximize $\Delta H = H^t - H^{t+1}$ for each test applied to the system under diagnosis.

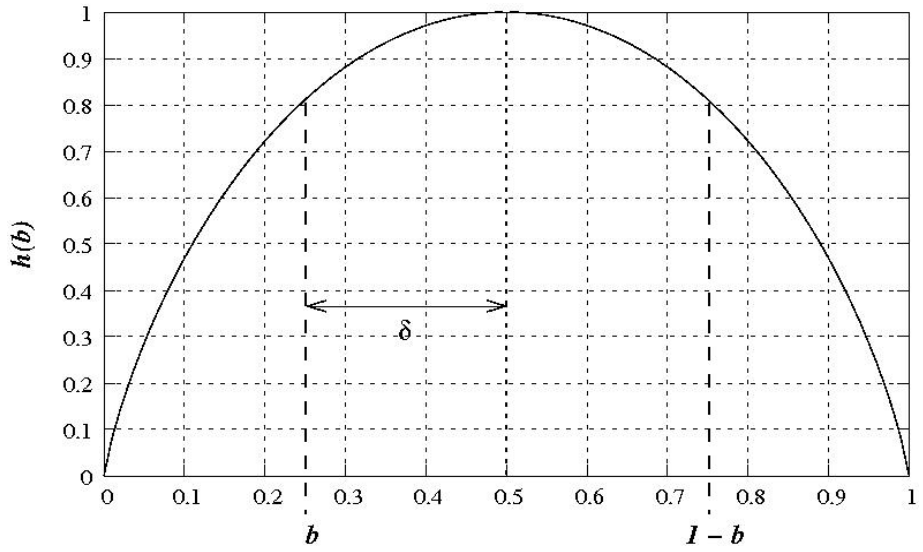

Figure 4. Module entropy $h(b_i)$, with notional module probability b_i indicated. Note that by symmetry, $h(b_i) = h(1 - b_i)$, with distance 2δ between these states.

Entropy is computationally attractive, though h_i may be less intuitive to analysts and diagnosticians when deciding which modules to replace. The probability b_i offers some insight into the likelihood that the module M_i should be replaced, such that a reasonable decision criterion D_i would be:

$$D_i = \begin{cases} \text{KEEP} & \text{if } b_i = \min(b_i, 1-b_i) \\ \text{REPLACE} & \text{if } b_i = \max(b_i, 1-b_i) \end{cases} \quad (4.4)$$

In effect, if the probability that M_i is BAD is above ½, we should replace it; if the probability is below ½, we should keep it. If, for example, a particular module has a b_i = 0.70, we replace it knowing that this informed guess should be correct 70% of the time; 30% of the time we will unnecessarily replace a GOOD module. Our number of correct guesses across the system will increase as each b_i is adjusted by testing away from ½ towards either zero or one (Figure 4). Thus, minimizing system entropy H in a step-wise process is equivalent to maximizing the number of correct replacement decisions, or correct diagnoses.

4.2. Simple Step-wise Testing

We seek to minimize the entropy of the probability vector **K** (Equation 4.1) by applying tests in step-wise fashion to update the component probabilities. For each candidate T_j in our diagnostic suite, we can compute a candidate ΔH, and then choose the test that causes the maximum reduction in entropy (largest ΔH).

In forming these predicted ΔH, we must account for both possible test outcomes. Let the event P_j represent the execution of test T_j with a PASS result. Similarly, let F_j represent the event of a FAIL result. To estimate the reduction in entropy *possible* by execution of test T_j at some point t in testing, we use the weighted sum:

$$\Delta H(T_j) = H^t - P(P_j)\sum_i h(B_i \mid P_j) - P(F_j)\sum_i h(B_i \mid F_j) \quad (4.5)$$

The entropy of the Bayesian result from either outcome is computed with:

$$h(B_i \mid P_j) = -P(B_i \mid P_j)\log_2 P(B_i \mid P_j) - P(G_i \mid P_j)\log_2 P(G_i \mid P_j) \quad (4.6)$$

$$h(B_i \mid F_j) = -P(B_i \mid F_j)\log_2 P(B_i \mid F_j) - P(G_i \mid F_j)\log_2 P(G_i \mid F_j)$$

We first consider those probabilities that describe whether a test will likely PASS or FAIL. If T_j only covers one module, the simple probability that this test will PASS becomes:

$$\begin{aligned} P(P_j) &= P(M_i \text{ is GOOD}) \bigcup P(M_i \text{ is BAD but undetected}) \\ &= (1-b_i) + (1-\alpha_{ij})b_i \\ &= 1 - b_i\alpha_{ij} \end{aligned}$$

We note that if T_j has no coverage on M_i, then $\alpha_{ij} = 0$, and this test will always PASS.

The complement to this probability is:

$$\begin{aligned} P(F_j) &= P(M_i \text{ is BAD and detected}) \\ &= \alpha_{ij} b_i \end{aligned}$$

We note that if T_j has perfect coverage on M_i ($\alpha_{ij} = 1$), then the probability that this test will pass reduces to the probability that the covered module is BAD.

In practice, we expect a given T_j will cover multiple modules, requiring that for a PASS event all modules are either GOOD, or BAD but undetected.

$$P(P_j) = \prod_{i=1}^{m} (1 - \alpha_{ij} b_i) \quad (4.7)$$

We can then compute a FAIL event for T_j as the complement of a PASS, thus:

$$P(F_j) = 1 - \prod_{i=1}^{m} (1 - \alpha_{ij} b_i) \tag{4.8}$$

Even though these products (Equation 4.7, 4.8) are computed over all modules in the system, we note that for those modules with no coverage by T_j, α_{ij} reduces to zero, and the product is unaffected.

The Bayesian results required for Equation 4.6 can be computed with:

$$P(B_i \mid F_j) = \frac{P(F_j \mid B_i) P(B_i)}{P(F_j \mid B_i) P(B_i) + P(F_j \mid G_i) P(G_i)} \tag{4.9}$$

$$P(B_i \mid P_j) = \frac{P(P_j \mid B_i) P(B_i)}{P(P_j \mid B_i) P(B_i) + P(P_j \mid G_i) P(G_i)}$$

Individual terms are computed as:

$$P(F_j \mid B_i) = \text{Given } M_i \text{ is BAD, probability } T_j \text{ will FAIL} = \alpha_{ij} b_i$$
$$P(F_j \mid G_i) = \text{Given } M_i \text{ is GOOD, probability } T_j \text{ will FAIL} = 1 - \prod_{k \neq i} (1 - \alpha_{kj} b_k)$$

$$P(P_j \mid B_i) = \text{Given } M_i \text{ is BAD, probability } T_j \text{ will PASS} = 1 - \alpha_{ij}$$
$$P(P_j \mid G_i) = \text{Given } M_i \text{ is GOOD, probability } T_j \text{ will PASS} = \prod_{k \neq i} (1 - \alpha_{kj} b_k)$$

$$P(B_i) = \text{Prior probability that } M_i \text{ is BAD} = b_i$$
$$P(G_i) = \text{Prior probability that } M_i \text{ is GOOD} = 1 - b_i$$

We note that in the case of $P(F_j|G_i)$ and $P(P_j|G_i)$, we must examine *all other* modules covered by test T_j to compute these probabilities.

This analysis provides a tractable, one-step method for sequencing tests by maximum reduction in entropy—though the computation grows approximately as the product of the number of modules, *m,* and number of tests, *p*. Additional insight may

be possible if we consider in our prediction the next best two tests (or perhaps *n* tests) in reducing entropy, though the computational cost grows as mp^n.

We have assumed implicitly in this analysis that all tests carry the same cost in some unified measure of time and money. We next discuss briefly the analysis with variable cost per test.

4.3. Variable Cost per Test

We can estimate the information gain possible with any test in our model system (Equation 4.5). With the inclusion of information about the cost per test $C(T_j)$, we can modify our objective function to compute, in effect, a cost per bit, or:

$$\Phi(T_j) = \frac{\Delta H(T_j)}{C(T_j)} \tag{4.10}$$

Our step-wise strategy, then, is to choose not the largest ΔH but the largest Φ. Although the analysis in Section 4.2 does not change, this additional model element permits a broader range of investigation in computational scenarios.

For example, with this extension of the present analysis (Equation 4.10), we could examine a scenario in which the diagnostic resources were limited by some finite purse (in terms of *C*) that, when exhausted, required the operators to make a replacement decision. In this case, a simple step-wise scenario would likely be less effective. Indeed, this particular example is more similar to the classic knapsack problem (see, for example, Corman, Leiserson & Rivest, 1990).

Given data on both test and module cost, we could also examine, at every iteration, whether the next best test (or next best *n* tests) cost more than simply replacing the current "best" candidates in the system of modules. Stochastic simulation of this scenario, given approximate real-world data, should yield significant insight into the physical systems under maintenance.

4.4. Summary

We have presented a mathematical framework to support the conceptual model of testing described in Section 3. Upon finding our system is down (or BAD), our notional diagnostic algorithm is:

1. Form the initial vector $\mathbf{K}^0$ from the given module failure rates.
2. From our diagnostic test suite, choose that T_j which maximizes ΔH.
3. After performing the selected test, update $\mathbf{K}^t$ to $\mathbf{K}^{t+1}$.
4. If we have not reached our stopping criteria, return to (2). In practice, stopping criteria might include:
   a. System entropy is very close to zero.
   b. Time or resources have expired.
   c. Cost of the next test exceeds cost of replacing candidate modules.
   d. Actual change in entropy on this cycle is very close to zero.

If we use entropy reduction as an objective measure, a simple analysis demonstrates the general utility of this approach, while additional physical data (e.g., cost per test) could easily be incorporated into the computation. We next discuss the implementation of these ideas in a computer simulation, then review results from idealized scenarios.

THIS PAGE INTENTIONALLY LEFT BLANK

5. Simulation Results and Analysis

To demonstrate the feasibility of the ideas developed in this study, a simulation was developed suitable for desktop computing. Because no physical system data were immediately available, distributions of modules and test coverages were constructed randomly subject to certain design constraints. While these scenarios provide some insight into this approach to systems testing, sufficient flexibility exists in the computer code to extend the model easily to real-world systems.

5.1. Model Details

A Java development environment was selected based on the strong numerical facilities available under most implementations and the widely portable nature of most Java code. Simulations were run primarily on a Windows Vista (x64) workstation, while portability tests were run on both Ubuntu Linux 8.04 and Mac OS X 10.5 (Leopard) machines. The simulation source code appears in Appendix A.

The code implements object models of Tests and Modules, collected under a System object. In most scenarios, 30 modules and 60 tests were constructed within the system, with test coverages spread randomly by test over some number of modules, nominally no fewer than 2, no more than five. That is, for each test T_j, a random integer q was chosen from {2, 3, 4, 5}, and then q modules were randomly selected from the system set and connected to T_j with random coverages. Initial failure rates were assigned to modules from a uniform distribution on the interval (0,1). While the code is quickly reconfigurable for more robust or physically realistic scenarios, these parameters were fixed for an initial comparison among simple test strategies.

The best-next test strategy, based on reduction of entropy, was described in Section 4. To make at least initial comparisons with the simulation code, a worst-next test strategy was implemented within the software to explore a pathological

case in which every test selected maximized entropy, or equivalently, increased uncertainty. As a baseline scenario, a random test strategy was implemented as well, with tests selected at random from the system suite.

Prior to the start of a set of trials, a failure deck was created based on the relative failure rates of modules within the system. Similar to a deck of playing cards, modules appear in the failure deck based on their standing relative to the minimum failure rate in the system; thus, if the minimum failure rate across the system is 0.2, a module with a failure rate of 0.6 will appear three times within this failure deck. The same deck is employed across all trials to simulate the relative appearance of failures in a physical system.

Prior to the start of a simulation, a test deck with one entry for each test is created (copied) from the system configuration. Strategies that compute the next best (or next worst) test operate on this deck. As a test is executed, it is removed from the deck, insuring that no test in our system will be executed more than once per trial. This also reduces the search space for the next test. A new test deck must be generated with each trial.

A single trial is processed in the following manner:

1. All module b_i are initialized from failure-rate data.
2. A module is selected from the failure deck, and a defect is planted in this module.
3. A test is chosen based on a simple strategy (best, random, worst).
4. The test is applied to the system object.
5. All affected b_i are updated based on the outcome of (4).
6. If we still have a test in the test deck, we return to (3).

Using a 2 GHz Intel processor, a simulation of 1000 trials required on average about 2.5 minutes for a randomized configuration with 60 tests and 30 modules. For a larger system configuration with 100 tests and 50 modules, run-time averaged

about 5 minutes for 1000 trials. In general, a ratio of 2:1 between tests and modules appeared to guarantee a correct diagnosis was obtainable, with the random configuration of coverages between tests and modules constrained to no fewer than 2 and no more than 5 modules per test.

5.2. Results from Initial Experiments

Over some number of trials (nominally 100 to 1000), the module traces for each strategy were aggregated. In these initial experiments, no stopping criteria were applied, and with the idealized scenario, the best-next strategy showed little improvement after 40 tests (of 60) were executed (Figure 5).

Entropy variance (Figure 6) for the best-next strategy shows a peak at about Test 19, with the caveat that Test 19 would be a different system object for each of the trials. This peak is consistent with the reduction in steepness of descent in the best-next mean entropy trace (Figure 5) and the increase of the maximum probability function to greater than 90%.

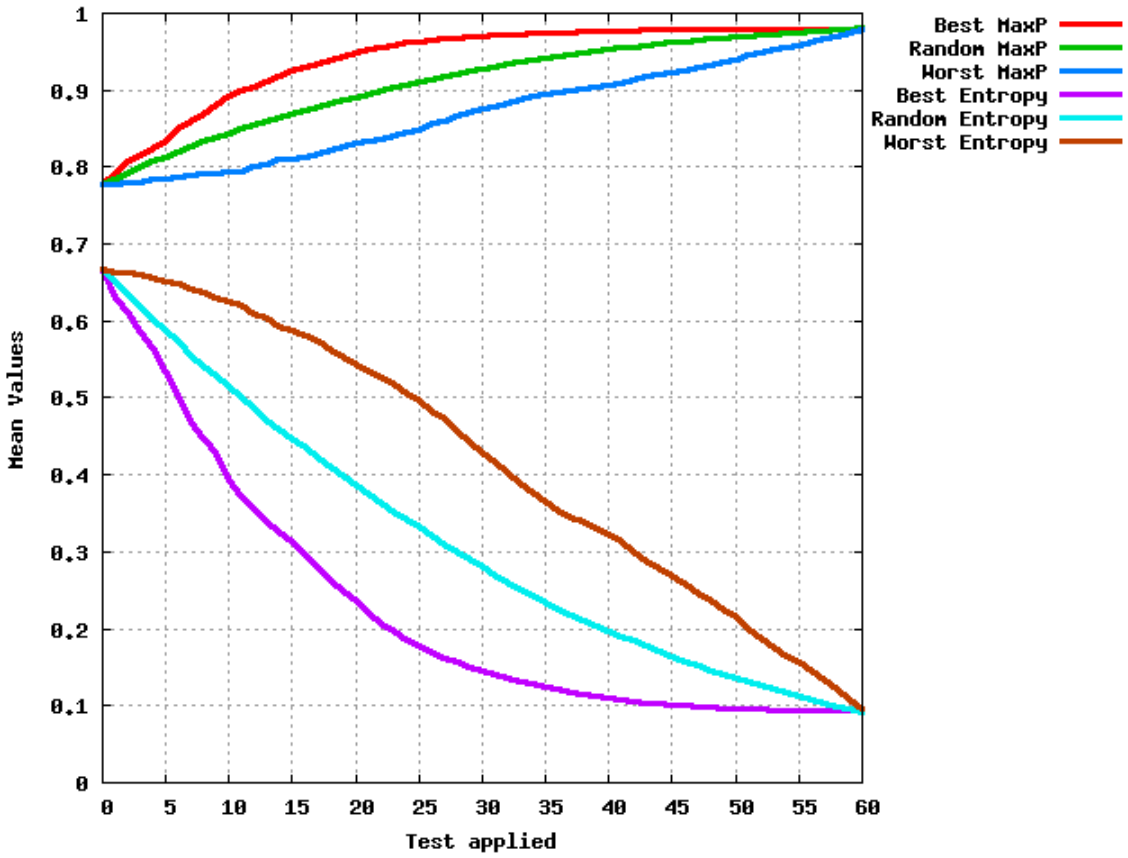

Figure 5. Mean diagnostic traces from 100 trials using best, random, and worst next-test strategies. Both system entropy (bottom traces) and maximum probability (top traces) are depicted, with all 60 tests applied—though in practice, we would likely stop sooner.

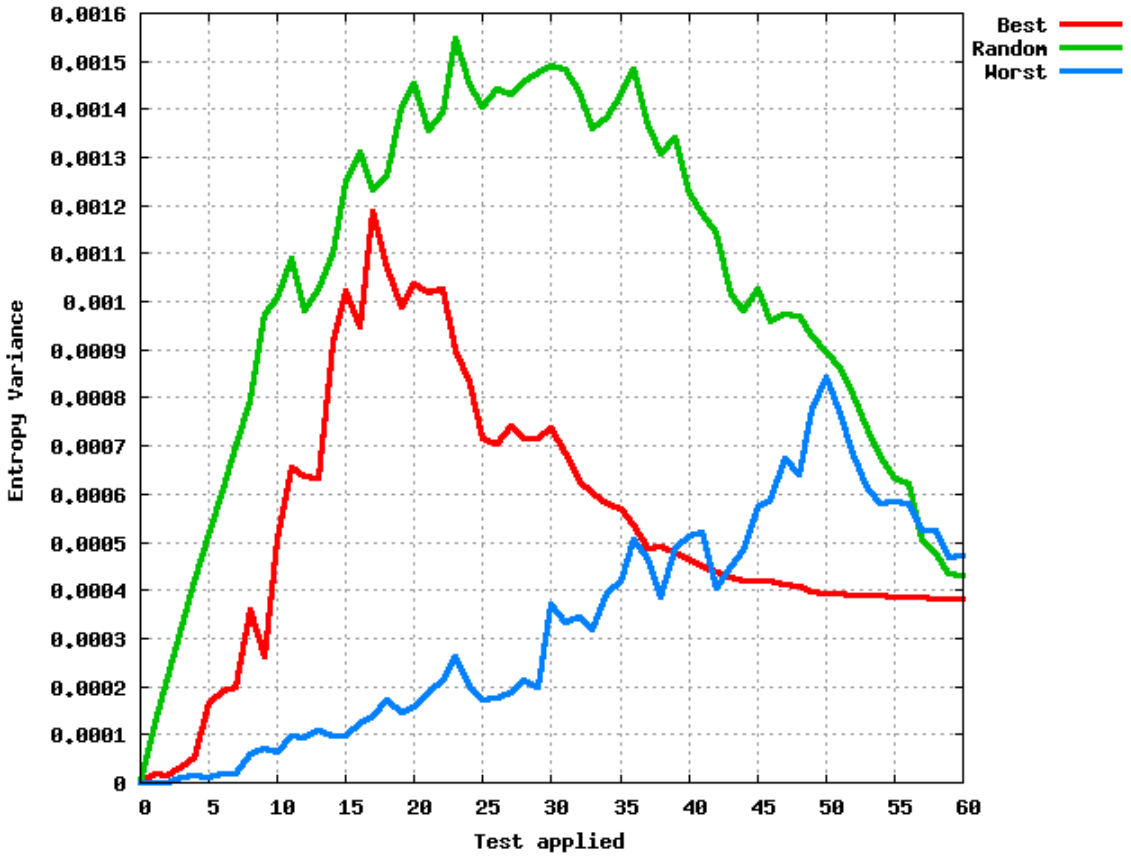

Figure 6. Diagnostic trace of variance in entropy from 100 trials using best, random, and worst next-test strategies, with all 60 tests applied. This set of trials matches the mean data depicted in Figure 7.

The distribution of model probabilities from one of the best-next trials shows the evolution of a correct diagnosis (Figures 7-10) as system testing unfolds. Although solid lines are used to highlight this dynamic in these figures, the module probabilities are, in fact, discrete. Early in testing, about Test 5, the module probabilities seem unremarkable compared to the true state (Figure 7). The best-next strategy strongly identifies Module 2 as the defective candidate, though several other modules still keep the aggregate entropy relatively high (*H = 0.40,* Figure 7). By Test 25, however, Module 2 shows a relatively large $b_i = 0.84$ with an overall aggregate entropy (*H=0.25,* Figure 8). Additional testing refines the individual module probabilities, so that by Test 30, a correct diagnosis appears evident (Figure

8). This slow down in reduction of entropy is evident in the mean curves (Figure 5) as testing proceeds past the *t=25.*

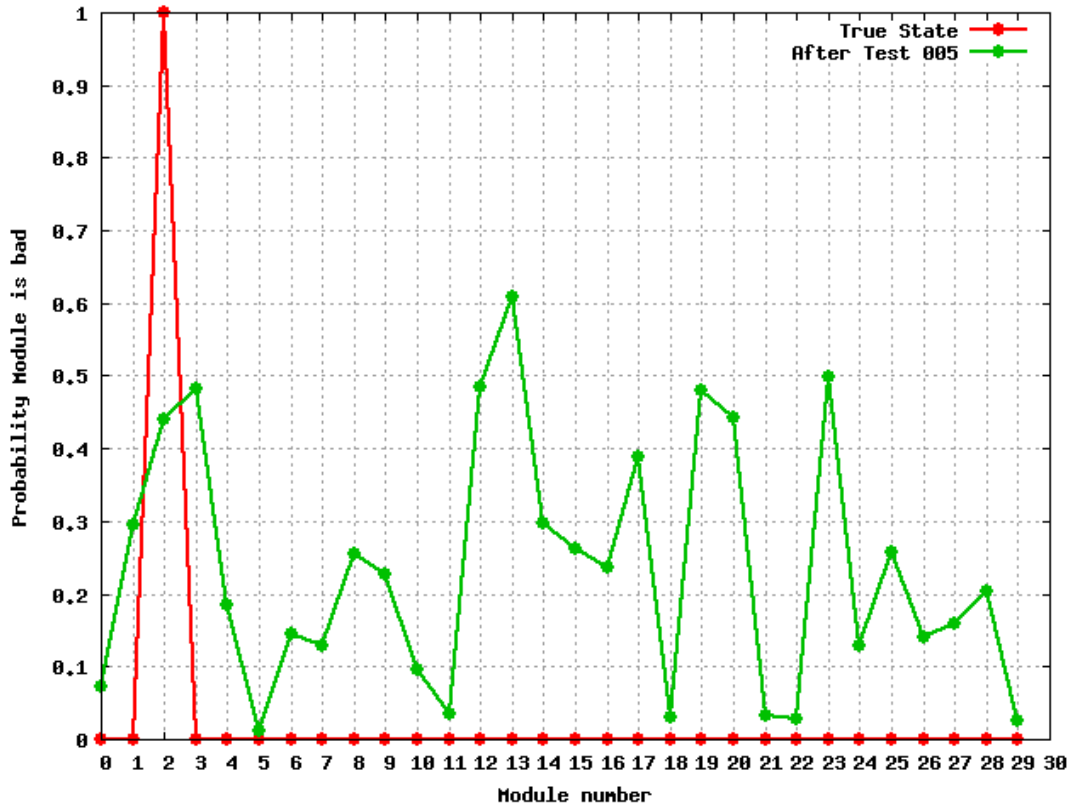

Figure 7. From a best-next test trial, module probabilities (b_i, in green) are shown versus the true state (in red) after test 5; from the system log, system entropy at this state is H=0.64.

6. Summary and Future Work

In this study, we have developed a simple but effective framework to examine the testing of complex systems. The idealized numerical experiments conducted in this study support the use of entropy reduction as an effective means to guide diagnostic testing, though these initial simulations can provide only simple insights. Real-world failure rates and coverages are needed to further investigate the usefulness of this approach for diagnosing physical systems.

We are confident that in the future, additional avenues of research will open up with more realistic scenarios—scenarios with which to exercise and develop this model. For example, simulation studies could inform the design of test suites for new weapons systems. By using available cost data for both tests and replaceable units, further research could help to develop or refine a diagnostic strategy to balance the cost of expensive, granular testing against the cost of routine maintenance. When modeling a fielded system, decision-makers could use real-time failure rate data to update the simulation and further improve fidelity.

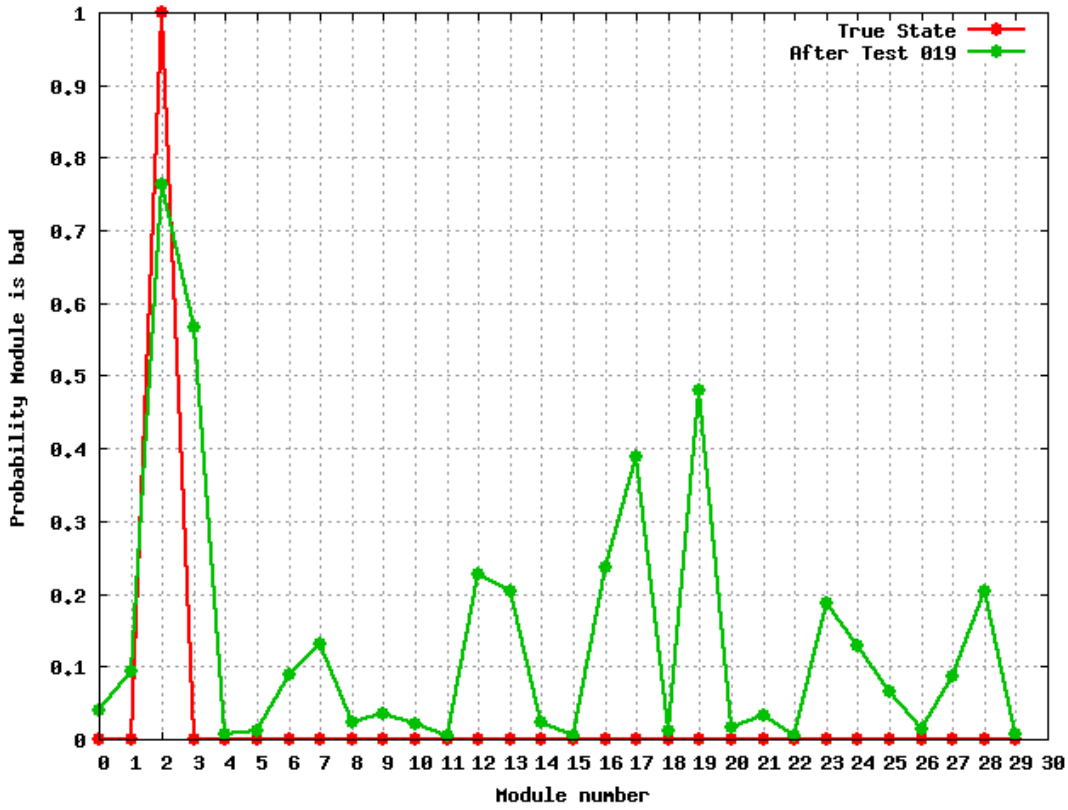

Figure 8. As in Figure 7, module probabilities after test 19, with H=0.40.

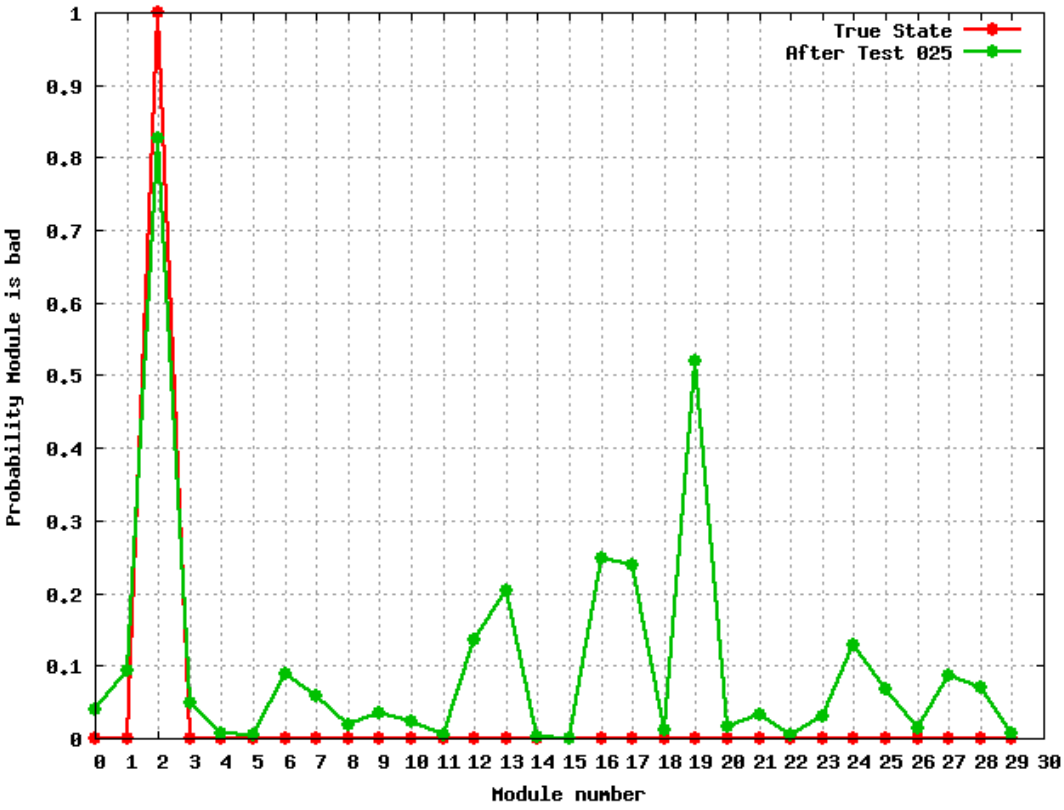

Figure 9. As in Figure 7, module probabilities after test 25, with H=0.25.

Using the flexible but precise language of our conceptual model, we can investigate the underlying probabilistic relationships of existing, complex systems. Although the original motivation for this work was the diagnostic testing of mechanical and electronic systems, with little modification, we were able to model classic regression testing scenarios in simulation code to estimate the degree and cost of testing following system upgrades.

Over the life of any complex system, particularly a weapon system, the initial costs for development are typically dwarfed by the long-term support and maintenance of the system. In this research we have made significant progress in understanding how we might better control this cost using a disciplined approach to diagnostic and regression testing, balancing our acceptable risk against available budget in dollars and time. Further research will expand and improve this rigorous

framework by investigating operational scenarios using real-world component and test structure information.

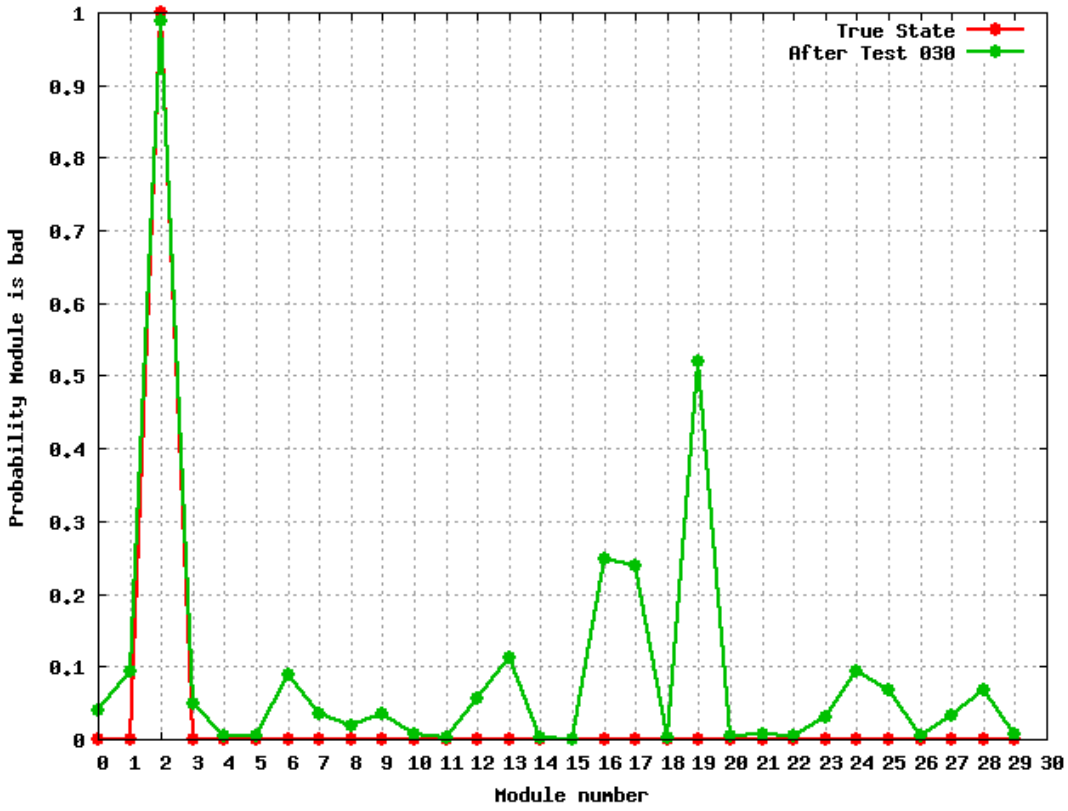

Figure 10. As in Figure 7, module probabilities after test 30, with H=0.16.

List of References

Athans, M. (1987). Command and control (C2) theory: A challenge to control science. *IEEE Transactions on Automatic Control, 32*(4), 286–293.

Barford, L., Kanevsky, V., & Kamas, L. (2004). Bayesian fault diagnosis in large-scale measurement systems. In *Proceedings* of the IMTC 2004: Instrumentation and Measurement Technology Conference (pp. 1234–1239). Como, Italy: IEEE.

Barlow, R.E., & Proschan, F. (1965). *Mathematical theory of reliability.* Philadelphia: John Wiley and Sons.

Bovaird, R.L. (1961). Characteristics of optimal maintenance policies. *Management Science, 7*(3), 238–253.

Caruso, J. (1995). The challenge of the increased use of COTS: a developer's perspective. *Proceedings of the Third Workshop on Parallel and Distributed Real-Time Systems* (pp. 155-159). Santa Barbara, California: IEEE.

Corman, T.H., Leiserson, C.E., & Rivest, R.L. (1990). *Introduction to algorithms*. Cambridge, MA: MIT Press.

Cover, T.M., & Thomas, J.A. (1991). *Elements of information theory*. New York: John Wiley and Sons.

Dalcher, D. (2000). Smooth seas - rough sailing: The case of the lame ship. *Seventh International Conference on Engineering of Computer Based Systems (EBCS 2000)* (pp. 393-395). Edinburgh, Scotland: IEEE.

Dolk, D. (2000l). Model integration in the data warehouse era. *European Journal of Operational Research, 122(2),* 199-218.

Fishman, G.S. (1990). How errors in component reliability affect system reliability. *Operations Research, 38*(4), 728–732.

Garey, M.R. (1972). Optimal binary identification procedures. *SIAM Journal on Applied Mathematics, 23*(2), 173–186.

Kottemann, J.. & Dolk, D. (1993). Model integration and a theory of models. *Decision Support Systems, 9* (1, 51-63.

Leung, H., & White, L. (1991). A Cost Model to Compare Regression Test Strategies. *Proceediings of the Conference on Software Maintenance* (pp. 201-208). Sorrento, Italy: IEEE.

Mao, C., & Lu, Y. (2005). Regression testing for component-based software systems by enhancing change information. *Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05)* (pp. 1-8). IEEE.

Moore, E.F., & Shannon, C.E. (1956a). Reliable circuits using less reliable relays, Part I. *Journal of the Franklin Institute, 262,* 191–208.

Moore, E.F., & Shannon, C.E. (1956b). Reliable circuits using less reliable relays, Part II. *Journal of the Franklin Institute, 262*, 281–298.

Nissen, M., & Buettner, R. (2004). Agent-based modeling of knowledge dynamics. *Knowledge Management Research & Practice, 2*, 169-183.

Power, D.J. (2004). Specifying an expanded framework for classifying and describing Decision Support Systems. *Communications of the ACM, 13,*158-166.

Rothermel, G. (2001). Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering, 27*(10), 929–948.

Shannon, C.E. (1948). A mathematical theory of communication. *Bell System Technical Journal, 27*, 379-423, 623-656; July, October.

Sobel, M., & Groll, P.A. (1966). Binomial group-testing with an unknown proportion of defectives. *Technometrics, 8*(4), 631–656.

Tsai, W. (2001). End-to-end integration testing design. *25th Annual International Computer Software and Applications Conference (COMPSAC)* (pp. 166-171). Chicago, IL: IEEE.

von Neumann, J. (1952). Probabilistic logics and the synthesis of reliable organisms from unreliable components. In *Automata Studies*, *Annals of Mathematics Studies* (pp. 45–98). (Vol. AM-34). Princeton, NJ: Princeton University Press.

Weyuker, E.J. (1998). Testing component-based software: A cautionary tale. *IEEE Software*, 15(5), 54–59.

White, L., & Leung, H. (1992). A firewall concept for both control-flow and data-flow in regression integration testing. *Proceedings of the Conference on Software Maintenance* (pp. 262-270). IEEE.

Appendix A. Simulation Source Code

The simulation code developed in this study, although research quality, is reasonably modular, with separate software objects for Modules, Tests, and Coverages gathered under a System object. An intermediate software object, the Probe, was developed to capture the relationship between a Test and a Module, though this was not explicitly modeled in Section 3. Additional code used to drive the simulation and provide I/O support is also documented here.

A.1 Main.java

Main.java

```
//
//  ============================================================================
//  Project: Risk-based Testing Simulation
//           Pfeiffer, Kanevsky, Housel
//           Department of Information Sciences
//           Naval Postgraduate School
//
//  Date:    1 Oct 2008
//  ============================================================================
//
import java.util.Random;
import java.util.ArrayList;
import java.io.*;
//
//  package: Main.java
//  ------------------
//
//    This is the default class containing the public main() function
//    that drives the simulation.
//
//
public class Main {
    //
    //  Global attributes available to all simulation objects
    //
    //  ... log messages from the simulation go to a standard location
    //
    private static Logger logger = new Logger("simulation.log");
    //
    //  ... one random number generator is used for the simulation
    //      all random number requests are made to this object
    //
    private static Random Generator = new Random();
    //
    //  ... the FailureDeck for deciding which modules receive
    //      defects at the start of a trial
    //
    private static ArrayList<Module> FailureDeck = new ArrayList<Module>();

    //
    //  ... method build() constructs an "empty" system with
    //      the required number of modules but no coverages
    //      connecting the two objects
    //
    static void build(SystemObject s, int nModule, int nTest) {

            for (int i = 0; i < nModule; i++) {
                String name = String.format("M%03d",i);
                double frate = 0.5 * Generator.nextDouble();
                Module m = new Module(name,frate);
                s.addModule(m);
            }

        for (int i = 0; i < nTest; i++) {
                String name = String.format("T%03d",i);
                double frate = Generator.nextDouble();
            // double cost  = Generator.nextDouble();
            double cost  = 1.0;
                Test t = new Test(name);
            t.setCost(cost);
                s.addTest(t);
        }
```

```
}

//
//  ... method configure() constructs an "empty" system with
//      the required number of modules but no coverages
//      connecting the two objects
//
static void configure(SystemObject s) {

    int[] ModuleDeck = generateRandomList(s.getModuleCount());

    int[] TestDeck   = generateRandomList(s.getTestCount());

    int nModule = s.getModuleCount();

    int nTest   = s.getTestCount();

            //
            //  ... per module connect to npeak tests where
            //      npeak is a random integer on [2,5]
            //
            for (int i = 0; i < ModuleDeck.length; i++) {
        Module md = s.getModule(ModuleDeck[i]);
        int npeak = 2 + Generator.nextInt(3);
                for (int j = 0; j < npeak; j++) {
            int q = Generator.nextInt(TestDeck.length);
            Test t = s.getTest(TestDeck[q]);
            if ( t.getCoverageOn(md) == 0.0 ) {
              double cp = Generator.nextDouble();
              double fr = Generator.nextDouble();
              t.addProbe(md,cp,fr);
            }
                }
            }
            //
            //  ... per test, connect to npeak modules where
            //      npeak is a random integer on [2,5]
            //
    for (int i = 0; i < TestDeck.length; i++) {
        Test t = s.getTest(TestDeck[i]);
        int npeak = 2 + Generator.nextInt(3);
        for (int j = 0; j < npeak; j++) {
            int q = Generator.nextInt(ModuleDeck.length);
            Module md = s.getModule(ModuleDeck[q]);
            if ( t.getCoverageOn(md) == 0.0 ) {
              double cp = Generator.nextDouble();
              double fr = Generator.nextDouble();
              t.addProbe(md,cp,fr);
            }
        }
    }

}


//
//  ... utility method, generate a random list of nSize elements
//
static int[] generateRandomList ( int nSize ) {
  int [] result = new int[nSize];
  for (int i = 0; i < nSize; i++)
    result[i] = i;
  for (int i = 0; i < nSize; i++) {
    int p = Generator.nextInt(nSize);
    int q = Generator.nextInt(nSize);
```

```
    int r = result[p];
    result[p] = result[q];
    result[q] = r;
   }
   return result;
}

//
//  ... method: create Failure Deck used in deciding in which
//              modules the simulation will plant defects
//
static void createFailureDeck(SystemObject s, int nTrial) {
    //
    //  ... find the min and max
    //      of component reliabilities
    //
          int nModule = s.getModuleCount();
          double min = 1.0;
          double max = 0.0;
    double sum = 0.0;
          for (int i = 0; i < nModule; i++) {
             Module md = s.getModule(i);
       sum += md.getFailureRate();
             if ( md.getFailureRate() > max )
                      max = md.getFailureRate();
             if ( md.getFailureRate() < min )
                      min = md.getFailureRate();
          }
    //
    //  ... populate the deck ...
    //
          for (int i = 0; i < nModule; i++) {
             Module md = s.getModule(i);
             int factor =
         (int) ((md.getFailureRate() / sum) * nTrial + 0.5);
       logger.write(".. Failure Deck: %s(%e) appears %5d times",
         md.getName(), md.getFailureRate(), factor);
             for (int j = 0; j < factor; j++) {
                      FailureDeck.add(md);
             }
          }
    //
    //  ... shuffle the deck ...
    //
          int nDeck = FailureDeck.size();
    logger.write(".. Failure Deck has %5d entries ..",nDeck);
          for (int i = 0; i < nDeck; i++) {
             int p = Generator.nextInt(nDeck);
             int q = Generator.nextInt(nDeck);
             Module mx = FailureDeck.get(p);
             Module my = FailureDeck.get(q);
             FailureDeck.set(p,my);
             FailureDeck.set(q,mx);
          }
    //
    //  ... done ...
    //
}

//
//  ... method: select next test at random
//
static Test getRandomNextTest(ArrayList<Test> tlist) {
  int q = Generator.nextInt(tlist.size());
  Test t = tlist.get(q);
  tlist.remove(q);
```

```
  return t;
}

//
//  ... method: select next best test based on
//              forecast entropy reduction
//
static Test getBestNextTest(SystemObject s, ArrayList<Test> tlist) {
  double dmax = -9999.0;
  int imax = 0;
  for (int i=0; i < tlist.size(); i++) {
    Test t = tlist.get(i);
    double dh = s.deltaEntropy(t);
    double dc = t.getCost();
    double df = dh / dc;
    if ( df > dmax ) {
      dmax = df;
      imax = i;
    }
  }
  Test tmax = tlist.get(imax);
  tlist.remove(imax);
  return tmax;
}

//
//  ... method: select worst next test based on
//              increase in entropy
//
static Test getWorstNextTest(SystemObject s, ArrayList<Test> tlist) {
  double dmin = 9999.0;
  int imin = 0;
  for (int i=0; i < tlist.size(); i++) {
    Test t = tlist.get(i);
    double dh = s.deltaEntropy(t);
    double dc = t.getCost();
    double df = dh / dc;
    if ( df < dmin ) {
      dmin = df;
      imin = i;
    }
  }
  Test tmin = tlist.get(imin);
  tlist.remove(imin);
  return tmin;
}

//
//  ... method: plant defect in module selected at random
//              from Failure Deck, at a point selected at
//              random on the interval [0,1]
//
static int plantDefect(SystemObject s, int nTrial) {

        if ( FailureDeck.size() == 0 )
            createFailureDeck(s,nTrial);

        Module mi = FailureDeck.get(0);
        FailureDeck.remove(0);

        double defect = Generator.nextDouble();

        mi.setDefectAt(defect);

    int imod = s.getModuleIndex(mi);
```

```
            logger.write("... Setting defect at %f in module %s",
                                    defect, mi.getName());

    return imod;

}

//
//  ... method: execute test on the system object, update
//              module probabilities based on PASS or FAIL
//
static void updateOnTest(SystemObject s, Test t) {

            Test.Result result = t.applyTest();

            String sResult = "PASS";
            if ( result == Test.Result.FAIL )
                sResult = "FAIL";
            logger.write("%s %s", t.getName(), sResult);

    ArrayList<Module> mlist = t.getModulesProbed();
    int nModule = mlist.size();
    for (int i=0; i < nModule; i++) {
      Module md = mlist.get(i);
      double bi = md.getBad();
      if ( result == Test.Result.FAIL ) {
        bi = md.computeBadGivenFail(t);
        logger.write("%s P(Bi|Fj) = %f", md.getName(), bi);
      }
      else {
        bi = md.computeBadGivenPass(t);
        logger.write("%s P(Bi|Pj) = %f", md.getName(), bi);
      }
      md.setBad(bi);
    }
}


//
//  ... method: utility routine to compute mean by trial
//              in the 2D matrix generated by a simulation
//
static double[] computeMeanByTrial(double[][] data) {

  int nTrial = data.length;
  int nTest  = data[0].length;

  double[] mean = new double[nTest];

  for (int i=0; i < nTest;  i++) {
    for (int j=0; j < nTrial; j++) {
      mean[i] += data[j][i];
    }
    mean[i] /= nTrial;
  }

  return mean;

}

//
//  ... method: utility routine to compute variance by trial
//              in the 2D matrix generated by a simulation
//
static double[] computeVarianceByTrial(double[][] data) {
```

```
  double[] mean = computeMeanByTrial(data);

  int nTrial = data.length;
  int nTest  = data[0].length;

  double[] var  = new double[nTest];

  for (int i=0; i < nTest;  i++) {
    for (int j=0; j < nTrial; j++) {
      var[i] += (mean[i] - data[j][i])*(mean[i] - data[j][i]);
    }
    var[i] /= (nTrial - 1);
  }
  return var;
}

//
//  ... method: main call to execute simulation
//
static void doSimulation() {

        // long seed = 271828;
        // long seed = 314159;
        // Generator.setSeed(seed);
        int nModule = 30;
        int nTest   = 60;
        int nTrial  = 100;

  logger.write("================");
  logger.write("... Starting ...");
  logger.write("================");

        SystemObject s = new SystemObject(logger);

  build(s,nModule,nTest);

  configure(s);

        s.describeModules();

        s.describeTests();

        logger.write("System entropy = %f, distance = %f",
    s.Entropy(),s.Distance());

  s.dumpStateToFile("output/test000.dat");

        double[][] entropyRandom = new double[nTrial][nTest + 1];
  double[][] entropyBest   = new double[nTrial][nTest + 1];
        double[][] entropyWorst  = new double[nTrial][nTest + 1];
  double[][] maxpRandom    = new double[nTrial][nTest + 1];
  double[][] maxpBest      = new double[nTrial][nTest + 1];
  double[][] maxpWorst     = new double[nTrial][nTest + 1];

        double h0    = s.Entropy();
  double maxp  = s.MaxProbability();
        for (int i=0; i < nTrial; i++) {
           entropyRandom[i][0] = h0;
           entropyBest[i][0]   = h0;
           entropyWorst[i][0]  = h0;
     maxpRandom[i][0]    = maxp;
     maxpBest[i][0]      = maxp;
     maxpWorst[i][0]     = maxp;
        }

        ArrayList<Test> TestList;
```

```
for (int j=0; j < nTrial; j++) {

    logger.write("## TRIAL %03d ##",j);
    int i;

    int imod = plantDefect(s,nTrial);
    s.dumpTrueStateToFile("output/true.dat");

        TestList = s.copyTestList();
        i = 0;
        while ( TestList.size() > 0 ) {
                Test t = getRandomNextTest(TestList);
                logger.write("RANDOM Test=%s delta H = %f",
                                t.getName(),
                                s.deltaEntropy(t));
                updateOnTest(s,t);
                double h = s.Entropy();
                logger.write("... Entropy = %f, Distance = %f",
                                h,s.Distance());
                i++;
                entropyRandom[j][i] = h;
      maxpRandom[j][i] = s.MaxProbability();
                // s.dumpStateToFile(String.format("output/test%03d.dat",i));
        }
        s.ResetBadValues();

        TestList = s.copyTestList();
        i = 0;
        while ( TestList.size() > 0 ) {
                Test t = getBestNextTest(s,TestList);
                logger.write("BEST Test=%s delta H = %f",
                                t.getName(),
                                s.deltaEntropy(t));
                updateOnTest(s,t);
                double h = s.Entropy();
                logger.write("... Entropy = %f, Distance = %f",
                                h,s.Distance());
                i++;
                entropyBest[j][i] = h;
      maxpBest[j][i] = s.MaxProbability();
                s.dumpStateToFile(String.format("output/test%03d.dat",i));
        }
        s.ResetBadValues();

        TestList = s.copyTestList();
        i = 0;
        while ( TestList.size() > 0 ) {
                Test t = getWorstNextTest(s,TestList);
                logger.write("WORST Test=%s delta H = %f",
                                t.getName(),
                                s.deltaEntropy(t));
                updateOnTest(s,t);

                double h = s.Entropy();
                logger.write("... Entropy = %f, Distance = %f",
                                h,s.Distance());
                i++;
                entropyWorst[j][i] = h;
      maxpWorst[j][i] = s.MaxProbability();
                // s.dumpStateToFile(String.format("output/test%03d.dat",i));
        }
        s.Reset();

} // end for(j)
```

Main.java

```
            double[] meanBest   = computeMeanByTrial(entropyBest);
            double[] meanWorst  = computeMeanByTrial(entropyWorst);
            double[] meanRandom = computeMeanByTrial(entropyRandom);

        double[] varBest = computeVarianceByTrial(entropyBest);
        double[] varWorst = computeVarianceByTrial(entropyWorst);
        double[] varRandom = computeVarianceByTrial(entropyRandom);

        double[] meanBestMaxP   = computeMeanByTrial(maxpBest);
        double[] meanWorstMaxP  = computeMeanByTrial(maxpWorst);
        double[] meanRandomMaxP = computeMeanByTrial(maxpRandom);

            Utility.dumpArrayToFile(meanRandom, "entropy-random.dat");
            Utility.dumpArrayToFile(meanBest, "entropy-best.dat");
            Utility.dumpArrayToFile(meanWorst, "entropy-worst.dat");

        Utility.dumpArrayToFile(meanRandomMaxP, "maxp-random.dat");
        Utility.dumpArrayToFile(meanBestMaxP, "maxp-best.dat");
        Utility.dumpArrayToFile(meanWorstMaxP, "maxp-worst.dat");

        Utility.dumpArrayToFile(varBest,"entropy-var-best.dat");
        Utility.dumpArrayToFile(varWorst,"entropy-var-worst.dat");
        Utility.dumpArrayToFile(varRandom,"entropy-var-random.dat");

        logger.write("Observed Failures ...");
        s.sortModulesByFailureRate();
        for (int i=0; i < nModule; i++) {
          Module md = s.getModule(i);
          logger.write("... %s (%e) failed %5d times",
            md.getName(), md.getFailureRate(), md.getFailureCount());
        }

        logger.write("============");
        logger.write("... DONE ...");
        logger.write("============");

    } // end doSimulation()


    //
    //  ... standard main() ...
    //
    public static void main(String args[]) {

            doSimulation();

    }

}
//
//  ================
//  end package Main
//  ================
//
```

A.2 SystemObject.java

SystemObject.java

```
//
//  ============================================================================
//  Project: Risk-based Testing Simulation
//           Pfeiffer, Kanevsky, Housel
//           Department of Information Sciences
//           Naval Postgraduate School
//
//  Date:    1 Oct 2008
//  ============================================================================
//
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.io.*;
//
//  package: SystemObject.java
//  --------------------------
//
//    The SystemObject is the model for the system and container
//    for Module and Test objects.
//
//    This theoretical System is comprised of a collection of
//    Modules with known (or estimated) failure rates, and a
//    collection of Tests, each of which exercises one or more
//    Modules within the system.
//
public class SystemObject {

    //
    //  ... attributes ...
    //
    ArrayList<Module> ModuleList = new ArrayList<Module>();
    ArrayList<Test> TestList     = new ArrayList<Test>();
    Logger Log;

    //
    //  ... methods ...
    //
    SystemObject() {
      this.Log = new Logger("systemobject.log");
    }

    SystemObject(Logger log) {
      this.Log = log;
    }

    void addModule(Module m) {
            this.ModuleList.add(m);
    }

    void addModuleList(Module[] mlist) {
      for (int i=0; i < mlist.length; i++) {
        this.ModuleList.add(mlist[i]);
      }
    }

    Module getModule(int q) {
            return this.ModuleList.get(q);
    }

    int getModuleCount() {
       return this.ModuleList.size();
```

```
}

Module getModuleByName(String name) {
  for (int i = 0; i < this.ModuleList.size(); i++) {
    Module m = this.ModuleList.get(i);
    String current = m.getName();
    if ( current.equals(name) ) return m;
  }
  return null;
}

int getModuleIndex(Module m) {
        int i;
        for (i = 0; i < this.ModuleList.size(); i++) {
            if ( this.ModuleList.get(i) == m ) return i;
        }
        return -1;
}

void addTest(Test t) {
        this.TestList.add(t);
}

void addTestList(Test[] tlist) {
  for (int i=0; i < tlist.length; i++) {
    this.TestList.add(tlist[i]);
  }
}

Test getTest(int q) {
        return this.TestList.get(q);
}

int getTestCount() {
    return this.TestList.size();
}

ArrayList<Test> copyTestList() {
  ArrayList<Test> tlist = new ArrayList<Test>();
  ArrayList<Test> olist = this.TestList;
  for (int i=0; i < olist.size(); i++)
    tlist.add(olist.get(i));
  return tlist;
}

double MaxProbability() {
  double maxp = 0.0;
  int nModule = this.ModuleList.size();
  for (int i=0; i < nModule; i++) {
    double p = this.getModule(i).getBad();
    maxp += Math.max(p, 1-p);
  }
  maxp /= this.ModuleList.size();
  return maxp;
}

double Entropy() {
  double esum = 0.0;
  int nModule = this.ModuleList.size();
  for (int i = 0; i < nModule; i++) {
    esum += this.ModuleList.get(i).Entropy();
  }
  esum /= nModule;
  return esum;
}
```

```
double deltaEntropy(Test t) {
    int nModule = this.getModuleCount();
            double delta = 0.0;
            for (int i = 0; i  < nModule; i++)
                delta += this.getModule(i).deltaEntropy(t);
            return delta;
}

double[] getTrueStateVector() {
  int nModule = this.getModuleCount();
  double[] trueState = new double[nModule];
  for (int i=0; i < nModule; i++) {
    trueState[i] = 0.0;
    if ( this.getModule(i).hasDefect() )
      trueState[i] = 1.0;
  }
  return trueState;
}

double Distance() {
  int nModule = this.getModuleCount();
  double[] trueState = this.getTrueStateVector();
  double sum = 0.0;
  for (int i=0; i < nModule; i++) {
    double dx = trueState[i] - this.getModule(i).getBad();
    sum += dx*dx;
  }
  return Math.sqrt(sum);
}

void describeTests() {

            int nTest = this.TestList.size();
            int i,j;
            for (i = 0; i < nTest; i++) {
                Test t = this.TestList.get(i);
                this.Log.write("Test %s, Cost(%f) Pj(%f)",
          t.getName(),t.getCost(),t.probabilityPass());
                int nProbe = t.ProbeList.size();
                for (j = 0; j < nProbe; j++) {
                        Probe p = t.ProbeList.get(j);
                        Module m = p.getModule();
                        this.Log.write("   Module %s, coverage = %f",
                                        m.getName(), p.getFraction());
                }
            }
}

void describeModules() {
            int nModule = this.ModuleList.size();
            int nTest = this.TestList.size();
            int i,j,k;
            for (i = 0; i < nModule; i++) {
                Module m = this.ModuleList.get(i);
                this.Log.write("Module %s: Failure rate(%8.2e) H(%8.2e)",
                                        m.getName(),m.getFailureRate(),m.Entropy());
        int nProbe = m.getProbeCount();
                for (j = 0; j < nProbe; j++) {
            Probe pb = m.getProbe(j);
                        Test t = pb.getTest();
                        int nGraphic = 30;
                        this.Log.write("  Test %s (%5.2f) %s",
                                        t.getName(),
                                        pb.getFraction(),
                                        pb.getCoverageGraphic(nGraphic));
                } // end for(j)
```

```
            } // end for(i)
  } // end describeModules()


  void dumpTrueStateToFile(String outfile) {
    try {
      PrintWriter output =
        new PrintWriter(new FileWriter(new File(outfile)));
      double[] trueState = this.getTrueStateVector();
      for (int i = 0; i < trueState.length; i++) {
        output.printf("%f\n",trueState[i]);
      }
      output.close();
    }
    catch(IOException ioException) {
      System.out.println("Error writing file");
      System.exit(0);
    }
  }

  void dumpStateToFile(String outfile) {
    try {
      PrintWriter output =
        new PrintWriter(new FileWriter(new File(outfile)));
      int nModule = this.getModuleCount();
      for (int i = 0; i < nModule; i++) {
        output.printf("%f\n",this.getModule(i).getBad());
      }
      output.close();
    }
    catch(IOException ioException) {
      System.out.println("Error writing file");
      System.exit(0);
    }
  }


  void Reset() {
      int nModule = this.ModuleList.size();
      for (int i = 0; i < nModule; i++) {
                Module md = this.ModuleList.get(i);
                md.Reset();
            }
  }

  void ResetBadValues() {
      int nModule = this.ModuleList.size();
      for (int i = 0; i < nModule; i++) {
                Module md = this.ModuleList.get(i);
                md.ResetBadValue();
            }
  }

  void sortModulesByFailureRate() {
    Collections.sort(this.ModuleList, new byFailureRate());
  }

class byFailureRate implements java.util.Comparator<Module> {
public int compare(Module x, Module y) {
  int result = 0;
  if ( x.getFailureRate() > y.getFailureRate() )
    result = 1;
  else
    result = -1;
  return result;
  }
```

SystemObject.java

```
  } // end class byFailureRate
}
//
//  ======================
//  end class SystemObject
//  ======================
//
```

A.3 Module.java

Module.java

```
//
//  ============================================================================
//  Project: Risk-based Testing Simulation
//           Pfeiffer, Kanevsky, Housel
//           Department of Information Sciences
//           Naval Postgraduate School
//
//  Date:    1 Oct 2008
//  ============================================================================
//
import java.util.ArrayList;
//
//  package: Module.java
//  --------------------
//
//    The Module represents the smallest replaceable unit within
//    the System.  We assume only a simple failure rate to describe
//    the reliability of the Module.
//
public class Module {
    //
    //  constant NO_DEFECT is used to indicate
    //  that our device is working correctly
    //
    public static final double NO_DEFECT = 9999.0;

    //
    //  ... attributes ...

    String Name;
    double FailureRate;
    double Bad;
    double Defect;
    double Cost;
    int    FailureCount;
    ArrayList<Probe> ProbeList;

    //
    //  ... methods ...
    //

    //
    //  ... initialize a Module ...
    //
    Module(String name, double frate) {
            //
            //  ... user supplied a Name and FailureRate
            //
       Name = name;
```

```
    FailureRate = frate;
        Bad = frate;
        Cost = 1.0;
        Defect = NO_DEFECT;
    FailureCount = 0;
    ProbeList = new ArrayList<Probe>();
}

String getName() {
  return this.Name;
}

void setName(String name) {
        this.Name = name;
}

double getFailureRate() {
        return this.FailureRate;
}

void setFailureRate(double frate) {
        this.FailureRate = frate;
}

double getBad() {
  return this.Bad;
}

void setBad(double bad) {
  this.Bad = bad;
}

double Entropy() {
  double p = this.Bad;
  double e = Utility.entropy(p);
  return e;
}


double deltaEntropy(Test t) {
        double h = this.Entropy();
        double pj = t.probabilityPass();
        double fj = 1.0 - pj;
        double bifj = this.computeBadGivenFail(t);
        double bipj = this.computeBadGivenPass(t);
        double hfail = Utility.entropy(bifj);
        double hpass = Utility.entropy(bipj);
        double result = h - hpass*pj - hfail*fj;
        return result;
}


void addProbe(Probe p) {
  this.ProbeList.add(p);
}

Probe getProbe(int i) {
  return this.ProbeList.get(i);
}

int getProbeCount() {
  return this.ProbeList.size();
}

//
//  ... plant a defect ...
```

Module.java

```
//
void setDefectAt(double p) {
          this.Defect = p;
   FailureCount++;
}

int getFailureCount() {
  return this.FailureCount;
}

//
//  ... remove the defect ...
//
void Reset() {
          this.Defect = NO_DEFECT;
   this.Bad = this.FailureRate;
}

void ResetBadValue() {
   this.Bad = this.FailureRate;
}


boolean hasDefect() {
   if ( this.Defect == NO_DEFECT )
     return false;
   else
     return true;
}

//
//  ... test our Module within a specified arc
//      about a specified center point
//
//      return TRUE if Module is defective
//
//      return FALSE if Module appears to be
//        functioning normally
//
boolean containsDefect(Coverage coverage) {
          //
          //  ... apply the test by "looking" within the
          //      user-specified arc for any defect
          //      returning TRUE if we are BAD and
          //      FALSE if we are GOOD or UNKNOWN
          //
   if ( coverage.containsPoint(this.Defect) ) {
             return true;
   }
   else
             return false;

} // end containsDefect()


//
//  computeBadGivenFail
//  -------------------
//
//    Compute the probability that given test Tj = FAIL
//    our module is BAD
//
//    P(Bi|Fj) =              P(Fj|Bi)P(Bi)
//                  -----------------------------
//                   P(Fj|Bi)P(Bi) + P(Fj|Gi)P(Gi)
//
```

```
//
//    where:
//
//      P(Bi) = current probability Mi is BAD
//
//      P(Gi) = 1 - P(Bi)
//
//      P(Fj|Bi) = probability of detecting failure, or
//                 the coverage alpha_ij
//
//      P(Fj|Gi) = probability that even though Mi is good
//                 some other module failed and was detected
//                 by Tj
//
double computeBadGivenFail(Test t) {
           double alpha = t.getCoverageOn(this);
           if ( alpha == 0.0 ) return this.Bad;
           double bi = this.Bad;
           double gi = 1.0 - bi;
           double fjbi = alpha;
    int nProbe = t.getProbeCount();
           double prod = 1.0;
           for (int i=0; i < nProbe; i++) {
              Probe pb = t.getProbe(i);
              Module md = pb.getModule();
              if ( md != this )
                        prod = prod*(1.0 - t.getCoverageOn(md)*md.getBad());
           }
           double fjgi = 1 - prod;
           double result = (fjbi*bi) / (fjbi*bi + fjgi*gi);
           return result;
}

//
//  computeBadGivenPass
//  -------------------
//
//    Compute the probability that given test Tj = PASS
//    our module is BAD
//
//    P(Bi|Pj) =                P(Pj|Bi)P(Bi)
//                  ------------------------------
//                   P(Pj|Bi)P(Bi) + P(Pj|Gi)P(Gi)
//
//
//    where:
//
//      P(Bi) = current probability Mi is BAD
//
//      P(Gi) = 1 - P(Bi)
//
//      P(Pj|Bi) = probability of non-detect of failure
//                 or 1 - alpha_ij
//
//      P(Pj|Gi) = probability some other module failed
//                 in the set of modules covered by Tj
//
double computeBadGivenPass(Test t) {
           double alpha = t.getCoverageOn(this);
    if ( alpha == 0.0 ) return this.Bad;
           double bi = this.Bad;
           double gi = 1.0 - bi;
           double pjbi = 1.0 - alpha;
    int nProbe = t.getProbeCount();
           double prod = 1.0;
           for (int i=0; i < nProbe; i++) {
```

Module.java

```
                Probe pb = t.getProbe(i);
                Module md = pb.getModule();
                if ( md != this )
                        prod = prod*(1.0 - t.getCoverageOn(md)*md.getBad());
            }
            double pjgi = prod;
            double result = (pjbi*bi) / (pjbi*bi + pjgi*gi);
            return result;
    }



}
//
//  ====================
//  end package: Module
//  ====================
//
```

A.4 Test.java

Test.java

```
//
//  ============================================================================
//  Project: Risk-based Testing Simulation
//           Pfeiffer, Kanevsky, Housel
//           Department of Information Sciences
//           Naval Postgraduate School
//
//  Date:    1 Oct 2008
//  ============================================================================
//
import java.util.ArrayList;
//
//  package: Test.java
//  ------------------
//
//  A Test is the smallest diagnostic executable within the System.
//
//  We treat a Test as a collection of Probes on Modules, with each
//  Probe testing some fraction on the interval [0,1) over a
//  specific Module.
//
//  When a Test is executed, only one result is returned.
//  The convention is
//
//    PASS: No faulty devices indicated
//
//    FAIL: At least one faulty device indicated
//
//  This result is the aggregation of all Probes applied
//  so that, for example, if a Test covers five Modules and the
//  Test returns TRUE, we know at least one of those five devices
//  is faulty.
//
public class Test {

    public enum Result { PASS, FAIL }

    //
    //  ... attributes ...
    //
    String Name;
    double Cost;
```

Test.java

```
ArrayList<Probe> ProbeList;

//
//  ... methods ...
//
Test(String name) {
        this.ProbeList = new ArrayList<Probe>();
        this.setName(name);
   this.setCost(1.0);
}

void setName(String name) {
        this.Name = name;
}

String getName() {
        return this.Name;
}

void setCost(double c) {
   this.Cost = c;
}

double getCost() {
   return this.Cost;
}

void addProbe(Module m, double cp, double cov) {
        Probe pb = new Probe(this,m,cp,cov);
   boolean Inserted = false;
   for (int i = 0; i < this.ProbeList.size(); i++) {
     Probe pbi = this.ProbeList.get(i);
     if (m == pbi.getModule()) {
       // System.out.printf("debug: replacing probe on %s\n",
       //    m.getName());
       this.ProbeList.set(i,pb);
       Inserted = true;
     }
   }
   if ( !Inserted ) {
     this.ProbeList.add(pb);
     m.addProbe(pb);
   }
}

Probe getProbe(int i) {
  return this.ProbeList.get(i);
}

int getProbeCount() {
  return this.ProbeList.size();
}

Test.Result applyTest() {
        Test.Result result = Test.Result.PASS;
        int i;
        int n = this.ProbeList.size();
        for (i = 0; i < n; i++) {
            Probe pb = this.ProbeList.get(i);
            Probe.Result pbres = pb.applyProbe();
            if ( pbres == Probe.Result.FAIL )
                    result = Test.Result.FAIL;
        }
        return result;
}
```

Test.java

```
   ArrayList<Module> getModulesProbed() {
            ArrayList<Module> dlist = new ArrayList<Module>();
            int nProbe = this.ProbeList.size();
            int i;
            for (i = 0; i < nProbe; i++) {
               Probe p = this.ProbeList.get(i);
               Module d = p.getModule();
               dlist.add(d);
            }
            return dlist;
   }

   double getCoverageOn(Module m) {
            int nProbe = this.ProbeList.size();
            double result = 0.0;
            for (int i=0; i < nProbe; i++) {
               Probe pb = this.ProbeList.get(i);
               if ( m == pb.getModule() )
                        result = pb.getFraction();
            }
      if ( result > 1.0 ) {
        System.out.printf("WARNING: coverage on module %s = %f\n",
          m.getName(), result);
      }
            return result;
   }

   //
   //  probabilityPass()
   //  -----------------
   //
   //    Return the probability that a test will pass
   //    based on the coverage of the test and the
   //    a priori probability the covered modules are bad
   //
   double probabilityPass() {
            double result = 1.0;
            ArrayList<Probe> pblist = this.ProbeList;
            for (int i=0; i < pblist.size(); i++) {
               Probe pb = pblist.get(i);
               double b = pb.getModule().getBad();
               double a = pb.getFraction();
               result = result * ( 1.0 - a*b );
            }
            return result;
   }

}
//
//  ==================
//  end package: Test
//  ==================
//
```

A.5 Probe.java

Probe.java

```
//
//  ============================================================================
//  Project: Risk-based Testing Simulation
//           Pfeiffer, Kanevsky, Housel
//           Department of Information Sciences
//           Naval Postgraduate School
//
//  Date:    1 Oct 2008
//  ============================================================================
//
//  package Probe.java
//  ------------------
//
//    Each Probe describes the coverage of a Test on a specific
//    Module, using a center point and fraction (or arc length)
//    to describe the portion of the Module exercised when a
//    specific Test is applied.
//
//    A Test is a collection (ArrayList) of Probes.
//
public class Probe {

    public enum Result { PASS, FAIL }
    //
    //  ... attributes ...
    //
    Module module;
    Test test;
    Coverage coverage;
    //
    //  ... methods ...
    //
    Probe (Test t, Module m, double cp, double f) {
        test = t;
        module = m;
        coverage = new Coverage(cp, f);
    }

    Probe.Result applyProbe() {
            if ( this.module.containsDefect(this.coverage) )
                return Probe.Result.FAIL;
            else
                return Probe.Result.PASS;
    }

    Module getModule() { return this.module; }

    Test getTest() { return this.test; }

    double getFraction() {
        return this.coverage.measure();
    }

    String getCoverageGraphic(int nGraphic) {
        String result = coverage.getCoverageGraphic(nGraphic);
            return result;
    }
}
//
//  ==================
//  end package: Probe
//  ==================
//
```

A.6 Coverage.java

Coverage.java

```
//
//  ============================================================================
//  Project: Risk-based Testing Simulation
//           Pfeiffer, Kanevsky, Housel
//           Department of Information Sciences
//           Naval Postgraduate School
//
//  Date:    1 Oct 2008
//  ============================================================================
//
import java.util.ArrayList;
//
//  package: Coverage.java
//  ----------------------
//
//    The Coverage class is used to model the relationship between
//    the System objects Test and Module.
//
//    The Coverage object represents an arc in which a module may
//    be inspected by a test.
//
public class Coverage {
    //
    //  -----------------------
    //  private class: Interval
    //  -----------------------
    //
    private class Interval {

                double Left;
                double Right;

                Interval(double left, double right) {
                    this.Left = left;
                    this.Right = right;
                }

        void set(double left, double right) {
                    this.Left = left;
                    this.Right = right;
        }

                double measure() {
                    double result = (this.Right - this.Left);
                    return result;
                }

                boolean containsPoint(double p) {
                    if ( this.Left <= p && p <= this.Right )
                            return true;
                    else
                            return false;
                }

    }
    //  -------------------
    //  end class: Interval
    //  -------------------
    //
    //  ---------------------------
    //  private class: IntervalList
    //  ---------------------------
    //
```

Coverage.java

```
private class IntervalList {

        ArrayList<Interval> List;

        IntervalList() {
           List = new ArrayList<Interval>();
        }

        void addInterval(double left, double right) {
           if ( left < 0.0 ) {
                   List.add(new Interval(1.0 + left, 1.0));
                   List.add(new Interval(0.0, right));
           }
           else if ( right > 1.0 ) {
                   List.add(new Interval(left, 1.0));
                   List.add(new Interval(0.0, right - 1.0));
           }
           else
                   List.add(new Interval(left, right));

        } // end addInterval()

        void addArc(double center, double fraction) {
           double left = center - fraction/2.0;
           double right = center + fraction/2.0;
           this.addInterval(left, right);
        } // end addArc()

        Interval get(int k) {
           return List.get(k);
        }

        boolean containsPoint(double point) {

           int nSize = List.size();

           for (int i = 0; i < nSize; i++)
                   if ( List.get(i).containsPoint(point) ) return true;

           return false;
        }

        double measure() {
           double sum = 0.0;
           int nSize = List.size();
           for (int i = 0; i < nSize; i++)
                   sum += List.get(i).measure();
           return sum;
        }

        String getCoverageGraphic(int nGraphic) {

           char[] coverage = new char[nGraphic];
           for (int i=0; i < nGraphic; i++) coverage[i] = '.';

           for (int k=0; k < this.List.size(); k++) {
                   Interval v = this.get(k);
                   int left  = (int) (nGraphic * v.Left + 0.5);
                   int right = (int) (nGraphic * v.Right + 0.5);
                   for (int p=left; p < right; p++)  coverage[p] = 'x';
           }

           String coverageGraphic = new String(coverage);

           return coverageGraphic;
```

Coverage.java

```
            }
    }
    //
    //  ------------------------------
    //  end private class IntervalList
    //  ------------------------------
    //
    //  --------------------------
    //  main body of class Coverage
    //  --------------------------
    //
    IntervalList List;

    Coverage (double centerpoint, double fraction) {
            List = new IntervalList();
            this.List.addArc(centerpoint,fraction);
    }

    double measure() {
            return this.List.measure();
    }

    boolean containsPoint(double p) {
            return this.List.containsPoint(p);
    }

    String getCoverageGraphic(int nGraphic) {
            return this.List.getCoverageGraphic(nGraphic);
    }
    //
    //  end class: Coverage
    //

}
//
//  ======================
//  end package: Coverage
//  ======================
//
```

A.7 Utility.java

Utility.java

```
//
//  ============================================================================
//  Project: Risk-based Testing Simulation
//           Pfeiffer, Kanevsky, Housel
//           Department of Information Sciences
//           Naval Postgraduate School
//
//  Date:    1 Oct 2008
//  ============================================================================
//
import java.io.*;
//
//  package: Utility.java
//  ---------------------
//
//    This is a catch-all class with those subroutines that do not
//    fit neatly into another object
//
public class Utility {

    public static double entropy(double p) {
            double h = 0.0;
            if ( p == 0.0 || p == 1.0 )
                h = 0.0;
            else {
                h += -p * Math.log(p);
                h += -(1.0 - p) * Math.log(1.0 - p);
                h /= Math.log(2.0);
            }
            return h;
    }

    public static void dumpArrayToFile(double[] list, String outfile) {
            try {
                PrintWriter output =
                        new PrintWriter(new FileWriter(new File(outfile)));
                int nSize = list.length;
                for (int i = 0; i < nSize; i++)
                        output.printf("%f\n",list[i]);
                output.close();
            }
            catch(IOException ioException) {
                System.out.println("Error writing file");
                System.exit(0);
            }
    }
}
//
//  ====================
//  end package: Utility
//  ====================
//
```

A.8 Logger.java

Logger.java

```
//
//  ============================================================================
//  Project: Risk-based Testing Simulation
//           Pfeiffer, Kanevsky, Housel
//           Department of Information Sciences
//           Naval Postgraduate School
//
//  Date:    1 Oct 2008
//  ============================================================================
//
import java.io.*;
import java.util.Date;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
//
//  package: Logger.java
//  --------------------
//
//    The Logger class is a simple utility object for writing all
//    messages with date-time stamp into a single log file.
//
public class Logger {

   private static String OutputFile;
   private static boolean Append = false;
   private static DateFormat dateFormat;
   private static PrintWriter handle;

   Logger(String filename) {
     this.OutputFile = filename;
     this.dateFormat =
       new SimpleDateFormat("yyyy-MM-dd HH:mm:ss SSS");
     openLog();
   }

   private void openLog() {
     String outfile = this.OutputFile;
     try {
       this.handle =
         new PrintWriter(new FileWriter(new File(outfile),this.Append));
     }
     catch(IOException ioException) {
       System.err.println("Error opening log file");
       System.exit(0);
     }
   }

   public void write(String format, Object... args) {
     Date date = new Date();
     String outstr = String.format(format,args);
     String logmsg =
       String.format("%s %s\n",this.dateFormat.format(date),outstr);
     this.handle.write(logmsg);
     this.handle.flush();
     this.Append = true;
   } // end write()
}
//
//  ===================
//  end package: Logger
//  ===================
//
```

2003 - 2008 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 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 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 Eared 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