Proceedings of the fifth workshop on software assessment

Michael, James Bret
Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/515

Downloaded from NPS Archive: Calhoun
NAVAL
POSTGRADUATE
SCHOOL
MONTEREY, CALIFORNIA

Proceedings of the
Fifth Workshop on Software Assessment

by
Dale S. Caffall
James B. Michael
Jeffrey M. Voas
22 March 2006

Approved for public release; distribution is unlimited.
Prepared for: IEEE Reliability Society
This report was prepared for the Institute of Electrical and Electronic Engineers (IEEE). Funding in support of the workshop and preparation of this report was provided by the IEEE Reliability Society and the Naval Postgraduate School. The workshop was held in conjunction with the 2005 IEEE International Symposium on Software Reliability Engineering.

Reproduction of all or part of this report is authorized.

This report was prepared by:

James Bret Michael, Associate Professor
Department of Computer Science

Reviewed by: 

Released by:

Peter J. Denning, Chairman and Professor
Department of Computer Science

Leonard A. Ferrari
Associate Provost and Dean of Research
The Fifth Workshop on Software Assessment was held in conjunction with the IEEE Sixteenth International Symposium on Software Reliability Engineering. The workshop serves as an annual forum for practitioners and educators to discuss software assessment. This document contains the proceedings of the workshop.
THIS PAGE IS INTENTIONALLY LEFT BLANK
Proceedings of the
Fifth Workshop on Software Assessment

Gleacher Center, University of Chicago, Chicago, Illinois
November 8, 2005

Held in conjunction with the IEEE International
Symposium on Software Reliability Engineering

Sponsored by
Naval Postgraduate School
IEEE Reliability Society
THIS PAGE IS INTENTIONALLY LEFT BLANK
Preface

The Fifth Workshop on Software Assessment was held in conjunction with the IEEE Sixteenth International Symposium on Software Reliability Engineering. The workshop serves as an annual forum for practitioners and educators to discuss software assessment.

We are grateful for the financial and organizational support provided by the IEEE Reliability Society and the Naval Postgraduate School.

Jeff Voas, Workshop Co-Chair
Butch Caffall, Workshop Co-Chair
Bret Michael, Publications Chair
# Table of Contents

A Discrete Lognormal Model of Defect Occurrence Counts with Applicability to Network Security Defects, *Robert E. Mullen* and *Swapna S. Gokhale*  

Homeland Security Airport Security Model, *Norman F. Schneidewind* 

Quantifying Quality of Non-Functional Quality Attributes using Customer Survey Metrics, *Vanitha Sugumaran* and *Janet Drake* 

Web Services Testing for Reliability Assessment, *Jia Zhang* and *Zongwei Luo* 

On the Design of Safety Kernels for Complex Real-Time Systems, *J. Bret Michael* 

Analyses of Milestone Readiness Levels in Software IV&V, *Ganesh J. Pai*, *Joanne Bechta-Dugan*, and *Khalid Leeteef* 

A Discrete Lognormal Model of Defect Occurrence Counts
With Applicability to Network Security Defects

Robert E. Mullen
Cisco Systems
Boxborough, MA 01719
bomullen@cisco.com

Swapna S. Gokhale
University of Connecticut
Storrs, CT 06269
ssg@enr.uconn.edu

Abstract

Existing arguments and evidence suggests that the distribution of occurrence rates of software defects is lognormal and that the first occurrence times of defects follow the Laplace transform of the lognormal. We extend this research to hypothesize and confirm that the distribution of occurrence counts of software defects, including security related defects, follow the Discrete Lognormal.

In this paper we summarize previous evidence for the lognormal, place the most recent evidence in context, and call attention to the unified quantified perspective the lognormal affords. We outline additional applications of the lognormal to important problems of software engineering and suggest enabling research.

Keywords: Discrete Lognormal, Lognormal, Poisson-Lognormal, Software Reliability, Software Maintenance.

1. INTRODUCTION

This paper provides context for the slides we presented at WOSA, Chicago, 2005. The slides present a summary of research demonstrating the natural and central role of the lognormal in software.

Engineering disciplines are often based on deriving a variety of properties of constructed objects from a few underlying physical processes. We suggest several observable properties of software systems are in fact related, being grounded in the conditional nature of software execution. This leads to the emergence of a lognormal distribution of rates of events in software systems --- including failure rates of defects. From the lognormal distribution of rates we can derive the distribution of first-occurrence times and the distribution of occurrence counts of defects.

The derivations and evidence are summarized in the slides and references. In this paper we also propose opportunities for further research and wider application of the lognormal. Some provide ways to increase confidence in applying the lognormal while others attempt to unlock the value of the insights by applying them to pressing software engineering problems.

2. LOGNORMAL DISTRIBUTION OF RATES

Whether one views software from the point of view of code execution, state-space, or an operational profile, the probability of any given event is determined by a multiplicative process. It is proportional to the product of the conditioned probabilities of its preconditions. [M98b] In typical systems there are sufficient factors (or preconditions) to bring the multiplicative form of the Central Limit Theorem into play. [AB69] Thus a lognormal distribution of event rates emerges in a natural way from all three analytical perspectives.

The proposed lognormal failure rate distribution was validated [M98b] by analyzing careful studies of failure rates of faults previously published by IBM [A84] and Boeing [NS82] [NSS84]. Bishop and Bloomfield [BB03] measured both the distribution of execution rates of code blocks and the distribution of failure rates of faults in the 10,000 line PREPRO application of the European Space Agency. Both were well fit by the lognormal.

We believe there are several directions in which further research may be fruitful. It would be instructive to examine a large, detailed operational profile or a specific state-space for additional confirmation by direct rate measurement. It would be especially valuable to understand how the parameters of the lognormal, especially sigma, are affected by the nature and size of the systems.

3. RELIABILITY GROWTH MODEL

Many have recognized existing models as being either too optimistic or too pessimistic. The dividing line is related, if not identical, to the division between finite and infinite failure models. The problem cannot be escaped by combining or weighting models to create super models or choosing which model to use on the fly. The accuracy and tractability of each model are more apparent than actual, because the real uncertainties relating to predicting software failure rates have been moved out of the models and into the model selection process.
Miller [M85] pointed out the mathematical transformation from a rate distribution to a first occurrence time (discovery time) is equivalent to the Laplace Transform of the rate distribution. If failure rates are lognormal then the distribution of first-failure-times is equivalent to the Laplace Transform of the lognormal.

Mullen [M98a] derived the Lognormal Software Reliability Growth Model by approximating the Laplace Transform of the lognormal. This model was validated using Stratus Computer data as well as data gathered by Musa. Increased testing should increase code coverage according to the same function. Gokhale and Mullen [GM04, GM05] showed the model fits four types of code coverage growth as the number of tests increase. This was done by repetitive-run experiments which Miller proposed as the surest way to uncover the form of reliability growth.

It would be useful to know the extent to which applying priors to the lognormal parameters improve prediction in real-life situations. This may be a very fruitful approach since according to [BB03] sigma seems to change slowly with size and complexity, and since there is extensive literature on estimating $N$ (the number of defects). A study of the accuracy of predictions, perhaps using sequential likelihood methods, would also be valuable.

4. OCCURRENCE COUNT MODEL

A second derivation is to compute the distribution of occurrence counts of defects. If failure rates are lognormal then the distribution of occurrence counts of defects can be derived and shown to follow the Discrete Lognormal defined in [JKK93].

There are several related questions needing further research. Will the result still hold if fix-times are variable? What are the uncertainties in the measurements of the LN parameters? How closely do the parameters of the lognormal agree when measured by direct rate, via the LT-LN in SWRGM, or via the D-LN occurrence count data? The sigma values, generally less than 2.0, seem low for the large size of these software products [MG05]; is this effect of heavy prior testing? Can knowledge of the form of the occurrence rate distribution and its parameters be used to quantitatively evaluate defect repair/ship strategies or even deduce optimal ones?

5. OTHER OPPORTUNITIES

The lognormal model has several advantages over earlier models. Its genesis is apparent since the mathematical form of the model is directly traceable to the structure of the subject of the model. This mathematical link between software structures and the lognormal distribution is based on the central limit theorem, a profound result of probability theory. The assumptions about software systems on which the model is founded are equivalent or similar to those successfully used within many sub-disciplines of software engineering. The lognormal distribution is one that has been applied in reliability modeling as well as a variety of other disciplines. Most importantly, the lognormal is very well supported by previous studies of the failure rate distributions in both laboratory and commercial environments.

There are many opportunities for future research. Several were noted above, especially those which re-validate or apply either the rate distribution or one of the derived distributions. In this section we take a broader unified view. If the lognormal appears to be nearly ubiquitous then the challenge is no longer to find it but rather to apply it.

First, studies similar to those done with other models are needed. These include studies of the ability of the model to predict future fault counts, the application of the model to determination of the optimal release time, optimum maintenance strategies, and the optimization of test strategies. Robust solutions of each of these problems depend on knowing the form of the distribution of the failure rates of defects.

Second, there is the opportunity to take advantage of the fact that the lognormal has its roots in the complexity of software states, program flows, and operational profiles, and to try to use such information to estimate the parameters in advance of execution. It would be very useful to have methods for estimating the parameters, especially $\mu$, given preliminary information about states, flows, or operational profiles. Although the “true” operational profile is usually more complex than that of any analyst, the analyst’s operational profile may be used to set a lower bound on the variance of the log-rates of operations within the actual system. Under what conditions is this bound tight or loose?

Third, techniques can be exploited to make use of information from prior releases or similar products. It is likely that similar systems will have similar parameters, allowing real use of prior information. Additional theory and experiment is needed to develop reasonable quantitative guidelines and expectations.

The analytical properties of the lognormal and its related functions also need study. What is the effect of prior testing or changes in the operational profile? How much data is needed in order to ensure predictions of a given level of accuracy? Can the lognormal perspective illuminate the reasons certain reliability growth models work in some situations and not others? We have seen the Discrete Pareto is close competitor to the Discrete Lognormal in our occurrence-count data; it would be useful to know the conditions under which that holds.

What is especially promising is the potential to share information from structural knowledge (or size), operational profile, reliability growth, occurrence counts, and so on, when determining parameters, and then being able to apply those parameters with additional confidence.

REFERENCES


A Discrete Lognormal Model of Defect Occurrence Counts
With Applicability to Network Security Defects

WOSA 2005 Chicago

Robert Mullen
Cisco Systems
Boxborough MA
bomullen@cisco.com

Swapna Gokhale
University of Connecticut
Storrs CN
ssg@engr.uconn.edu

Outline

• Introduction and Origins
  • Operational Profile, Program Flow, State Space

• Lognormal applications and evidence
  • Rates, SRGM, Test Coverage, Occurrence Counts

• Occurrence Counts
  • Discrete-LN, Data Collection, Security Defects
**Relationship to Other Disciplines, Prediction, and Software Structures**

Conventional reliability theory does not apply to software.
Dick Hamlet (1992)

Inference and prediction steps will be easier if the class of possible models can be restricted a priori, i.e. before testing or, ideally, before the program is written.
D. R. Miller (1985)

An underlying fault/failure model is central to a better understanding of reliability amplification,... the heart of the theoretical problem is finding a proper home for the failure rate, and we believe that it should be assigned to the program computation and its data-state values, instead of to points in its input domain.
Dick Hamlet and Jeff Voas (1993)

---

**Event Taxonomy**

<table>
<thead>
<tr>
<th>COMMON ~ 90%</th>
<th>LESS COMMON ~ 9%</th>
<th>RARE &lt; 1%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read/write</td>
<td>Open/close</td>
<td>Create/delete</td>
</tr>
<tr>
<td>Local</td>
<td>Nearby</td>
<td>Distant</td>
</tr>
<tr>
<td>By book</td>
<td>User error</td>
<td>UBD</td>
</tr>
<tr>
<td>IO works</td>
<td>IO error</td>
<td>IO removed</td>
</tr>
<tr>
<td>Queue modest</td>
<td>Queue long</td>
<td>Queue overflow</td>
</tr>
<tr>
<td>Resource available</td>
<td>Resource wait</td>
<td>Resurse gone</td>
</tr>
<tr>
<td>Process OK</td>
<td>Process slow</td>
<td>Process died</td>
</tr>
<tr>
<td>7200</td>
<td>GSR/12000</td>
<td>ESR/10000</td>
</tr>
<tr>
<td>Monitoring</td>
<td>Debugging</td>
<td>Installing</td>
</tr>
<tr>
<td>IP</td>
<td>MPLS VPN</td>
<td>IPSEC</td>
</tr>
<tr>
<td>Normal running</td>
<td>Start/shutdown</td>
<td>Crash/recovery</td>
</tr>
</tbody>
</table>

Most customer found defects are corner cases --- combinations of two or more events from the rightmost columns. These may be < .01%
Although they are less common or rare, they are essential to the product. The customer already paid for them to work ~ 100%.
Multiplicative origin of failure rates

• E.N. Adams: (in production code) “the typical design error requires very unusual circumstances to manifest itself, possibly in many cases the coincidence of very unusual circumstances.”

• This suggests the failure rates of faults (both rare and common) are determined by a multiplicative process.

• The more factors needed to cause a failure and the more rare they are, the more rare the failure is.

• As the number of factors increases the logarithm of the product approaches the Gaussian or Normal distribution.

\[ \prod_{j=1}^{n} X_j \lim_{n \to \infty} \]

Iyer and Rossetti: “During periods of stress or uncommon workload patterns, rarely used code can be executed, leading to the discovery of errors.”

\[ X_j \]

Background of the lognormal

• A random variable X is lognormally distributed if log (X) is normally distributed.

• The lognormal is used in many other disciplines including hardware reliability engineering.

• Why hasn’t it been used for software before?
  • T = 0 is a problem if used for failure times.
  • Some attempted use for interfailure times.
  • Here we use it for failure rates.
Alternative Conditions for Central Limit Theorem

If i.i.d. and $X_j$ has finite second moment.
  – Lindeberg-Levy

If factors not identically distributed . . .
  – Liapounoff, see also Petrov

If factors not independent . . .
  – Loeve

If variable number of factors . . .
  – Feller

Interpretation of lognormal parameters

• $N$ is the number of distinct events (code blocks, defects, etc).
  $N$ can be affected by the software process and the size of the system.

• $\mu$ is the mean of the log-rates.
  $\mu$ will increase if system speed increases.

• $\sigma$ is the standard deviation of the log-rates.
  $\sigma$ increases slowly with size. Proportional to square root of depth of conditionals or to square root of log2 number of blocks (for example).
*Ramifications of lognormal

Multiplicative Rates

\[ n \prod_{j=1}^{n} X_j \]

States, Usage, Code

Limiting Distribution = Lognormal

Trouble Tickets = Discrete-LN

Test Strategy
Statistical, but with an accelerated profile: run a much higher percentage of less common triggers to drive interactions.

Ten times the rare rates will find rare-rare interactions 100 times as fast.

Equivalent to Heat/Power/Temp “corner testing” of HW.

SRGM = Code Coverage = Laplace Transform of LN

Release Strategy
Is it ready? Which is best?

Repair Strategy
Risk vs. Benefit?

Evidence for the lognormal

- LN Defect Rate Distribution
  - 9 IBM products (Adams-1984)
  - 12 Boeing experimental programs (Nagel-)
  - European Space Agency PREPRO application (Bishop-2003)

- LN Block Execution Rate Distribution
  - PREPRO (Bishop-2003)

- Laplace Transform-LN, Reliability Growth Model, Code Coverage
  - 10 “Musa” data sets (Mullen-1998)
  - 2 Stratus Releases (Mullen-1998)
  - 4 Code Coverage Metrics SHARPE (Gokhale-2004, 2005)

- Discrete-LN, Occurrence Count Distribution
  - 12 Cisco subsets by year, product, severity, ODC (ISSRE-2005)
  - 3 Security related subsets (Mullen-2005, QOP)
Range of sigma values

Quantitative change $\Rightarrow$ qualitative change.
Variation of sigma changes behavior dramatically.

$\sigma \sim 0$ Identity of LNET with BET.
$\sigma \sim 1$ Intermediate behavior by LNET.
$\sigma \sim 2$ Affinity of LNET with LPET in some cases.
$\sigma > 3$ Novel and useful long tailed distribution, heavier than LPET for a long time.
Musa, Juhlin: Operational Profile

- Compute probabilities by multiplication.
- Assumed probabilities approx. independent.

Call: internal, abbreviated, answered, put on hold.

• Probability is \(0.35 \times 0.70 \times 0.60 \times 0.10 = 0.0147\)

From software structures to rates

We can color the nodes in the tree of an operational profile according to the log of their rates.

If we group them by the log of their rates we approach a normal distribution.
Avritzer & Larson: Telecommunications

• Handles calls of 5 types, each with arrival and service times.
• State Vector: \((n_1,n_2,n_3,n_4,n_5)\)
• Kleinrock independence approximation of \(n(i)\)
• Probability of each state equals the product of probabilities of \(n(i)\) for each call.
• Tested most probable states for great gain in efficiency.

Evidence for the Fault Rate Distribution

• Reality Check: Actual customer usage.
  • Use Adams study of 9 IBM products
  • indirect, commercial, large systems, many bugs.

• Miller: “Best is repetitive-run experiments.”
  • Use 12 experiments by Nagel et al.
  • direct, academic, small systems, few bugs.
Adams’ Model of Software Defect Occurrences

- Each defect, in a given product, has a characteristic rate $R$.
- Each occurrence of that defect is an event in a Poisson process, with rate $R$.
- Running time of the Poisson process is the cumulative time of all users.
- Software defects have different rates, some are more “virulent.”
- The error rate of the product is the sum of the error rates of all defects.
- The rate of a given defect remains unchanged until it is removed.

Therefore the encountering of defects through time depends on:

- The number of defects in the product.
- The distribution of rates, $R$, of the defects, in the product
- The rate of use of the product, over time.
- The schedule of installing fixes for defects after they are discovered.

Adams: Example

Figure 1

Lognormal Fitted to Adams Product 2.6

<table>
<thead>
<tr>
<th>Lognormal Fitted to Adams Product 2.6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relative Num Faults</td>
</tr>
<tr>
<td>35</td>
</tr>
<tr>
<td>30</td>
</tr>
<tr>
<td>25</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Low failure rate of faults</th>
<th>High</th>
</tr>
</thead>
</table>

[Graph showing lognormal fitting to data]
Adams’ data and some lognormal fits

- Relative number of defects is normalized to 100 by Adams. We extrapolate N to lower rates in the table.
- (The shape of the low-rate left-side has been determined by other studies.)

<table>
<thead>
<tr>
<th>Prod#</th>
<th>Sigma</th>
<th>Mu</th>
<th>N</th>
<th>R²</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>3.267</td>
<td>-6.220</td>
<td>309</td>
<td>.994</td>
</tr>
<tr>
<td>2.2</td>
<td>3.508</td>
<td>-6.752</td>
<td>355</td>
<td>.994</td>
</tr>
<tr>
<td>2.3</td>
<td>3.442</td>
<td>-6.488</td>
<td>319</td>
<td>.995</td>
</tr>
<tr>
<td>2.4</td>
<td>2.565</td>
<td>-4.642</td>
<td>191</td>
<td>.994</td>
</tr>
<tr>
<td>2.5</td>
<td>3.501</td>
<td>-6.859</td>
<td>369</td>
<td>.991</td>
</tr>
<tr>
<td>2.6</td>
<td>2.585</td>
<td>-4.443</td>
<td>185</td>
<td>.999</td>
</tr>
<tr>
<td>2.7</td>
<td>3.301</td>
<td>-6.272</td>
<td>313</td>
<td>.994</td>
</tr>
<tr>
<td>2.8</td>
<td>3.472</td>
<td>-6.185</td>
<td>297</td>
<td>.996</td>
</tr>
<tr>
<td>2.9</td>
<td>2.321</td>
<td>-3.883</td>
<td>157</td>
<td>.998</td>
</tr>
</tbody>
</table>

ave R² = .995

Repetitive Run Experiments

- 12 programs, several programmers and specs.
- Total 97 faults, thousands of executions.
- The failure rate of each fault was determined by running input cases randomly based on an operational profile.
- Defects were not fixed until the failure rate of each fault was determined.
Repetitive Run
Shapiro-Wilks Test for normality of small samples

- Shapiro-Wilks value
- .10 level of significance

Value vs Number of Faults

good
poor

Repetitive Run: LN vs Gamma

Lognormal significantly better: 8 experiments

Log-Likelihood

Gamma slightly better: 4 experiments
Summary of Evidence on Rate Distribution

Adams Field Data
• Visually good fit.
• Very high values for correlation coefficient.
• Lognormal fits every case better than power-law.

Repetitive Run Experiments
• Pooled data appears lognormal.
• Log-rates pass test for normality.
• The lognormal is astronomically more likely to generate the data than is the gamma.

Revisit Earlier Statements
• Conventional reliability calculations and methods do work for software.
• Whether the home of the failure rate is in the input space, state space, or code paths, the mathematical form is, in the limit, the same.
• The lognormal distribution of failure rates of software faults can provide a solid basis for additional modeling.
Software Reliability Growth Modeling

- Different models --- even finite and infinite --- can appear arbitrarily similar over a finite interval.
- In practice it can be impossible to decide between two models on the basis of one unreplicated experience.
- "Inference and prediction steps will be easier if the class of possible models can be restricted a priori, i.e. before testing or, ideally, before the program is written."

---

Miller's statements imply: Use all available data and knowledge of software systems to determine the form of the model. Use data from a specific debugging experience to determine the parameters of the model.
Transformation from Rates to Times

- Defects with higher failure rates occur sooner, on average.
- X-axes are log-scaled. Each rate-class is color coded.
  - Black and red are the highest rate, yellow and brown are lowest.
- On the left, the Lognormal distribution of rates.
- On the right, the distribution of first failure times.
  - It is not Lognormal, it is the Laplace transform of the Lognormal.

Mean Functions for First Failures

**Lognormal failure rate distribution**

\[
dL(\lambda) = \frac{1}{\lambda \sigma \sqrt{2 \pi}} e^{-(\ln(\lambda) - \mu)^2 / 2\sigma^2} d\lambda
\]

**Mean number before time** \( t \)

\[
M(t) = N - N \cdot \int_{\lambda=0}^{\infty} \exp(-\lambda t) dL(\lambda)
\]

**Mean rate at time** \( t \)

\[
m(t) = N \cdot \int_{\lambda=0}^{\infty} \lambda \cdot \exp(-\lambda t) dL(\lambda)
\]

The mathematical form of the mean function is formally equivalent to the Laplace Transform of the lognormal.
From Rates to Cumulative Faults

- Rate-classes are color coded. X-axes are log-scaled.
- On the left, the Lognormal distribution of rates.
- On the right, the cumulative distribution of first failure times, which equals the cumulative faults discovered.
- More rate-buckets creates a more exact approximation to the incremental and cumulative functions.

Reliability Growth: Stratus Data

- Manufacturer of Fault Tolerant Mini-computers.
- Used 24x7 in banks, brokerage, and telecom.
- Sites monitored by Remote Service Network.
- The execution time for each release is known.
- Releases presented were unchanged over life.
- For each week we know
  - system hours on the release
  - number of defects reported for the first time against that release.
- Thus we can determine the first failure time of each fault, thereby the number of first-failures as a function of cumulative execution time.
CODE COVERAGE GROWTH
Experimental Subject and Tools

- **SHARPE Application was “System under Test”**
  - Developed at Duke, solves stochastic models
  - Real program, 35,412 lines of C code in 29 program files

- **Test Suite for SHARPE**
  - 735 test cases achieve 93.5% Block coverage (275 tests suffice)

- **Test Instrumentation Tools**
  - Telcordia SW Visualization and Analysis Tool Suite (TSVAT)
  - Automatic Test Analyzer for C (ATAC)
  - Instruments code, executes tests, and measures coverage

- **Procedure**
  - Replicated test sequence (randomized) 10 times

**share.c cumulative decision coverage**
(share.c is just one program in SHARPE)

- Linear chart shows problems with exponential and Log-Poisson.
- Log chart shows exponential is far off the mark, in this case.
Entire SHARPE : (blocks)
- Left: cumulative % coverage vs log test count. The 10 replications and their average are also shown for each interval. Much less noisy.
- Right: incremental % coverage vs log test count. The s.d. of the 10 runs are shown at each point.

Whole of SHARPE -- four coverage metrics
- Shows percent incremental coverage per test for four coverage metrics (block, decision, P.uses and C.uses).
- For each metric the lognormal model comes closer to the data than the log-Poisson.
SHARPE SUMMARY
Note value of replicated test sequences

<table>
<thead>
<tr>
<th></th>
<th>One file</th>
<th>Whole application</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>~ 3,000 LOC</td>
<td>~ 30,000 LOC</td>
</tr>
<tr>
<td>Single Test Sequence</td>
<td>Very noisy. Other models sometimes fit data as well as lognormal.</td>
<td>Less noisy. Lognormal always significantly better.</td>
</tr>
<tr>
<td>Replicated Test Sequence (10X)</td>
<td>Less noisy. Lognormal is significantly better.</td>
<td>Overwhelming support for lognormal (vs exp or log-Poisson models)</td>
</tr>
</tbody>
</table>

Defect Occurrence Counts

- At Cisco, a Trouble-ticket is written when a defect occurs at a customer. From them we can determine and fit the number of encounters or occurrences of each defect.
- We studied the distribution of Trouble-tickets among defects written within a given year.
  - How many defects have one ticket?
  - How many have two, three … ?
  - How many have none? (Can’t observe, but can estimate.)
- Assumption: over the interval studied, fixes are not put in service.
*The distribution of defect occurrences is Discrete-Lognormal (Poisson-Lognormal)*

- Each defect has a rate $\lambda$ generated from the lognormal distribution.
- Within a time interval, the number of occurrences of an unrepaired defect having rate $\lambda$ is distributed as a Poisson random variable with rate $\lambda$.
- The number of defects with $X$ occurrences is the sum (over all rates) of the probabilities of getting exactly $X$ occurrences.
- The exact form of the occurrence-count distribution is simply that of a mixed Poisson distribution in which the rate is distributed according to the lognormal.

$$\text{Poisson}(\lambda) \bigodot \text{Lognormal}(\mu, \sigma)$$

- This is the Discrete Lognormal or Poisson-Lognormal [Johnson, 1993].
  - Related to the Lognormal Software Reliability Growth Model
  - Differs from log-Poisson SW reliability growth model, which is not a distribution

---

**From software rates to number of occurrences**

**Contributions of rate groups to occurrence-counts**

Assuming 50 time units. Note defects in bucket zero did not occur yet. The overall distribution is the sum of Poisson distributions of the various colors or rates
Example: three years, large system

First Year

Second Year

Third Year

Year 2: LN and Pareto

Example: by Product

- Data format: for product T, about 9% of the bugs had 3 Tickets. About 1% had 9 Tickets.
- Three very different Products, each millions of LOC
- Product W is the oldest and in the most mature market.
- Charts show percentages in each count-bucket. Top is data, below is fitted P-LN. Y-axis is log %.
- Below, we compute the rate per-year (not per device-year).

<table>
<thead>
<tr>
<th>Product</th>
<th>Tickets per bug</th>
<th>LN sigma</th>
<th>LN mu</th>
<th>Mean Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>2.9</td>
<td>2.1</td>
<td>-2.3</td>
<td>.90</td>
</tr>
<tr>
<td>S</td>
<td>2.4</td>
<td>3.3</td>
<td>-7.0</td>
<td>.20</td>
</tr>
<tr>
<td>W</td>
<td>1.6</td>
<td>1.8</td>
<td>-3.3</td>
<td>.18</td>
</tr>
</tbody>
</table>
Example: by Severity

- The more trouble tickets a defect generates, the more likely it is to be classified S1 (critical) or S2 (severe). Moderate defects (S3) tend to have 3/4 as many tickets.
- From the lognormal perspective, the spread in rates of S1 defects is also much larger, sigma = 3.5 rather than 2.0 like S2 and S3.
- The wider spread in S1 rates suggests high rate, as well as high impact, affect classification.
- Charts show percentages in each count-bucket. Top is data, below is fitted P-LN. Y-axis is log-scale.

Example: ODC Origin

(Base Code, New code, Bad Fix)

- Top chart shows both data and fit for Defects with ODC Origin: BadFix. This has actual counts for selected product. Fit and Data are better (higher counts) for New and Base code.
- On lower chart, note fitted curves are similar, consistent with Adams.
- Note defects with a higher number of incidents are less common among Base Code defects and more common among BadFix defects. BadFixes generate 50% more tickets per defect.
- Makes sense: the shorter the prior exposure of changed code, the more chance high-rate defects have survived.
# Occurrence Counts: Parameters (Four sets of three)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Product R 3 Years</th>
<th>Product R, Year 1 ODC Age(Origin)</th>
<th>Other Products</th>
<th>Product R, Year 1 Defect Severity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Year 1</td>
<td>Year 2</td>
<td>Year 3</td>
<td>Base Code</td>
</tr>
<tr>
<td>LN sigma</td>
<td>2.03</td>
<td>1.88</td>
<td>1.92</td>
<td>1.66</td>
</tr>
<tr>
<td>LN mu</td>
<td>-2.96</td>
<td>-2.80</td>
<td>-3.17</td>
<td>-1.77</td>
</tr>
<tr>
<td>Pareto a</td>
<td>0.984</td>
<td>1.067</td>
<td>1.122</td>
<td>1.036</td>
</tr>
<tr>
<td>Pareto k</td>
<td>0.155</td>
<td>0.16</td>
<td>0.141</td>
<td>0.239</td>
</tr>
<tr>
<td>d.f.</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>10</td>
</tr>
<tr>
<td>D-LN Chi-square</td>
<td>24.61</td>
<td>35.25</td>
<td>26.41</td>
<td>10.82</td>
</tr>
<tr>
<td>D-Pareto Chi-square</td>
<td>31.20</td>
<td>59.93</td>
<td>28.45</td>
<td>23.33</td>
</tr>
<tr>
<td>D-Pareto Signif.</td>
<td>n.s.</td>
<td>.0001</td>
<td>n.s.</td>
<td>.01</td>
</tr>
<tr>
<td>Defects studied</td>
<td>&gt;1000</td>
<td>&gt;1000</td>
<td>&gt;1000</td>
<td>&gt;1000</td>
</tr>
<tr>
<td>Tic/Defect observed</td>
<td>2.44</td>
<td>2.20</td>
<td>2.05</td>
<td>2.51</td>
</tr>
<tr>
<td>Tic/Defect LN calc.</td>
<td>2.44</td>
<td>2.22</td>
<td>2.05</td>
<td>2.52</td>
</tr>
<tr>
<td>Mean Rate LN calc.</td>
<td>0.41</td>
<td>0.36</td>
<td>0.26</td>
<td>0.68</td>
</tr>
</tbody>
</table>

# QOP (Quality of Protection)

Software Defects and Security Defects

- QoP will inevitably be affected by software defects.
- Strategy: attempt to apply ordinary software reliability engineering to security related defects as one step toward QoP.
- Reason & evidence imply software defect rates are lognormal and discovery times follow Laplace transform of lognormal.
- Lognormal rates imply Discrete-LN occurrence counts.
- Hypothesis: security defect occurrence counts follow D-LN.
- Three classes of security defects are shown to follow D-LN, confirming both the strategy and the hypothesis.
Problem and motivation

• Assessment of Quality of Protection (QoP) is predominantly qualitative. Unlike QOS or Reliability...

• Two-fold relationship between software defects and security:
  – Defect may be exploited directly for a security breach.
  – May manifest as a field failure, and repair of it may introduce another defect which may be exploited and cause QoP degradation.

• Occurrence rates of software defects in general and security defects in particular will be an important component of QoP

Problem and motivation (contd..)

• Occurrence rates of defects varies widely, from rarely to pervasive.

• Empirical observations and theoretical justification suggests that the:
  – Distribution of defect occurrence rates is lognormal.
  – Distribution of defect occurrence counts is Discrete-Lognormal.

• Confirm the D-LN hypothesis by analyzing three sets of data.

• Link results from prior studies of software reliability growth, test coverage, defect failure rates and code execution rates to occurrence rates of security defects affecting QoP.
* Percentages of Defects with Specific Number of Tickets per Defect

- Three sets of data relating to security related defects.
- Shows percent of defects with a given number of tickets.
- Col 2: Defects with ODC Impact value of Security (within large system).
- Col 3: All defects within a security related product.
- Col 4: All defects in a suite of security related products

<table>
<thead>
<tr>
<th>Tickets per defect</th>
<th>ODC Security</th>
<th>Security Product</th>
<th>Security Suite</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>62.34</td>
<td>61.62</td>
<td>58.95</td>
</tr>
<tr>
<td>2</td>
<td>15.58</td>
<td>18.38</td>
<td>16.32</td>
</tr>
<tr>
<td>3</td>
<td>6.49</td>
<td>5.41</td>
<td>10.00</td>
</tr>
<tr>
<td>4</td>
<td>5.19</td>
<td>6.49</td>
<td>3.68</td>
</tr>
<tr>
<td>5</td>
<td>1.30</td>
<td>1.08</td>
<td>4.21</td>
</tr>
<tr>
<td>6</td>
<td>3.90</td>
<td>1.08</td>
<td>1.58</td>
</tr>
<tr>
<td>7</td>
<td>1.30</td>
<td>0.00</td>
<td>1.05</td>
</tr>
<tr>
<td>8</td>
<td>1.30</td>
<td>0.00</td>
<td>0.53</td>
</tr>
<tr>
<td>9</td>
<td>0.00</td>
<td>1.08</td>
<td>0.53</td>
</tr>
<tr>
<td>10</td>
<td>0.00</td>
<td>1.62</td>
<td>1.05</td>
</tr>
<tr>
<td>11-40</td>
<td>2.60</td>
<td>3.24</td>
<td>2.12</td>
</tr>
</tbody>
</table>

Analysis and discussion

- Alternative model: Pareto rates (for $\lambda > k$)
  \[ N*(1-(k/\lambda)^\alpha) \]
- Model fitting
- Model comparison
- Comparative analysis of defect subsets
Data and fitted discrete-lognormal for QOP subjects.

<table>
<thead>
<tr>
<th>Tickets</th>
<th>ODC Security Data</th>
<th>Security Product Data</th>
<th>Security Suite Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ODC Security DLN</td>
<td>Security Product DLN</td>
<td>Security Suite DLN</td>
</tr>
<tr>
<td>1</td>
<td>148</td>
<td>48.08</td>
<td>114</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>12.17</td>
<td>34</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>5.31</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2.94</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1.85</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>1.27</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0.92</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0.69</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0.54</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0.43</td>
<td>3</td>
</tr>
<tr>
<td>&gt;10</td>
<td>2</td>
<td>2.45</td>
<td>6</td>
</tr>
</tbody>
</table>

*Lognormal results*

- Lognormal is not rejected at .05 level by this data.
*Relationship to Pareto

- Pareto is also not rejected.
- Pareto chi-square are slightly larger (worse) than LN but there is no significant difference.

<table>
<thead>
<tr>
<th></th>
<th>ODC Security</th>
<th>Security Product</th>
<th>Security Suite</th>
</tr>
</thead>
<tbody>
<tr>
<td>LN sigma</td>
<td>2.63</td>
<td>2.40</td>
<td>1.75</td>
</tr>
<tr>
<td>LN mu</td>
<td>-5.1</td>
<td>-4.27</td>
<td>-2.23</td>
</tr>
<tr>
<td>Pareto a</td>
<td>.832</td>
<td>.943</td>
<td>1.045</td>
</tr>
<tr>
<td>Pareto k</td>
<td>.057</td>
<td>.111</td>
<td>.187</td>
</tr>
<tr>
<td>d.f.</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>LN Chi-square, LLH</td>
<td>-24.09</td>
<td>-39.28</td>
<td>-31.33</td>
</tr>
<tr>
<td>Pareto Chi-square,</td>
<td>.95</td>
<td>6.05</td>
<td>5.45</td>
</tr>
<tr>
<td>LLH</td>
<td>-24.11</td>
<td>-38.87</td>
<td>-32.50</td>
</tr>
<tr>
<td>Defects studied</td>
<td>77</td>
<td>185</td>
<td>190</td>
</tr>
</tbody>
</table>

* Comparison of defect subsets

- Visually the curves are similar.
- This is not always the case.
Advantages of Lognormal

- Fits data.
- Linked to structure of software and its usage.
  - This may allow reasoning about how changes will affect the defect rates and therefore QoP.
- Cleaner separation of number of bugs, their rates, and the spread in their rates.
- Symmetry (on log scale) allows inferences about distribution of low-rate defects.
  - This is crucial because low rate defects can become high rate during an exploit.
- Though it can mimic part of a lognormal distribution the Pareto fails at low rates (see Perline, 2005)

* References (short list)

- S. Gokhale and R. Mullen, "From the Lognormal to Test Coverage, ISSRE 2004.
**Step 1: How to fit the Discrete-Lognormal**

e.g. for \( \mu = -1, \sigma = 1, N=100 \)

- Define wide range of rates \( R(i) \) geometrically spaced. Factor of 1:1,000,000 or more overall.

- Use Excel lognormdist (sig, mu) function to determine percentage at each rate \( R(i) \). Sum (weights) = 1

- Determine each rate’s contribution to each count-bucket according to Poisson distribution and weight.

- Sum contributions to each count-bucket, multiplying by \( N \), total number of defects, including ones that have not yet happened

- You will need rates that differ by a smaller factor and yet span a larger range.

<table>
<thead>
<tr>
<th>Rate</th>
<th>.05</th>
<th>.135</th>
<th>.688</th>
<th>1.00</th>
<th>2.72</th>
<th>7.39</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log Rate</td>
<td>-3</td>
<td>-2</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>% at Rate</td>
<td>5.4</td>
<td>24.2</td>
<td>49.9</td>
<td>24.2</td>
<td>5.4</td>
<td>.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tick</th>
<th>Tot</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>63.1</td>
</tr>
<tr>
<td>1</td>
<td>23.1</td>
</tr>
<tr>
<td>2</td>
<td>7.8</td>
</tr>
<tr>
<td>3</td>
<td>2.9</td>
</tr>
<tr>
<td>4</td>
<td>1.24</td>
</tr>
<tr>
<td>5</td>
<td>.567</td>
</tr>
<tr>
<td>6</td>
<td>.274</td>
</tr>
<tr>
<td>7</td>
<td>.145</td>
</tr>
</tbody>
</table>

**Step 2: Solving for best Discrete-Lognormal**

Generate Fit Bugs col as Function of Sigma, Mu and N (as on prev slide).

An adjacent column has the \( D_i \), the number of bugs with \( i \) tickets.

Set chisq(i) = \((D_i - F_i)^2 / F_i\).

Sum chisq(i) over all counts.

Allow Excel “solver” to minimize the SUM by varying the Sigma, Mu, and N used to generate the \( F_i \) column.

Note Tickets = Num * Bugs

Preventing high-rate 15% of bugs prevents 57% of tickets in this case.

Above shows min-chi-square fit. Ultimately we used max-log-likelihood for fitting.
Opportunities for Research

- Gather additional failure rate distribution data, test it for lognormal.
- To what extent are analyst’s operational profiles found to match the lognormal?
- What is effect of varying operational profile on shape of distribution, e.g. after test?
- Other opportunities for application of lognormal insights to software.
- Use lognormal as prior in other applications.

<table>
<thead>
<tr>
<th>Triggering Conditions</th>
<th>Multiplicative Rates</th>
<th>Limiting Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMON</td>
<td>UNCOMMON</td>
<td>RARE</td>
</tr>
<tr>
<td>Read</td>
<td>Open</td>
<td>Create</td>
</tr>
<tr>
<td>Local</td>
<td>Nearby</td>
<td>Distant</td>
</tr>
<tr>
<td>By book</td>
<td>User error</td>
<td>UBD</td>
</tr>
<tr>
<td>IO works</td>
<td>IO error</td>
<td>Removed</td>
</tr>
<tr>
<td>ETC</td>
<td>ETC</td>
<td>ETC</td>
</tr>
</tbody>
</table>

Test Strategy
Statistical, but with an accelerated profile: run a much higher percentage of less common triggers to drive interactions.
Ten x the rare rates will find rare-rare interactions 100 times as fast.
Equivalent to Heat/Power/Temp “corner testing” of HW.

SRGM = Coverage Growth = Laplace Transform of LN

Release Strategy
Is it ready? Which is best?

Repair Strategy
Risk vs. Benefit?
Homeland Security Airport Security Model

Norman F. Schneidewind
IEEE Congressional Fellow, IEEE Fellow
Professor of Information Sciences, Naval Postgraduate School
nschneid@nps.navy.mil

Abstract

This model provides a framework for helping to understand and analyze the airport security problem. By modeling the security process, and identifying the weak points, we were able to make recommendations for possible Federal initiatives through legislative or management action to close the identified security loopholes. Passenger flow through the ticket counter, security station, and gate, which potentially includes terrorists, is modeled and quantified. A probability model estimates the probability of a terrorist escaping detection at the various stations. This probability is a function of the reliability of a proposed security database and the reliability of security equipment. The influence of these reliabilities on the probability of non-detection is studied. In addition, a commonly overlooked security problem -- overloading security personnel with passenger traffic to the extent that they are distracted from thoroughly checking passengers -- is modeled and analyzed. Model quantitative results are used to delineate the implications for changes in security policy at the nation’s airports.

Introduction

A model of airport security is proposed and executed. The model involves the flow of a group of passengers, who wish to board a given aircraft, to ticket counters, security stations, airline gates, and aircraft. By confining the security problem in this way, the very difficult problem of airport security analysis is simplified. Why do we develop such a model? An important reason is: "Airports and ticket counters have been attacked, and even airline offices have not been spared in terrorist attempts to intimidate governments and prevent the western public from flying. Terrorists simply cannot leave airports alone, nor does it make sense to do so, since they are the weak point in Western defenses" [JOH91]. And this was written before 9/11! By way of historical perspective, in 1973 the Federal Aviation Administration (FAA) specified that the three critical security areas in airports are the ticket counter, boarding gate, and the aircraft [JOH91]. Curiously, the security station (i.e., luggage x-ray station) is omitted.

According to [JOH91], technology has not kept pace with the threat: terrorists exploit existing technology, airports upgrade their technology, but terrorists outwit that technology, in a never-ending cycle. “we will always be in a position where deterrence presupposes a rational adversary” [JOH91].

This model contains new concepts as follows: improve the reliability of airport security equipment; implement a security database in airports that do not have this capability; improve the reliability of security databases in airports where they exist; and alleviate queuing problems at airline passenger stations and airport security facilities. If these measures are implemented by Congressional funding and enabling legislation, the threat of terrorist attacks should be reduced at the nation’s airports.
The severity of the problem is dramatized by the following findings:

**Pre 9/11 Aviation Unpreparedness**

WASHINGTON - The Federal Aviation Administration received repeated warnings in the months prior to Sept. 11, 2001, about al-Qaeda and its desire to attack airlines, according to a previously undisclosed report by the commission that investigated the terror attacks [MSN05].

The report by the 9/11 commission that investigated the suicide airliner attacks on the World Trade Center and the Pentagon detailed 52 such warnings given to FAA leaders from April to Sept. 10, 2001, about the radical Islamic terrorist group and its leader, Osama bin Laden. The commission report, written last August, said five security warnings mentioned al-Qaeda’s training for hijackings and two reports concerned suicide operations not connected to aviation. However, none of the warnings pinpointed what would happen on Sept. 11. FAA spokeswoman Laura Brown said the agency received intelligence from other agencies, which it passed on to airlines and airports. But, she said, “We had no specific information about means or methods that would have enabled us to tailor any countermeasures.” Brown also said the FAA was in the process of tightening security at the time of the attacks. “We were spending $100 million a year to deploy explosive detection equipment at the airports,” she said. The agency was also close to issuing a regulation that would have set higher standards for screeners and, for the first time, give it direct control over the screening work force. [911] However, there are few airports, today, that have explosive detection equipment installed. In addition, simulated tests have shown that it is possible to pass screener detection in major U.S. airports, while carrying concealed weapons. Thus, the need for a model that can pinpoint vulnerabilities in airport security.

Findings from the 9/11 Commission: [911]

- Aviation officials were “lulled into a false sense of security” and “intelligence that indicated a real and growing threat leading up to 9/11 did not stimulate significant increases in security procedures.”
- Of the FAA’s 105 daily intelligence summaries between April 1, 2001, and Sept. 10, 2001, 52 mentioned Osama bin Laden, al Qaeda, or both, “mostly in regard to overseas threats.”
- The FAA did not expand the use of in-flight air marshals or tighten airport screening for weapons. It said FAA officials were more concerned with reducing airline congestion, lessening delays and easing air carriers’ financial problems than thwarting a terrorist attack.
- A proposed rule to improve passenger screening and other security measures ordered by Congress in 1996 had been held up by the Office of Management and Budget and was still not in effect when the attacks occurred, according to the FAA.

**Passenger and Baggage Screening**

The Aviation and Transportation Security Act (ATSA) made overall aviation transportation security a direct federal responsibility for the first time [DHS05]. The Transportation Security Administration’s (TSA) responsibilities include ensuring screening of passengers through a mix of federal and private screeners and technology. The screener workforce consists of 45,000 screeners located at 448 airports. The screeners are supported by
technology, including x-ray machines, explosive trace detection machines, and explosive
detection systems. U.S. air carriers transport 12.5 tons of cargo, 2.8 tons of which is secured on
passenger planes. The remaining 9.7 million tons is shipped in cargo planes; air freight remains a
serious threat to the nation. TSA is charged with closing this security vulnerability. While
obviously important, air freight security is beyond the scope of this research.

Despite all of the above, according to [BEN05], “We are spending nearly $5 billion each year
on passenger and baggage screening systems, yet lethal weapons still are getting past security and
onto planes. While we have devoted enormous attention and resources to improving aviation
security, it is still far too easy for a terrorist to get a weapon on a passenger plane. The
Department of Homeland Security (DHS) Inspector General, the Government Accountability
Office (GAO), and the TSA have conducted tests on TSA screeners at the nation’s airports and
found surprisingly high failure rates. An alarming number of prohibited items are still not being
detected during checks of passengers, carry-on items, and checked baggage”. In addition,
according to [CKE05], DHS has been slow to deploy equipment and technology that could aid
airport screeners in detecting concealed weapons and explosives.

Air Cargo

“While airline passengers may be screened, cargo beneath their feet is not. The TSA has
identified two critical risks to air cargo “(1) The hostile takeover of an all-cargo aircraft leading to
its use as a weapon; and (2) the use of cargo to introduce an explosive device onboard a passenger
aircraft in order to cause catastrophic damage. Terrorists have exploited the lack of cargo security
on several occasions. For example, a device in a baggage container of Pan Am Flight 103 caused
the flight to explode in 1988 over Lockerbie, Scotland.4. An explosion aboard a U.S. airliner in
1979 was caused by a parcel linked to the “Unabomber” Theodore Kaczynski and shipped as air
cargo. While Congress has mandated tripling air cargo screening, a large portion of commercial
air cargo remain unscreened. TSA relies heavily on the “Known Shipper” program, under which
only approved companies may ship cargo on passenger aircraft. A company can become a
“Known Shipper” with practically no security checks” [BEN05].

PRINCIPLES OF MODELING AND SYSTEMS ENGINEERING

Since modeling is the central tool used in this research, it is appropriate to outline the
methodology and spirit of this quantitative approach to problem solving. In particular, we
describe the operations research (OR) approach to model development [HIL01] and systems
thinking as exemplified in the field of systems engineering [TUR93]. First, we outline the steps in
an OR study, annotated with the relevance to the airport security model.

1. Define the problem of interest and gather relevant data.

   The problem of interest is to improve the security of the nation’s airports. An
   important facet of problem definition is to identify the decision makers. For airport
   security, these are the managers in the FAA, TSA, and airport and airline executives.

   Unfortunately, with few exceptions, there is not much published data on airport
   security available. Our search of the Transportation Research Information Services
   and the Transportation Research Board. databases did not yield relevant data, such as
   airline terrorist threat incidents. Thus, we resort to the use of randomized
hypothetical, but realistic data, and sensitivity analysis to compensate for the data void. We also subject the model to extreme value testing (e.g., using values of probability of terrorist non detection that seem unlikely, but, nevertheless, might occur in an airport security system), as a form of sensitivity analysis, to note the effect on the solution [HIL01].

2. Formulate a mathematical model to represent the problem.

Since little is known with certainty about the details of the airport security problem, we use a probabilistic approach to estimating the quantities of interest, such as the probability of not detecting a terrorist by the time he reaches the gate, if he has not been detected prior to this point. No model can be a complete representation of the real system. If it were, it would be incomprehensible and mathematically intractable. Thus, we extract from the real world of airport security the key factors, such as the probability of non detection, as opposed to attempting to model every movement of a terrorist in an airport. Note that our focus is on non detection because we wish to emphasize the probability of a terrorist escaping apprehension.

3. Develop a computer-based procedure for deriving solutions to the problem from the model.

A spreadsheet approach is used because sensitivity analysis of the solutions can be performed conveniently and plots of the solutions can be obtained easily.

4. Test the model and refine it as needed.

Although we are unable to test the model in an airport at this time, we perform reality checks on the solutions. That is, we check the model assumptions, solutions, and sensitivity analyses to see whether they comport with reality (e.g., a solution of 99.9% probability of terrorist detection would be considered unrealistically optimistic). If such a solution emerged, we would modify the model to produce a more realistic result.

5. Prepare for the ongoing application of the model as prescribed by management.

This step is beyond the scope of this research because, at this stage, the model is a proposal that may be considered for implementation by FAA, TSA, and airport, and airline managers. The details of implementation would be a decision taken by theses managers.

6. Implement the model.

Examples of implementation details are the following: training of airport personnel in the revised passenger security process, implementing the security database and terrorist detection procedures, and installing equipment to detect biological, chemical, and nuclear weapons. Biological agents and toxins are of particular concern [CSI04].

A key piece of legislation pertaining to biological terror is the Intelligence Reform and Terrorism Prevention Act of 2004 that contains various provisions to promote and accelerate the use of biometric technology for secure identification. The law provides for the use of biometric technology in airport access control and law enforcement travel [USS05].
Systems Engineering Concepts

Now, we explore how systems engineering concepts can be applied to airport security. In this vein, an aspect of the origin of systems thinking was the realization that that particular objects are comprised of components and these components are interrelated and independent [TUR93]. The following Table 0 portrays the airport security example, showing current security holes that could be rectified by using a database ID check at the Security Station and Gate:

<table>
<thead>
<tr>
<th>Components</th>
<th>Related By</th>
<th>Security Control</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Boarding Pass and</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ID</td>
<td></td>
</tr>
<tr>
<td>Ticket Counter</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Does not apply</td>
</tr>
<tr>
<td>Security Station</td>
<td>x</td>
<td>Security Hole</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Gate</td>
<td>x</td>
<td>Security Hole</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Does not apply</td>
</tr>
<tr>
<td>Passengers</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Non Terrorists</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

One of the critical developments relative to the origins of systems thinking is that of cause and effect. When a particular component behaves in a certain way, a different component in the related object reacts in a predictable way. [TUR93] For example:

If a passenger (component P) fails a database ID check, then an agent (component A) reacts to detain component P.

Furthermore, the behavior of component P can only be understood by identifying and characterizing the impact of components on each other (e.g., component A checks the database) and the influence of the components on the object (i.e., airport security system) [TUR93].

OBJECTIVE

According to [HIL01], the first order of business in an OR study is to define the objective. Accordingly, we state that our objective is to identify weak points (e.g., security station check) and links (e.g., passenger flow between security check station and gate) in the security process for the purpose of influencing government legislation and regulations to strengthen the process. We feel the subject of this research is extremely important because “America is not sufficiently prepared to fully respond to a catastrophic terrorist attack on U.S. soil that involves chemical, radiological, or nuclear weapons” [CSI04].

MOTIVATION

Consistent with the objective, we relate our experience at an airport that indicates the need for improvement in airport security. Instead of focusing on security measures, like a high reliability and comprehensive security database, which would significantly enhance security, the TSA, in some instances, spends considerable time on trivial matters. For example, we were recently passengers at the one of the nation’s airports. We were carrying a stapler in our briefcase. After the case went through the x-ray machine, and signaled an alert, the TSA agent asked to...
open the case. She saw that the “offending object” was the stapler. She then removed the stapler from the case, put it in a basket, and sent the case and basket through the x-ray machine again. It seemed obvious that the only object in the case -- the only metal object -- that could have signaled an alert was the stapler. Thus, the process should have stopped after the case was opened. Instead of paying TSA personnel to spend time on trivial searches, the TSA should invest in a security database and in improving the reliability of security station equipment.

In addition to seemingly non productive security processes, as described above, certain proposed legislation, does not appear to be helpful. For example, a provision of immigration bill HR418, which passed the House of Representatives, would require: “that information on anyone convicted of using a false driver’s license to board an airplane be added to aviation security screening databases” [HOU05]. The trouble with this provision is that it “closes the barn door after the horse is out of the barn”. No terrorist is going to try to use the same identification again, if his identification had been discovered as false! It is important to note that there have been proposals for standardizing the driver’s license [WAR05], which could become, in effect, a national identification card. With such a card, it would be difficult to fake identification; thus, the probability of non detection would be decreased. This is an issue currently being debated by Congress. It is not clear that such legislation will be passed because of the opposition of privacy advocates.

Other examples of airport security problems that motivate our research are the following:

**Background on Airport Security Issues**

**Selected Items from Terrorist Detection History**

This section illustrates why airport security is a problem and why we are motivated to study the problem. A critical aspect of successful terrorist and weapons detection is the quality and appropriateness of the detection tests. The following reports from the media illustrate some of the problems in conducting successful tests:

**HOW NOT TO TEST AIRPORT SECURITY, SCHNEIER ON SECURITY, DECEMBER 20, 2004[BBC05]**

If this were fiction, no one would believe it. Four days after police at Charles de Gaulle Airport slipped some plastic explosives into a random passenger’s bag as part of an exercise for sniffer dogs, it is still missing -- and authorities are stumped and embarrassed. It is perfectly reasonable to plant an explosive-filled suitcase in an airport in order to test security. It is not okay to plant it in someone's bag without his knowledge and permission. (The explosive residue could remain on the suitcase long after the test, and might be picked up by one of those trace mass spectrometers that detects the chemical residue associated with bombs.) But if you are going to plant plastic explosives in the suitcase of some innocent passenger, shouldn't you at least write down which suitcase it was?

**US airport security loses 'bomb' [BBC05]**

Security screeners at a US airport lost track of a bag containing fake explosives and allowed to be loaded on a flight to Amsterdam. The "bomb" was planted in luggage for training exercise at Newark Liberty International Airport. A scanning machine raised the alarm, but the bag was not searched and airport staff lost track of it. "At no time did the bag pose a threat and at no time was anyone in danger," said a transport security spokeswoman.
**Airport Security Data**

Since, with certain exceptions, airport security data is either classified or unavailable, we have had to resort, in this model, to use hypothetical but realistic data to illustrate the principles of the model. See the Appendix for the spreadsheet data and the results of example computations. In future research, we will attempt to collect data about security attacks from reports, web sites, and the Department of Homeland Security (DHS).

Information flow rates, queue characteristics, etc., which are used in the analytic model, are expected or mean values. If instantaneous values of these variables are desired, simulation must be used. The values of quantities used in the examples are for illustrative purposes. Sensitivity analysis is performed to protect against choosing certain values in the examples. As Cordesman points out, probabilities based on history may be worthless (e.g., pattern of past no indicator of 9/11 attack). It is better to use “what if” analysis [COR, p. 25]. It is important to consider worst case scenarios [COR, p. 33]. Many of the model variables are randomized to provide further protection against bias. An example of “what if” analysis is covered in the What If section.

**Threats**

Now, we consider the flow of passengers, wherein one or more could be terrorists, and a threat to innocent passengers, through the ticketing, security checking, and boarding process, as depicted in Figure 0.

**Definitions**

Refer to Figure 0 when reading the definitions:

Facilities: ticket counter (A), security station (S), and gate (G).

\[ P_t \] is the probability that a passenger (on the aircraft) is a terrorist, mean \( \approx \) 0.05 [CRS04], \( N \) is the estimated number of possible terrorists who are ticketed on Plane P (\( N \) is assumed to be in the range 1,,10), and \( C \) is the capacity of P. A mean value for \( P \) would be appropriate to use, if \( C \) were a constant. However, just considering the Boeing Company alone, there are ten commercial models, with the capacity in number of passengers, shown in Table 1 [BOE05]. Therefore, it is appropriate to consider \( P_t \) as a variable, and to calculate it as \( P_t = N / C \).

<table>
<thead>
<tr>
<th>Table 1. Boeing Company Commercial Aircraft Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>717</td>
</tr>
<tr>
<td>737</td>
</tr>
<tr>
<td>747</td>
</tr>
<tr>
<td>757-300</td>
</tr>
<tr>
<td>757-200</td>
</tr>
<tr>
<td>767</td>
</tr>
<tr>
<td>777</td>
</tr>
<tr>
<td>787-3</td>
</tr>
<tr>
<td>787-8</td>
</tr>
<tr>
<td>787-9</td>
</tr>
</tbody>
</table>
PA is the probability that the terrorist will be detected at the ticket counter by querying the security database. Specifically, this probability is a function of the accuracy and completeness of the security database and of the type of identification IA presented by the passenger at the ticket counter. Although the probabilities of detection at the ticket counter, security station, and gate differ in the real world, they are treated as equal in this model because 1) we have no evidence to the contrary and 2) the assumption of equality is mitigated by randomizing these quantities in the model.

The probability that the terrorist will be detected at the ticket counter is of particular relevance in light of the El Al airlines practice of requiring complete identification of the passenger when purchasing a ticket to allow security officials to compile a reference file on the passenger [JOH91]. Although this is an excellent practice, it is not clear that it would be acceptable to American airline passengers.

RD is the reliability of the security database. Specifically, this is the probability of the database operating without failure during the security checks at the three stations. It is assumed that the reliabilities at the three facilities are equal, since this feature is new in airports, with little information available about operating characteristics.

PS is the probability that the terrorist will be detected at the security station by querying the security database or by performing the luggage check. Specifically, this probability is a function of the accuracy and completeness of the security database and of the type of identification IS presented by the passenger at the security station and the accuracy of the luggage checking equipment.

RS is the reliability of the security checking equipment. Specifically, this is the probability of the security checking equipment at the three facilities operating without failure during the security checks. As in the case of RD, it is assumed that the reliabilities at the three facilities are equal, because we have no information to the contrary.

PG is the probability that the terrorist will be detected at the gate by querying the security database. Specifically, this probability is a function of the accuracy and completeness of the security database and of the type of identification IG presented by the passenger at the gate.

In later sections, we use the following additional definitions:

PAf, probability of non detection at the ticket counter.

PSf, probability of non detection at the security station.

PGf, probability of non detection at the gate.

PGs, probability of detection at the gate.

Rd, overall reliability that is decomposed into the reliabilities of the primary and secondary security databases, Rd1 and Rd2, respectively.
**Assumption:** The events and variables in the analysis are assumed to be independent. Thus, their probabilities can be multiplied. This assumption seems reasonable because there is no dependence among the events of security checking at the three facilities and facility reliabilities.

**Terrorist Scenario**

Before we begin the scenario, let us consider the fact that multiple checks against a database are needed, even if this seems counter intuitive, for the following reason:

Assume that X is not a Muslim, but is part of a terrorist plot. X has a ticket under a false name – the name of Y and a false photo ID with the name of Y. X passes the check at the airline check in counter. Next, X gives his ticket to Y, a Muslim, who has a photo ID. Y goes to the security station and presents “his ticket” and photo ID. Although Y’s name on his ticket and ID match, a search of the database shows that the ticketed person, X, is not a Muslim, and Y is detained for further investigation. Of course, if the database contains Y’s photo, X would have been stopped at the ticket counter, but the reviewer did not make this point.

Picture the scenario shown in Figure 0, where a passenger, who may have biological, chemical, or nuclear weapons in his luggage, stops first at the ticket counter (A) to check in. In this model, airline and security personnel access a security database that contains information about people who are considered possible security threats; their identification is designated by T. Passengers have identification $I_A$ at the ticket counter, $I_S$ at the security station, and $I_G$ at the gate.
The reason for the three identifications is that a passenger could use a different identification at each facility. If a database check results in $T = I_A$, or $T = I_S$, or $T = I_G$, the passenger is detained for interrogation. At the start of the interrogation, the passenger is assumed to not be a terrorist; however, subsequent questioning may suggest otherwise. If the passenger passes the ticket counter check, he proceeds to the security station (S), which is staffed by TSA and airport personnel. The same database check process takes place again. Why? The reason is that no database and computer system is 100% reliable. It is possible that the passenger is a terrorist and the ticket counter check failed to reveal this fact. Of course, the converse is possible. This is why there should be presumed innocence at the start of the interrogation. Unfortunately, currently, the drivers license is the main means of passenger identification, and it is not standardized among the states. As Richard Clark points out, airline agents make no attempt to validate passenger identification [CLA05]. Perhaps, a national identification card is needed, but this might be considered a violation of civil liberties.

This process is repeated at the gate (G). If a terrorist manages to pass all three checks, he is allowed on board the aircraft. Of course, we want this event to have a very low probability. Subsequent sections will address how this could be achieved.

Events

The events pertinent to the process of terrorist detection are listed below.

1. Terrorist detected at ticket counter (A)
2. Terrorist not detected at ticket counter (A)
3. Terrorist detected at security station (S)
4. Terrorist not detected at security station (S)
5. Terrorist detected at gate (G)
6. Terrorist not detected at gate (G)

Definitions

The nomenclature of stations and their associated events are defined below.

A, S, and G are called stations

Events 1, 3, and 5 are independent (i.e., detection at a given station does not depend on detection at other stations).

Events 2, 4, and 6 are independent (i.e., non detection at a given station does not depend on non detection at other stations).

Event Sequences

The sequence of events that transpire in the attempt to detect a terrorist is captures in the event transitions that follow.

A. Start → Event 1 → Terrorist stopped for interrogation at A
B. Start → Event 2 → Event 3 → Terrorist stopped for interrogation at S
C. Start → Event 2 → Event 4 → Event 5 → Terrorist stopped for interrogation at G

D. Start → Event 2 → Event 4 → Event 6 → Terrorist not detected at A, S, and G

Event Sequences A, B, C, and D are independent (i.e., the fact that a terrorist is stopped at a given station does not depend on being stopped at other stations)

The event sequences A, B, C, and D and events 1, ..., 6 in the passenger flow process are depicted in Figure ASG.

**Probabilities**

p: probability of terrorist not being detected on any given attempt at passing a single security check, independent of his location in the airport at any given time. This is a function of the accuracy and comprehensiveness of the security database and of the accuracy of security equipment (e.g., luggage x-ray equipment). Thus, p becomes the key probability in the model because we are modeling the process of the terrorist attempting to go undetected at the ticket counter, security station, and gate.

1 – p: probability of terrorist being detected on a given attempt at passing a single security check. This probability is also a function of the accuracy and comprehensiveness of the security database and of the accuracy of security equipment.
**Binomial Distribution**

The binomial distribution describes the possible number of times \( n \) that a particular event (e.g., terrorist *non* detection) will occur in a sequence of observations (e.g., at the ticket counter, security station, and gate). The binomial distribution is used when a researcher is interested in the probability of an event occurring. The binomial distribution is specified by the number of observations, \( x \) (e.g., number of times a passenger is subjected to a security check), and the probability of occurrence, which is denoted by \( p \) (e.g., probability of *non* detection).

**Definitions**

\( n \) trails: number of possible attempts by terrorist to avoid detection

\( n = 3 \) (A, S, G)

\( x \): given number of attempts by terrorist to avoid detection

**Apply Binomial Distribution**

Our objective is to estimate the probability of *non* detection at A, S, and G, as a function of \( p \), for the purpose of determining the threat posed by a terrorist at each of these stations. Therefore, we have, according to the binomial distribution,

\[
P = \frac{n!}{x!(n-x)!} p^x (1-p)^{n-x}
\]

(1)

Why is it necessary to use the probability \( P \) when \( p \) has already been defined? The reason is that \( p \) does not take into account the **number of times** \( x \) that the terrorist attempts detection out of \( n = 3 \) possible attempts. The probability \( p \) only pertains to the event of *non* detection, **independent** of the number of attempts.

The following Table *Event* summarizes the application of the binomial distribution as it is applied to the quantities \( n \), \( x \), \( P \), and \( p \) and the events 2, 4, 6, and 5, showing that the \( \sum P \) exhausts the probability space.
### Probabilities of Events

In the sections that follow, we describe the airport security events, model the related probabilities of events, determine key points and values on the probability functions, and determine local minima and maxima of the functions by using the calculus. The key points and values, and the local minima and maxima, characterize the probability of non detection (our airport security metric), and identify the optimal non detection probabilities at the ticket counter, security station, and gate that imply policy decisions for government, airport, and airline managers. In developing an optimal solution, we strive for optimality across all entities and personnel within the scope of this research -- airlines; airport security personnel, security database, and equipment; FAA; and TSA -- rather than a single entity [HIL01]. This is achieved by modeling the ticket counter, security station, and gate as a single integrated security system.

It is important to note that an “optimal solution” provided by a model may not be optimal in the eyes of the decision makers responsible for airport security. They are the final arbiters of what constitutes a good security policy [HIL01].

#### Event 2: Terrorist not detected at ticket counter (A)

\[ x = 1 \text{ attempt at non detection at A; } n = 3 \text{ possible attempts} \]

Applying the binomial distribution, the probability of Event 2 = \( P_{Af} \):

\[
P_{Af} = \frac{3!}{1!2!} p^1 (1-p)^2 = 3p(1-p)^2
\]

\[ (2) \]

\[
P_{Af} = 3p - 6p^2 + 3p^3
\]

\[ (3) \]
For \( P_{Af} = 1, \) \( 3 \, p - 6 \, p^2 + 3 \, p^3 = 1, \) \( 3 \, p - 6 \, p^2 + 3 \, p^3 - 1 = 0 \)

(4)

Solving for the roots of (4), \( p = 1.475; \) this value is obviously infeasible.

For \( P_{Af} = 0, \) \( p = 0, \) \( p = 1; \) only \( p = 0 \) is a realistic solution (i.e., \( P_{Af} \) should = 0 when \( p = 0; \) it should not equal 0 when \( p = 1). \) The rate of change of \( P_{Af} \) with \( p \) is given by equation (5):

\[
\frac{dP_{Af}}{dp} = 3 - 12p + 9p^2 = 0, \quad 3p^2 - 4p + 1 = 0
\]

(5)

\[
p = \frac{4 \pm \sqrt{16 - 4(3)(1)}}{6} = \frac{4 \pm 2}{6} = \frac{2}{3} \pm \frac{1}{3}
\]

(6)

\( p_1 = 1, \) \( p_2 = 1/3 \)

\[
\frac{d^2P_{Af}}{dp^2} = -12 + 18p
\]

(7)

For \( p_1 = 1, \) \( \frac{d^2P_{Af}}{dp^2} = -12 + 18 = 6 \Rightarrow P_{Af} \) minimum

For \( p_2 = .3333, \) \( \frac{d^2P_{Af}}{dp^2} = -12 + (18)(.3333) = -6.0 \Rightarrow P_{Af} \) maximum

For \( p_1 = 1, P_{Af} = (3)(1) - (6)(1) + (3)(11) = 0 \)

For \( p_2 = .35, P_{Af} = (3)(.3333) - (6)(.3333)^2 + (3)(.3333)^3 = .4444 \)

As shown in Figure 1, where \( P_{Af} \) is plotted against \( p, \) the maximum value of \( P_{Af} = .4444 \) occurs at \( p = .3333. \) After that, \( P_{Af} \) decreases with \( p, \) becoming 0 at \( p = 1. \) The reason for the decrease is that the binomial representation of equation (2) is not only a function of probability of non-detection \( p \) but also a function of the probability of detection \( (1-p) \) at the ticket counter. At \( p = .3333, p^1 \) begins to exceed \( (1-p)^2 \) in equation (2). Thus the optimal \( p \) represents the resolution of these counteracting factors. The policy implication suggested by this result is that the FAA, airport managers, and airline managers would attempt to improve security at ticket counters (e.g., use of computerized security database) so that \( p \) would be reduced to a value much lower than .3333.
Event 4: Terrorist  *not* detected at security station (S)

\[ x = 2 \text{ attempts at non-detection at S; } n = 3 \text{ possible attempts} \]

Applying the binomial distribution, the probability of Event 4 = \( P_{Sf} \):

\[
\begin{align*}
P_{Sf} &= \frac{3!}{2!1!} p^2 (1-p) = 3 p^2 (1-p) \\
&= 3 p^2 - 3 p^3 \\
&= 3 p^2 - 3 p^3 - 1 = 0 \\
\end{align*}
\]

(8)

For \( P_{Sf} = 1 \), \( 3 p^2 - 3 p^3 = 1 \), \( 3 p^2 - 3 p^3 - 1 = 0 \)

Solving for the roots of (8), \( p = 1.264 \); this value is obviously infeasible

For \( P_{Sf} = 0 \), \( p = 0 \), \( p = 1 \); only \( p = 1 \) is realistic solution (i.e., \( P_{Sf} \) should = 0 when \( p = 0 \); it should not equal 0 when \( p = 1 \)). The rate of change of \( P_{Sf} \) with \( p \) is given by equation (10):
\[ \frac{dP_{Sf}}{dp} = 6p - 9p^2 = 0, 2p - 3p^2 = 0 \]  
\[ p = \frac{-2 \pm \sqrt{4 - 4(-30(0))}}{-6} = \frac{-2 \pm \sqrt{144}}{-6} = \frac{1}{3} \pm \frac{1.4192}{6} \]

\( p_1 = 0.65, \ p_2 \approx 0 \)

\[ \frac{d^2P_{Sf}}{d^2p} = 6 - 18p \]

For \( p_1 = 0.65 \), \[ \frac{d^2P_{Sf}}{d^2p} = 6 - (18)(0.65) = -5.7 \Rightarrow P_{Sf} \text{ maximum} \]

For \( p_1 = 0 \), \[ \frac{d^2P_{Sf}}{d^2p} = 6 - (18)(0) = 6 \Rightarrow P_{Sf} \text{ minimum} \]

For \( p_1 = 0.65 \), \( P_{Sf} = (3)(0.65)^2 - 3(0.65)^3 = 0.4436 \)

For \( p_1 = 0 \), \( P_{Sf} = 0 \)

As shown in Figure 2, where \( P_{Sf} \) is plotted against \( p \), the maximum value of \( P_{Sf} = 0.4436 \) occurs at \( p = 0.65 \). After that, \( P_{Sf} \) decreases with \( p \), becoming 0 at \( p = 1 \). The reason for the decrease is that, while the binomial representation of equation (2) increases with \( p \), \( (1-p) \) -- the probability of terrorist being detected -- decreases with \( p \). Thus the optimal \( p \) represents the resolution of these counteracting factors. The policy implication suggested by this result is that the FAA, airport managers, and TSA managers would attempt to improve security at security stations (e.g., use of computerized luggage checking system) so that \( p \) would be reduced to a value much lower than 0.65.
Event 6: Terrorist not Detected at gate (G)

\[ x = 3 \text{ attempts at non detection out of } n = 3 \text{ total attempts} \]

Applying the binomial distribution, the probability of Event 6 is given by

\[ P_{Gf} = \frac{3!}{3!0!} p^3(1-p)^0 = p^3 \]

(12)

As shown in Figure 3, where \( P_{Gf} \) is plotted against \( p \), the maximum value of \( P_{Gf} = 1.0 \) occurs at \( p = 1.0 \). In this case, \( P_{Gf} \) increases monotonically. The reason for this is that there is only a \( p \) non detection term in equation (12); no 1-p detection term. The policy implication suggested by this result is that the FAA, airport managers, airline managers, and TSA managers would attempt to improve security at the ticket counters and at security stations to the extent that terrorists would be detected before they reach the gate, because after they reach the gate, there is little opportunity for detection, as shown in Figure 3.
Since the gate is the last place to stop the terrorist within the scope of the model -- $P_{Gf}$ is our metric of the quality of the security system -- the lower the better – consistent with cost, personnel and technology constraints. Decision makers could gauge the performance of their security system against this metric [HIL01].

**Event 5: Terrorist Detected at gate (G):**

$x = 0$ attempts at *non detection* out of $n = 3$ total attempts is equivalent to a successful detection.

Applying the binomial distribution, the probability of Event 5 (successful detection at G) =

$$P_{Gs} = \frac{3!}{0!3!} p^0 (1-p)^3 = 1 - 3p + 3p^2 - p^3$$

This is also equal to:

$$P_{Gs} = 1 - P_{Af} - P_{Sf} - P_{Gf} = 1 - (3p^2 - 3p^3) - (3p^2 - 3p^3) - p^3 = 1 - 3p + 3p^2 - p^3$$

For $P_{Gs} = 1$, $1 - 3p + 3p^2 - p^3 = 1$, $-3p + 3p^2 - p^3 = 0$

Solving for the roots of (13), no feasible roots were found

For $P_{Gs} = 0$, $p = 0$

$$\frac{dP_{Gs}}{dp} = -3 + 6p - 3p^2 = 0, 3p^2 - 6p + 3 = 0, p^2 - 2p + 1 = 0$$

(14)
Figure 4. Probability of Detection at Gate, $P_{Gs}$ vs. $p$

$$p = \frac{2 \pm \sqrt{4 - (4)(1)(1)}}{2} = \frac{2}{2} = 1$$

$$\frac{d^2 P_{Gs}}{d^2 p} = -6 - 6p, \text{ for } p_1 = 1, \frac{d^2 P_{Gs}}{d^2 p} = 0$$

(15)

$$\frac{d^3 P_{Gs}}{d^3 p} = -6 \neq 0, \text{ for } p_1 = 1, P_{Gs} \text{ is neither a minimum nor maximum}$$

(16)

For $p_1 = 1$, $P_{Gs} = 1 - 3 + 3 - 1 = 0$

For $p_2 = 0$, $P_{Gs} = 1 - 0 + 0 - 0 = 1$

As shown in Figure 4, where $P_{Gs}$ is plotted against $p$, the maximum value of $P_{Gs} = 1.0$ occurs at $p = 0$. In this case, $P_{Gs}$ decreases monotonically. The reason for this is that there is only a $1 - p$ detection term in equation (13); no $p$ non detection term. The policy implication suggested by this result is that the FAA, airport managers, and airline managers would attempt to improve security at the gate so that $p$ is not significantly greater than 0, because $P_{Gs}$ decreases rapidly thereafter, as shown in Figure 4.

**Effectiveness of Detection at Gate**

Now, we combine probability of non detection $P_{Gs}$ with the reliability of the security database $R_d$ and the reliability of the security equipment $R_s$ to produce the effectiveness at the gate. Before we present this effectiveness equation, we elaborate on the characteristics of $R_d$ and $R_s$, and show how redundancy increases reliability and, therefore, effectiveness.
Definitions

Definitions that characterize the redundant security database and security equipment, and their reliabilities are presented below.

\( d_1 \): primary security database
\( d_2 \): secondary security database
\( s_1 \): primary security equipment
\( s_2 \): secondary security equipment

\( R_{do} \): overall reliability of security database
\( R_{d1} \): reliability of primary security database
\( R_{d2} \): reliability of secondary security database

\( R_{so} \): overall reliability of security equipment
\( R_{s1} \): reliability of primary security equipment (.96, 1.00, with a mean = .98 [CRS04])
\( R_{s2} \): reliability of secondary security equipment (.96, 1.00, with a mean = .98 CRS04)

Assumptions

The assumptions upon which the computations of reliability rest are as follows:

\( d_1 \) and \( d_2 \) are independent (i.e., failure of \( d_2 \) does not affect reliability of \( d_1 \))

\( s_1 \) and \( s_2 \) are independent (i.e., failure of \( s_1 \) does not affect reliability of \( s_2 \))

Reliability Equations

The reliability of parallel components is computed below.

\( R_{do} = R_{d1} + R_{d2} - R_{d1} \cdot R_{d2} \): reliability of two components in parallel (i.e., redundancy) (16)

\( R_{so} = R_{s1} + R_{s2} - R_{s1} \cdot R_{s2} \): reliability of two components in parallel (i.e., redundancy) (17)

The redundancy characteristics are re-elaborated in Figures 5 and 6.

Data Values

Mean values of security database and security equipment reliabilities were obtained from Congressional Research Service reports as follows:

\( R_{d1}, R_{d2}, R_{s1}, R_{s2} \): specified between 0 and 1, and randomized, with a mean \( \equiv .96 \) [CRS04]

Probability that the passenger is a terrorist = \( P_t \): specified between 0 and 1, and randomized, with a mean \( \equiv .05 \) [CRS04]

Sensitivity Analysis

Randomization of \( R_{d1}, R_{d2}, R_{s1}, R_{s2}, \) and \( P_t \), within the specified constraints, provides a degree of sensitivity analysis.

Effectiveness

The effectiveness of terrorist detection at the gate is obtained by melding \( P_{tg} \) with the reliabilities obtained by redundant component analysis, as shown in equation (18).
Effectiveness is a better metric of ability to detect terrorists than $P_{Gs}$ alone, because, whereas $P_{Gs}$ is a function of the accuracy and speed of the database and equipment, it does not include reliability. If accuracy and speed are high, but reliability is low, overall effectiveness of detection will be low.

Effectiveness of security measures at the gate $= E_G$:

$$E_G = P_{Gs} R_{do} R_{so} = (1 - p)^3 (R_{d1} + R_{d2} - R_{d1} R_{d2}) (R_{s1} + R_{s2} - R_{s1} R_{s2})$$  \hspace{1cm} (18)

Also, using equations (16) and (17), $E_G = (1 - p)^3 R_{do} R_{so}$

$$\frac{dE_G}{dp} = -3(1-p)^2 R_{do} R_{so} = 0, (1-p)^2 = 0, 1 - 2p + p^2 = 0, p_1 = 1, p_2 = 0$$  \hspace{1cm} (19)

$$\frac{d^2E_G}{dp^2} = 6(1-p)R_{do} R_{so}$$  \hspace{1cm} (20)

For $p_1 = p_2 = 1$, $\frac{d^2E_G}{dp^2} = 0$, $\Rightarrow$ neither minimum or maximum.

For $p_1 = 1$, $E_G = 0$

For $p_2 = 0$, $E_G = R_{do} R_{so}$

The policy implication of Figure 7, where $E_G$ is plotted against $p$ is to make $R_{d1}$, $R_{d2}$, $R_{s1}$, $R_{s2}$ as high as possible, because from equations (16) and (17), this will maximize $R_{do}$ and $R_{so}$, respectively. Of course, this plan must be consistent with cost and technical considerations (i.e., state of the practice with respect to achieving reliability). Doing this will maximize $E_G$ at probability of non detection $= p = 0$, as can be seen in Figure 7.
Figure 5. Redundancy in Security Database System
Primary Security Database Equipment

- $d_1$: Primary Security Database
- $R_{d1}$: Primary Security Database Reliability
- $d_2$: Secondary Security Database
- $R_{d2}$: Secondary Security Database Reliability

- Periodic Backup: Primary → Secondary
- If Primary fails, Secondary takes over
- Primary updated by Secondary and takes over, after coming back on line

Figure 6. Redundancy in Security Equipment System

- $R_{s1}$: Primary Security Equipment $s_1$
- $R_{s2}$: Secondary Security Equipment $s_2$
- $R_{s1}$: Primary Security Equipment Reliability
- $R_{s2}$: Secondary Security Equipment Reliability

If Primary fails, Secondary takes over


**Evaluation of Relative Terrorist Threats at the Ticket Counter, Security Station, and Gate**

Figure 8 quantifies what seems intuitive about a terrorist escaping detection at the ticket counter, security station, and gate. That is, the more stations that fail to detect the terrorist, the easier it is for him to go undetected at succeeding stations. In Figure 8, this is portrayed by the optimal probability of non detection $p$ increasing from ticket counter to security station, and, finally, the probability of non detection increasing monotonically with $p$, at the gate. The policy implication is clear: stop the terrorist as soon as possible, preferably at the ticket counter. This objective is crucial when we consider that the terrorist’s chances of achieving at least partial success (i.e., high probability of non detection) exceeds 75 per cent, according to [JOH91]. According to the model, as Figure 8 shows, this level of success would not be achieved at the ticket counter or security station but could be accomplished at the gate.

Unfortunately, the ticket counters are under the control of the airlines and are the entity least subject to control by the government. A compromise solution might be to emphasize detection at the security station because it is under the control of the TSA and airport management and has the advantage of containing luggage checking equipment and, in the future may be equipped with a security database, as an additional check on passengers.
As shown in Figure 0, the security process actors are airline ticket agents, security station personnel (TSA and airport luggage screeners), and airline personnel at the gate; not shown is the flight crew. The security measures exercised by the crew are beyond the scope of this research. For the security system to work, there must be communication and coordination among these actors. The 9/11 report states that better coordination between the FAA and the airlines is needed [911, p. 10]. In addition, as stated in [JOH91]: “Good airport security involves outthinking the terrorist. It also involves cooperation among all agencies that can, together, block security loopholes that begins with ticket purchase and ends when the plane takes off”. In response, TSA is developing a computer network to tie together administrative, passenger screening, and baggage screening areas [DHS05].

The TSA has obvious influence over airport and airline security personnel. In the model, this is accomplished, in part, by the security database. This capability seems to be lacking in airports at present. In addition to the database, an important contributor to terrorist detection is the number and quality of airport screeners. With respect to the former, TSA reports that the number of screeners has dropped from 60,000 to 45,000 due to insufficient funding [CSI04]. An additional concern is that the DHS Inspector General issued a report in September 2004 stating that Federal screening improvements were needed in training, equipment, and technology, policy and procedures, and management and supervision [SEC05]. Improvements in equipment and
In the model, the flight crew does not have access to the database because airline personnel at the gate would provide a security check prior to passenger boarding. The gate represents a further opportunity for passenger security database checking, before the passenger boards [JOH91]. However, having a fourth security check on board the aircraft might be a feature to consider.

**Information Flow**

Information flow, as opposed to physical flow, which is shown in Figure 0, is shown in Figure 9. This figure shows the important quantities associated with queuing at the various security checking facilities. An objective of this section is to expose security vulnerabilities that may not have been recognized heretofore.

**Definitions**

Refer to Figure 9 when reading the definitions:

\( \lambda_A \): mean rate at which passengers approach the ticket counter in passengers per minute

\( \lambda_S \): passenger input rate at Security Station: mean rate at which passengers approach the security station in passengers per minute

\( \lambda_G \): passenger input rate at Gate: mean rate at which passengers approach the gate in passengers per minute

\( \mu_A \): passenger service rate at Ticket Counter: mean rate passengers can be served at the ticket counter in passengers per minute

\( \mu_S \): passenger service rate at Security Station: mean rate passengers can go through the security check at the security station in passengers per minute

\( \mu_G \): passenger service rate at Gate: mean rate passengers can be boarded at the gate in passengers per minute

\( \rho_A \): utilization at Ticket Counter: mean fraction of time that ticket counter is busy serving passengers

\( \rho_S \): utilization at Security Station: mean fraction of time security station is busy doing security checks on passengers
Passenger Security Processing Scenario

Passengers approach the ticket counter with a mean input rate of $\lambda_A$ passengers per minute. The queue characteristics at the ticket counter are the number of agent stations (i.e., servers) $M_A$, queue utilization $\rho_A$, and queue service rate $\mu_A$ in passengers per minute. In addition, the terrorist could be carrying on luggage $C_o$, checking luggage $C_h$, or requesting that luggage be delivered to the cargo bay of the plane $C_a$. The concern about cargo has received increase emphasis of late because TSA is not only responsible for passenger security but cargo security as well [CSI04].

It is interesting to note the practice of El Al Airlines that first x-rays baggage destined for the cargo hold and then subjects it to depressurization to simulate flight conditions [JOH91]. The concept is that either the x-rays will expose weapons or depressurization will cause premature detonation. In addition, it has been recommended that vapor sniffing machines be added to the x-ray capability [JOH91].

Furthermore, “the success of TSA in fulfilling its aviation security mission depends heavily on the quality of its staff and the capability and reliability of the equipment (i.e., overall reliability $R_s$) to screen passengers and cargo in order to identify terrorists and terrorists’ weapons, while minimizing disruption to public mobility and commerce “[SKI05].
Integrating Probability of Non Detection with Queue Characteristics

Now, we integrate the probability of non detection with the queue characteristics of the stations, such as the station service rate. Why would there be this relationship? The answer is that as the personnel at the stations are pressured to process passengers at increasing rates, their ability to detect terrorists decreases as they are distracted by the growing passenger flow rate. Therefore, we expect the probability of non detection to increase with increasing service rate (i.e., increasing number of passengers serviced per unit time).

To determine the mean input rate at the ticket counter $\lambda_A$, compute equation (23):

$$\lambda_A = \frac{C}{t}$$  \hspace{1cm} (23)

where $C =$ plane mean capacity = 400 passengers (assumed) and $t$ is the time required to process passengers at the three facilities = 100 minutes (assumed). Therefore, $\lambda_A = 4$ passengers per minute (mean).

A security vulnerability could be created by the agents becoming overloaded by the size of the queue with the result that security checking becomes inadequate. Indeed, this very factor was discovered in the airports of Europe where passengers going through the screening process produced the assembly line effect, causing security personnel to become much less vigilant [JOH91]. This vulnerability is represented by the queue utilization $\rho_A$, as given by equation (24), taking into account the overall reliability of the security checking equipment $R_{so}$. Recall from Figure 0, that we are concerned with the reliability $R_{so}$ of the ticket counter, security station, and gate. Thus, $R_{so}$ appears in the queuing equations below.

From queuing theory [HIL01], we produce equation (24):

$$\rho_A = \frac{\lambda_A R_{so}}{\mu_A M_A}$$  \hspace{1cm} (24)

Solving for $\mu_A$ yields equation (25):

$$\mu_A = \frac{\lambda_A R_{so}}{\rho_A M_A}$$  \hspace{1cm} (25)

Since the service rate at the ticket counter $\mu_A =$ the input rate at the security station $\lambda_s$, (see Figure 9), we can develop equation (26) for the service rate $\mu_s$ at the security station:
\[ \mu_S = \frac{\mu_A R_{so}}{\rho_S M_S} \] 

(26)

Since the service rate at the security station \( \mu_S \) = the input rate at the gate \( \lambda_G \) (see Figure 9), we can develop equation (27) for the service rate \( \mu_G \) at the gate:

\[ \mu_G = \frac{\mu_S R_{so}}{\rho_G M_G} \] 

(27)

Equations (25, 26, and 27) indicate that the vulnerability could be mitigated by reducing \( R_{so} \) or increasing the number of servers. Interestingly, reducing \( R_{so} \), while helping to close this vulnerability, would decrease the Effectiveness of Detection at the gate! (see equation 18). Thus, there is a tradeoff between \( R_{so} \) and the number of servers, as they affect the Effectiveness of Detection and service rate, respectively.

At this point, we provide an example, to illustrate the analysis of the results of the example calculations of the relationships between service rate and number of servers, between probability of non detection and service rate, and between probability that the passenger is a terrorist and the estimated number of terrorists:

The data used in this example are the following:

\[ R_{so} = R_{s1} + R_{s2} - R_{s1} R_{s2} \]  
(28)

\( R_{s1}, R_{s2}, R_{s1}, R_{s2} \): mean \( \approx .98 \)

\( \rho_A = .8 \) (assumed from observation of ticket counter operations)

\( \rho_S = .9 \) (assumed from observation of security station operations: higher service rate requirement that ticket counter)

\( M_A = 1, \ldots, 20 \)

\( M_S = 1, \ldots, 20 \)

\( P_t = \text{probability that passenger is a terrorist} = N / C \) (on plane) 
(29)

\( N= 0, \ldots, 10 \) terrorists on plane

\( C: \text{Capacity of plane} = \text{number of passengers}. \) See Table 1

Sample size = 10 or 20 depending on variable
Example Calculations

Ticket Counter

In Figure 10, we see that the service rate at the ticket counter $\mu_A$ decreases rapidly with the number of servers $M_A$, at first, but then decreases less rapidly later, reaching an optimal value at $\mu_A = .62$ passengers per minute; this occurs at $M_A = 8$ servers. The optimal $\mu_A$ is obtained from Figure 11, where the probability of non-detection at ticket counter $P_{Af}$ is maximum at $\mu_A = .62$. A possible explanation for this relationship in Figure 11 is that initially the ticket agents are unable to cope with the passenger input rate $\lambda_A$, thus allowing an increase in the probability of non-detection $P_{Af}$. Eventually, at service rate $\mu_A = .62$, the agents adjust, get the security process under control, and $P_{Af}$ decreases. The policy implication for airline managers is that providing more than eight agents, (see Figure 10), from a security standpoint, would be a waste of money and personnel. Of course, this may not be the correct policy from the standpoint of customer satisfaction.

Figure 10. Service Rate at Ticket Counter $u_A$ vs. Number of Servers $M_A$
Figure 11. Probability of Non Detection at Ticket Counter, $P_{AF}$ vs. Service Rate $u_A$

![Graph showing the relationship between $P_{AF}$ and $u_A$. The graph indicates a desired service rate region and a minimum tolerable service rate: $u_A = 0.2$.

Security

Figure 12, illustrates dramatically the danger in letting a terrorist past the ticket counter: In Figure 11, we saw that after the service rate at the ticket counter reached 0.62 passengers per minute, the probability of non-detection steadily decreases. No such benign condition exists at the security station, where the probability of non-detection increase monotonically with service rate. Therefore, the policy implication for airline and airport managers is to stop the terrorist at the ticket counter!

Analysis of Terrorist Factors

Figure 13 shows the relationship between the probability that a passenger is a terrorist $P_t$ and the estimated number of passengers who are terrorists. We see that the $P_t$ ranges between 0.001 and 0.039, for $N$ ranging between 0 and 9, respectively; these are significant probabilities in light of the damage that terrorists did on 9/11, where a priori the probability of such a successful attack was considered insignificant. Therefore, since the number of terrorists, and their probabilities of occurrence, are areas not under the control of airport management, the policy implication is that they must be prepared to handle a number of incidents of high severity in the foreseeable future, and it behooves them to greatly improve the reliability and accuracy of detection hardware and software.
Figure 12. Probability of Non Detection at the Security Station, $P_{Gf}$, vs. Service Rate, $u_s$

Figure 13. Probability that Passenger is a Terrorist, $P_t$, vs. Number of Terrorists, $N$
Countermeasures

Countermeasures to the threats, consistent with the previous section, involve the following:

Ticket Counter and Gate

Reduce the service rate of airline personnel at the ticket counter ($\mu_A$) and gate ($\mu_G$) by increasing the number of agents $M_A$, $M_G$, respectively. The first countermeasure may not be attractive to the airlines because of the personnel cost involved and limited space for additional personnel. The second countermeasure may be even less attractive because the number of gates is governed by airport design. More space could be considered when airports are redesigned or when new airports are constructed.

Security Station

Reduce the service rate of the security station ($\mu_S$) by increasing the number of stations $M_S$. This option is attractive because the TSA controls TSA personnel and equipment and airport screeners. Thus, Congressional funding might be considered for more security personnel and equipment, airport space permitting.

Increase the reliability of the security station equipment $R_S$. This quantity influences security not only at the security station, but at the ticket counter and gate as well.

Security Database

As we have seen, the quality of the security database can have a pervasive effect on the ability to detect terrorists. Since the security database is accessed by all facilities, its reliability could be considered a high priority for Congressional funding.

What If Analysis

Terrorist with False Identification

One of the critical situations that would mitigate against terrorist detection: what if the terrorist carries a false identification? This means that the security database checks at the ticket counter, security station, and gate could fail. Then, the only facility to catch the terrorist is the x-ray equipment at the security station for checking luggage. The implication of this is that the terrorist could only be stopped at the security station. With only the luggage check to stop a terrorist, we now need a probability of non detection $P_{Sf} < .65$, as shown in Figure 8, for the security station, as opposed to a probability of non detection $P_{Af} < .35$, for the ticket counter, also shown in Figure 8. At the security station, $P_{Sf}$ does not decline until .65, as opposed to $P_{Af}$ declining at .35, in the case of the ticket counter. If the database check were working at the ticket counter, it would be easier to catch the terrorist there, as shown in Figure 8. However, what if the ticket counter security check is not working? What is the policy implication of this scenario? It means that the TSA and airport and airline managers need to operate as though the security
database will fail at all stations (even with redundant equipment), and the only line of defense is luggage x-ray equipment. Each manager must assess the security threat in a worst case scenario and guarantee that an adequate level of security is maintained.

**Database and Equipment Failures**

Another “what if” situation, with less adverse consequences than the above, is when the primary security database equipment $d_1$ fails or the primary security equipment $s_1$ fails. This adversity is covered by the redundant back up equipment $d_2$ and $s_2$, respectively. The implication for management in this case is to have a switchover and repair policy that can bring all units back on line as soon as feasible.

**Security Station Performance**

“What if” the queue characteristics that were evaluated in the “Integrating Probability of Non Detection with Queue Characteristics” section, do not hold. For example, what if we use the Department of Transportation goal that passenger wait time for security processing not exceed 10 minutes [CRS04] (e.g., at the security station): $t_{sw} \leq 10$ minutes.

Actually, the Bureau of Transportation Statistics found, in 2003 [CRS04], that $t_{sw} = 18$ minutes (mean). We will evaluate and compare the goal with the real world experience to note the effect of passenger wait time on the probability of non detection at the security station. Continuing the analysis, and noting that the reciprocal of the service time is equal to the service rate (i.e., $\mu_s = 1 / t_{ss}$), we have the following equations:

$$t_{st} = \text{total time in security station system (wait time} = t_{sw} + \text{service time} = t_{ss}) \quad (30)$$

Case 1: $t_{sw} \leq 10$ minutes per passenger:

$$t_{st} \leq (10 + t_{ss}) \quad (31)$$

$$t_{ss} \geq (t_{st} - 10) \quad (32)$$

$$t_{st} \leq (1 / \mu_s) + 10 \text{ minutes per passenger} \quad (33)$$

Case 2: $t_{sw} = 18$ minutes per passenger (mean):

$$t_{st} = 18 + t_{ss} \quad (34)$$

$$t_{ss} = t_{st} - 18 \quad (35)$$

$$t_{st} = (1 / \mu_s) + 18 \text{ minutes per passenger (mean)} \quad (36)$$

Now from equation (26), we have:

$$\mu_s = \frac{\mu_A R_{so} \rho S M_S}{\rho S M_S}$$
Combining (26) with (33), we obtain equations (37) for Case 1:

\[
\frac{\rho}{\mu_A R_{so}} + 10 \leq t_{st}
\]  

(Case 1)

Combining (26) with (36), we obtain equations (38) for Case 2:

\[
\frac{\rho}{\mu_A R_{so}} + 18 = t_{st}
\]  

(Case 2)

The results of this analysis are shown in Figure 14 for Case 1 (equation 37) and Case 2 (equation 38). The two curves are separated by a time of 8 minutes, as the two equations indicate. By providing upper bound and mean values, a band of total time of processing passengers \( t_{st} \) is provided that airport managers can use for estimating \( t_{st} \) for a given value of probability of non-detection \( P_{sf} \) at the security station. In addition, we note that if the managers lose control of passenger processing time, and it exceeds the allowable band, \( P_{sf} \) will increase.
Conclusions

Based on the foregoing analysis that showed a preference for security improvements that could be implemented under Federal control and funding, we reach the following conclusions:

security station

Increase the accuracy and reliability of security checking equipment

Increase the number and quality of security personnel and servers

All Facilities

Implement, if the system does not exist, a high accuracy and reliable security database at the nation’s airports

For existing databases, increase their accuracy and reliability

Maintain a centralized database of all passenger flight activity and perform security checks at all security points, using this database.

Include the use of fingerprints or photographs in the database of passengers as one way of positively identifying each passenger on each flight. However, this kind of surveillance would likely face serious legal and privacy challenges [BBC05].

Future Research Directions

The focus of future research will be to visit several major airports and hold discussions with airport managers for the purpose of collecting detailed security data and to obtain their opinions about the validity of the model. In addition, we will attempt to validate the model, based on the collected data, and to modify the model, if necessary.

References


[AIR] AirSafe.com (undated)


### Appendix: Spreadsheet Data and Computations

<table>
<thead>
<tr>
<th>p</th>
<th>$P_{A1}$</th>
<th>$P_{A2}$</th>
<th>$P_{C1}$</th>
<th>$P_{C2}$</th>
<th>$R_{A1}$</th>
<th>$R_{A2}$</th>
<th>$R_{C1}$</th>
<th>$R_{C2}$</th>
<th>$E_G$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>1.0000</td>
<td>0.9942</td>
<td>0.9897</td>
<td>0.9822</td>
<td>0.9991</td>
<td>0.9999</td>
</tr>
<tr>
<td>0.05</td>
<td>0.1354</td>
<td>0.0071</td>
<td>0.0001</td>
<td>0.8574</td>
<td>0.9840</td>
<td>0.9776</td>
<td>0.9757</td>
<td>0.9608</td>
<td>0.8563</td>
</tr>
<tr>
<td>0.1</td>
<td>0.2430</td>
<td>0.0270</td>
<td>0.0010</td>
<td>0.7290</td>
<td>0.9708</td>
<td>0.9817</td>
<td>0.9892</td>
<td>0.9764</td>
<td>0.7284</td>
</tr>
<tr>
<td>0.15</td>
<td>0.3251</td>
<td>0.0574</td>
<td>0.0034</td>
<td>0.6141</td>
<td>0.9906</td>
<td>0.9910</td>
<td>0.9861</td>
<td>0.9983</td>
<td>0.6141</td>
</tr>
<tr>
<td>0.2</td>
<td>0.3840</td>
<td>0.0960</td>
<td>0.0080</td>
<td>0.5120</td>
<td>0.9901</td>
<td>0.9990</td>
<td>0.9819</td>
<td>0.9612</td>
<td>0.5116</td>
</tr>
<tr>
<td>0.25</td>
<td>0.4219</td>
<td>0.1406</td>
<td>0.0156</td>
<td>0.4219</td>
<td>0.9606</td>
<td>0.9757</td>
<td>0.9755</td>
<td>0.9937</td>
<td>0.4214</td>
</tr>
<tr>
<td>0.3</td>
<td>0.4410</td>
<td>0.1890</td>
<td>0.0270</td>
<td>0.3430</td>
<td>0.9858</td>
<td>0.9848</td>
<td>0.9617</td>
<td>0.9771</td>
<td>0.3426</td>
</tr>
<tr>
<td>0.35</td>
<td>0.4436</td>
<td>0.2389</td>
<td>0.0429</td>
<td>0.2746</td>
<td>0.9777</td>
<td>0.9940</td>
<td>0.9914</td>
<td>0.9950</td>
<td>0.2746</td>
</tr>
<tr>
<td>0.4</td>
<td>0.4320</td>
<td>0.2880</td>
<td>0.0640</td>
<td>0.2160</td>
<td>0.9655</td>
<td>0.9624</td>
<td>0.9704</td>
<td>0.9708</td>
<td>0.2158</td>
</tr>
<tr>
<td>0.45</td>
<td>0.4084</td>
<td>0.3341</td>
<td>0.0911</td>
<td>0.1664</td>
<td>0.9882</td>
<td>0.9756</td>
<td>0.9829</td>
<td>0.9705</td>
<td>0.1662</td>
</tr>
<tr>
<td>0.5</td>
<td>0.3750</td>
<td>0.3750</td>
<td>0.1250</td>
<td>0.1250</td>
<td>0.9660</td>
<td>0.9807</td>
<td>0.9826</td>
<td>0.9610</td>
<td>0.1248</td>
</tr>
<tr>
<td>0.55</td>
<td>0.3341</td>
<td>0.4084</td>
<td>0.1664</td>
<td>0.0911</td>
<td>0.9854</td>
<td>0.9984</td>
<td>0.9623</td>
<td>0.9831</td>
<td>0.0911</td>
</tr>
<tr>
<td>0.6</td>
<td>0.2880</td>
<td>0.4320</td>
<td>0.2160</td>
<td>0.0640</td>
<td>0.9805</td>
<td>0.9803</td>
<td>0.9832</td>
<td>0.9899</td>
<td>0.0640</td>
</tr>
<tr>
<td>0.65</td>
<td>0.2389</td>
<td>0.4436</td>
<td>0.2746</td>
<td>0.0429</td>
<td>0.9807</td>
<td>0.9844</td>
<td>0.9706</td>
<td>0.9987</td>
<td>0.0429</td>
</tr>
<tr>
<td>0.7</td>
<td>0.1890</td>
<td>0.4410</td>
<td>0.3430</td>
<td>0.0270</td>
<td>0.9683</td>
<td>0.9992</td>
<td>0.9750</td>
<td>0.9916</td>
<td>0.0270</td>
</tr>
<tr>
<td>0.75</td>
<td>0.1406</td>
<td>0.4219</td>
<td>0.4219</td>
<td>0.0156</td>
<td>0.9655</td>
<td>0.9673</td>
<td>0.9889</td>
<td>0.9875</td>
<td>0.0156</td>
</tr>
<tr>
<td>0.8</td>
<td>0.0960</td>
<td>0.3840</td>
<td>0.5120</td>
<td>0.0080</td>
<td>0.9882</td>
<td>0.9965</td>
<td>0.9905</td>
<td>0.9658</td>
<td>0.0080</td>
</tr>
<tr>
<td>0.85</td>
<td>0.0574</td>
<td>0.3251</td>
<td>0.6141</td>
<td>0.0034</td>
<td>0.9652</td>
<td>0.9669</td>
<td>0.9655</td>
<td>0.9652</td>
<td>0.0034</td>
</tr>
<tr>
<td>0.9</td>
<td>0.0270</td>
<td>0.2430</td>
<td>0.7290</td>
<td>0.0010</td>
<td>0.9974</td>
<td>0.9852</td>
<td>0.9940</td>
<td>0.9851</td>
<td>0.0010</td>
</tr>
<tr>
<td>1</td>
<td>0.0000</td>
<td>0.0000</td>
<td>1.0000</td>
<td>0.0000</td>
<td>0.9645</td>
<td>0.9697</td>
<td>0.9734</td>
<td>0.9752</td>
<td>0.0000</td>
</tr>
</tbody>
</table>

**mean** 0.9774 0.9846 0.9789 0.9805
<table>
<thead>
<tr>
<th>R_{m0}</th>
<th>R_{n0}</th>
<th>M_A</th>
<th>\mu_A</th>
<th>M_S</th>
<th>\mu_S</th>
<th>M_G</th>
<th>\mu_G</th>
<th>P</th>
<th>N</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9999</td>
<td>1.0000</td>
<td>1</td>
<td>5.00</td>
<td>1</td>
<td>5.555</td>
<td>1</td>
<td>7.936</td>
<td>0.051</td>
<td>5</td>
<td>106</td>
</tr>
<tr>
<td>0.9996</td>
<td>0.9990</td>
<td>2</td>
<td>2.50</td>
<td>2</td>
<td>1.386</td>
<td>2</td>
<td>0.989</td>
<td>0.035</td>
<td>7</td>
<td>189</td>
</tr>
<tr>
<td>0.9995</td>
<td>0.9997</td>
<td>3</td>
<td>1.67</td>
<td>3</td>
<td>0.617</td>
<td>3</td>
<td>0.294</td>
<td>0.001</td>
<td>0</td>
<td>524</td>
</tr>
<tr>
<td>0.9999</td>
<td>1.0000</td>
<td>4</td>
<td>1.25</td>
<td>4</td>
<td>0.347</td>
<td>4</td>
<td>0.124</td>
<td>0.016</td>
<td>4</td>
<td>280</td>
</tr>
<tr>
<td>1.0000</td>
<td>0.9993</td>
<td>5</td>
<td>1.00</td>
<td>5</td>
<td>0.222</td>
<td>5</td>
<td>0.063</td>
<td>0.017</td>
<td>4</td>
<td>228</td>
</tr>
<tr>
<td>0.9990</td>
<td>0.9996</td>
<td>6</td>
<td>0.83</td>
<td>6</td>
<td>0.154</td>
<td>6</td>
<td>0.037</td>
<td>0.023</td>
<td>9</td>
<td>375</td>
</tr>
<tr>
<td>0.9998</td>
<td>0.9991</td>
<td>7</td>
<td>0.71</td>
<td>7</td>
<td>0.113</td>
<td>7</td>
<td>0.023</td>
<td>0.011</td>
<td>6</td>
<td>550</td>
</tr>
<tr>
<td>0.9999</td>
<td>1.0000</td>
<td>8</td>
<td>0.62</td>
<td>8</td>
<td>0.087</td>
<td>8</td>
<td>0.015</td>
<td>0.005</td>
<td>2</td>
<td>296</td>
</tr>
<tr>
<td>0.9997</td>
<td>0.9991</td>
<td>9</td>
<td>0.56</td>
<td>9</td>
<td>0.068</td>
<td>9</td>
<td>0.011</td>
<td>0.039</td>
<td>9</td>
<td>223</td>
</tr>
<tr>
<td>0.9997</td>
<td>0.9995</td>
<td>10</td>
<td>0.50</td>
<td>10</td>
<td>0.055</td>
<td>10</td>
<td>0.008</td>
<td>0.019</td>
<td>5</td>
<td>259</td>
</tr>
<tr>
<td>0.9993</td>
<td>0.9993</td>
<td>11</td>
<td>0.45</td>
<td>11</td>
<td>0.046</td>
<td>11</td>
<td>0.006</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.0000</td>
<td>0.9994</td>
<td>12</td>
<td>0.42</td>
<td>12</td>
<td>0.039</td>
<td>12</td>
<td>0.005</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.9996</td>
<td>1.0000</td>
<td>13</td>
<td>0.38</td>
<td>13</td>
<td>0.033</td>
<td>13</td>
<td>0.004</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.9997</td>
<td>1.0000</td>
<td>14</td>
<td>0.36</td>
<td>14</td>
<td>0.028</td>
<td>14</td>
<td>0.003</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.0000</td>
<td>0.9998</td>
<td>15</td>
<td>0.33</td>
<td>15</td>
<td>0.025</td>
<td>15</td>
<td>0.002</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.9989</td>
<td>0.9999</td>
<td>16</td>
<td>0.31</td>
<td>16</td>
<td>0.022</td>
<td>16</td>
<td>0.002</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.0000</td>
<td>0.9996</td>
<td>17</td>
<td>0.29</td>
<td>17</td>
<td>0.019</td>
<td>17</td>
<td>0.002</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.9988</td>
<td>0.9986</td>
<td>18</td>
<td>0.28</td>
<td>18</td>
<td>0.017</td>
<td>18</td>
<td>0.001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.0000</td>
<td>0.9999</td>
<td>19</td>
<td>0.26</td>
<td>19</td>
<td>0.015</td>
<td>19</td>
<td>0.001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.9989</td>
<td>0.9993</td>
<td>20</td>
<td>0.25</td>
<td>20</td>
<td>0.014</td>
<td>20</td>
<td>0.001</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0.9996 0.9996
Quantifying Quality of Non-Functional Quality Attributes Using Customer Survey Metrics

Vanitha Sugumaran* and Janet Drake**

* IT Analyst, John Deere, Waterloo, Iowa- 50614
Sugumaravanitha@JohnDeere.com
**Department of Computer Science, University of Northern Iowa,
Cedar Falls, IA- 50614
drake@cs.uni.edu

Abstract

Satisfying customers is an essential element in software development because it enables service providers to benchmark their performance and to identify areas that require improvement (Das et al., 1999). However, one of the key challenges in the software quality research is to quantify the quality of various non-functional attributes. The goal of this study is to quantify quality of non-functional quality attributes using a customer survey approach. In order to quantify quality from the customer’s perspective the following steps were followed a) Identified quality characteristics, b) developed questionnaire and a tool to conduct a quality requirement collection, c) quantified the Quality Expectation Score (QES) from expectation survey, d) conducted satisfaction survey after product delivery and quantified the Quality Satisfaction Score (QSS), and e) finally compared QES with QSS to judge how well the customer’s expectations were met. The methodology was designed and tested with a product as a case study. The survey raw data were converted into information that can be used by managers in understanding customer’s expectation and satisfaction. The result of the expectation survey showed the importance of different attributes and customer’s expectation. The satisfaction survey results explained where improvements were needed. Final comparison between satisfaction and expectation survey showed how the quality satisfaction score

---

1 The case study was not conducted in John Deere.
can help software engineers and other technical staff to identify opportunities for ongoing process improvements and to monitor the impact of those improvements.

1. Introduction

Software plays an important role both in our personal and work lives. As the software take control of so many aspects of our life, the failure of software results in immense losses. For example, software bugs are costing the U.S. economy an estimated $59.5 billion each year, in which more than half of the cost is borne by end users and the remainder by the developers and vendors (National Institute of Standards and Technology [NIST]), 2002). In another study by Boston Globe (2000) reported that the cost of failed projects for the U.S. IT industry in the year 2000 was estimated at $84 billion. Thus, delivering “quality product” or “quality services” is important concern in software development field.

To understand and measure quality, researchers have often built models of how quality characteristics relate to one another. Best known quality models are ISO 9126, McCall’s model, Boehm’s model and Dromey’s model using functional and non-functional attributes. Although many researchers mentioned the importance of non-functional attributes, non-functional issues have received little attention compared to functional issues in the software quality measurements. “Non-functional requirements are explicit statements of some aspect of the design, development or performance in the very broadest sense of an application” (Acquisition Management System, 2005). During the software development process non-functional requirements are not very often taken into account (Rosa et al., 2001) Franch (1997) precisely summarizes why non-functionality is so important. He says “the lack of non-functional issues in software components has some negative effects on many software development tasks that include specification, implementation, maintenance and reusability”. In every definition of quality the satisfaction of user’s expectations is emphasized (Stavrinoudis et al., 2005).

In addition, satisfying customers is also an essential element in software development because it enables service providers to benchmark their performance and to identify areas that require improvement. However, one of the key challenges in software
quality research is to quantify the quality of various non-functional requirement attributes from the customer perspective. By getting the user’s perception of software quality, we can proceed towards corrective actions and define market strategies. (Stavrinoudis et al., 2005). As Svanberg, et al. (2005) points out that quality can not be added to the systems as an afterthought, it has to be built into the system right from the beginning. In order to add quality from the beginning, it is important to know what the customer’s expectations regarding quality. Most of the researchers have focused on how to conduct customer satisfaction surveys efficiently and how customer’s opinion changes progressively as they use the software.

The goal of this study is to quantify quality of non-functional quality attributes using customer survey approach. This was achieved through surveying the customers before the product development to get their non-functional quality expectation level and after the product development to get their quality satisfaction level.

2. Methodology

In order to quantify quality from the customer’s view the following nine steps were followed (Figure 1). This study was designed and tested with a small company where the end product requirement was developed. The end product’s main functionalities are data entry, data manipulation and visualization. First step was to identify the important non-functional attributes to end product. Several Meetings were organized with customers resulting in identifying the five most important non-functional attributes Reliability, Maintainability, Security, Performance and Usability. For this study the customers answered two surveys, first survey was to help identify their quality expectations for these five non-functional attributes and second survey was to measure their satisfaction level. The first survey was taken in the analysis phase and the second survey was conducted two months after the product delivery.
The second step involved the development of questionnaire. After successfully identifying the quality characteristics, a questionnaire was developed. The questions were created focusing on what would be important to day to day users, IT managers and the business owners within the company. The questionnaire was given at the end of the functional requirement collection but before customers signing off the functional requirement document. Similarly, for each non-functional quality attributes (*Reliability*...
(RE), Maintainability (MA), Security (SE), Performance (PE) and Usability (US)) five questions that were well related to the product were developed. Customers were asked to give numerical score (NS) of 1 to 5, 1 being unimportant and 5 being very important. A numerical score was calculated using an average of all the five questions. Then the customers were also asked to weigh (W) the characteristics to get the relative importance. W of all the characteristics is equal to 1. Importance of weighting factor in survey was discussed by several authors (Voas, 2003). To make this weighing process easier, customers were asked how much they would be willing to spend for each characteristic if they were given 100% budget. Finally the quality expectation score was calculated using the following formula.

\[
\text{Quality Expectation Score (QES)} = \sum_{i=1}^{5} \text{NES}_i \cdot W_i
\]

Where \(\text{NES}_i\) = mean of all the 5 expectation questions for a quality characteristic

The overall Quality Expectation Score was calculated using total Quality Expectation Score (QES) upon total number (n).

Overall Quality Expectation Score (QES) = \(QES_j / n\)

After creating relevant questions, the customers were educated to understand the terminology used to explain the characteristic of the quality. We gave a presentation covering the meaning of Reliability, Maintainability, Security, Performance and Usability and the goal of this experiment.

In third step, a survey tool was developed because it was easy for the customers to answer the questions, save the survey results directly into the database and for the data manipulation. In order to achieve this goal, the user interface for the end product was developed in Java and on the backend SQL server was used as the RDBMS.
customer’s expectation responses were collected and stored in the database. After collecting the quality expectations from the customers, the data was analyzed and interpreted.

After the product was delivered to the customer, the survey tool was used again for the satisfaction survey. The feedback was received through another survey called “satisfaction survey”. The expectation survey was used to understand the customers’ need before the product developed, whereas the satisfaction survey was conducted to study how the customer was satisfied with the end product. Once the product was delivered and the users started using the product, the same questionnaire was given again to understand the customer satisfaction with few changes. The changes include instead of asking the expectation score, the users were asked to rate the questions based on how satisfied they are with the product and they were not asked to weigh the characteristics. Customers were again asked to give numerical score of 1 to 5, 1 being unsatisfied and 5 being satisfied. Customers used the same tool with some word changes for the second survey. Finally, quality satisfaction score was stored in the database and later analyzed. Similar to expectation, a Quality satisfaction score was developed with the following formula.

\[
5
\]

\[
\text{Quality Satisfaction Score (QSS)}_{j} \text{ for one customer} = \sum_{i=1}^{5} \text{NSS}_{i} \ast W_{i}
\]

Overall Quality Satisfaction Score (QSS) = \( \frac{\text{QSS}_{j}}{n} \)

NSS – Numerical score from satisfaction survey
W – weigh from original survey
n – Number of customers surveyed

This score was compared with the Expectation Score to see how much the users were satisfied.
3. Results and Discussion

We have used four ways to view the customer survey results for both expectation and satisfaction. The first approach examines the distribution of the detailed response data for each key quality requirement. The second method compares the quality expectation between attributes. The third view shows the order of relative importance of the attributes from the weighing method and final method summarizes the overall quantification score of the quality.

3.1. Quality Expectation Survey Results

3.1.1 Distribution of Each Key Quality Requirement

To produce this report, the survey responses to all five questions related to those quality characteristic were averaged for expectation level and for importance. For each requirement, the value was plotted as bar graph. The expectation number is shown on the X axis, and Y-axis shows the expectation questions. In addition, Standard Deviation (STDEV) is also calculated to see how many customers are close to the mean value. In other words, STDEV helps to understand if customers are agreeing with each other. The STDEV is the short line on the X axis along with the expectation number. This view allows managers and technical staff to quickly identify expectation requirements from the customers. Table 1 shows customer expectation survey questions for each quality attribute. In this paper we only give a detailed description of one quality characteristics – reliability.
Table 1 Customer Expectation Survey Questions.

<table>
<thead>
<tr>
<th>Quality Attribute</th>
<th>Questions</th>
</tr>
</thead>
</table>
| **Reliability**   | RE1. Implementation of the required functionalities without crashes or service interruptions  
                    RE2. Implementation of the required functionalities without any errors or problems  
                    RE3. Ability to recover from crash in timely manner  
                    RE4. Ability to recover from crash without losing data  
                    RE5. Being able to do a task without doing workaround |
| **Maintainability** | MA1: Ability to add functionalities to the software in future.  
                         MA2: Ability to correct the errors reported by the users in timely manner  
                         MA3. Availability of software 24/7  
                         MA4. Availability of technical support in case of difficulty  
                         MA5. Software is not affected by multiple users |
| **Security**      | SE1. Editing department specific information should be restricted to certain users.  
                         SE2. Deleting department specific information should be restricted to certain users.  
                         SE3. Importing data from other sources should be restricted to certain users.  
                         SE4. Importance of restricting users to view certain sections (example: Admin section)  
                         SE5. Importance of restricting users to adding new user should be restricted to certain users. |
| **Performance**   | PE1. There should not be any performance delays with multiple users.  
                         PE2. Updates must show immediately.  
                         PE3. All new records must show up immediately.  
                         PE4. Software should have same performance while running on different platform.  
                         PE5. Application should start-up immediately. |
| **Usability**     | US1. Importance of having screens similar to the application currently in use.  
                         US2. In case of error the displayed message should be easily understandable.  
                         US4. Software should display status messages in case of operational delays.  
                         US5. Software should be easily learnable for novice users. |
Figure 2. Reliability (RE) expectation (EX) summary (RE_EX_STDV = Reliability Expectation Standard Deviation).

Figure 2 illustrates an example for reliability expectation report that summarizes the survey results and indicates the current customer expectation level with each of the reliability quality attribute requirements. From this figure it is clear that the overall expectation for reliability shows the value above 4 that indicates the customer expectations for all the five questions are very important. It is also clear for IT managers and developers from figure 2 that customer was very keen on crash free product and more emphasis should be given for these attributes. Summary report also reveals that the “implementation of the required functionalities without crashes or service interruptions” is much more important to the customer than other reliability requirements. It is evident from the standard deviation that everyone agrees. It is also interesting to note that the customers are some what willing to have a workaround if we can not satisfy a functional requirement.
3.1.2 Comparison between Five Expectation Quality Attributes

Figure 3 shows the customer’s expectation for all the quality attributes. This was calculated by averaging all the responses of the user surveyed. It is clear from the chart is that customer’s overall expectation was very high with all scores above 3.5. Within these five attributes, reliability and performance have the highest value and it shows that customer wants very reliable software. Security is the least important attribute. The order of importance was Reliability > Maintainability > Performance > Usability > Security.

![Expectation Quality Attributes Comparison](image)

*Figure 3. Comparison between expectation quality attributes.*

3.1.3 Weighting between the Quality Attributes

Users were asked to weigh the attributes to show which quality attribute is important to this particular product. In this Figure 4 it is clear that users rated usability and reliability were the most important quality attributes for this product. Security and performance were given relatively low importance. The order of relative importance was Usability > Reliability > Maintainance > Performance > Security.
3.1.4 Quantifying the Overall Quality Expectation Score (QES)

The main goal of this research is to quantify the quality and provide a single score which will provide not only the developers but also all the stakeholders to understand the overall customer’s expectation and satisfaction. The overall Quality Expectation score (QES) was calculated using the formula given below.

\[
QES = (M_{RE} \times W_{RE}) + (M_{MA} \times W_{MA}) + (M_{SE} \times W_{SE}) + (M_{PE} \times W_{PE}) + (M_{US} \times W_{US})
\]

Where: MES: Mean expectation survey for each quality attribute and W: Weighing given by the customer for that characteristic.

In this case study, overall QES was 4.34. This shows that the quality expectations were very high for the customer. There are nine customers surveyed in this example. We are aware that this number is not statistically significant, but these 9 people are 100% customers involved with this software.

From the quality expectation survey it was easy to understand what the customers are really looking for with respect to quality. During the functional requirement collection users were stating that security was very important, but the survey actually showed that...
users were really looking for reliable and usable software. This expectation survey also
gave us an opportunity to discuss other problems related to quality and helped to improve
customer relationship. It gave the opportunity for the customers to convey what they
really wanted instead of a round table meeting, where their thoughts can be influenced by
others. This process made users understand that a software product has its own
characteristics. It is also interesting to note that IT administrator’s expectation is very
different from end users expectation (Figure 5).

![IT Admin vs. User](image)

*Figure 5. Quality expectations between IT admin and end users*

### 3.2. Quality Satisfaction Survey

After the quality requirements were collected, the product was developed,
tested and delivered to the customers. Customers were using the product for
approximately three months, and then the satisfaction survey was conducted. Figure 7
shows the summary of Quality Satisfaction Survey results for all five attributes. The
expected results are also shown in Figure 6. In the satisfaction survey same questions
were asked but users were asked to rate the questions based on how satisfied they were
with the product.

Even though we analyzed all five attributes key quality requirements in the
project, in this example, we are demonstrating only the reliability satisfaction. The
comparison graph clearly depicts that overall none of the reliability qualities are met the
customer’s expectations (Figure 6). The graph shows that the satisfaction level for attribute reliability is more than 2.5 when comparing with expectation level; it is obvious that the product’s reliability needs some improvement, especially for Reliability question 1 which has very low satisfaction level 2.78. The result shows that product is not as stable as the users wanted. These scores give the technical staff and managers to quickly view the product status. As we look closely, Reliability question 1, Reliability question 2 and Reliability question 5 i.e. crash, error handling and workarounds are required more attention than other two questions.
Figure 6. Quality Satisfaction Score summary for all five attributes. EX stands for Expectation, SA – Stands for Satisfaction.
3.2.2 Overall Comparison between Expectation and Satisfaction survey

The Figure 7 below shows customer expectation and the satisfaction results. This graph helps to see where improvement is needed. It is clear from the graph that customer’s satisfaction scores were lower than their expectation scores. Out of these five attributes, the customer had higher satisfaction for usability and maintainability attributes. The least satisfaction was noted for security attribute. Even though the customer reliability expectations were high, the product did not achieve the higher customer satisfaction. These score shows the developers can do work more on reliability related issue.

![Graph showing expectation vs. satisfaction](image)

Figure 7. Comparison between expectation and satisfaction survey scores

3.2.3 Quantifying the Overall Quality Satisfaction Score (QSS)

Quality satisfaction for a customer is calculated in this tool, the same weight is used from the expectation survey.

Quality satisfaction score (QSS) for a customer

\[
\text{QSS} = (\text{NS}_\text{RE} \times \text{W}_\text{RE}) + (\text{NS}_\text{MA} \times \text{W}_\text{MA}) + (\text{NS}_\text{SE} \times \text{W}_\text{SE}) + (\text{NS}_\text{PE} \times \text{W}_\text{PE}) + (\text{NS}_\text{US} \times \text{W}_\text{US})
\]

Where \( \text{NS}_x \) = mean of all the 5 satisfaction questions for a quality characteristic for \( x \), and \( x \) is one of the 5 quality characteristics.

Overall satisfaction score is = 3.19

The overall satisfaction score of 3.19 shows clearly that product needs improvement.
Expectation score is 4.34 and the satisfaction is 3.19. We have met 73.5% of customer’s expectation. This helps to understand how much our customers are satisfied with our product.

3.3. Limitations of this Study

It is important to critically evaluate the study. Some of the limitations of this study include survey audience and sample size, questionnaire development, importance of weighting,

The first problem was related to survey audience particularly to figure out whom to survey and sample size. We discussed about whether the survey is to be given to all the stakeholders or only to the end users. As literature clearly says that correctly determining the target population is critical and if you do not survey the right kinds of people, you will not successfully meet the product goals. In this study we have collected the functional and quality requirements from the business owners, IT administrators and the end users. The challenge in this approach was the level of technical knowledge from this varied group. In the beginning of the process, end users did not understand the technical details and how it impacts the product. We have successfully educated the end users and then the survey was done. The next issue was about how many people you need to survey. Statisticians know that a small, representative sample will reflect the group from which it is drawn. The larger the sample, the more precisely it reflects the target group. However, the rate of improvement in the precision decreases as your sample size increases. As mentioned before, in this research we had only 9 customers using the product with varied background. This may not be statistically significant. However, our goal was to validate our methodology. Future study will explore the sample size and its impact on customer survey.

The second problem was creating the questionnaire. We have chosen five important questions for each quality attribute. We believe if more questions were asked subjects would not think carefully about each questions. Before the survey was given to the customers we piloted the questions with one end user to make sure the questions were understandable. We have created the questions related to the product, and the questions can be modified to fit another product.

The third problem was related to explaining the weighting to the users, we asked them to divide the budget of 100% among the characteristics, and this way they understood how to weigh the characteristics. The weighing helped to get the relative importance of the attributes (Voas, 2003) to end product. From the expectation survey results, it seemed that Reliability was an important attribute for this product, but the weighting show that users really valued Usability.

We also debated sending out the paper survey or to asking them to take a survey using a tool. Tool has minimized our data entry work and it has been reused to give the satisfaction survey.
4. Conclusion and Future Direction

Understanding what the customer wants and expects is very important to any type of business. This study developed a methodology to quantify the quality of non-functional quality attributes using customer survey approach. The survey raw data were converted into information that can be used by managers in understanding customer’s expectation and satisfaction. Of the non-functional quality characteristics, Reliability, Maintainability, Security, Performance and Usability were identified as important attributes. The result of the expectation survey showed the importance of different attributes and customer’s expectation. Within these five attributes, reliability and performance have the highest value and it shows that customer wants very reliable software. The order of importance was Reliability > Maintainability > Performance > Usability > Security. The detailed reliability expectation score showed an overall expectation score of above 4. Customers were very keen on a crash free product. On the other hand, security was the least important attributes in customer expectation. From the customer survey, the average security expectation score was only 3.6. The second approach was to identify the relative importance between the quality attributes using weighting method. It also showed that usability and reliability are the most important quality attributes for this product. The order of relative importance was Usability > Reliability > Maintainability > Performance > Security. The final method produced the overall quality expectation score that provided a single score which helps the developer to understand the overall customer’s expectation. The overall quality expectation score was 4.34 which indicate the quality expectations were very high for the customer. These three quantification methods help the developers and managers to understand the overall customer expectations of the product. Expectation survey result has allowed developers to set a goal related to non-functional quality. While testing the product special attention was given towards non-functional quality expectations.

Comparison between satisfaction and expectation survey showed how quality satisfaction score can help software engineers and other technical staff to identify opportunities for ongoing process improvements and to monitor the impact of those improvements. Overall for all the five attributes, the customers’ satisfaction scores were lower than their expectation scores. Out of these five attributes, the customer had higher satisfaction for usability and maintainability attributes. The least satisfaction was noted for security attribute. The 3.19 satisfaction score and 4.34 expectation score shows that product can be improved.

The method used in this study to quantify the quality through customer survey provides a tool for the developers and managers to track the product development and successful process. However, the present study has certain limitations including the survey audience background and sample size. The future direction of the research will explore how customers with different background (for example IT admin versus end users) impact...
the result? The second limitation is related to the sample size used in this study. The future study will also concentrate on the development of online or web-based customer survey and also study the optimum number of questions to be asked to the customer when conduction survey. Further more, future direction will answer the question such as “after the delivery should we conduct surveys in fixed time intervals?” and “Will it help show how customer’s satisfaction levels are improving?”

Acknowledgements
The authors wish to acknowledge the company T8Design, where this case study was conducted, and the customers who took the survey. Authors also would like to thank Dr. Voas for his timely advice and comments.

References


About the Authors:

Vanitha Sugumaran is working as an IT analyst in John Deere. Before she joined John Deere, she worked at T8Design as a Software Engineer. She is also working on her master’s degree in computer science from University of Northern Iowa and will be graduating this December. Her area of research interests includes improving software quality, improving customer satisfaction and requirement analysis.

Janet Drake is an Associate Professor of Computer Science at the University of Northern Iowa. She received her PhD from the University of Minnesota. Her area of interest is software engineering with emphasis is requirements engineering and software testing.
Web Services Testing for Reliability Assessment

Jia Zhang**,1, Zongwei Luo**

* Department of Computer Science, Northern Illinois University, DeKalb, IL 60115, USA
** E-business Technology Institute, The University of Hong Kong, Pokfulam Road, Hong Kong

jiazhang@cs.niu.edu, zwluo@eti.hku.hk

Abstract

How to effectively and efficiently test and assess available Web services with similar functionalities published by different service providers remains a challenge. In this paper, we present a boundary value-based automatic test case generation approach.

1. Introduction

According to the Stencil Group, one leading industry analyst firm, Web services are defined as “loosely coupled, reusable software components that semantically encapsulate discrete functionality and are and programmatically accessible over standard Internet protocols.” [1] This paradigm of Web services has been changing the face of Internet from a repository of data into a repository of services in several significant ways [2]. First, the model of Web services facilitates Business-to-Business (B2B) e-Commerce within and across organizational boundaries, by means of business organizations enabling universal Internet access to their software services through standard programmatic interfaces [3]. Second, the Web services technology enables cross-language and cross-platform interoperability for distributed computing and resource sharing. Third, the paradigm of Web services enables rapid development of new business software by integrating published Web services as components. In short, the paradigm of Web services is widely considered to be the strategic model for the next generation of Internet computing [4, 5].

The backbone of the Web services paradigm comprises of three fundamental techniques: (1) communication protocols, (2) service descriptions, and (3) service registration and discovery [6, 7]. Each category has its own ad hoc standard: the Simple Object Access Protocol (SOAP) [8] acts as a lightweight protocol for exchanging structured and typed information between Web services; the Web Service Description Language (WSDL) is an eXtensible Markup Language (XML)-based description language that is used to describe the programmatic interfaces of Web services [9]; and the Universal Description, Discovery, and Integration (UDDI) standard [10] provides a mechanism to publish, register, and locate Web services.

This SOAP+WSDL+UDDI foundation ensures publication, discovery, and transportation of a specific Web service. However, as more and more Web services are published onto the Internet on a daily basis, it is apparently that a service requester may need to make a decision facing a large set of available Web services with similar functionalities. How to effectively and efficiently test, assess, and select a Web service that match most predefined requirements becomes critical [2]. Moreover, a Web service may utilize other third-party Web services as components, which fact further worsens the problem.

Typically, a service requester uses quantitative or qualitative Quality of Service (QoS) to assess and distinguish a Web service. The QoS of a Web service is normally measured against a set of persistent software attributes, or so-called “ilities,” such as reliability, scalability, efficiency, security, reusability, adaptability, interoperability, maintainability, availability, portability, etc [11]. If a Web service scores high in the evaluation of these attributes (here we omit the exception that some attributes naturally conflict with each other, such as the fault tolerance and testability,) it is considered as of high quality [2].

Last fifty years of software development history has witnessed the establishment of a research branch software testing, which contains a wealth of theories, technologies, methodologies, and tools in order to guide the verification process of a software product against each of the above attributes [12]. However, Web services possess unique features; thus, their testing and assessments require further investigation. The model of Web services implies that Web service components can be dynamically searched, located, and invoked at run time [13-16]. This distinctive feature of dynamic discovery and invocation requires highly efficient testing and assessment of Web services components.

In this paper, we aim to explore effective and efficient techniques of automatic Web services test case generation.
for reliability verification of Web services-oriented systems. By “Web services-oriented system”, we mean a software system that consists of components that will be fulfilled by Web services. It can be a standalone Web service or a system composed of multiple Web services components. Here we adopt the standard definition of software reliability: Musa defined software reliability as the probability of failure-free operation of a computer program for a specified time in a specified environment [17]. Our essential idea is to automatically elicit test cases from WSDL documents. The remainder of the paper is organized as follows. We will first introduce our boundary value-based automatic test case generation approach. Then we will discuss our preliminary experiments. Finally we will draw conclusions and discuss future work.

The remainder of this paper is organized as follows. In Section 2, we discuss related work. In Section 3, we discuss our boundary values-based Web services reliability testing approach. In Section 4, we discuss in detail how to generate test cases. In Section 5, we present our preliminary experiments. In Section 6, we make conclusions and discuss future work.

2. Related work

Casati et al. suggested that Web service providers define service quality metrics, which contain non-functional parameters specifying the cost, duration, and other characteristics of a service, to help service requestors make decisions over multiple candidates [18]. However, their work remains as a high-level abstraction without technical discussions such as how to construct service quality metrics.

Simulation has been utilized to validate and monitor Web services composition. Narayanan and Mellraith translate DAML-S service descriptions of composite services into a Petri nets formalism in order to provide decision procedures for Web services simulation, verification, and composition [19]. Cardoso and Sheth use simulation to validate Web services composition based upon a mathematical Quality of Service (QoS) model that emphasizes on timeliness, cost of service, and reliability [20, 21]. Miller and colleagues focused on utilizing simulation analysis to monitor Web process composition [22]. Lerner uses parameterized state machine to verify process models [23]. Contrast with their work, our research focuses on efficiently generate test cases to assess Web services and concentrate on reliability attribute.

Researchers argued that the selection of component services should be performed during the execution of a composite service, instead of at design time [13-15]. Zeng et al. proposed a global planning selection approach that not only takes into account multiple criteria (e.g., price, duration, reliability), but also global constraints and preferences set by service requestors (e.g., budget constraints) [14]. Menascé argued that a Web service should be characterized by its functionality, QoS attributes, and cost [13]. He proposed that the total execution time and cost of the whole composite Web service be calculated for the selection of each individual service component. However, there still lacks a comprehensive QoS model of Web services. Contrast with their work, our research intends to investigate how to assess reliability of Web services.

Zhang et al. [24] explored how to measure reliability of Web services using the techniques of mobile agents. Menascé stated that QoS measures should be evaluated from different perspectives: from service providers’ perspective and from service requestors’ perspective [25]. Menascé presented a way to calculate response time for composite Web services on the perspective of service requestors [25, 26]. Compared with those work, this research explores how to generate test cases for Web services reliability assessment.

Offutt and Xu propose to adopt data perturbation technique to generate test cases of testing message communications between pairs of Web services. Data perturbation includes two approaches: data value perturbation modifies values according to the data types specified by Web services; and interaction perturbation tests on RPC communication and data communication [27]. Their goal is to use mutation analysis to find faults from Web services. In contrast with their approach, this research aims to help service requestors automatically create test cases to select Web services found from public registry. From a service requestor’s perspective, a Web service is a complete black box with its published WSDL definitions. Therefore, the basis of our test cases generation is the found WSDL definition files of the Web services. In addition, their research uses machine-related boundary values as data perturbation strategy (e.g., largest number for a double data type). Our research proposes much finer-grain strategy to find boundary values based upon constraining facets and XML schema-referenced data type standards.

3. Boundary values-based Web services testing

Our research applies boundary values and faulty data to test Web services candidates. Our major strategy is to generate test cases to eliminate Web service candidates. The core challenge is: how to create appropriate test cases to efficiently test a Web service candidate. Since it is obviously impractical to test every piece of datum in the possible input space outlined by the operational profiles, the question can be broken down into the following two
questions: (1) How to decide possible test case space? and
(2) How many test cases are sufficient and necessary to
obtain the full state of a remote Web service?
In order to answer these two questions, let us re-
examine the goal of the test cases that we are interested.
As we discussed earlier, the test cases intend to carry back
the full states of a remote Web service. The full states of
the Web service are based upon the entire input space,
which ought to involve the input data that are able to test
both the functional and non-functional attributes of the
Web service. The test data related to the functional
attributes intend to test whether the Web service fulfill the
functional requirements from the operational profile; and
the ones related to non-functional attributes intend to test
the ilities of the Web service, which in the context of the
specific scenario in our paper imply the reliability of the
Web service, and more specifically its interoperability.
Our proposed solution is to utilize boundary values
together with faulty data perturbed from boundary values
to quickly verify the reliability of a Web service
candidate. Each test case will test a Web service upon a
function call which signature contains several parameters.
Each parameter requires a specific data type with implicit
boundary constraints. Our approach focuses on finding out
the boundary values for each input parameter’s data type.
Let us take a simple example: suppose that a Web ser-
vice exposes a WSDL interface that includes a string-type
parameter defined as follows:

<part name="loginId" type="xs:string"/>

For this parameter, we can test on boundary values
such as: null, "" (empty string), short string (i.e., one
character long), very long string (e.g., 200 characters
long), string containing “new line” character, non-string
value (i.e., integer 3), etc.
For every WSDL interface exposed by a Web service,
we will list boundary values for each input parameter.
Then we will assemble all boundary values for each input
parameter to obtain a list of test cases. For example,
suppose a Web service interface contains three input
parameters, each one being a string type without further
constraints. As shown above, each parameter can have
five boundary values. Assembling them together, we will
get a list of fifteen different test cases for the functional
call.
In short, our strategy of generating test cases to test the
correctness of a Web service is to find boundary values
for each parameter. These boundary values are definitely
within the input domain. In order to test the fault tolerance
of a Web service, we perturb the boundary values to
generate faulty data as test cases. Injecting faulty data to
verify fault tolerance is not new. Traditional software
testing establishes the fault injection technique [28, 29].
However, applying the traditional fault injection technique
to the domain of Web services testing remains
challenging. In more detail, we adopted the basic concept
of the fault injection but we need to solve corresponding
technical issues.
Here we will first briefly review the concept of fault
injection and then discuss the technical challenges we are
facing in the domain of Web services. Derived from the
technique used in traditional industry for a long time (e.g.,
automobile manufacture), fault injection is a set of
techniques that provide worst-case predictions for how
badly a system will behave in the future [28, 29]. More
specifically, the Interface Propagation Analysis (IPA)
technique proposed by Voas and colleagues is an
advanced fault injection technique to test upon black-box
like software systems [30]. We believe that IPA is a right
candidate concept to test the reliability of Web services
due to the following reason: similar to normally called
Commercial-off-the-shelf (COTS) components, users of
Web services have no access to their internal source code.
Users can only access Web services via Simple Object
Access Protocol (SOAP) [31] request messages, and get
results from Web services via SOAP response messages.
Therefore, Web services exhibit as black-box systems
from users’ perspectives.
The IPA technique suggests to inject corrupted data to
the input of a black-box system [28], and monitors the
output of the system to obtain knowledge of its fault
tolerance, as shown in Figure 1(a). IPA can help us test
the vulnerability of a Web service serving as a component

![Figure 1. Injecting faulty data for interoperability testing](image-url)
in a software system with respect to two levels: (1) the Web service in isolation, and (2) the Web service as a component interoperative with other parts of a system. As shown in Figure 1(b), the second level can be considered along with two scenarios: (2.1) when the Web service component returns corrupted information or no information at all, and (2.2) when the Web service fails to interoperate with other components of the system. In short, IPA can be applied to test the degree of how a system can tolerate a Web service as a component; or in other words, IPA can help test the interoperability of a Web service in a system.

However, although the basic concept of IPA seems appropriate to be applied to test both the fault tolerance and interoperability of Web services, how to apply the IPA technique in the domain of Web services remains a challenge. The core challenge of the IPA technique is how to create corrupted data for a testing component. Voas and colleagues proposed to perturb the input domain to find corrupted data [28]. In traditional component-based testing, a testing component is already deployed in its execution environment; thus, it is feasible to conduct arbitrary amount of testing over the testing component. When we deal with Web services, on the other hand, we are facing remote Web components so that network traffic needs to be considered imperatively, let alone the fact that some Web services might have access charges associated. Furthermore, unlike traditional software components, Web services found from public registries oftentimes reveal limited information except the access prototypes defined in WSDL. Therefore, our strategy of designing faulty data to test the fault tolerance of a Web service focuses on perturbing the boundary values for each input parameter’s data type. Let us take a simple example: suppose that a Web service login function requires a string-type input parameter with a length limitation of 6 to 16 characters. 6 and 16 character-long strings are both boundary values for the input parameter. Perturbing these two boundary values, we can obtain 5 and 17 character-long strings, which can be used as faulty data to test the fault tolerance of the Web service.

The generated faulty test cases can also be used to test the interoperability of a Web service, as shown in Figure 1(b). As we discussed earlier, we shall actually test the interoperability of its substitute - X' - of the remote Web service. Faulty data should be injected into the X', and then we will monitor the output of X' and the output of its successor X, and so on, as shown in Figure 1(b). Notice that the output of X' will be the input of X. Since X' is a substitute, in order to enable the testing, X' should already hold the corresponding output values that the remote Web service will produce from the expected faulty data. In other words, in order to facilitate the local interoperability testing, the corresponding mobile agent should not only carry the test data from the normal input domain specified by the WSDL interface of the Web service, but also the faulty input data to acquire a comprehensive state of the remote Web service.

In summary, the faulty data should be divided into two sets with different purposes: (1) to test the Web service in isolation, as shown in Figure 1(a); and (2) to test the Web service as a component in the system environment, as shown in Figure 1(b). In order to test the vulnerability of a Web service in isolation, we will perturb each boundary values to generate faulty test cases, and monitor and analyze the post-condition of the Web service to decide whether the output events from the Web service is undesirable. It should be noted that certain amount of testing should be performed to achieve particular level of assurance. On the other hand, in order to test the interoperability of a Web service in its final operating environment, specific operation scenarios and profiles need to be considered, in addition to our proposed boundary value perturbing approach. Exploring generating test cases based upon operation profiles is an area of future research.

4. The design of test cases

To be specific, a test case of a Web service is a set of mappings between input variables and their values. Each test case can be used to generate a SOAP input message to test a corresponding Web service.

4.1 The design of test cases for correctness of individual Web services

As we discussed in the previous section, we will test the correctness of a Web service by applying test cases with boundary values elicited from the Web services interfaces written in WSDL. Here we will first briefly introduce the related WSDL specification on Web services interface definition; then we will discuss in detail how we extract test cases (including both testing data and faulty data) from WSDL definition.
WSDL is “an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information” [32]. As shown in Figure 2, using the WSDL, a Web service is defined as a set of ports, each publishing a collection of port types that bind to network addresses using common binding mechanism. Every port type is a published operation that is accessible through messages. Messages are in turn categorized into input messages containing incoming data arguments and output messages containing results. Each message consists of data elements; and every data element must belong to a data type, either a XML Schema Definition (XSD) simple type or a XSD complex type. In summary, in order to design test cases for a Web service, our basis is its WSDL operations, input messages, and output messages. For simplicity, we do not consider to design test cases on the binding of the Web service.

Our basic strategy is to design test cases based upon boundary values of each formal argument of the published WSDL definition of the Web service. It should be noted that one major motivation is to enable automatic test case generation. Therefore, our challenge here turns into how to find efficient boundary values for each formal argument. Since each input parameter must be a XML-allowed data type, it can be either XML built-in types or user-defined compound types, as shown in Figure 3. Let us discuss XML built-in type first.

4.2 Boundary values for XML built-in primitive type

As shown in Figure 3, XML built-in types include built-in simple types and built-in complex types. The former can be in turn divided into built-in primitive types and built-in derived types. A built-in complex type is defined in terms of built-in primitive types and built-in derived types by unioning their value spaces and lexical spaces. Built-in derived types actually depend on built-in simple types [33]. In other words, built-in primitive types are base types, and other types can be derived in terms of primitive types. Thus, we only need to investigate how to extract boundary values for built-in primitive types.

As shown in Table I, W3C Specification of the XML Schema language defines nineteen built-in primitive types: string, decimal, boolean, duration, dateTime, time, date, gYearMonth, gYear, gMonthDay, gDay, gMonth, base64Binary, hexBinary, float, double, anyURI, QName, and NOTATION [34] [33]. For each primitive type, W3C XML specification defines a set of constraining facets. Each constraining facet restricts an aspect of the value space of a built-in primitive type (e.g., minimum value,
maximum value, etc). Taking string as an example, it has six constraining facets: length, minLength, maxLength, pattern, enumeration, and whiteSpace.

As summarized in Table I, there are altogether twelve kinds of constraining facets: (1) length: the number of units of length based upon data types, (2) minLength: the minimum number of units of length, (3) maxLength: the maximum number of units of length, (4) pattern: regular expression that restricts the lexical spaces to literals, (5) enumeration: a set of values, (6) whiteSpace: space, tab, line feed, and carriage return, (7) maxInclusive: inclusive upper bound, (8) minInclusive: inclusive lower bound, (9) maxExclusive: exclusive upper bound, (10) minExclusive: exclusive lower bound, (11) totalDigits: maximum number of digits, and (12) fractionDigits: maximum number of digits in the fractional part. Detailed information about each constraining facets can be found in W3C XML Schema [33].

We believe that these constraining facets can be used as guidelines to identify boundary values. For example, consider a WSDL input argument that is an XML data type string with constraining facet of length: <length value = ‘8’>. We can identify a boundary value of a string with the length of 8-character long. For each input parameter, we will then search for its constraining facets. These constraining facets will be part of the corresponding XML schema definition, which can be either included in the corresponding WSDL definitions, or referenced by separate XSD files. The keywords for the twelve constraining facets will be utilized to search for the corresponding specifications. Using the example above, the keyword “length” can be used to search in the corresponding XSD specifications for the constraining facet length and its specified value 8.

The twelve constraining facets can be divided into five categories based upon how they can be used to identify boundary value-based test cases. (1) Four constraining facets explicitly specify the boundary values to test: maxInclusive, minInclusive, maxExclusive, and minExclusive. (2) One constraining facet explicitly defines the set of values to test: enumeration. (3) Five constraining facets define the length of a test case: length, minLength, maxLength, totalDigits, and fractionDigits. (4) WhiteSpace facet guides to generate test cases on spaces. (5) Pattern facet guides to generate test cases based upon regular expressions specified. The first, second, and fourth categories explicitly define the boundary values that can be used. The third category specifies the length of test cases. The fifth category specifies the rules to validate test cases, which deserve

<table>
<thead>
<tr>
<th>Table I. Constraining facets of XML built-in primitive types</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>length</strong></td>
</tr>
<tr>
<td>string</td>
</tr>
<tr>
<td>boolean</td>
</tr>
<tr>
<td>decimal</td>
</tr>
<tr>
<td>float</td>
</tr>
<tr>
<td>double</td>
</tr>
<tr>
<td>duration</td>
</tr>
<tr>
<td>dateTime</td>
</tr>
<tr>
<td>time</td>
</tr>
<tr>
<td>date</td>
</tr>
<tr>
<td>gYearMonth</td>
</tr>
<tr>
<td>gYear</td>
</tr>
<tr>
<td>gMonthDay</td>
</tr>
<tr>
<td>gDay</td>
</tr>
<tr>
<td>gMonth</td>
</tr>
<tr>
<td>hexBinary</td>
</tr>
<tr>
<td>base64Binary</td>
</tr>
<tr>
<td>anyURI</td>
</tr>
<tr>
<td>QName</td>
</tr>
<tr>
<td>NOTATION</td>
</tr>
</tbody>
</table>
further separate investigation and will not be discussed in this paper.

Therefore, for each input parameter defined in a WSDL document, we will obtain a set of possible constraining facets from Table I based upon the XML data type of the parameter. The keywords of this set of possible constraining facets will be used to search from the corresponding schema definitions for defined boundary values or rules. Note that the constraining facets summarized in Table I are possible facets for each data type. If for a defined input parameter, there is no value defined for a possible constraining facet, an implicit constraint value should be used based upon the corresponding IEEE standards [33]. For example, consider an input parameter with type float, if there is no value defined for a possible constraining facet, say maxInclusive, an implicit value is actually defined. XML schema adopts for the type float the IEEE single-precision 32-bit floating point type. Thus the basic value space of a float consists of the values \( m \times 2^e \), where \( m \) is an integer whose absolute value is less than \( 2^{\text{24}} \), and \( e \) is an integer between \(-149 \) and \( 104 \). Therefore, an implicit maxInclusive for a type float is \( 2^{\text{24}} x 2^{104} - 1 = 2^{128} - 1 \). Similar rules should be applied to other numeric XML data types, such as double. Similarly, ISO standards of Gregorian time values should be applied to time-related data types: duration, date, time, date, gYearMonth, gYear, gMonthDay, gDay, and gMonth.

For each XML primitive type, we believe that its comprehensive boundary values can be extracted from three dimensions: (1) XML constraining facets, (2) operational profiles, and (3) semantic meanings. XML constraining facets provide generic guidelines for us to find boundary values; and the operational profiles of a Web service will help us find more efficient boundary values. For example, let us consider a login id field with type string. From the XML constraining facets of string, we know that we need to test on the length of the string. A specific operational profile can help us decide to test whether the string can accept more than 16 characters. In addition, operational profiles can help decide the boundary values for patterns testing, as defined by the corresponding XML constraining facets. Taking the login id field as an example again, the operational profiles may help to generate boundary values to test the string such as: whether the field accepts a string containing only digits, whether the field is case sensitive, whether the first character can be a digit, etc.

Finally, the semantic meanings of an argument can further facilitate boundary values elicitation. Taking an input field of credit card expiration year as an example, it is intuitive for us to test the following cases: whether the input year is a future year or a past year, whether the year is way too far in the future, whether the combination of the year and the month represents a date in the future, whether the month is between 1 to 12, etc.

Although operational profiles and semantic meanings can facilitate more accurate and comprehensive boundary value elicitation, they mainly require manual involvement. On the other hand, constraining facets-based boundary value elicitation can mainly be performed through automatic process, following the methods we discussed in this section. Regarding to Web services testing, automatic test case generation is critical due to the unique time and dynamic feature requirements. In this research, we focus on test case generation based upon constraining facets-based approach. Automating test case generation based upon operational profiles and semantic meanings will be a future research topic.

4.3 Boundary values for XML compound type

As shown in Figure 3, based upon the nineteen built-in primitive types, XML schema defines twenty-five built-in derived data types, such as normalizedString, token, language, etc [33]. In addition, complex data types can be composed of primitive types and derived types. Furthermore, users can define their own data types. In general, each user-derived data type must be defined in terms of another data type in one of three ways [33]: 1) by assigning constraining facets that restrict the value space of the user-derived data type to a subset of that of its base type; 2) by creating a list of data types whose value space consists of finite-length sequences of values of its item types; or 3) by creating a union data type whose value space consists of the union of the value space of its member types. In other words, for each compound data type, it is associated with a hierarchical tree of how it is composed of simpler data types. Each leaf element of the tree is an XSD built-in primitive data type. Therefore, for a compound data type, we can navigate through its hierarchy tree and design test cases based upon each leaf element that is a XSD built-in primitive data type.

Figure 4 shows a simplified XSD compound data type AddressBook. The address book for a person contains three elements: his/her id as a double type, name as a string type, and addresses as a list of address information. Each address is composed of five elements: an address type as a double type, an address line as a string type, a city as a string type, a state as a string type, and a zip code as a string type. Therefore, there are seven leaf elements in this AddressBook data type: id, name, addrType, addrLine, city, state, and zipCode. Each element belongs to a XSD built-in primitive data type, either double or string. Then for each element, we can apply our method of designing boundary values for XSD built-in primitive data types, as we discussed in the previous section. Since we prefer to locate errors if there is any, each test case only focuses on testing one boundary value of one leaf element, without combining several boundary values of multiple
elements. In other words, we have purposely limited the boundary values to single parameter to avoid the explosion that occurs in the number of combinatorial edge values that could be set at each SOAP input message. Accumulating all of these test cases together, we will obtain a set of test cases targeting at testing the overall AddressBook data type.

4.4 The design of test cases for fault tolerance of individual Web services

In this section we will discuss how to perturb boundary values to validate fault tolerance of an individual Web service. Our approach is again based upon XML schema constraining facets. In the last section we discussed how to extract boundary values from the WSDL definition of a Web service to efficiently test its correctness. These elicited boundary values can be perturbed to generate faulty data. Since our boundary values are generated from constraining facets, it is straightforward to generate faulty data in terms of constraining facets also. Table II summaries our methods of creating faulty data based on each constraining facet.

For length, since it defines the exact length of the characters/digits to be used in an argument, two test cases will be generated, one with (length+1) and one with

Table II. Perturbation strategy to generate faulty data

<table>
<thead>
<tr>
<th>Constraining facets</th>
<th>Perturbation strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>+1/-1</td>
</tr>
<tr>
<td>minLength</td>
<td>-1</td>
</tr>
<tr>
<td>maxLength</td>
<td>+1</td>
</tr>
<tr>
<td>pattern</td>
<td>--</td>
</tr>
<tr>
<td>enumeration</td>
<td>Values outside of the set</td>
</tr>
<tr>
<td>whiteSpace</td>
<td>Null/tabs/space/multiple spaces</td>
</tr>
<tr>
<td>maxInclusive</td>
<td>+1</td>
</tr>
<tr>
<td>maxExclusive</td>
<td>The value</td>
</tr>
<tr>
<td>minExclusive</td>
<td>-1</td>
</tr>
<tr>
<td>minInclusive</td>
<td>Use the value</td>
</tr>
<tr>
<td>totalDigits</td>
<td>+1</td>
</tr>
<tr>
<td>fractionDigits</td>
<td>+1</td>
</tr>
</tbody>
</table>
(length-1). For \textit{minLength}, since it defines the minimum length of the digits to be used in an argument, one test case will be generated with smaller length of \((\text{minLength}-1)\). For \textit{maxLength}, since it defines the maximum length of the characters/digits to be used in an argument, one test case will be generated with larger length of \((\text{maxLength}+1)\). For \textit{enumeration}, since it defines explicitly the set of values to be used, we can generate one or more test cases with values out of the set defined. For \textit{whiteSpace}, we can generate test cases with value null, one or multiple white spaces, or tabs. For \textit{maxInclusive}, since it defines the largest value that can be used, one test case can be generated with the value of \((\text{maxInclusive}+1)\).

For \textit{maxExclusive}, since it defines the smallest value that can not be used, one test case can be generated with the value of \((\text{minExclusive})\). The approach to perturb regular expression patterns will not be discussed in this paper.

Table II summarizes our perturbation algorithm over each constraining facet. Recall that using the algorithm discussed in the previous section, a suite of test cases with boundary values will be generated. For each such test case, we will iterate through each input argument, find out from which constraining facet it is generated from, and perturb the data using the algorithm defined in Table II. Each perturbation will create a new test case. A suite of test cases will then be generated by combining all such test cases.

It should be noted that the test cases with faulty data generated from our strategy obviously do not cover all faulty data domain. It is by no means our objective to test a Web service with any possible faulty data though. Our goal is to find efficient faulty data to eliminate a Web service candidate. Our strategy covers faulty data violating constraining facets that definitely should be tested. In addition, as shown in Table II, our approach of generating faulty test cases can be easily automated, which meets the requirements of Web services testing. Faulty data can be further elicited from operational profiles and semantic meanings, which topic will be our future research topic.

5. Experiments

The third set of experiments performed intended to test the effectiveness and efficiency of our boundary value-based test cases and faulty test case generation algorithm.

In \(Y_1\) and \(Y_2\), we embedded various types of errors, (1) computational faults, such as changing the algorithm to translate retrieved double-type grade value into character-type grade value, (2) input SOAP processing faults, such as errors of parsing incoming SOAP messages, (3) output SOAP processing faults, such as errors of generating SOAP response messages, (4) data exception handling faults, such as improper handling over boundary values, (5) incorrect method calls, such as calling wrong methods, (6) database access errors, such as incorrect SQL calls to relational database underneath, and (7) other errors, such as random errors. In order to facilitate the experiments, we carefully implant code to throw meaningful exceptions if an error is found. Table III summarized the number of errors seeded. The total number of seeded errors is 18.

We performed three categories of test case generation: (1) manually and randomly pick up test cases from the input space, (2) manually go through possible test case from input data space, and (3) automatically generate test cases using our boundary value-based approach. The results are shown in Figure 5.

We found that random test case selection is the least robust algorithm to find errors. As the number of test cases increased, the second exhaustive approach can find more and more errors. Meanwhile, it should be noted that both the first and the second algorithms have to go through manual process of test case generation. As shown in Figure 5, we found that our boundary value-based test case generation approach is efficient to find most errors. It found 16 out of 18 seeded errors (88.89%). If the number of test cases increased by randomly picking up more test cases in addition to automatically generated test cases, no more errors were found.

<table>
<thead>
<tr>
<th>Error type</th>
<th>Number of errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computational faults</td>
<td>3</td>
</tr>
<tr>
<td>Input SOAP processing faults</td>
<td>2</td>
</tr>
<tr>
<td>Output SOAP processing faults</td>
<td>2</td>
</tr>
<tr>
<td>Data exception handling faults</td>
<td>3</td>
</tr>
<tr>
<td>Incorrect method calls</td>
<td>3</td>
</tr>
<tr>
<td>Database access errors</td>
<td>3</td>
</tr>
<tr>
<td>Other errors</td>
<td>2</td>
</tr>
</tbody>
</table>

Table III. Distribution of errors seeded into the services
We also found that our algorithm is good at finding errors such as SOAP processing faults, data exception handling errors, as well as incorrect method calls. One database access error and one computational error were not found from our algorithm. If the database retrieval of a grade is incorrect, then the error of translating the grade in double to string was not found. In other words, it is difficult to test those errors depending on database retrieval.

In order to further test the efficiency of three algorithms, we chose to set up the reliability decision threshold to zero, which means that all test cases will be conducted. The testing results are similar to that was described above.

In summary, our preliminary experiments show that our test case generation algorithm is efficient.

6. Conclusions

In this paper we propose a boundary value-based approach to automatically generate test cases for Web services reliability assessment. Based upon their limited exposed interfaces, our approach is appropriate to test the reliability of Web services candidates. By constructing test data including normal data and corrupted data, our approach is capable of certifying whether the tested Web service thoroughly fulfill the functional requirements as desired. By perturbing the test data to imitate unusual events, our approach is capable of testing whether the hosts of Web services act maliciously or errantly at invocation times.

Our future work will include: (1) constructing code generation tools for test case generation, (2) exploring comprehensive selection criteria to ensure reliability of Web services, and (3) conducting more case studies.

REFERENCES

On the Provision of Safety Assurance via Safety Kernels for Modern Weapon Systems

J. Bret Michael1, Anil Nerode2, and Duminda Wijesekera3
1Naval Postgraduate School, Monterey, Calif., bmichael@nps.edu
2George Mason University, Fairfax, Va., dwijesek@gmu.edu
3Cornell University, Ithaca, N.Y., anil@math.cornell.edu

Abstract*

In this paper we discuss some of the challenges and approaches for providing safety assurance for modern weapon systems via software-based safety kernels. We argue that software-centric approaches for designing and verifying safety kernels are flawed. We claim that the design and verification of safety kernels for complex event-driven real-time systems is a matter of physics and dynamical system analysis of system design. We describe an approach for rapidly prototyping safety kernels (and plants and controllers) using an agent-based safety-kernel architecture. The approach utilizes multiagent modeling and hybrid automata.

Keywords: System safety, safety kernel, software, hybrid automata, verification

1. INTRODUCTION

A safety kernel is a component specifically designed to reduce the probability of occurrence of mishaps by performing one or more of the following types of fail-safe functions in a system: detecting, tolerating, and isolating faults. The design of a safety kernel should be based on the following information obtained from hazard analyses of the target system:

- Hazard causal factors, along with frequency and severity of hazards and mishaps
- Complexity of the system protected by the safety kernel
- Number of safety-related functions in the system

Based on the foregoing information, decisions need to be made about the following:

- How much control is to be exercised by the safety kernel over safety-related functions?
- What type of safety-kernels architecture should be employed?
- Which parts of the safety-kernel functions should be allocated to software, hardware, and humans?

The safety kernel needs both sufficient control and functionality to return the system that has entered an unsafe state to a safe (or less risky) state.

So why in the past did so few safety-critical systems have safety kernels? Many factors played a role in stymieing the introduction of safety kernels into safety-critical systems. For example, consider design experience. There was no concept of a safety kernel in the analog version of legacy systems that have since been reengineered to incorporate digital technology. Another factor is engineering judgment. Up until recently there was little experience upon which to judge the prudence of using safety kernels rather than the tried-and-true safety-engineering techniques—the use of a safety kernel was viewed as posing an untenable level of risk not only by the developer and operator of a system, but also by safety certification and accreditation boards.

2. SAFETY KERNELS FOR WEAPON SYSTEMS

The first weapon within the U.S. Navy’s arsenal to use a safety kernel was the fire-by-wire Rolling Airframe Missile (RAM) Guided Missile Weapon System (GMWS). RAM is quick-reaction, fire-and-forget missile designed to destroy anti-ship missiles.

The GMWS is an example of a complex event-driven real-time system. The system performs real-time processing of sensor data in order to detect, track, and target threat objects. GMWS and the RAM itself are both safety-critical because they control the release of energy—energy that can cause death or injury to humans, property damage, or damage to the environment. For example, both inadvertent launches and premature detonation of a RAM in close proximity of the mother ship are hazards which need to be controlled.

Let’s take a look at the Launcher Control/No-Point No-Fire (LC/NPFN) subsystem of the GMWS. The LC/NPFN system monitors the potential movement of the missile launcher into a NPNF zone. The launcher is mounted to the deck of a ship. The NPNF zone consists of the deck, the ships superstructure (e.g., bridge, antennas), and other areas...
provide for fault detection, fault tolerance, and fault isolation. In contrast, multiple redundant safety kernel architectures only provide for fault detection. In other words, any changes to the superstructure or movement of the weapons system to either a different mounting position or another platform would require the reengineering of the electromechanical system. This was one of the reasons the U.S. Navy assumed the risk of introducing the use of software-controlled dynamic NPNF zones.

The software-controlled NPNF function of the LC/NPNF subsystem is an example of a safety kernel. The NPNF safety kernel detects the potential movement of the launcher into a NPNF zone. If the NPNF safety kernel determines that the Launch Control System (LCS) will move the weapon into the NPNF zone, the kernel takes control of the LCS and executes an orderly shutdown of the GMWS. However, the NPNF processor does not provide the fidelity or control necessary to train or elevate the launcher to prosecute an engagement. The NPNF processor performs the following two tasks: (i) stops the launcher movement and (ii) interrupts concurrently the Launcher Control Processor and the firing circuit to the missile (in order to preclude arming and firing of the RAM).

The use of a software-controlled NPNF safety kernel for the LC/NPNF subsystem is a double-edged sword: it increased the complexity of the system (i.e., in terms of state-space) and approximately doubled the size of control software of the GMWS, but on the other hand it reduced both the overall level of risk of putting the system into operation and the need to make costly changes to the GMWS hardware or the ship’s superstructure.

Examples of safety kernel architectures that lie within the middle of the spectrum are safety executives and monitor-actuator patterns. The former initiates fail-safe processing: monitors the state of a system and ensures that the software cannot enter a potentially unsafe state, in addition to coordinating recovery from faults. The latter returns a system to a known less-risky state and resumes processing via monitoring the actuation functions of another process and the state of the actuators. The RAM GMWS LC/NPNF is an example of the application of the monitor-actuator pattern.

4. Need for a New Approaches to Providing Safety Assurance

Safety design requirements are typically easier to implement in the weapons system or weapons-related system than in an external system. The way in which safety design requirements are implemented in U.S. Navy weapons systems is relatively homogeneous: known safety attributes and characteristics of these systems are already relatively well known to the system safety programs.

So why consider the use of safety kernels? Assessing the level of control to be exercised over the weapons or weapons-related system becomes increasing challenging as the level of system integration and complexity increases. Such assessments are especially problematic to perform for system-of-systems. A system-of-systems is an amalgamation of legacy systems and developing systems that provide an enhanced capability greater than that of any of the individual systems within the system-of-systems. Systems-of-systems are a great departure from standalone systems. There is uncertainty and risk associated with assumptions and unknowns regarding the interfaces between the component systems. There is also uncertainty and risk associated with system interoperability issues.

Let’s take a look at a real-world system-of-systems—the Ballistic Missile Defense System (BMDS). Like the RAM GMWS, the battle manager element of the BMDS is a real-time, event-driven complex system. However, it is also asynchronous and distributed. The battle manager must interface to a large number of heterogeneous legacy, organic, and even foreign systems, some of which may not have undergone sufficient safety assessment. In addition, the configuration of the system needs to adapt via plug-and-play components to changes in the environment; that is, the system needs to be readily reconfigurable during operation. Thus, traditional approaches for providing safety assurance of BMDS will not be cost effective and do not lend themselves well to verification.

5. What is the way forward?

System safety relies on predictability. There is a need to know what the system must guard against (i.e., hazards). How does one handle unanticipated hazards? Adaptive systems can have lots of configurations; it is hard to
characterize these configurations because each instance of a component has a different view of the system.

Moreover, the set of things in the environment is neither closed nor stable. However, it might be possible to create a sufficiently large closed world so that one can deal with all of the system hazards. Even so, there will still be a challenge to validate an upper bound on the probability of a system failure leading to a given hazard.

The lack of predictability is at odds with the view of a system for which safety, reliability, and other forms of dependability engineering (i.e., dependability encompasses all of the “ilities”) typically rely.

One of the promising new approaches to providing safety assurance for weapon systems is to think in terms of dependability disciplines.

5.1 CHALLENGES

In order to explore a dependability-disciplines view of providing safety assurance for weapon systems, one must treat a system or system-of-systems in terms of integration properties in at least two respects: (i) to identify the emergent requirements of the weapon system from the collaborations (i.e., determine value-added, rather than what must not happen) and (ii) to certify that the legacy, organic, and foreign systems meet constraints of the well-defined “plug-in slots” (i.e., system interfaces) of the plug-and-play reconfigurable system.

We can construct a wish list for weapon systems, to include, for example, the following desires:

- Coordinated battle management at system-of-systems level vice system level
- Predictable behavior of system-of-systems
- Integrated systems vice interconnected systems, bringing together legacy systems, new system developments, and nondevelopmental (e.g., commercial- and government-off-the-shelf) items
- Minimal effort for modifications to system-of-systems
- System architectures that outlive their components

5.2 ACHIEVING DEPENDABILITY

We advocate a departure from business as usual in the engineering of weapon systems, especially those that a part of systems-of-systems, by requiring the following:

- Spartan and Draconian designs of the system
- Distinguishing up front which system requirements are stable from those that are expected to change
- Institutionalizing the invariant part of the principles of operation of the system
- Taking a positive approach to handling emergent properties of systems, thinking in terms of integration
- Defining emergent requirements and ensure realization

A Spartan safety kernel provides for liveness properties with service guarantees. The Spartan safety kernel only provides services needed to achieve critical requirements in a timely manner.

A Draconian global structure provides for safety with non-interference guarantees. The structure affords for the visible dependencies to be much less than potential dependencies, with fault containment at boundaries and no invisible interactions.

The safety executive resides within the Spartan safety kernel. The safety executive monitors in a cyclic fashion the high-level functionality (i.e., execution of high-level processes) of the system or system-of-systems to ensure that the processes follow the desired sequence of execution. For example, in the case of ballistic missile defense, the safety executive of the safety kernel associated with one of the replicated battle managers would monitor whether the battle manager’s processes for prosecuting a missile engagement execute in the required manner.

5.3 BATTLE MANAGER DEVELOPMENT

In addition to the Spartan safety kernel, we foresee the need for the weapon system to also have one or more battle management kernels—kernels that contain only the basic functions of battle management. Derived from the kill chain, these basic battle-management functions will manage the use of the system’s computing resources to ensure that all time-critical, battle-management events are processed as efficiently as possible. All other weapon-system functionality is to be placed in components that interface with the battle management kernel via the aforementioned well-defined plug-in slots. The Spartan safety kernels monitor the behavior of the battle management kernels, taking action as needed to enforce safety policy, while the battle management kernels are responsible for monitoring the behavior of the weapon system or system-of-systems of weapons, taking action as needed to enforce the rules of engagement and other policy and doctrine related to prosecuting an engagement.
6. DESIGN AND VERIFICATION OF SAFETY KERNELES

Existing software-centric approaches for designing and verifying safety kernels are flawed: these approaches rely on models of faults which occur at discrete times and can be identified. This has little to do with systems that are governed by a continuous sequence of messages which alter the controlled behavior of devices (e.g., networks of weapons and sensors). In essence, today’s popular approaches are based on verifying that a pure software system obeys its software specification and does not send out ineligible signals.

What is needed is a verification approach that provides the dependability engineer with the ability to determine what sequences of control actions would cause a catastrophe. No one control action in the sequence has any meaning, but the whole sequence may send the system spiraling out of control (i.e., positive feedback, not control).

However, we do not believe that the design and verification of safety kernels is a software problem. We claim that the design of safety kernels for complex event-driven real-time systems is a matter of physics and dynamical system analysis of system design.

For verification purposes, one needs to provide simulation or mathematical evidence for complex systems that the real systems of physical devices are similar enough to their finite automaton approximations so that the finite automata controls would control the system in the real world.

Hybrid automata are needed for prototyping and verifying safety kernels. The automata can be used to represent complex real-time systems as distributed systems of interacting physical (e.g., sensors and launch systems) and rule-based (e.g., decision makers and threat evaluators) agents. Physical devices are modeled as Buchi finite state automata on infinite strings; Buchi finite state automata differ from finite automata in that they operate on infinite words and have a different acceptance condition. Agent networks of Buchi automata constitute a special case of Rabin automata (have strong-fairness acceptance) which is computationally feasible and represent many significant aspects of multiagent systems.

Doctrine, policy (including safety policy) and organizational structures are treated as rule-based constraints on system behavior.

Every agent is modeled as Datalog program, which can then be used to run simulations with the aim of determining the effects of inserting, modifying, or deleting physical and rule-based agents. This approach provides a means for rapid prototyping of safety kernels (and plants and controllers) using an agent-based safety-kernel architecture.

We believe that Datalog is a good choice of modeling languages because it is expressive enough to represent the agents, but yet compiles decently to real-time deterministic Buchi automata.

The next step we intend to take is to develop a professional-grade simulation test bed, with the aim of providing a means for verifying the design of safety kernels (and plants and controllers).
Analysis of Milestone Readiness Levels
During the Software Requirements Development Phase

Ganesh J. Pai and Joanne Bechta-Dugan
University of Virginia, Dept. of ECE
{gpai | jbd}@virginia.edu

Khalid Lateef
TITAN Corporation, Greenbelt, MD
Khalid.Lateef@titan.com

Abstract

The independent verification and validation (IV&V) process provides assessment to the project as to whether the developer’s artifacts have met a pre-determined “readiness” level. In practice, this is quantified using engineering judgment. We describe the application of Bayesian Belief networks (BBN) to analyze the readiness of software requirements specifications during software requirements review (SRR) milestone. The method proposed in this paper brings an additional level of rigor to IV&V analysis and the input provided to developers. Starting with a dataflow model of the IV&V process, we construct a BBN semi-automatically. Then, we provide a quantitative interpretation of the readiness level of software requirements in terms of artifact properties. The results of IV&V analysis are used as evidence in the BBN to obtain a posterior distribution of readiness. We illustrate our approach by applying it to two example systems.

1. Introduction and motivation

Independent verification and validation (IV&V) of software adds value to software projects by identifying potential defects in the software product and by uncovering shortcomings which may exist in the software development process [1]. It is typically performed in parallel with software development, and scheduled readiness review milestones are used to communicate IV&V task results to the developers. One of the main tasks in the IV&V process is building a case that (1) an artifact has met some pre-determined level of acceptability; i.e., analyzing the readiness of the artifact, and (2) the subsequent phase of the development process can proceed.

The IV&V analysis level, determined from a criticality analysis and risk assessment process usually dictates the effort invested and the type of activities performed [2] e.g. a comprehensive analysis would require using formal techniques in addition to a broader set of analysis techniques, whereas the so-called lightweight formal methods can be applied at less critical levels to maintain rigor. It is especially valuable to evaluate requirements readiness, since requirements errors are most often responsible for software failures and they are the most expensive to correct when not discovered early in the development lifecycle [3]. One of the main benefits of using formal techniques is that it provides a clearer and more confident assessment of the readiness level. However, in practice, formal techniques are mainly applied to critical elements while the rest of the analysis is comparatively less rigorous. Consequently, the readiness evaluation of an artifact is fuzzy and it may be quantified using engineering judgment.

We use Bayesian networks (BBN) to assess the readiness level of software artifacts, specifically requirements specifications by combining evidence from diverse sources. The overall approach is to first build an annotated process model which describes the IV&V process being used, the entities involved in the process and their respective properties. Thereafter we build the BBN structure semi-automatically from the process model to encode the case for readiness i.e. the rationale with which the analyst can argue whether an artifact is ready at the desired level (Sections 2 and 3). The BBN numerical specification is generated from elicitation,
empirical data and from metrics applied to the artifacts. We characterize readiness in terms of the
generic properties desired from requirements specifications and describe how to quantify it as a
conditional probability distribution (Section 4). To illustrate our methodology, we apply it to
software requirements specifications for two real example systems (1) a fault protection system of
the Cassini deep space probe, and (2) the non-volatile memory management module for a real-
time space system (Section 5).

2. Modeling the IV&V process

The overall IV&V process is usually specified as a set of work instructions. These resemble
guidelines which provide a high-level overview and a flow of the activities that the IV&V team
must perform for a given analysis level. The rigor with which these activities are implemented,
affects the quality, and in turn, the artifact readiness level. Thus an IV&V team which applies
formal techniques to analyze an artifact is less likely to miss potential problems compared to a
team which either does not apply or is unfamiliar with formal analysis. Consequently, their
respective assessments of the readiness level will differ. Our rationale is that modeling the IV&V
process to capture such diverse factors as skill/expertise of the IV&V team or compliance with
process specifications will provide insight into the artifact readiness level and the reasoning used
to arrive at the assessment. To model the IV&V process, we use model the dataflow within a
process, where a process or process activity has input entities, is enacted by agents and produces
output entities. Each of these can have properties which can be quantified or qualified.
Figure 1 shows a simplification of the IV&V process activities and input and output entities at the requirements development phase; some of the properties of interest are also shown. This model itself is a refinement of a higher level process which includes activities such as the identification of relevant inputs, and criticality based prioritization of artifacts for review.

The box labeled “Software requirements analysis” shows some of the activities prescribed in the IV&V literature [2] for requirements analysis. The model captures the notion that software requirements analysis takes as input not only the prioritized software requirements, but also system software documentation, system requirements, and relevant IV&V documentation. Additionally, the model also captures the idea that the team which enacts the process may use tools or approaches tailored for a particular domain. The suitability of these agents influences the
quality of the analysis to some degree since these agents are used to execute the process. We can further decompose the sub-activities shown, to include activity specific tasks: for example, in analyzing whether a requirement described using scenarios is internally complete, some of the tasks would include checking that (for a required functionality): all scenarios and their relevant pre- and post-conditions have been defined, the conditions for any temporal transitions within the scenarios have been defined, all relevant actors have been identified and that the scenarios identified can be stepped through to completion. Again, such tasks may be performed using formal methods if the requirements are critical or using relatively less rigorous techniques otherwise.

For our purpose of building a BBN, this relatively informal process model suffices. Furthermore, it is simple and high-level enough to build so that the IV&V team can quickly and easily build a model of their process. This model was informally validated by an IV&V analyst who had implemented these processes from an original set of work instructions. Once the process models are built, we semi-automatically build a BBN from the process models.

3. Constructing the BBN model

3.1 Bayesian networks

A BBN is a concise representation of a joint probability distribution on a set of statistical variables, encoded as an acyclic graph of nodes and directed edges [4]. The nodes model random variables which can be discrete or continuous. Edges model the probabilistic relations between the nodes. Each node has an associated conditional probability distribution \( p(A|\pi(A)) \) which characterizes the relationship of the node with its immediate parents \( \pi(A) \).

The joint probability distribution for a node is computed by marginalization, whereas the conditional posterior distribution for the nodes given evidence, i.e. observations about the state of a node, is computed using Bayes’ rule. The qualitative part of a BBN is encoded in the structure of the digraph, while the conditional probability distributions for the nodes encode the quantitative portion. One of the strengths of a BBN is that both subjective judgment and empirical data can be used as input. Furthermore, as evidence becomes available, we can update the model and refine its assessments. The fundamental tasks of mathematical modeling using a BBN are: (1) identifying a belief structure that best describes the phenomenon being modeled, and (2) specifying the conditional probability distributions on the nodes.

The BBN structure is built semi-automatically from the process model [5]. Briefly, this method constructs nodes for each of the entities in the process model and for their properties. It appropriately directs arcs between the nodes, such that the observable variable is a leaf node, i.e. from entities to their properties. It also directs arcs from the nodes for the input, and the process entities, to the nodes for the output entities. Since entities relevant to a process activity can exist across levels of abstraction, the algorithm generates subnets which repeat at these different abstraction levels. Although such subnets can be pruned algorithmically, checking whether it produces the correct belief structure is important, i.e. we want to inspect whether the model makes practical sense. Currently, we prune the model manually as this gives us the opportunity to modify the structure of the BBN to account for dependencies between nodes that may not have been explicitly captured or may not be representable in the process model. Thus, the procedure mainly builds a generic structure of dependencies; the BBN used for analysis may be modified by appropriately re-directing transitions or by including or eliminating nodes.

An additional change to the network structure is binary factorization [6], which splits nodes with three or more input arcs and creates logical intermediate nodes that preserve the numerical specification of the original network. Essentially, this operation simplifies both computation and
specification of conditional node probability tables. The latter is beneficial since the tables increase exponentially in size with the number of parents and their respective states.

### 3.2 BBN model for IV&V of SRS

A partial BBN obtained by for the IV&V is shown in figure 2, labeled as the subnet software requirements analysis. Subnet 1 models the influence of process activities on Readiness. In this paper, we refine the completeness analysis activity of the IV&V software requirements analysis process. The BBN for this activity are represented in subnet 1a and subnet 1b.

![Partial BBN structure for IV&V process of requirements](image)

**Figure 2:** Partial BBN structure for IV&V process of requirements
Similar subnets are generated for the remaining activities in the IV&V requirements analysis process. Shaded nodes represent nodes replicated across levels of abstraction as a result of algorithmic construction. We prune the BBN by simply eliminating subnet 1a, since the influences modeled by this subnet has already been captured. The resulting two BBN can be pruned further by re-directing arcs between the appropriate nodes. Alternatively, we can simply use the data obtained from the lower level BBN as evidence for the higher level BBN. The node labeled “SW requirements with IVV recommendations” has arcs to its properties, of which two have been shown in the figure i.e., Completeness and Correctness. Together, these nodes compose the subnet Readiness, and the probability distribution on the root node in this subnet i.e., SW requirements with IVV recommendations represents the evaluation of readiness at the software requirements review milestone.

3.3 Numerical specification

Specifying root and intermediate node probabilities in the BBN encodes its quantitative portion. These model the nature and the weight of the probabilistic relations between related nodes. The relation between nodes need not be only probabilistic. However a deterministic relationship can be easily transformed into the appropriate probabilistic function. For root nodes (nodes without parents), we specify a (prior) probability distribution which reflects either the IV&V team’s initial belief or the available data. Typically, prior distributions are specified such that all available data are considered. In the absence of relevant data, an alternative to a subjective prior is a non-informative prior [7].

We model each node as random variable (say $X$) whose states are mapped to an ordinal scale. i.e. $X$: {Very Low, Low, Medium, High, Very High}. These states can be further mapped to either a monotonically increasing, continuous numeric scale or a discrete numeric scale with integer values. The probability distribution across its states can be specified from (1) historical data or (2) a prior belief (such as a uniform distribution). Given data, i.e. observations ($\theta$) on the set of child nodes ($\Theta$), Bayes’ rule is used to compute the posterior distribution ($p[X | \Theta = \theta]$). For intermediate or child nodes (nodes with parents), we specify a continuous conditional probability density function (pdf); then, the corresponding discrete distribution is easy to build. For our analyses, we use a simple procedure developed by Neil et al. [6] to construct the conditional pdf as a tail-truncated normal distribution $tN(\mu, \sigma)$. This is a normal distribution truncated at both tails and normalized such that the resulting distribution is proper (integrates to 1).

4. Readiness of an artifact

As mentioned earlier, IV&V analysts are typically required to assess the ‘readiness’ or ‘maturity’ of an artifact at a milestone in the process. Admittedly, readiness is an imprecise and fuzzy term; this value is, in practice, largely quantified by using engineering judgment. Provided it meets some pre-determined value, development is allowed to proceed to the subsequent development phase. Readiness assessment is essentially an inference task, i.e. the IV&V team constructs a case for readiness based on the results of their analyses. The IV&V process model and the resulting BBN formalize the procedures and the evidence used to construct this case.

To mathematically characterize readiness, we model the readiness of an artifact as a discrete random variable $M$, qualified on a five point ordinal scale, i.e. $M$: {very low, low, medium, high, very high}. Properties desired from an artifact are modeled as a vector of random variables $X$, each of which can assume some state $x$, also on the same five point scale. Readiness is then, some probabilistic function of the artifact properties and it is assessed as:
\[ p[(M \geq m) \mid (X = \{x\})] \]

(1)

Given some initial prior distribution on \( M \), results of IV&V analysis provides evidence to \( X \) and equation (1) is the posterior distribution of \((M \mid X)\) computed using Bayes’ rule. The result is the probability that the readiness level is some value \( m \). In figure 2, this is modeled as the subnet readiness, where the root node SW requirements with IV&V recommendations represents requirements readiness as function of the desired properties of requirements. Thus, we can interpret readiness (1) with respect to individual properties or (2) as a function of all the properties. The former allows us to make statements of the form “The requirements are ready at a level \( M \) with respect to property \( X_1 \), but not with respect to property \( X_2 \)”.

5. Application to example systems

5.1 Non-volatile memory load component example

For this system, IV&V analysis was performed mainly for the requirements analysis stage and is scenario and inspection based. Since the example provided here is mainly to illustrate our approach, we describe analysis of one of the criteria, i.e. completeness, which composes overall component readiness. The requirements for the non-volatile memory load component were expressed as use-cases and were supplemented with natural language descriptions. In addition to use cases, design stage models for the component were also available in terms unified modeling language (UML) constructs i.e. class diagrams, statechart diagrams, component diagrams, etc. Figure 3 shows the use case diagram of the module and figure 4 shows the natural language specifications corresponding to the module from the requirements specifications document.

![Use case diagram of “non-volatile memory load” operations](image-url)
The EX-3-MODULE-1 FSW, in Initialize Mode and Ground Load State, shall load data from the ground into non-volatile memory upon receiving non-volatile memory load command packets.

A1. The EX-3-MODULE-1 FSW, upon receipt of an off-SBC non-volatile write request, shall append Reed-Solomon parity symbols and write the requested data into the specified non-volatile memory.

A2. The EX-3-MODULE-1 FSW shall load the data given at the address specified into non-volatile memory upon receipt of a non-volatile memory load command.

A2.1. The EX-3-MODULE-1 FSW shall reject the non-volatile memory load command and report the rejection to Actor 1 if the load address, load size, or actual data length are invalid.

Figure 4. Natural language specifications for non-volatile memory load operations

5.2 Example analysis

The IV&V analysis procedure was essentially to build a structured use-case description from the natural language specifications, with minimum changes to the text of the natural language, so as to try and preserve the original intent of the specifications. We believe that organizing the natural language into a structured description permits us to identify methodically, missing conditions, scenarios and statements with ambiguity. Of course, we may also convert such a structured description into formal statements and apply formal analysis to reason about the desired properties of the requirements specifications. Table 1 provides metrics which were computed once the structured use-cases were constructed. Other techniques such as use-case animation or executing the specifications may also be applied.
Table 1. Metrics computed on use-case specification

<table>
<thead>
<tr>
<th>Metric</th>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actors</td>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>Missing actors</td>
<td>$A_{missing}$</td>
<td>0</td>
</tr>
<tr>
<td>$A_{TOTAL} = A + A_{missing}$</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Normal scenarios</td>
<td>NS</td>
<td>1</td>
</tr>
<tr>
<td>Missing normal scenarios</td>
<td>$NS_{missing}$</td>
<td>1</td>
</tr>
<tr>
<td>$NS_{TOTAL} = NS + NS_{missing}$</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Exceptional scenarios</td>
<td>ES</td>
<td>1</td>
</tr>
<tr>
<td>Missing exceptional scenarios</td>
<td>$ES_{missing}$</td>
<td>1</td>
</tr>
<tr>
<td>$ES_{TOTAL} = ES + ES_{missing}$</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Operations</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Missing operations</td>
<td></td>
<td>≥ 2</td>
</tr>
<tr>
<td>$Op_{TOTAL} = Op + Op_{missing}$</td>
<td>≥ 4</td>
<td></td>
</tr>
<tr>
<td>Invariant conditions</td>
<td>I</td>
<td>0</td>
</tr>
<tr>
<td>Missing invariant conditions</td>
<td>$I_{missing}$</td>
<td>2</td>
</tr>
<tr>
<td>$I_{TOTAL} = I + I_{missing}$</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Pre-conditions</td>
<td>PreC</td>
<td>5</td>
</tr>
<tr>
<td>Missing pre-conditions</td>
<td>$PreC_{missing}$</td>
<td>15</td>
</tr>
<tr>
<td>$PreC_{TOTAL} = PreC + PreC_{missing}$</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Post-conditions</td>
<td>PostC</td>
<td>3</td>
</tr>
<tr>
<td>Missing post-conditions</td>
<td>$PostC_{missing}$</td>
<td>≥ 2</td>
</tr>
<tr>
<td>$PostC_{TOTAL} = PostC + PostC_{missing}$</td>
<td>≥ 5</td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Evidence provided for Completeness analysis

<table>
<thead>
<tr>
<th>Node</th>
<th>Node State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exceptional Scenarios</td>
<td>0.4 – 0.5</td>
</tr>
<tr>
<td>Normal Scenarios</td>
<td>0.4 – 0.5</td>
</tr>
<tr>
<td>Actors</td>
<td>0.9 – 1.0</td>
</tr>
<tr>
<td>Operations</td>
<td>0.4 – 0.5</td>
</tr>
<tr>
<td>Post conditions</td>
<td>0.4 – 0.5</td>
</tr>
<tr>
<td>Pre-conditions</td>
<td>0.2 – 0.3</td>
</tr>
<tr>
<td>Invariant conditions</td>
<td>0.0 – 0.1</td>
</tr>
</tbody>
</table>

Scale: (0.0 – 0.2: Very low), (0.2 – 0.4: Low), (0.4 – 0.6: Medium), (0.6 – 0.8: High), (0.8 – 1.0: Very high)

Given the metrics computed on the use-case specifications and the BBN constructed from the IV&V process model, we may use the BBN which models the activities that evaluate completeness of use-cases. Figure 5 and Table 2 show the BBN model and the evidences provided to the BBN, respectively. Essentially, we compute the evidence from the metrics; the nodes in the BBN are assigned an initial prior distribution, assuming that child nodes are normally distributed as functions of the weighted average of parent nodes. Additionally, a variance factor expresses our degree of belief in the priors [6].

114
Figure 5. Example analysis (completeness) OOD-EX-3 system

Figure 6. Example analysis of readiness given completeness
Given our assumptions of the priors and the evidence obtained from the IV&V analysis, the BBN computes the level of completeness for these specifications as approximately Medium, with ~37% probability. Since the specifications and the design approach uses a model based development approach, we provide an initial assumption of High quality for the tools and approaches used during the requirements phase.

Figure 6 shows the BBN model for readiness computed given completeness at level Medium. Essentially, we use the BBN of figure 5, to provide evidence into the BBN at the higher level of abstraction. Given completeness alone and that other nodes are in an unknown state, Readiness of the non-volatile memory load module requirements is between Medium and High.

To illustrate how the BBN can be used to evaluate readiness given all other properties, we make certain assumptions for the other nodes in this BBN. These assumptions are stated in table 3. For example, we assume that the system requirements have a high quality; the consistency of the requirements is medium, whereas the properties of correctness and clarity are in state high. We observe that given these assumptions, the BBN computes the readiness of the requirements to lie at a High level, with approx. 75% probability (Figure 7).

Table 3: Evidence provided for Readiness analysis

<table>
<thead>
<tr>
<th>Node</th>
<th>Prior</th>
<th>Node State (Evidence)</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Requirements</td>
<td>N(0.5,0.1)</td>
<td>0.7 – 0.8 (High)</td>
</tr>
<tr>
<td>IV&amp;V Documentation</td>
<td>N(0.4,0.1)</td>
<td>0.5 – 0.6 (Medium)</td>
</tr>
<tr>
<td>IV&amp;V Team</td>
<td>N(0.5,0.1)</td>
<td>0.7 – 0.8 (High)</td>
</tr>
<tr>
<td>Tools/Approaches</td>
<td>N(0.5,0.1)</td>
<td>0.7 – 0.8 (High)</td>
</tr>
<tr>
<td>Prioritized SW Requirements</td>
<td>N(0.5,0.1)</td>
<td>No evidence</td>
</tr>
<tr>
<td>Consistence</td>
<td>0.5 – 0.6 (Medium)</td>
<td></td>
</tr>
<tr>
<td>Complexity¹</td>
<td>0.7 – 0.8 (Low)</td>
<td></td>
</tr>
<tr>
<td>Correct</td>
<td>0.7 – 0.8 (High)</td>
<td></td>
</tr>
<tr>
<td><strong>Complete</strong></td>
<td><strong>0.4 – 0.6 (Medium)</strong></td>
<td></td>
</tr>
<tr>
<td>Clarity</td>
<td>0.7 – 0.8 (High)</td>
<td></td>
</tr>
</tbody>
</table>

¹ Complexity is measured on the same scale but has an inverse relationship with readiness. Hence higher values correspond with lower complexity.
5.3 Fault protection system example

In this section, we apply our methodology to evaluate readiness for a fault protection system of the Cassini deep space probe. For this system, lightweight formal methods had been applied by Easterbrook et al. [8] at the requirements specification stage, resulting in a finding of 37 issues. To summarize these, there were 11 undocumented assumptions, of which some were significant, 10 cases of inadequate requirements, 9 inconsistency problems, 6 cases of ambiguous terminology and 1 logical error. From the details their IV&V process, we provide prior distributions for the BBN nodes (table 4).

<table>
<thead>
<tr>
<th>Node</th>
<th>Prior</th>
<th>Node State</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Requirements</td>
<td>N(0.6,0.2)</td>
<td>Medium</td>
</tr>
<tr>
<td>IV&amp;V Documentation</td>
<td>N(0.75,0.1)</td>
<td>High</td>
</tr>
<tr>
<td>IV&amp;V Team</td>
<td>N(0.75,0.1)</td>
<td>High</td>
</tr>
<tr>
<td>Tools/Approaches</td>
<td>N(0.75,0.1)</td>
<td>High</td>
</tr>
<tr>
<td>Prioritized SW Req</td>
<td>N(0.5,0.1)</td>
<td>Medium</td>
</tr>
</tbody>
</table>

Our rationale for using these priors is as follows: the IV&V documentation, and the tools used were qualified as having the state High since the application of formal methods provides a stronger assurance of detecting potential errors. The state of the node IV&V Team was qualified as High since the team performing the analysis had expertise in performing both formal analysis as well as IV&V. The prioritized software requirements and system requirements were qualified as having the state Medium as we believe that the developers of the requirements have several years of domain experience in building space probes. Therefore the requirements would be expected to have an appreciable quality a priori.
Since more information about the particular IV&V process used for this system was unavailable, we directly provide evidence to the BBN shown in figure 8. The result of IV&V analysis is interpreted as pessimistic evidence and this evidence is used to assess the readiness of the fault protection system requirements. In this figure, most of the mass of the distribution of M is defined over the interval [0.3-0.5) with more than 50% of the mass in the interval [0.3-0.4) \rightarrow \{\text{Low}\}. The variance of the distribution is also small indicating a greater degree of credibility. We feel that this assessment of the readiness level and the decision to revise the requirements (i.e. not proceed to the development phase) is consistent with the results of IV&V analysis in reference [8].

6. Related work

Neil et al. have conducted research on building object-oriented BBN from process models [9]. Their work models the underlying process of inference and represents the BBN at a higher level of abstraction. Our work differs primarily in modeling an enacted process and building the BBN from the parameters of the input entities, and the process itself. To the best of our knowledge, readiness assessment in a systematic and quantitative fashion in the context of IV&V has not been performed before. Bayesian networks have been used in analyzing software quality [10]; however, quality has been assessed in terms of defect content of artifacts. In our work, BBN are applied to the IV&V process to estimate readiness in terms of artifact properties. The notion of modeling processes is not new [11], however process models have been mainly used to specify and simulate processes.

Our notation is currently simple enough to model dataflow in a process and is sufficient for building a BBN. It is straightforward to extend and formalize the model or map it to existing process modeling formalisms so as to get the benefits of traditional process models. Additionally, we use the process model to analyze how properties of the process and its inputs influence the output of the process.
7. Summary and conclusions

The BBN model constructed from the IV&V process model captures the diverse factors that affect the readiness of an artifact. The assessment of readiness is performed by obtaining evidence from IV&V analysis and using this data in the BBN model. In the light of evidence, the assessment of readiness is updated indicating the likelihood that it is in some particular state.

The BBN also models the intuitive notion that an informal process is less likely to detect errors or issues in an artifact as compared with a process that employs formal methods. This is evident from the variance parameter for the readiness distributions, shown in figures 7 and 8. The latter has a lower variance indicating that we are more confident in this assessment of readiness (as we should be, given that formal methods were used). Thus, the BBN allows using data from both formal and informal IV&V processes in analyzing readiness. In practice, since both of these are employed in the IV&V of a complete system, the BBN provides an elegant framework to aggregate arguments from both sources. Additionally, parameters which influence the IV&V process, and in turn the assessment of readiness of the artifact, (such as IV&V documentation, appropriateness of the tools or methods used, expertise of the analysts) are also modeled. We can measure properties of interest in the artifact and use these easily within the BBN formalism.

The BBN numerical specification requires an identification of prior probability distributions (both conditional and unconditional). These are specified either by elicitation from expert opinion or from empirical/historical data. We assess readiness using the results of IV&V analysis as evidence in the BBN. Evidence can be supplied in the form of metrics applicable to artifacts, problems found from formal or informal analysis, etc. The BBN is capable of modeling the intuitive notion that an informal process is less likely to detect errors or issues in an artifact as compared with a process that employs formal methods. We validate the IV&V process model and in-turn the BBN model via consultation with IV&V practitioner(s).

The BBN encodes a comprehensive argument for artifact readiness level, quantifies this level, and indicates potential problem areas to the development team/customer. Thus, the BBN structure is a mechanism to formalize (1) the IV&V process and (2) the underlying reasoning used to assess readiness. Thus, this approach provides a mathematical basis for the so-called “go/no-go” decision. The BBN is versatile enough to model both probabilistic and deterministic relations. Consequently, it has greater expressive power compared to a functional form or a checklist based approach for assessing readiness levels.

Acknowledgements

We would like to thank the NASA IV&V center, which supported this work under NASA Grant NAG5-11953 and TITAN Corporation for supplying the example system used in this paper.

References


Abstract
This study examines the code quality for four widely used versions of the UNIX operating system (OS) used by industry, government and educational institutions. All four of these operating systems are written primarily in the C programming language and run on a variety of different hardware platforms. One system is a commercial derivative of System V Release 4 UNIX. Cooperating closely with this particular vendor [Vendor A] we tracked the release cycle for this commercial operating system’s kernel from the time the product was first distributed widely to industry through its current release. We examined and analyzed over 6.7 million lines of kernel code during the history of all major releases of this operating system. The vendor closely cooperated with us, providing defect reports on the product that allowed us to better understand the product and to correlate the design and implementation with the known defects at the functional level. In addition, we obtained a competitor’s kernel source [Vendor B] and compared these two vendor kernel source codes to open source OS kernel code to gain some insight into how code quality manifests itself in open source kernels versus commercial source kernels. Linux and FreeBSD were used as examples of open source kernels for this comparison. One result from our work that generalized across operating systems (except Linux) was a common trend in high risk networking code. Another result that surprised us was that the riskiest functions crossed the boundaries of the various subsystems analyzed. In terms of comparison between systems, Linux numbers are dramatically more positive in comparison to the other three OSs.

INDEX TERMS: Software Metrics, Software Quality, Operating Systems.

1.0 Goals and General Approach
The goal of this research is to objectively measure operating system kernel code quality. In this work we analyzed the two commercial versions of UNIX and two open source kernels. We seek to assess the quality of these systems by observing the evolution of the releases of the kernels using objective methodologies. The quality of the final product, in this case the operating system kernel code, will reflect the level of quality of the processes that were used in its construction.

The approach we used examines the source code through the prism of some key static metrics that for procedurally intensive kernel code can serve as a powerful predictor of risk, maintenance and development effort, error proneness, excessive program size, and code optimization levels. For example, this type of source code analysis can detect the potential likelihood or the risk that errors could be latent, the degree of difficulty for maintenance, and/or the degree to which the code is cyclomatically complex. We use well recognized static software source code metrics (ANSI/IEEE standard and derived) for comparing and profiling the four operating systems[1]. These metrics expose the syntactic associations and relationships in the code and the degree to which the code possesses predictive risk from a software engineering perspective.

Finally, for one of the commercial operating system releases [Vendor A], we have been able to validate the results of the predicted risk analysis with field faults. Specifically, the results associate the predicted at-risk code with serious defects that occurred at customer sites due to the code.

Our analysis breaks new ground because, to our knowledge, correlation of field faults with static metric predictors has not been validated at such a large scale as in a commercial operating system nor has a direct detailed quantitative comparison been made between commercial and open source kernels. The analysis also allows us to distinguish between the higher quality code and the lower quality code; this can be of great value in operating system improvement. In addition, the analysis reveals the quality of the associated software engineering practices and discipline used in the software construction process in terms of open source and closed source methods.

1 The names of the vendor products are not identified due to the terms of our research with the respective vendors.
2.0 Metrics Approach

The use of software metrics has been deemed a major factor in the transformation of software development from an art to an engineering discipline. Despite this, static source and instrumented source code analysis has not been widely adopted by software developers and engineers, nor fully appreciated for its discovery properties and predictive profiling abilities. Predictive profiling assists in identifying errors early during the development lifecycle. Unfortunately, the detection of errors late in the development lifecycle process is expensive to correct. Late detection also leads to code that is poorly supported and difficult to maintain[2].

Software metrics measure specific attributes of a software product, including the software development process, in a very precise way. The metrics approach allows us to:

- Distinguish the higher quality code versus lower quality code in each release of the kernel
- Determine subsystems in the kernel that are of higher quality versus lower quality and to observe subsystem quality as it evolves over time (trending)
- Determine the quality of code removed from the kernel between releases, to assess quality of code added to the kernel in a new release and to assess the change in quality between releases based on the code added or deleted
- Correlate the results of the predicted "risk" analysis to actual bugs associated with the same code and therefore to validate the predictive profiling ability of the metrics used in this study
- Obtain important information concerning the associated software engineering practices used to construct the software itself

In the next section we review the metrics that we use in this study. Because many will already be familiar with these metrics, readers with software engineering background may choose to skip this Section 2.2.

2.1 Software Science Metrics Background

In the early 1970s, Maurice Halstead of Purdue University observed that all computer software programs were made up of operators and operands or the key tokens in the code[3]. Halstead defined the four parameters upon which the rest of his theoretical framework was built: the number of unique operators, the number of unique operands, the total number of operators and the total number of operands. Halstead was able to derive a large number of relationships from the four basic parameters from which the core and extended ANSI/IEEE Software Science metrics have been built.

Halstead theorized that a well-written program with n1 unique operators and n2 unique operands should have a length of approximately:

\[ N^\wedge = [n1 \times \log_2(n1)] + [n2 \times \log_2(n2)] \]

From these observations Halstead defined the Purity Ratio as the ratio of \( N^\wedge \) to the actual length N (i.e., \( N^\wedge / N \)). A Purity Ratio of 1 suggests few impurities exist. It also indicates predictively the degree to which more code or less code was used to implement the function being analyzed. More code (P/R < 1) is predictively less well optimized, while less code (P/R ratio > 1) is predictively better well optimized.

Another interesting code-level syntactic relationship that Halstead developed is called Volume or V. If a program has n unique operators and operands or \( n1 + n2 \), then it would take \( \log_2(n) \) “bits” to uniquely represent each. If there are N total usages of those operators and operands, then the number of “bits” to represent the program is defined as \( V = N \times \log_2 n \). Halstead suggested that Volume was a reasonable measure of program size. Correspondingly, potential volume is a metric for denoting the corresponding parameters in an algorithm's shortest possible form.

A derivative of the Volume metric calculates predicted errors. People tend to make mistakes, on the average, every E0 mental comparisons. Therefore, the number of errors (B^\wedge), that would be expected in a program would be calculated as \( B^\wedge = [N \times \log_2(n)] / E0 \). Thus, the number of errors predicted by \( B^\wedge \) is simply an estimate of how many errors existed in the code upon completion of the coding phase. This metric is normally converted to predicted errors per thousands of lines of source code (KSLOC).

Another measure suggested by Halstead is the abstraction level of a program called L. This metric provides the relationship between Program Volume and Potential Volume. Only the most clear algorithm can have a level of unity. Halstead also proposed an “L^\wedge” metric that could be calculated from the basic token parameters. In fact, because \( L^\wedge \) can be derived from a simple analysis of the source program without having to know much about the design or application, it is usually used in most studies in place of L. More specifically \( L^\wedge \) is used when the value of Potential Volume is not known.

Another related metric is called Effort, or E. Effort is based somewhat on Volume, but is “adjusted” to account for the level of abstraction at which the program is written. Effort is simply \( E = V / L^\wedge \). Halstead suggested that the Effort measure reflected the “difficulty” of converting the specifications for the code or its abstraction, into the actual symbolic representation of the code for a particular programming language.

A different approach for assessing program complexity is to consider the program’s control flow. A program’s control flow is based on the number and arrangement of

---

3 Declarations and specification statements were not part of this count as Halstead considered the functional execution code as more important.
decision statements within the code and one of the most popular measures in this regard is widely known as the McCabe Cyclomatic Complexity[4]. Cyclomatic complexity and its associated complexity measures measure the structural complexity of a program.

The cyclomatic number is calculated as follows: \( V(g) = E - N + 2 \) where \( N \) is the number of nodes in the graph, and \( E \) is the number of edges or lines connecting each node. The nodes represent computational statements or expressions, and the edges represent transfer of control between nodes.

Additional metrics including executable code counts with average per function, line of code counts with average number of lines per function, estimated development time and average per function, code comment percentages, compiler directives and nesting depth levels are also used. The maintainability index combines cyclomatic complexity with Halstead measures to produce a practical measure of maintainability. We use all of these measured in this paper. See Fenton, et al. [12] for additional information regarding code metrics.

### 2.2 Advantages and Limitations of Metrics for OS Source Code Analysis

The static metrics we used are commonly accepted by the software engineering community and the IEEE. These metrics come with many caveat emptors. There are widely differing opinions on the worth of Halstead measures, ranging from "convoluted... [and] unreliable" [13] to "among the strongest measures of maintainability" [14]. There is evidence that Halstead measures are useful during development, to assess code quality in computationally-dense applications[15].

Common practice today is to combine measures to suit the specific program environment. Most measures are amenable for use in combination with others (although some overlap). Oman[15] presents a comprehensive list of code metrics that are found in maintainability analysis work, and orders them by degree of influence on the maintainability. These include:

- lines of code per module
- lines of comments per module
- variable span per module
- lines of data declarations per module

We therefore selected these as intrinsically reasonable for the basis of our analysis. Certain desirable characteristics of the metrics can be identified. For the most part, the metrics we chose were:

- Simple to understand and comprehend
- Precisely defined and reproducible
- Objective, cost effective and informative
- Automated as much as possible
- Able to integrate with other measures

- Appropriate for many different application domains
- Applicable to the unit of analysis (a function) as defined in the programming language

Nonetheless, there are clear limitations to static code analysis. This work is not run-time, executable analysis and it will certainly not uncover or discover the majority of run-time conditions and faults. Traditional testing, instrumenting the source code and compiling, or inserting other types of technology in the object code may uncover those types of flaws. Therefore, there is clearly a category of risk that comes from the dynamic side of an operating system and the embedded concurrency in the OS, where our metrics provide limited value.

### 2.3 Complementary Metrics

There are some metrics which could complement this work that we chose not to use. Function point measures provide a measure of functionality, with some significant limitations (at least in the basic function point enumeration method); the variant called engineering function points adds measurement of mathematical functionality that may complement Halstead measures. We chose not to use these metrics in order to simplify our analysis.

Our focus is on techniques and methods that allow errors to be detected early and methods that can prevent errors from arising in the first place. Techniques to promote a better understanding of the system and the location of risks while also assisting in the development, test and maintenance process are paramount. The metrics we use here serve as a powerful predictor of risk, of coupling and cohesiveness, maintenance and development effort, error proneness, excessive program size, and code optimization levels. This source code analysis, despite limitations, can detect the potential likelihood or the risk that errors could be latent, and/or the degree to which the code is cyclomatically complex (decision point/statement analysis) and the degree of difficulty for maintenance.

Hatten indicates[6] that static inspection of software is still not practiced often despite considerable evidence of its effectiveness when supported by analysis tools and software quality engineering techniques. This work bridges this gap.

### 2.4 Organization of the Remainder of the Paper

The remainder of the paper is organized in three sections. The first section presents results. Results are organized into five subsections. In Section 3.0 we discuss the methodology for the measurements. Section 3.1 presents the summary results discussing overall characteristics of the systems with respect to each of the metrics. Section 3.2 discusses predicted error rates of the systems analyzed. Section 3.3 and 3.4 discusses subsystem-level analysis of the kernels and summarizes subsystem complexity and quality across the observed kernels. The section also examines predicted risk profiling at the function level for each of the OSs using the Level of
Effort metric. Section 3.4 also discusses the statistical correlation of metrics with in-field observations for a Vendor A OS therefore validating the measures. Section 4.0 presents related work and Section 5.0 concludes.

3.0 Methodology

In our study we compared several different UNIX kernels including commercial and open source, and where possible for each kernel, trace its evolution over many releases [Vendor A, Linux, and FreeBSD]. Vendor B did not provide us with more than one snapshot. To assist in this process, we abstracted the respective kernels from each UNIX operating system and placed them in logical buckets (or subsystems) as shown in Table 1.

Every kernel source file was assigned to one subsystem and the results of our analysis were applied to each discrete subsystem as well as for the entire kernel as an aggregate. This approach helped us determine characteristic features of those subsystems - which subsystems in general are more complex, which subsystems are being changed the most during the evolution of the kernel and which subsystems were the most stable throughout the product’s lifetime.

The analysis was conducted using SET Labs metrics tools previously used in the NSA study[1]. We set the tools to ignore compiler directives. Consequently, every line of the source code was analyzed even if it was part of conditionally compiled code. Experiments were conducted to assess the impact of this approach on the results. Our analysis determined that the values of the metrics were not significantly changed whether we preprocessed the code or not. Another reason to ignore compiler directives is that we wanted all the code to be analyzed despite configuration or platform differences.

Once the analysis was completed the results files that were generated were analyzed with a set of specially created tools for this work. Perl scripts as well as commercial statistical analysis packages were used in the final analysis.

3.1 Summary Results

Table 2 provides summary level results of the analysis of the four kernels. Expected threshold ranges for the values are included in parentheses where appropriate.

<table>
<thead>
<tr>
<th>Subsystem(s)</th>
<th>Description of Subsystem</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeviceManagement</td>
<td>Generic drivers, logical devices, interactions with hardware etc.</td>
</tr>
<tr>
<td>FSScommon FSSufs FSSnfs FSSother FSSproc</td>
<td>Generic and some specific file system implementations</td>
</tr>
<tr>
<td>IOManagement</td>
<td>I/O operations and management</td>
</tr>
<tr>
<td>IPC</td>
<td>Inter Process Communication implementation</td>
</tr>
<tr>
<td>KernelThreads</td>
<td>Light weight processes and SMP implementation</td>
</tr>
<tr>
<td>Mem</td>
<td>Memory management, virtual memory, swapping etc.</td>
</tr>
<tr>
<td>NetCommon NetIP NetTCP NetUDP</td>
<td>Implementation of networking management and protocols</td>
</tr>
<tr>
<td>ProcessControl</td>
<td>Process creation, scheduling etc.</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Calls implementation</td>
</tr>
<tr>
<td>Security</td>
<td>Authentication implementation, Kerberos support, etc.</td>
</tr>
<tr>
<td>Signals</td>
<td>General Unix signals mechanism implementation</td>
</tr>
<tr>
<td>Other</td>
<td>Modules which we found difficult to assign to one of the existing buckets</td>
</tr>
</tbody>
</table>

Table 1 - Subsystems Used to Categorize the Study
but also the highest average number of SLOC. The three
quality significantly. For example, Vendor B has the
examined.
Also, what code there is to be maintained is very well
Vendor B kernel will have fewer functions to maintain.
The meaning of these metrics is that the
functions coupled with the highest percentage of
predicted error rate of 4.82 per KSLOC. (normalized threshold is 3-8 errors per KSLOC), except
code (KSLOC) and are in the mid to higher "five" range
average predicted errors per thousands of lines of source
been carefully coded and is relatively maintainable.
The four kernels are remarkably similar with respect to
average predicted errors per thousands of lines of source
code artifact,
and lowest average number of executable lines. FreeBSD
interface), and an improved virtual memory system.
hot plugging, support for enhanced power management
missing from Linux is support for a journaling file
simplicity manifests itself through a smaller source base
considering the size of the code base and the
conclusions that Linux is superior when comparing OSs
before making erroneous conclusions that Linux is superior when comparing OSs
without considering the size of the code base and the
overall set of features. Linux does not support as wide a
range of kernel features as found in the commercial
operating systems and in terms of supported features its
simplicity manifests itself through a smaller source base
that is perhaps easier to control and maintain. Notably
missing from Linux is support for a journaling file
system, the ability to name removable devices, support for
hot plugging, support for enhanced power management
called ACPI (Advanced Configuration & Power Interface),
and an improved virtual memory system.
A comparison of number of lines of code (minus blank
and comment lines) per function is shown in Table 2.
Vendor B has the highest average number of lines of code
per function with 41.64 while the Vendor A averages
42.67. However, Linux has an average of just under 26
while FreeBSD averages just over 37. Clearly, both

<table>
<thead>
<tr>
<th>Baseline UNIX OS Release, # of Functions</th>
<th>Purity Ratio (0.85-1.25+), # of Compiler Directives</th>
<th>Predicted Errors, Errors Per KSLOC (3-8)</th>
<th>Level of Effort in Hours, Months, Avg. per Function</th>
<th>Average Cyclomatic Complexity (7 or less), Avg. Nesting Depth</th>
<th>Total LOC, LOC-Blank(SLOC), Avg. per Function (62 or less), Avg. per function minus blank lines &amp; comments</th>
<th>Executable Code(%), Avg. per Function (25 or less)</th>
<th>Comment Lines, Percent of SLOC (min of 25%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vendor A 23,298 functions</td>
<td>1.22</td>
<td>7.993</td>
<td>30,793</td>
<td>6</td>
<td>1,553,750</td>
<td>431,999</td>
<td>362,416</td>
</tr>
<tr>
<td></td>
<td>45,432</td>
<td>5.89</td>
<td>192.46</td>
<td>1.32</td>
<td>1,356,632</td>
<td>18.54</td>
<td>27%</td>
</tr>
<tr>
<td>Linux 2.4 18,994 functions</td>
<td>1.37</td>
<td>3.613</td>
<td>9,416</td>
<td>4</td>
<td>765,389</td>
<td>217,193</td>
<td>169,797</td>
</tr>
<tr>
<td></td>
<td>38,626</td>
<td>5.47</td>
<td>58.85</td>
<td>1</td>
<td>660,689</td>
<td>11.43</td>
<td>26%</td>
</tr>
<tr>
<td>Vendor B 14,473 functions</td>
<td>1.18</td>
<td>5.072</td>
<td>21,559</td>
<td>6</td>
<td>1,180,655</td>
<td>272,903</td>
<td>448,502</td>
</tr>
<tr>
<td></td>
<td>31,113</td>
<td>4.82</td>
<td>134.74</td>
<td>2</td>
<td>1,051,231</td>
<td>18.86</td>
<td>43%</td>
</tr>
<tr>
<td>FreeBSD 4.1 15,023 functions</td>
<td>1.27</td>
<td>4.400</td>
<td>14,414</td>
<td>6</td>
<td>873,674</td>
<td>244,475</td>
<td>211,919</td>
</tr>
<tr>
<td></td>
<td>40,640</td>
<td>5.71</td>
<td>90.09</td>
<td>2</td>
<td>771,409</td>
<td>16.27</td>
<td>27%</td>
</tr>
</tbody>
</table>

Table 2 - Unix OS Kernel C Code Summary. Underlined measures represent best of the four releases.

Table 2 shows some important aspects of these systems. In terms of size, Vendor A has a kernel that has roughly twice the amount of executable code as Linux or FreeBSD. Both commercial platforms [Vendor A and B] have more executable code than their corresponding public domain kernel counterparts. Vendor A has 22% more functions than Linux. Vendor B has the fewest number of functions.

In quality metrics Linux 2.4 has the best values for purity ratio, level of effort, average cyclomatic complexity, and has the lowest average source lines of code per function and lowest average number of executable lines. FreeBSD is close behind. At a gross level, this indicates that the code we analyzed in both open source operating systems and proprietary ones has been fairly well optimized, has been carefully coded and is relatively maintainable.

The four kernels are remarkably similar with respect to average predicted errors per thousands of lines of source code (KSLOC) and are in the mid to higher “five” range (normalized threshold is 3-8 errors per KSLOC), except for Vendor B which has a statistically significant lower predicted error rate of 4.82 per KSLOC.

In addition, Vendor B also has the fewest number of functions coupled with the highest percentage of comments. The meaning of these metrics is that the Vendor B kernel will have fewer functions to maintain. Also, what code there is to be maintained is very well commented in comparison to the other systems we examined.

Our results show that vendor source code can vary in quality significantly. For example, Vendor B has the fewest functions and the highest percentage of comments, but also the highest average number of SLOC. The three
vendor counts are very similar. However, Linux is the most functionally compact code and lowest in complexity overall.

All of the Unix releases have the metrics within threshold bounds except for comment percentages. Vendor B is the true exception and stands out with a much higher average comment percentage than the others, indicating better possible maintainability.

Overall, the commercial vendor metric values are very similar. One may be able to surmise that the closed source development methodology may have some effect on this correlation. We note significant differences between the vendors in predicted error rates, number of functions and comment percentages.

3.11 OS Trend Analysis Using Kiviat Diagrams

Three trend releases over time were selected for the OSs. The following releases were chosen: Vendor A release b, d, f, and h; Linux releases 1.2, 2.0.01, 2.2 and 2.4; and FreeBSD releases 3.4, 3.51, 4.0, and 4.1. The metrics used to present this analysis for each subsystem considered the number of procedures (functions), purity ratio, the number of predicted errors, ratio of errors per thousands of lines of source code (KSLOC), average volume or program size, semicolon or live code counts; average of semicolons per procedure, comment percentages, and the number of asserts.

Figures 1-3 are trend analysis kiviat charts graphically representing the indicated metrics within a sampled family of OS source code releases and comparing the delta changes across these same releases.

The greatest single trend analysis pattern revealed for Vendor A is the significant increase in the number of asserts and in the number of overall functions. In fact, the number of asserts in release h is almost double the number from release f. Predicted error rates were above 6 and have held below 6 since release b. Since release b the amount of code has almost tripled reflecting substantial increases in the networking related subsystems. The number of total functions for the release h is approximately 2.5 times that of release b. Code comment

![Figure 1 – Multiple Trend Analysis for Vendor A](image-url)
percentages have held steady in the 27 percent range. However, estimated level of effort development hours have ranged from 1.24 down to 1.08 (rel d) and back up to 1.32 (rel h) on average per function overall across all subsystems indicating higher optimization levels with the use of fewer operators and operands in later releases.

For Linux the trend is quite clear. Every release has resulted in a number of quality improvements over time. For example, volume, lines of code, complexity, level of effort and predicted errors have dropped. Level of effort averages per function have dropped from 0.80 estimated hours to 0.50 estimated hours per function. Estimated cyclomatic complexity has gone from 6 in release 1.2 to 4 in release 2.4. The average amount of source lines of code per function has dropped from just over 41 to just under 35. Program size or volume numbers have also declined significant by almost a third and this in the face of significant increases in the number of functions and amount of code. Of note is the huge increase in the number of asserts since the releases beginning with 2.x.

Most of this assert increase has been in the common net, process control, net, and netIP subsystem code.

Figure 3 shows the remarkable stability of FreeBSD overall. With the exception of total number of functions (increased by some 40%), the delta shift for the rest of the metrics is quite small when compared to the other OSs. This may reflect a very mature design and highly cohesive and loosely coupled code.

3.2 Predicted Error Rate Summary Trends

Much of the analysis we conducted focuses on the predicted error rates of the various OS releases. Empirical validation for these predicted error rates is discussed in Section 3.42 and we revisit this topic to evaluate predicted error rate trends at the functional level in Section 3.4. This section focuses on summary-level analysis of this metric for an entire OS release across numerous releases.

The IEEE predicted error rate algorithm \( B^* \) was used for this calculation and then graphed to reveal the trend lines.
between releases (See [5]). The number of errors predicted by \( B^c \) is simply an estimate of the number of predicted errors in the code upon completion of the coding phase for that software. This predicted error rate is normalized based on the number and distribution of operators and operands used in the code.

Figures 4-6 presents predicted error rates for the baseline version of the code and all new functions added or removed from the base code between releases. Generally accepted threshold ranges for predicted errors are \( 38 \) errors predicted per thousands of lines of code (KSLOC).

Overall predicted error rates for the Vendor A releases from version C have steadily declined until the last release (see Figure 4). Analysis indicates that several changes and updates took place within this last release. Overall Vendor A releases (other than the last release) have leveled out in the mid to upper 5 range for predicted errors per KSLOC. But even this last release from Vendor A is just below 6 for predicted errors. Also note that Vendor A “new” and “removed” code is more error prone than the base code while the same is not true for Linux. In fact, there is actually more "volatility" in the predicted error rates for the newly added code in Linux (see Figure 5). Although within range (other than for the earliest releases) we observe that for the last couple of releases the volatility is much lower with a significant improvement in the predicted error (well below 6). With the exception of these two releases, the overall predicted error rates for Linux are essentially the same as Vendor A although there are more fluctuations in the quality of new and removed code in Linux in these metrics.

FreeBSD is very stable in predicted error rates and the new and removed code error rates are quite close. The overall error rate for the four FreeBSD releases is well below 6 and the last release has all the new and removed code error rates very near the base rate. Notably, as well, Vendor B has the lowest overall predicted error rate for all of the OSs with 4.83 and a rate that is well under 5. This is nearly a full point or more lower than all three of the other OSs predicted error rates.

In summary, FreeBSD and Vendor A have new and removed code that is higher in predicted errors than the base, but Linux has removed code and added code that often is better than the base but also just as likely to be worse than the base. In other words, Linux has both higher and lower error rates for removed and added code but Vendor A and FreeBSD are producing/removing code with more consistency than Linux. However, the last major 2.2 release and the 2.4 release both have removed and added code above the base rate. This would suggest that the review process for Linux is much broader in scope than the other OSs, and especially for the releases earlier than 2.2 and that earlier often resulted in less error prone code as the OS development base evolved over time.

### 3.3 Selected Critical Subsystem Analysis per OS

For the subsystem analysis between and within OSs, the IO Management, Kernel Threads and Net subsystems were compared. Figures 7 through 11 sample some of the metrics that changed the most over time across the OSs. Note that the same releases were tracked as used in Section 4.21.

Figure 7 reveals that the both the commercial and open source OS code have realized significant size increases over time in the Net subsystem arena. Vendor A has a large increase in the number of functions over time devoted to the Net subsystem particularly between the third trend release and the last.

One interesting observation is that FreeBSD has the greatest number of lines of code for the Net subsystem. Vendor A and Vendor B are very similar in size for earlier releases. Linux has the fewest number of lines of code for this subsystem. This suggests to us that the networking code for Linux had been elevated in priority for development effort in latter releases like 2.4. In fact, we later found out that Version 2.4 of Linux had substantive development effort in this area by adding threading support to the networking code.

Figure 8 shows the number of asserts in Vendor A’s kernel. The trend across the lifecycle of Vendor A’s kernel is that the number of asserts is greatest in IO Management, followed by network code, and followed by Kernel Threads. Vendor A makes extensive use of asserts throughout much of their code base.

Profiling asserts can be quite valuable in understanding where development effort is occurring or where likely instabilities can be found. The selected subsystems profiled show significant growth rates in asserts suggesting the subsystems profiled are being actively developed, debugged and scrutinized for errors by the developers.

Figure 9 focuses on Vendor A. It reveals the code growth over multiple releases for the IO Management, Kernel Threads, and Net subsystems. The fourth trend release indicates a significant increase in the amount of Net code. The figure also shows modest code growth increases in the IO management and the Kernel Threads subsystems.

Figure 10 gives the predicted errors per KSLOC for the networking subsystem. Some of the highest risk for all the OSs is in the networking area. Figure 11 shows that Vendor A, Vendor B and Linux are very similar in predicted errors with FreeBSD having the fewest overall predicted errors.

Overall, the networking subsystems (Netcommon, NetIP, NetUPD, etc.) are generally the most complex, followed by IO Management and Process Control and carry a lot more code, but even this can vary between OSs. Observe the correlation between code size increases in the
Figure 4 - Vendor A Predicted Error Rates (per KSLOC)

Figure 5 - Linux Predicted Error Rates (per KSLOC)

Figure 6 - FreeBSD Predicted Error Rates (per KSLOC)
Figure 7 - Trend of Lines of Code in Networking

<table>
<thead>
<tr>
<th>OS Releases</th>
<th>Vendor A</th>
<th>Linux</th>
<th>FreeBSD</th>
<th>Vendor B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32124</td>
<td>20451</td>
<td>110909</td>
<td>31619</td>
</tr>
<tr>
<td></td>
<td>36417</td>
<td>42793</td>
<td>111578</td>
<td></td>
</tr>
<tr>
<td></td>
<td>48539</td>
<td>111072</td>
<td>145795</td>
<td></td>
</tr>
<tr>
<td></td>
<td>114271</td>
<td>148313</td>
<td>158739</td>
<td></td>
</tr>
</tbody>
</table>

Figure 8 – Number of Asserts in Vendor A Kernel

<table>
<thead>
<tr>
<th>Subsystems</th>
<th>Vendor A</th>
<th>Linux</th>
<th>FreeBSD</th>
<th>Vendor B</th>
</tr>
</thead>
<tbody>
<tr>
<td>IO Management</td>
<td>1924</td>
<td>2472</td>
<td>2332</td>
<td>6124</td>
</tr>
<tr>
<td>Kernel Threads</td>
<td>133</td>
<td>187</td>
<td>209</td>
<td>504</td>
</tr>
<tr>
<td>Net</td>
<td>20</td>
<td>98</td>
<td>345</td>
<td>1303</td>
</tr>
</tbody>
</table>
Figure 9 - Vendor A code Growth Trends

Figure 10 – Predicted errors for Network Code
networking area in Figure 10 and predicted errors in Figure 11.

We also wanted to assess the degree of risk posed by the subsystems based on the Level of Effort metric. The Effort metric predictively reveals the degree of code change in "velocity and acceleration" over time and the relative magnitude of instability and error proneness in the software. It is also a very strong indicator of development implementation "time" by the engineers.

Figure 12 is a "ribbon" chart showing highest risk subsystems across OSs based on average estimated level of effort converted to hours. For Figure 12 the normalized summary risk threshold values for subsystems are as follows: Anything over 7.0 is at highest risk, between 4.5-7 is at risk, from 2-4.5 is borderline risk and below 2.0 is within bounds.

Figure 12 shows the predicted risk subsystem trend for highest average estimated level of effort in hours. Highest risk code is in the NetTCP area for Vendor A, B, and FreeBSD. Linux does not have any subsystem code at risk. For NetTCP Vendor B has the highest predicted risk with 13.6. FreeBSD is 9.71, Vendor A is 6.07, and Linux is 0.88.

Across all subsystems the highest estimated number for Linux is Fsufs with 1.62; Vendor A is NetTCP with 6.07, Vendor B is NetTCP with 13.6, FreeBSD is NetTCP with 9.71. The general observation is that for predicted risk, networking code stands out followed by the Security subsystem for FreeBSD. Everything else is well within scope and all OS have similar patterns. Table 3 summarizes this data.

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Subsystem</th>
<th>Level of Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vendor A</td>
<td>NetTCP</td>
<td>6.07</td>
</tr>
<tr>
<td></td>
<td>Net</td>
<td>2.88</td>
</tr>
<tr>
<td></td>
<td>System V IPC</td>
<td>2.42</td>
</tr>
<tr>
<td></td>
<td>Netcommon</td>
<td>2.34</td>
</tr>
<tr>
<td>Vendor B</td>
<td>NetTCP</td>
<td>13.6</td>
</tr>
<tr>
<td></td>
<td>Net UDP</td>
<td>5.45</td>
</tr>
<tr>
<td></td>
<td>Net</td>
<td>2.62</td>
</tr>
<tr>
<td></td>
<td>IO Management</td>
<td>2.02</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>NetTCP</td>
<td>9.71</td>
</tr>
<tr>
<td></td>
<td>Security</td>
<td>4.55</td>
</tr>
<tr>
<td></td>
<td>NetP</td>
<td>2.05</td>
</tr>
<tr>
<td>Linux</td>
<td>filesystem code UFS</td>
<td>1.62</td>
</tr>
</tbody>
</table>

Table 3 - Summary of Findings in Risk Analysis
The Linux numbers are significantly lower overall in comparison to the other three OSs. Based on our analysis over 150 million lines of code across many domains we have never seen a source code base of this size and functionality with as few “at risk” subsystems. This speaks well for the open source development method that is producing code with low numbers in estimated level of effort development times and reflects higher quality.

For Table 3 the Level of effort number was used to sort the list of functions for each OS across the four trend releases (with the exception of Vendor B) and determine the highest predicted level of risk with respect to error probability.

To summarize, the goal of this part of the analysis is to identify the subsystems at risk and therefore to identify code to operating systems designers for improved design and coding practices in the future. The risk analysis findings show a common trend that the networking code is the code at highest risk across all operating systems analyzed except Linux. Given the complexity of memory management code, process control, scheduling, virtual memory and a number of other subsystems in the kernel, this result is extremely significant. Networking code has been the source of changes for support of common new Internet standards including Ipv6; the growth of network functions and significant changes in this subsystem is evident across the various operating systems analyzed and supported by our results in Figure 7. Thus, our results show that networking code is evolving significantly and may pose higher risk to the stability of the kernel than many other more mature subsystems that often have been thought of as having very tricky or complex kernel code.

![Figure 13 - Predicted Risk Functions](image)

### 3.4 Function Level Risk Analysis (Level of Effort)

In this section we again use level of effort metric to evaluate kernel functions. The analysis provides the highest risk code at the function/procedure level based on the sorted Level of Effort metric. The results are meant to identify the weakest “links” in the kernel that should be considered for code scrappage and/or reimplementation.

#### 3.4. Methodology

Table 4 shows the breakpoints for classifying the code qualitatively based on our extensive past operational experience with this metric on other application-level source code. The values differ in Table 4 from those used in Figure 12 because the raw values are used to establish these thresholds where Figure 12 normalizes the value to level of effort in average hours per function. A paper that uses a similar approach and this metric is used in work of one of the coauthors[11].

<table>
<thead>
<tr>
<th>Risk Factor</th>
<th>Effort Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extremely high predicted risk</td>
<td>&gt;= 10,000,000</td>
</tr>
<tr>
<td>Very high predicted risk</td>
<td>&gt;= 2,000,000</td>
</tr>
<tr>
<td>High predicted risk</td>
<td>&gt;= 750,000</td>
</tr>
<tr>
<td>Borderline risk</td>
<td>&gt;= 200,000</td>
</tr>
</tbody>
</table>

**Table 4 - Risk Categories: Level of Effort Thresholds**

We determined the highest predicted risk functions by sorting the individual functions for all kernel functions for all OSs by the Level of Effort metric. We generated a report for the vendor using thresholds from Table 4 for all kernel functions.
3.42 Predicting Functional Risk in OS Code

Based on Table 4 we wanted to gain an understanding of the distribution of the predicted risk functions. We statistically sorted these functions based on the level of effort metric across the four latest “sampled” releases.

Figure 13 shows the distribution of functions that exceeded the normalized maximum threshold for each range. We began our analysis by observing that Linux does not have any functions that are extremely high in predicted risk and by far has the lowest number of predicted risk functions when compared to the others. Vendor B and Vendor A are quite similar in ratio of predicted risk code to threshold values. Finally, FreeBSD is second behind Linux in the lowest number of predicted risk functions per threshold range. Our limited sample suggests that the open source development process results in a much lower number of predicted risk code overall when compared to commercially developed code.

Table 5 shows the riskiest predicted functions for Vendor A Release f. The raw data shows that there are 16,423 individual functions with 345 ctl functions in the kernel; these functions are identified as ctl functions by having “ctl” as part of the function name. Thus, ctl functions represent 2.1% of the kernel functions. Despite the small percentage of these functions, several of these functions were predicted at highest risk. Four of the 10 predicted high risk functions presented in Table 5 are ctl functions and 2 of the four ctl functions marked with an asterisk had one or more defects.

Of the general population of functions (16,423) only 6.9% had actual defects. Thus, we could expect only 24 ctl functions of the 346 would have defects based on the rate in the general population. Interestingly, our results show for the 41 ctl functions (11.8%) have defects which indicates the tools are predicting functions with higher risk and higher actual defect rates. The io and network subsystem contains many of these functions.

This result is not entirely surprising since ctl functions are well-known for a variety of loopholes to alter kernel variables or parameters. Thus, the analysis shows in addition to identifying certain high risk subsystems, certain classes of high risk functions also exist in Unix kernels.

3.43 Maintainability Metrics

Coleman, et al.[10] points out that Halstead’s volume and effort metrics were the best predictors of maintainability. Experiments showed that the regression model that was most applicable was a four-metric polynomial based on Halstead’s effort metric and on metrics measuring extended cyclomatic complexity, lines of code and number of comments[10]. We used this metric to evaluate the functions in Vendor A’s kernel. Specifically, a comprehensive highest predicted risk evaluation was conducted including the calculation of a Maintainability Index (MI) for the code (the lower the number the less maintainable).

This MI is desirable both as an instantaneous measure of maintainability for the code and as a predictor of maintainability over time. The ability to measure and track maintainability is intended to help reduce or reverse a software systems tendency toward "code entropy" or degraded integrity, and for the purposes of reengineering, to indicate where it might be cheaper and/or less risky to rewrite the code than to change it[16].

The basic MI takes the form of a polynomial in the following form based on an average per code module/function measurement.

\[
171 - 5.2 \ln(\text{aveV}) - 0.23 \cdot \text{aveV}(g') - 16.2 \ln(\text{aveLOC}) - 50 \cdot \sin(\sqrt{2.4 \cdot \text{perCM}})
\]

where:
- \(\text{aveV} = \text{avg Halstead Volume V per module (pgm size)}\)
- \(\text{aveV}(g') = \text{avg extended cyclomatic complexity per module (Myer's Complexity)}\)
- \(\text{aveLOC} = \text{the average count of lines of code (LOC) per module; and, optionally perCM} = \text{average percent of lines of comments per module}\)

Table 6 shows results that closely match results in Table 5.
3.43 Validating Functional Risk in OS Code

To confirm our measurements of the quality of the source code and its correlation with operational defects, we empirically validated predicted risk by examining the function level bug fixes mapped to the predicted error values for Vendor A’s OS release f. Release f is not the most recent version of the kernel, but it is a version that allows us to validate predictive measures with data the vendor has gathered. Thus, working in close cooperation with Vendor A, we developed a tool that examines the whole source tree of the product (one of the UNIX version control systems) and extracts all of the bug fixes that were made to the product in a specific version down to individual functions within a source code file.

The defect extraction information was made based on the record in the history of the version control system.

Vendor A provided us with the complete history of changes the developers made along with the workspace for this earlier version of their OS and we were able to extract the number of distinct “fixes” for individual functions. These fixes are certain classes of faults that have a much higher priority to the vendor and are called escalations. These faults are serious enough that Vendor A invests a large amount of time and money in the correction of these defects and subsequently must produce patches to correct them. The actual escalation events that do occur in the field and this level of sensitive data has never, to our knowledge, ever been provided for an analysis of this kind that correlates static predictive measures at the source code with actual field faults in operating systems.

Table 7 data compares the predicted highest risk functions with the actual escalations from Vendor Release f. Release f for Vendor A contained 16,423 individual functions. Of these functions, 1,148 (or 6.9% of the total number of functions) had at least one escalation assigned to it. A number of functions had multiple escalations for the same function (multiple defects caused by the same function) for a total of 2,625 individual escalations.

The table shows a summary of the percentage of functions predicted to be at highest risk and how well the prediction matched the escalation data.

Note that the predicted risk functions for the highest risk factor category has a 100% hit rate, 33% for the next highest category, 29% for the third category and 22% for the fourth category. There are 728 escalations that map to the predicted risk functions representing 28% of the total number of escalations. The remaining 1897 escalations fall in the 14,223 functions not found at risk using our metrics. One way to interpret the result is that very weak links in the code base predicted statically should be fixed before deploying the code otherwise the increased cost of an escalation will be paid. The tools identified riskiest functions accurately.

<table>
<thead>
<tr>
<th>Predicted Risk Factor</th>
<th>Level of Effort Metric Threshold</th>
<th># of Predicted Risk Functions</th>
<th># of Vendor A Escalations (one to one)</th>
<th>Successful Predictions (hit rate)</th>
<th># of Vendor A Escalations (many to one)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Extremely high</td>
<td>&gt;= 10M</td>
<td>3</td>
<td>3</td>
<td>100%</td>
<td>7</td>
</tr>
<tr>
<td>2) Very high</td>
<td>&gt;=2M</td>
<td>66</td>
<td>22</td>
<td>33%</td>
<td>73</td>
</tr>
<tr>
<td>3) High</td>
<td>&gt;= 750K</td>
<td>202</td>
<td>58</td>
<td>29%</td>
<td>178</td>
</tr>
<tr>
<td>4) Borderline</td>
<td>&gt;= 200K</td>
<td>929</td>
<td>202</td>
<td>22%</td>
<td>570</td>
</tr>
<tr>
<td>Total</td>
<td>1200</td>
<td>285</td>
<td>24%</td>
<td></td>
<td>728</td>
</tr>
</tbody>
</table>

Table 7 - Vendor A Release F Predicted Risk/Escalation Comparison (sorted by Level of Effort)
3.43 Analysis of Escalation Data

The escalation data analysis itself shows some interesting results. Bugs in software and the associated subsequent changes and fixes to the same code are not usually distributed equally and tend to cluster. The above graph demonstrates this quite well in showing the clustering effect of the escalations around certain sets of OS functionality.

Figure 14 displays the Vendor A Release F escalation clustering where “bubble” size is determined by the Level of Effort metric. Specifically, the functions were sorted by the directory pathname in alphabetical order and then mapped against Vendor A escalations. The clustering of the escalations is given by the functions on the x-axis based on the alphabetical order of the functions and their containing directories in which the functions appear in the kernel.

Interestingly, not all escalations are distributed evenly in Vendor A’s Release F OS source code base. Results show the clear tendency for certain escalations to cluster around certain sets of functions in certain subdirectories. Each expanded oval represents the number of escalations associated with a given set of functions sorted by directory pathname. The higher the “floating” bubbles are in escalations and size, the greater the risk the sets of functions pose to the OS baseline. Our analysis of the alphabetical ordering with the subsystems delineation used in this paper indicates that clustering is primarily part of the io and tcp subsystems.

Overall results show a significant correlation between the actual bugs and the level of effort rates and reveals that the predicted level of effort metric is well correlated to the actual escalation events that occurred in terms of clustering. The data also shows the age-old software dilemma based on Pareto’s Law where 20% of the code contains 80% of the errors and 2.5-5% of the code predictively contains 95% of the most critical errors.

In overall significance, Table 7 and Figure 14 shows that static software metrics are a powerful predictor for the developer. The metrics can help identify error prone code before the software is tested and even compiled. The validation of this result has profound impact in the evaluation of large, frequently changing source code bases where manual inspection is not a reasonable option, peer review of code may not be feasible for the entire code base, and automated tools must be used.

4.0 Related Work

There have been few studies conducted that analyze the quality of operating system source code at the kernel level using classic static software engineering metrics. In Henry, et.al.[7] the authors present the results from a source code analysis of the UNIX operating system. The paper focuses on the use of the metrics and shows that there is a high correlation between "Volume", "Effort" and "Cyclomatic Complexity" in the analyzed code.

Significant results were also presented in Schneidewind and Hoffman [8]. The authors proved that the propensity to make programming errors and the rates of error detection are dependent on program complexity. Knowledge of these factors can be used to avoid error-prone structures in software design and to devise a testing strategy based on the anticipated difficulty of error detection and correction.

Significant relationships were also found between complexity measures (McCabe's Cyclomatic Complexity) and error characteristics. Similar results were reported in Ward’s work[9]. HP's Waltham Division showed success with software defect prevention using McCabe's Cyclomatic Complexity metric. The study specifically showed that McCabe’s metric provided automatic identification of potentially faulty software before actual
testing was started, the metric identified code modules that could benefit from code inspections, the metric provided for a means to have well-defined coding standards accepted throughout the lab, and it established effective code defect prevention strategies based on the restructuring of overly complex code.

Coleman, et al.[10] demonstrate how automated software maintainability analysis can be useful. Their work focuses on quantifying maintainability by calculating a “degree of fit” from a table of acceptable metrics ranges. When the metric value falls outside of the optimum range, it indicates that maintainability is lower; hence, there is a deviation (or penalty) on the components contribution to maintainability. The optimum range value, called the trigger point range, reflects the “goodness” of the program style. The authors evaluated approximately 50 regression models in an attempt to identify simple models that could be calculated from existing tools and still are generic enough to apply to a wide range of software systems. These metrics were used at the component level, subsystem level and whole system level to evaluate and compare software much in the way we have done in this paper. The authors state that in spite of research that has moved away from using Halstead metrics, all tests clearly indicated that Halstead’s volume and effort metrics were the best predictors of maintainability for the test data evaluated. Experiments showed that the regression model that was most applicable was a four-metric polynomial based on Halstead’s effort metric and on metrics measuring extended cyclomatic complexity, lines of code and number of comments.

Finally one of the coauthors presents an extensive overview of how to measure software quality[11]. To ensure cost-effective delivery of high-quality software, the National Security Agency analyzed code bases of more than 25 million lines of code. This case study illustrated the benefits of code-level measurement activities.

We are not aware of published studies that have looked at the quality of large amounts of operating system software code using the software engineering metrics we selected. It is of extreme significance to the operating system community to have a firm understanding of where kernel code tends to be complex and significantly more difficult to maintain throughout the product release cycle. This in turn can generalize to allow research to focus on simplifying design in these areas and for improving coding practices on the more complex portions of the kernel.

In order to gain an understanding of how well a software system such as the operating system kernel is actually coded in terms of code quality, our research group used standardized tools that have been used to assess code quality. Our study assesses kernel releases in a systematic manner tracking the improvements in discrete components and correlates quality improvements between different versions and different components. Over 10 million lines of code were analyzed and results are presented here of one vendor’s operating system kernel over several releases, another vendor’s source code (one snapshot) and two public domain versions of operating system kernels, namely Linux and FreeBSD through their evolution.

5.0 Conclusion

The paper presents the first analysis we are aware of that shows a strong correlation between the classic static IEEE metrics and actual escalations events evaluated at the functional level and summary level. We have been able to obtain traditionally sensitive data for this paper that has not been made available to the public and this data has provided significant results that are sweeping in scope due to the validation with “in field” defect data. The paper showed that the level of effort and associate predicted error and complexity metrics identified the riskiest code functions remarkably well. Consequently, this can allow developers to provide for important resource allocation for testing and modification of the higher predicted error source code before it is deployed in the field.

From an operating systems standpoint, the results are significant because the correlation pertains to operating system source code, was analyzed on a commercial vendor’s source base, and appears to have exceptionally strong practical significance for large, frequently changing codes where manual inspection is not a reasonable option.

The metrics used can assist in the development of better quality operating systems code because it is much cheaper to fix and update risky code early in the development cycle instead of waiting until the code is deployed later. In fact, a well-known conservative industry average is $10,000 per bug to fix code defects once the code is deployed in the field. The static metrics can provide a direct bottom-line impact on the containment of escalating costs associated with operating systems maintenance before defects escape downstream in the development cycle and become expensive to fix.

An exceptionally surprising result from our work that generalized across operating systems (except Linux) was a common trend in high risk networking code. Given the complexity of memory management code, process control, scheduling, virtual memory and a number of other subsystems in the kernel, this result is extremely significant. Our results show that networking code is evolving significantly as new standards emerge and may pose higher risk to the stability of the kernel than many other more mature subsystems that often have been thought of as having very tricky or complex kernel code.

Another result that surprised us was that the riskiest functions crossed the boundaries of the various subsystems analyzed. Various etl functions floated to the top of functions highest at risk when all subsystems were aggregated and analyzed. The risky values associated with the etl functions seems obvious once one considers
these functions purpose, their overall functionality, and the potential risk they pose to operational stability.

In terms of comparison between systems, Linux numbers are dramatically more positive overall in comparison to the other three OSs. Although this kernel is small, it is of exceptionally high quality. Based on our analysis over 150 million lines of code across many domains we have never seen a source code base of this size and functionality with as few “at risk” functions per code size.

The quality of commercial versus open source systems is difficult to assess based on this “limited sample” because the scope of the open source system was more limited in functionality than the commercial ones. However, a very good indicator of the methodology differences appeared in our analysis of new and removed code during the kernel trend profiling we performed.

In summary, FreeBSD and Vendor A have new and removed code that is higher in predicted errors than the base, but Linux has removed code and added code that is both better and worse than the base. Thus, Linux has both higher and lower error rates for removed and added code but Vendor A and FreeBSD are producing and removing only inferior code with more consistency than Linux. Our conclusion here is that the review process for Linux may be much broader in scope than the other OSs. Code is removed from Linux to improve it regardless of the fear that some good code is being discarded and consequently this often resulted in less error prone code as the kernel base evolved over time.

Overall, the results of this study have provided new information to the OS and software engineering community on many dimensions. We hope the results will be useful in improving the quality of operating systems source code, in containing maintenance costs, and in establishing the use of predictive analysis with profiling metrics to predict field faults and vendor escalations before they actually occur.

6.0 References


[5] redacted A web site specification (for more info)


INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center  
   Ft. Belvoir, Virginia

2. Dudley Knox Library  
   Naval Postgraduate School  
   Monterey, California

3. Paul E. Black  
   National Institute of Standards and Technology

4. Dale S. Caffall  
   Missile Defense Agency

5. Janet Drake  
   University of Northern Iowa

6. Thomas A. Drake  
   Independent Consultant

7. Joanne Bechta-Dugan  
   University of Virginia

8. Brett D. Fleisch  
   University of California, Riverside

9. Swapna S. Gokhale  
   University of Connecticut

10. Ilya Grishashvili  
    University of California, Riverside

11. Khalid Leetef  
    TITAN Corporation

12. Zongwei Luo  
    University of Hong Kong

13. James B. Michael  
    Naval Postgraduate School

14. Robert E. Mullen  
    Cisco Systems

15. Anil Nerode  
    Cornell University
16. Ganesh J. Pai  
University of Virginia

17. Norman F. Schneidewind  
Naval Postgraduate School

18. Vanitha Sugumaran  
John Deere Power Systems

19. Jeffrey M. Voas  
SAIC, Inc.

20. Duminda Wijesekera  
George Mason University

21. Jia Zhang  
Northern Illinois University