Defense Acquisition in Transition

6TH ANNUAL ACQUISITION RESEARCH SYMPOSIUM

How to Check If It Is Safe Not to Retest a Component

Dr. Valdis Berzins, berzins@nps.edu, 831-656-2610

Paul Dailey, prdailey@nps.edu

Software Engineering, Naval Postgraduate School

Context

- Expected long term benefits from Navy Open Architecture
  - Business benefits:
    - Flexible acquisition strategies and contracts that enable ***software reuse, easy systems upgrade***, and ***shared data*** throughout the Navy
  - Technical benefits:
    - Modular open architectures facilitate ***system adaptation, portability***, interoperability, ***upgrade-ability*** and ***long-term supportability***
- The Achilles Heel - Test and Evaluation
  - Current practices require ***retesting unchanged components*** after each system upgrade, typically every two years
  - Substantial budget and schedule are currently devoted to retesting
  - ***New technology, processes, and policies*** are needed to ***safely reduce*** this effort and free resources for testing new functionality
- Improvements sought by our research
  - Less time for testing, quicker response to changes
  - Improved reliability on larger scales without increasing testing cost

Scientific Roadmap - Objectives

- Safely reduce testing cost
  - Reduce the need for re-testing
  - Eventually eliminate integration test after every reconfiguration
  - Reduce cost of future system failures due to missed errors

- Make testing more effective by augmenting it with other quality assurance methods
  - Develop conceptually new and different methods to achieve dependability in Navy OA systems in presence of reuse, reconfiguration, changes and unpredictable environments

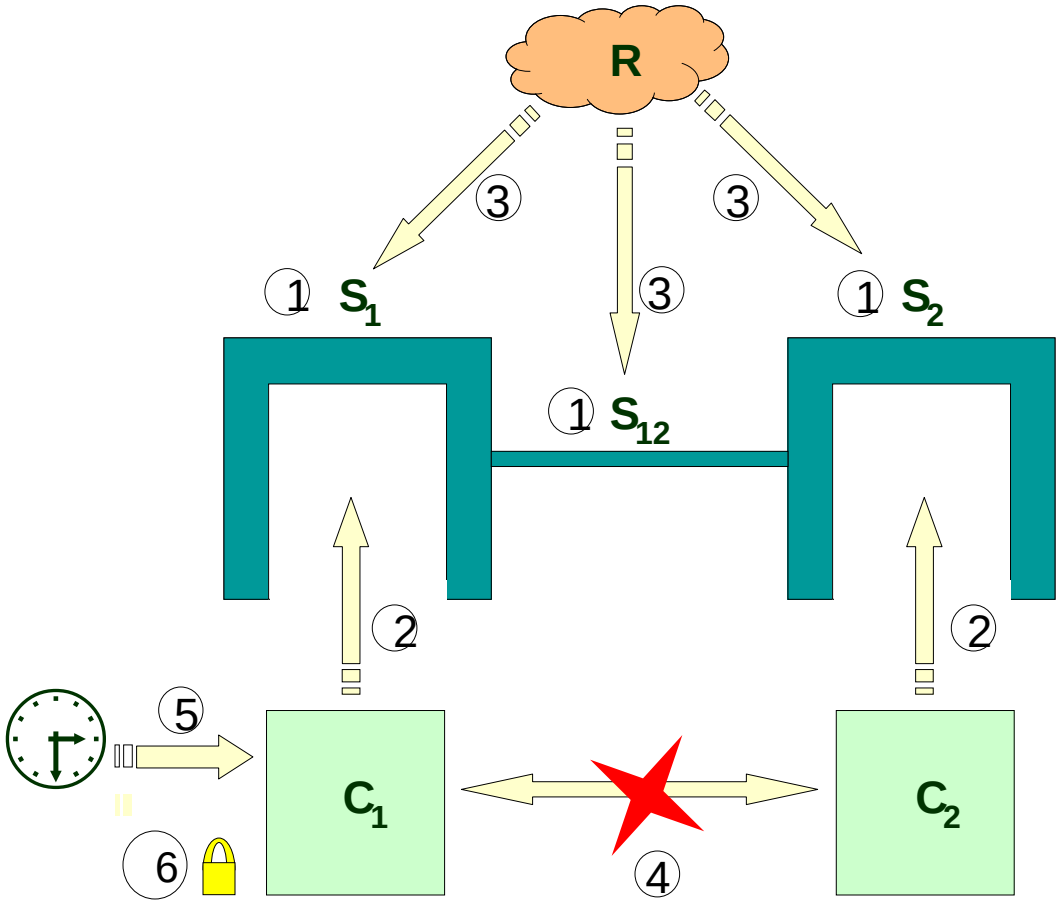
- Enable Persistent Open Architectures
  - The architecture should not have to change or be retested every time the system configuration changes
    - Methods that cover many configurations with one analysis
    - Avoid redundant retesting of previously existing modules and architectures

Scientific Roadmap - Approach

- Refine the open architecture concept to support system development and testing with interchangeable software parts that conform to persistent system standards
  - Requirements that are stable across all configurations
  - Both system-wide capabilities and subsystem/connection properties
- A Dependable Open Architecture should include:
  - Not only components and connections but also constraints expressing the most important dependability properties
  - Links to requirements, capabilities and standards
  - Variable parameters – KPP's / features / Load characteristics
  - Components and connectors should be swappable within compatibility groups defined by testable dependability properties
- Apply testing and systematic quality assurance at the architectural level as well as the system implementation level

Long Term Solution Approach

- The proposed QA method is globally decomposed into five major steps:



1. Formulate dependability contracts
2. Test Components vs. Standards
3. Verify Architecture vs. Requirements & Standards
4. Ensure noninterference among components
5. Monitor environment assumptions
6. Monitor changes to executables

R	Requirements
S_1	Standard for Component 1
S_2	Standard for Component 2
S_{12}	Standard for connection between components 1 and 2
C_1	Component 1
C_2	Component 2

See 2007 Acquisition Symposium Paper for details

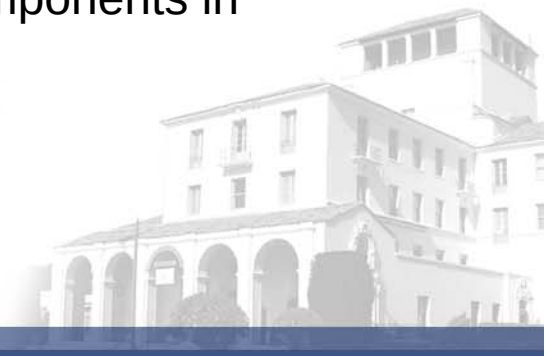
Short Term Problems

- Current Navy combat system test procedures require an integration test for every:
  - System configuration (platform)
  - Changed system configuration (upgrade)
- Open Architectures support frequent changes to configurations
  - Retesting is expensive and time consuming
- Open Architectures support component reuse across platforms
  - Component workloads subject to change
  - New workloads expose new faults

Recent Work - Approaches

- Reduce testing cost
  - Methods to ***identify components that do not need to be retested***
  - Methods to ***limit scope of retesting*** when it is needed
  - Methods to ***completely automate*** testing and analysis

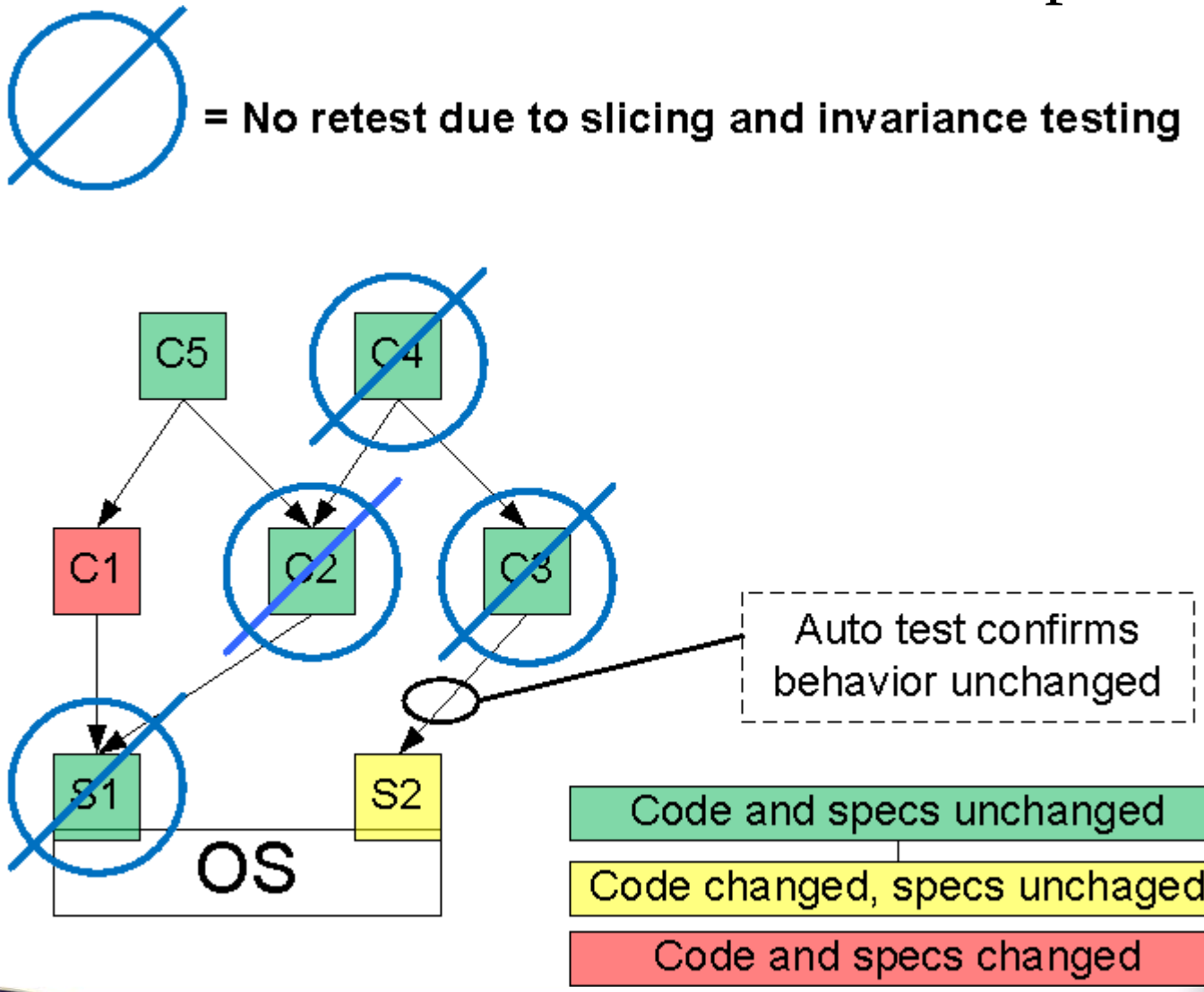
- Maintain safety
  - Program slicing to confirm unchanged behavior of unchanged code
  - Automated testing to confirm unchanged behavior of modified code
  - Operational profiles to efficiently test reusable components in different environments.



When Retesting a Service is Necessary

- When its slice or behavior has changed
- When requirements have changed
  - New functionality needs to be tested
  - Test all affected components
  - When the ***range of expected operating conditions*** has expanded
  - Even if there was no other change, new test scenarios are needed
  - Indicated by a modified operational profile
- When computing speeds or timing constraints have changed
  - Changed hardware processing rates can adversely affect scheduling algorithms and cause missed deadlines

Test Avoidance Example



Program Slicing

- Program slicing is a kind of automated dependency analysis
  - Same slice implies same behavior
  - Can be computed for large programs
  - Depends on the source code, language specific
- Slicing tools must handle arrays and objects correctly
  - Need to certify the tools to be used
- Unchanged component behavior depends on continued correspondence of machine code to source code
- Must certify absence of memory corrupting bugs
  - Tools exist: Valgrind, Insure++, Coverity,…
- Must ensure absence of runtime modifications due to cyber attacks
  - Cannot be detected by testing because modifications are not present in test loads
  - Need runtime checking, can be done using cryptographic signatures

How Much Invariance Testing is Enough?

- How many tests are needed to reach *high confidence*?
  - Stakeholder defines the acceptable risk threshold k
    - *The expected frequency of behavioral differences in a given service is at most one in k missions*.
- Number of test cases is computed for each service in the middleware interface to the operating system
  - It is determined by the following formula

$$T_s = (k\, e_s) \log_2 (k\, e_s)$$

    - Where s is a service, e_s is the mean number of executions of s per mission, k reflects stakeholder's tolerance for risk as above
- Test cases are independently drawn from the probability distribution characterizing the mission, a.k.a. *operational profile*
  - Statistical confidence level is $1 - 1/(k\, e_s)$
    - Probability of making a false positive conclusion matches the stakeholder's risk tolerance

Current Policy for Mishap Risk Assessment

FREQUENCY OF OCCURRENCE	MISHAP SEVERITY CATEGORIES			
	1 CATASTROPHIC	2 CRITICAL	3 MARGINAL	4 NEGLIGIBLE
A – FREQUENT P ≥ 10%	1A	2A	3A	4A
B – PROBABLE 10% > P ≥ 1%	1B	2B	3B	4B
C – OCCASIONAL 1% > P ≥ 0.1%	1C	2C	3C	4C
D – REMOTE .1% > P ≥ 0.0001%	1D	2D	3D	4D
E – IMPROBABLE 0.0001% > P	1E	2E	3E	4E

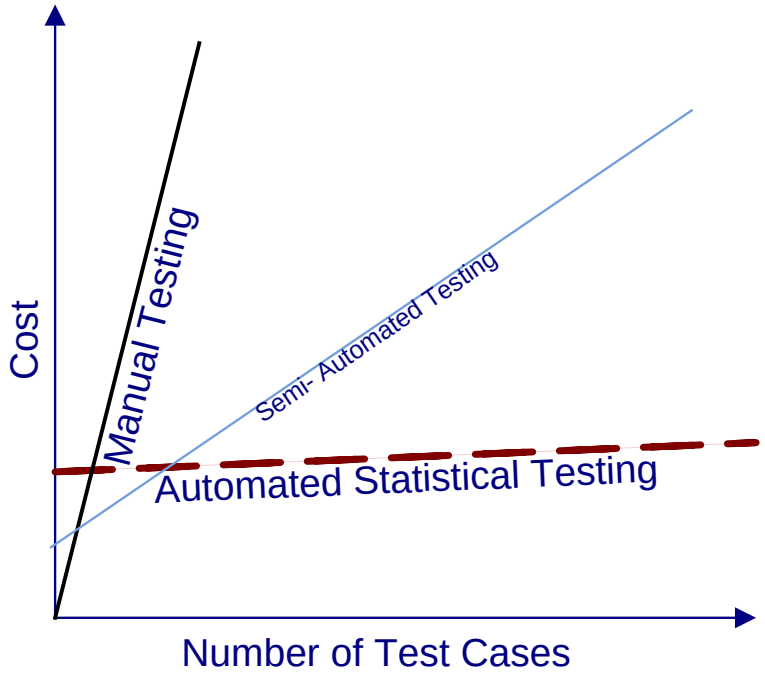
Cells:	Risk Level & Acceptance Authority:
1A, 1B, 1C, 2A, 2B:	HIGH – ASN (RDA)
1D, 2C, 3A, 3B:	SERIOUS - PEO-IWS
1E, 2D, 2E, 3C, 3D, 3E, 4A, 4B:	MEDIUM –PEO-IWS 3
4C, 4D, 4E:	LOW – PEO-IWS 3

P: Probability of occurrence in the lifetime of an individual system, ranges taken from MIL_STD-882D

Testing Efforts vs. Acceptable Risk

$N_s = k\ e_s$	C	T_s
10^3	.999	1.0×10^4
10^4	.9999	1.3×10^5
10^5	.99999	1.7×10^6
10^6	.999999	2.0×10^7
10^7	.9999999	2.3×10^8
10^8	.99999999	2.7×10^9
10^9	.999999999	3.0×10^{10}

Number of test cases required for different levels of risk tolerance



Testing cost characteristics

See paper in 2008 acquisition conference for details

Why Do We Need Operational Profiles

- Can be used to ***automate selection of test cases***
- Reliability of a system is determined by the operational profile
  - Real systems have bugs, specification errors, requirement omissions, etc.
  - System reliability varies from **0** (always fails) to **1** (never fails) in different environments
- Operational profiles have proved useful in practice
  - Example: reliability testing of telephone-switching software
- It takes human effort to produce an operational profile
  - Measure the frequency distributions of executions and associated input parameters for each service
    - Can be collected on- or off- line

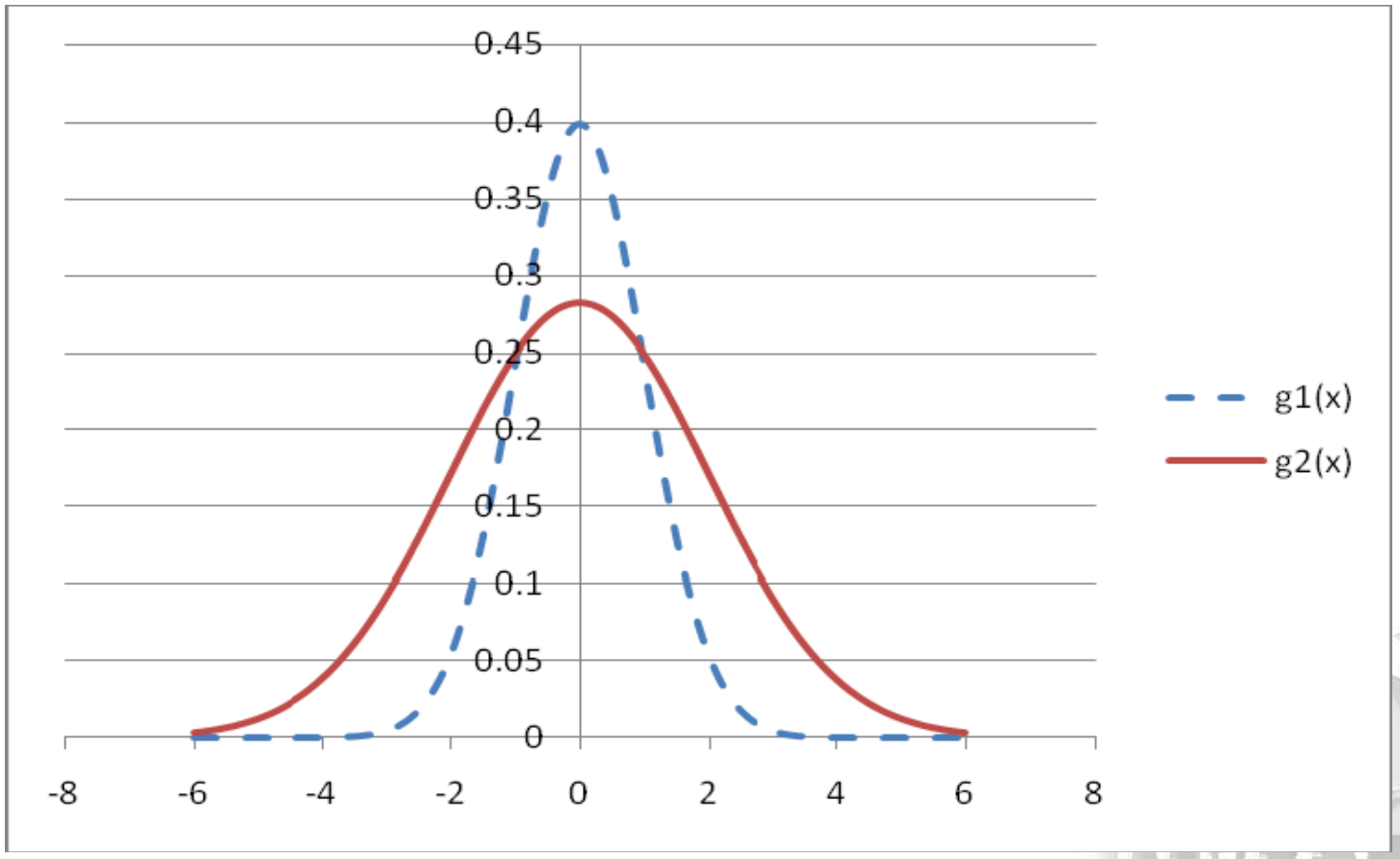
Benefits of Operational Profiles

- Reduces testing resources
  - Automatic generation of test cases
  - Efficient selection of test cases
  - Finds most frequent failures first
  - Supports reuse of previous test results
- Good software reliability checking
  - Statistically represents external environment
  - Suited for software reuse testing
- Ideal for Open Architecture applications by enabling automated statistical testing

Example of Using an Operational Profile for Reuse Testing

- Currently fielded software has been tested with N samples from operational profile g1(x) and functions reliably in that environment
- Software is being reused and placed in new environment represented by operational profile g2(x)
- What is the minimum amount of testing required to ensure operability and reliability in the new environment?

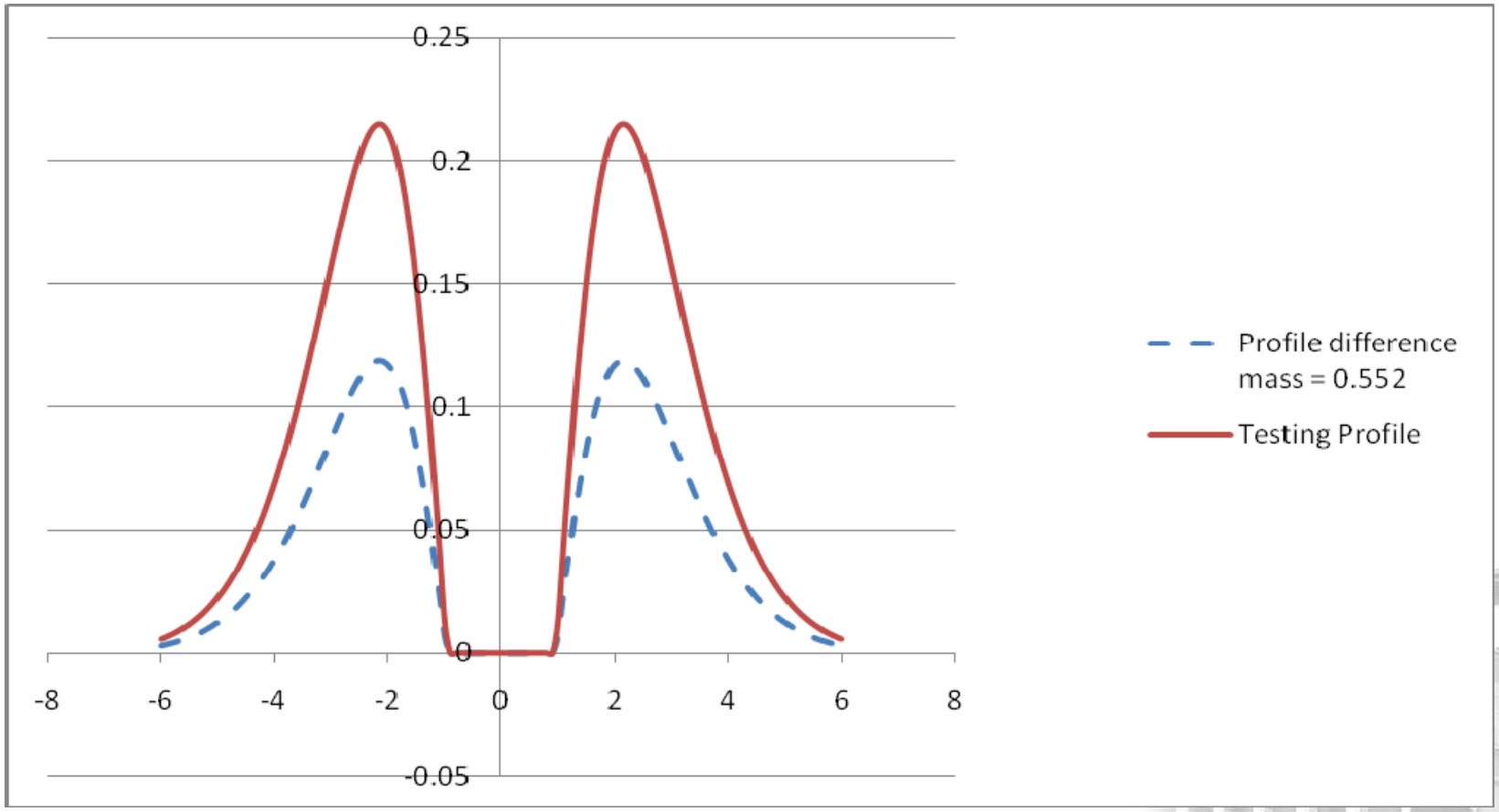
Operational Profile for Two Different Environments



Example of Using an Operational Profile for Reuse Testing (cont)

- Need additional testing in regions more likely in the new profile than in the old one
- The profile difference defines the needed test cases
  - Pd(x) = if g2(x) > g1(x) then g2(x) – g1(x) else 0
  - Must be scaled if reliability goals differ in the two environments
  - Must be normalized to become a probability distribution

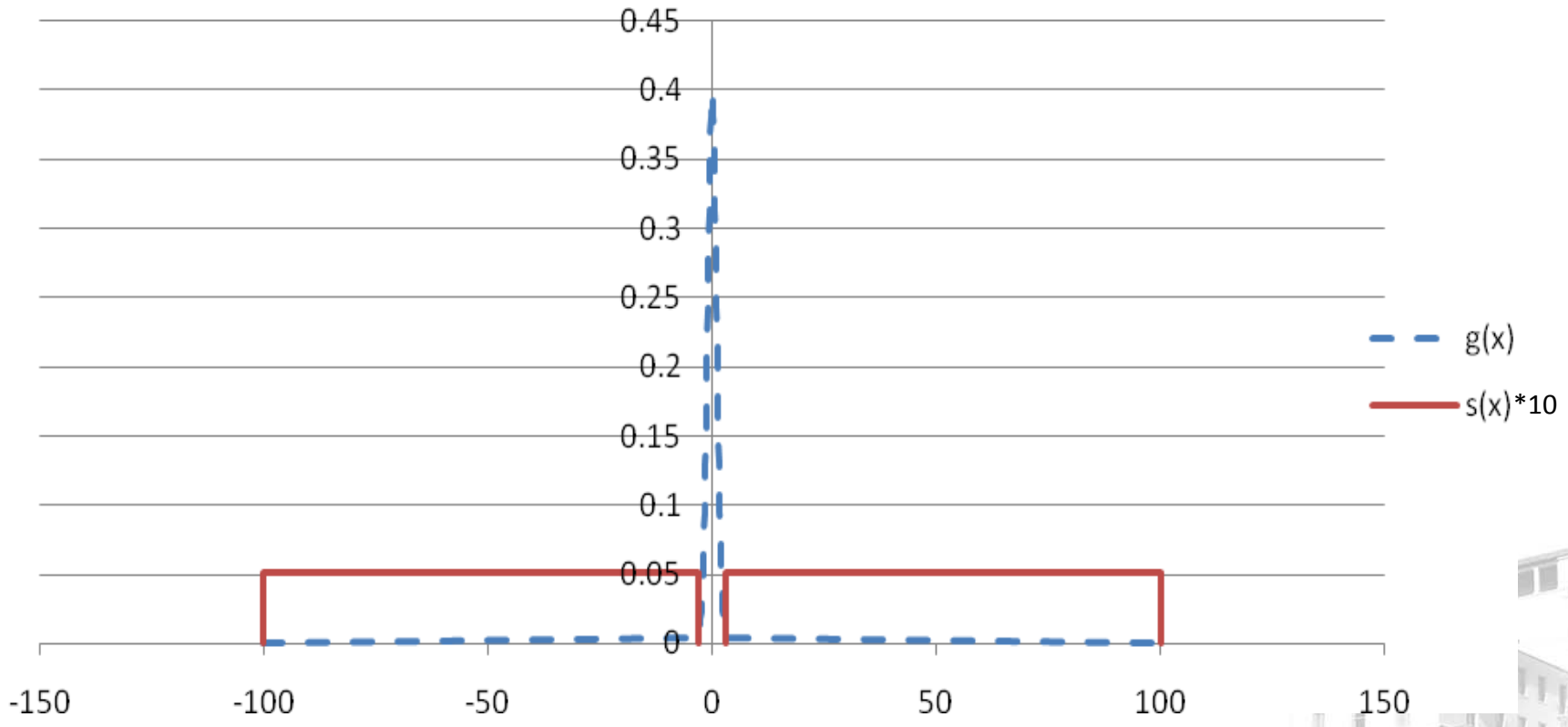
Derived Testing Profile



Example of Using an Operational Profile for Reuse Testing (cont)

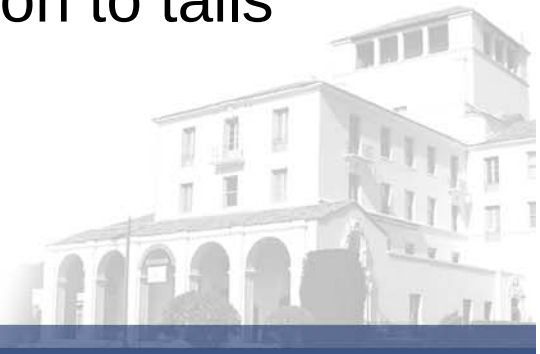
- How to stress test the software?
  - Safety or operationally critical software
  - Extended boundary condition testing
  - Checks for “unknown unknowns”, prevents surprises from the new environment
- Rough guideline: test out to 100 standard deviations

Stress Testing Profile



Methods for modeling operational profiles

- Identify all environment inputs and their dependencies
  - Possible use of conditional distributions
- Estimate distribution for each input
  - Mathematical analysis and use of histogram “bins” when raw historical data is available
  - Smoothing, interpolation & extrapolation to tails where raw data is missing



Methods for modeling operational profiles (cont)

- Use of Bayesian methods for estimating distributions of actual data
- Implementing Stress Test profiles
  - When not enough information is known about current or past operational environments
  - Always for safety critical software
- Calculate statistical confidence levels in the profile model based on sample size

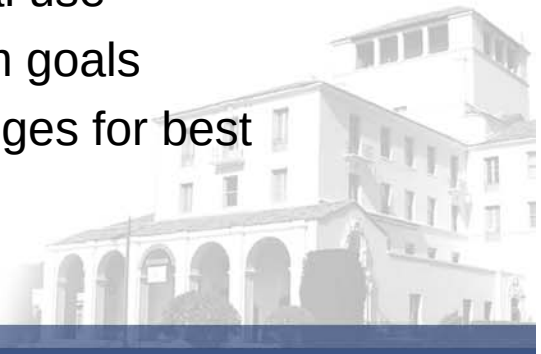
Acquisition Process Implications

- Requirements analysis needs to span the entire problem domain and system life, not just individual versions of the System of Systems
  - Same architecture must support all future versions and all platforms
  - Planned control of variation via ranges for parameters/features
- Re-orient development processes toward Design-to-Tolerances
  - Currently oriented towards Design-to-Fit, Test-to-Fit
- The architecture as a whole needs authority / priority
  - Responsible organization
  - Global system standards authority
  - Manage accountability for subsystems
  - Empower via change control, acceptance testing, budget control, contracts with incremental commitment

Acquisition Process Implications

- Domain requirements/Architecture development / QA need substantial time/resources/technology development
  - Must be included in the plan from the start
  - More detailed/precise standards and analysis needed
  - Shift from current requirements to likely requirements trajectories

- New QA technologies needed
  - Some known in labs but not used currently
  - Tailoring/improvement may be needed for practical use
  - Some areas need new methods to reach long term goals
  - Will need tech transfer, training, and process changes for best practical impact



Short Term Recommendations

- Testing profiles and statistical test results should be attached to reusable components in repositories.
- Operational Profiles should be measured based on observed data.
- Validity of pointers and storage recycling should be checked by tools especially if components not retested based on slicing.
- Absence of code modification should be checked at runtime via cryptographic signatures.
- Automated invariance testing should be applied to components whose specifications are unchanged but hardware or code affecting behavior has changed.

Short Term Recommendations (cont)

- Statistical testing should be performed for safety-critical and mission critical functions.
- Need uniform guidance for mission-critical reliability, analogous to MIL-STD-882D for system safety.
- Effectiveness and safety of slicing criteria for avoiding retesting should be validated with a case study/demo.
- Reusable components should monitor assumptions about their operating environment at runtime.

Conclusions

- The slicing and automated testing approach has a potential to reduce testing duration and costs
  - More research is recommended to substantiate the applicability of our approach to DoD systems
  - Experimental evaluation of slicing and invariance testing methods is needed
- Automated testing techniques can alleviate concerns about system risks due to technology innovations
- Measurement and analysis of the operational profiles of reusable components can be used to support analysis of changes in the operating environments
  - Hence determining whether additional testing is necessary

Backup Slides

Approach: Program Slicing [Weiser 84]

- What is a slice?
  - A self-contained subset of a program
    - Contains all of the code that affects its observable behavior
  - Determined by an observation point
    - Example: behavior of a single service
  - Contains only the relevant parts
- Why do slices matter?
  - Behavior invariance property:
    - *If a service has the same slice in two different versions of a program, it has the same behavior in both versions*
  - *If two slices are the same, the service does not have to be retested*
  - Slices can be computed on a large scale
    - Involves dependency tracing, data flow analysis, and control flow analysis

Invariance Testing Extends Program Slicing

- Used to check that behavior of modified code remains the same
  - Candidates: Open Architectures and higher level middleware
    - Enables effective slicing cutoff boundaries
  - Example: operating system interface
  - Example: upgrade from a deprecated interface
  - Example: baseline specific interfaces used by common components
- Enhances slicing to identify more components that do not need retesting
- Relies on a statistical inference with a very high confidence level
  - Needs large numbers of test cases
  - Economically feasible because this kind of test and analysis can be *completely automated*
    - Test cases - generate inputs by random sampling
    - Data analysis - compare outputs from two different software versions

Related Work

- Navy systems are designed with open architecture in mind
  - Hence encapsulating all system calls
- Program Slicing has been used in a wide variety of applications: testing, debugging, program understanding, reverse engineering, software maintenance, change merging, software metrics.
  - See paper for extended list of citations.
- Automate testing has been used to automatically generate open sets of test cases based on random samplings from implementations of operational profile distributions [Berzins and Chaki 2002]
- Prior work on quality assurance for flexible systems at the level:
  - Of requirements [Luqi, Zhang, Berzins & Qiao 2004] [Luqi & Lange 2006]
  - Of architectures [Berzins & Luqi 2006]][Luqi & Zhang 2006]