

NPS-AM-09-022



EXCERPT FROM THE PROCEEDINGS

OF THE
SIXTH ANNUAL ACQUISITION
RESEARCH SYMPOSIUM

HOW TO CHECK IF IT IS SAFE NOT TO RETEST A COMPONENT

Published: 22 April 2009

by

Valdis Berzins and Paul Dailey

**6th Annual Acquisition Research Symposium
of the Naval Postgraduate School:**

**Volume I:
Defense Acquisition in Transition**

May 13-14, 2009

Approved for public release, distribution is unlimited.

Prepared for: Naval Postgraduate School, Monterey, California 93943



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

Report Documentation Page

Form Approved
OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE APR 2009		2. REPORT TYPE		3. DATES COVERED 00-00-2009 to 00-00-2009	
4. TITLE AND SUBTITLE How to Check If It Is Safe Not to Retest a Component				5a. CONTRACT NUMBER	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S)				5d. PROJECT NUMBER	
				5e. TASK NUMBER	
				5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School, Monterey, CA, 93943				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)				10. SPONSOR/MONITOR'S ACRONYM(S)	
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT This paper focuses on ways to reduce testing effort and costs associated with technology-advancement upgrades to systems with open architectures. This situation is common in Navy and DoD contexts such as submarine, aircraft carrier, and airframe systems, and accounts for a substantial fraction of the testing effort. This paper describes methods for determining when testing of unmodified components can be reduced or avoided, and it outlines some methods for choosing test cases efficiently to focus retesting where it is needed, given information about past testing of the same component. Changes to the environment of a system can affect its reliability, even if the behavior of the system remains unchanged. The new capabilities added by a technology upgrade can interact with previously existing capabilities, changing the frequency of their usage as well as the range of input values and, hence, changing their effect on overall system reliability.					
15. SUBJECT TERMS					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT Same as Report (SAR)	18. NUMBER OF PAGES 53	19a. NAME OF RESPONSIBLE PERSON
a. REPORT unclassified	b. ABSTRACT unclassified	c. THIS PAGE unclassified			

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

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

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

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

Conference Website:
www.researchsymposium.org



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

Proceedings of the Annual Acquisition Research Program

The following article is taken as an excerpt from the proceedings of the annual Acquisition Research Program. This annual event showcases the research projects funded through the Acquisition Research Program at the Graduate School of Business and Public Policy at the Naval Postgraduate School. Featuring keynote speakers, plenary panels, multiple panel sessions, a student research poster show and social events, the Annual Acquisition Research Symposium offers a candid environment where high-ranking Department of Defense (DoD) officials, industry officials, accomplished faculty and military students are encouraged to collaborate on finding applicable solutions to the challenges facing acquisition policies and processes within the DoD today. By jointly and publicly questioning the norms of industry and academia, the resulting research benefits from myriad perspectives and collaborations which can identify better solutions and practices in acquisition, contract, financial, logistics and program management.

For further information regarding the Acquisition Research Program, electronic copies of additional research, or to learn more about becoming a sponsor, please visit our program website at:

www.acquisitionresearch.org

For further information on or to register for the next Acquisition Research Symposium during the third week of May, please visit our conference website at:

www.researchsymposium.org



THIS PAGE INTENTIONALLY LEFT BLANK



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

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

Valdis Berzins
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943
E-mail: berzins@nps.edu

Author: Paul Dailey is a systems engineer for the Office of Naval Intelligence in Washington, DC. He has also recently worked as a systems engineer for the Naval Surface Warfare Center Port Hueneme Division Detachment, Louisville, and has been working for the Department of the Navy for close to seven years. He holds an MS in Systems Engineering from the Naval Postgraduate School and a BS in Electrical and Computer Engineering from the University of Louisville. He is currently pursuing a PhD in Software Engineering from the Naval Postgraduate School, focusing his research on the automated testing of software.

Paul Dailey
Software Engineering PhD student
Naval Postgraduate School
Monterey, CA 93943
E-mail: prdailey@nps.edu

Abstract¹

This paper focuses on ways to reduce testing effort and costs associated with technology-advancement upgrades to systems with open architectures. This situation is common in Navy and DoD contexts such as submarine, aircraft carrier, and airframe systems, and accounts for a substantial fraction of the testing effort. This paper describes methods for determining when testing of unmodified components can be reduced or avoided, and it outlines some methods for choosing test cases efficiently to focus retesting where it is needed, given information about past testing of the same component. Changes to the environment of a system can affect its reliability, even if the behavior of the system remains unchanged. The new capabilities added by a technology upgrade can interact with previously existing capabilities, changing the frequency of their usage as well as the range of input values and, hence, changing their effect on overall system reliability.

Keywords: open architecture, reducing regression testing, automated testing, statistical testing, dependency analysis, reuse, technology upgrades.

¹ This research was supported in part by ARO under project P-45614-CI and, in part, by NAVSEA under project number 09WX11455.



Introduction

Current US Navy combat and weapon system test procedures require an integration test event with every change to the software or system configuration to certify that the software-intensive system-of-systems is stable and functional. As more systems are moving to a modular open architecture, software configurations are changing with increased frequency, requiring more testing, which is expensive and time-consuming.

The Navy's open architecture framework is intended to promote reuse and reduce costs. Ongoing research at the Naval Postgraduate School is developing improvements to the test and evaluation procedures that can contribute to these goals. Test and evaluation accounts for a large part of system-development cost, but the impact of open architecture ideas on this part of the process has been relatively modest so far. The purpose of this effort is to provide sound engineering approaches to better realize the potential benefits of Navy open architectures and to provide concrete means that support economical acquisition and effective sustainment of such systems.

The specific goals of this research are to enable: (1) identification of specific testing and checking procedures that do not need to be repeated after given changes to a system, (2) limiting the scope and reducing the cost of retesting when the latter is necessary, and (3) a single analysis to provide assurance that all possible configurations that can be generated in a model-driven architecture will satisfy given dependability requirements. This paper reports some results that address the first two of the goals listed above. A roadmap and technical approach for reaching the third goal are outlined in Berzins, Rodriquez, and Wessman (2007).

Technology upgrades are typically performed on a two-year cycle. They often involve migration to the best hardware and operating system version available at the time, where "best" implies a balanced trade-off between high performance and reliable operation. Typically, only a small fraction of the application code has been changed. However, current certification practices require all of the code to be retested prior to deployment, whether it has been modified or not. Retesting of an unchanged module can be avoided only if we can establish that it has not been adversely impacted by the change. Preliminary results on how to do that have been reported by Berzins (2008). In this paper, we further explore ways to determine whether it is safe not to retest an unchanged component under the assumption that the load characteristics of the component have not changed. We also address the problem of how to most effectively focus retesting for unchanged components in cases where the requirements and behavior of the component have not changed but the load characteristics have changed.

The latter situation has great importance for assuring reliability of reusable components. Many past cases of well-publicized software failures involved reuse of software components in new environments that had different characteristics than the contexts for which the components were originally designed. These components failed in their new environments despite the fact that they were well-tested and found to be reliable in the field under previous deployment conditions. Examples include the Patriot missile failure (Marshall, 1992) and the failure of the European Ariane 5 rocket (Jézéquel & Meyer, 1997, January).

The rest of this paper is organized as follows: Section 2 describes methods for deciding when re-testing of unchanged components can be safely reduced or eliminated



entirely; Section 3 presents methods for efficiently retesting reusable components for use in deployment environments with workloads that are different from previous deployments; Section 4 identifies some relevant previous work; and Section 5 presents our conclusions.

Deciding When Retesting Can Be Avoided

Our previous work identified two types of analysis that could enable safe avoidance of retesting unchanged components under certain conditions: program slicing and invariance testing (Berzins, 2008). These techniques are applicable in cases in which the requirements, code, expected workload and available resources of the component are unchanged. This section briefly reviews the approach and then examines in more detail what additional analysis needs to be done to safely reuse such components in the next release without retesting them.

Program slicing is a kind of dependency analysis that is based on the source code. Slicing algorithms are efficient enough to be used on practical, large-scale programs. If two different versions of a program have the same slice with respect to a service it provides, then that service has the same functional behavior in both versions, and retesting can be avoided if having the same functional behavior is sufficient to establish the reliability of the component (Gallagher, 1991, August).

Invariance testing is a kind of statistical, automated testing that is applicable to components whose code has changed but whose specifications and requirements remain the same. The purpose of an invariance test is to confirm that the changes to the code have not changed the behavior of the services it provides. In this kind of a situation, it is easy to implement a test oracle procedure (explained below) that enables affordable checking of large numbers of automatically executed test cases. Invariance testing can increase the number of components that can be certified not to need retesting when combined with program slicing (Berzins, 2008). Invariance testing can also be used to educe the cost of retesting modules that need to be retested, even though their requirements remain unchanged. This includes unchanged components that depend on other modified components, which are identified by program slicing methods, as well as unchanged components whose expected workload has changed (see section 3).

We can omit retesting of a service if slicing and invariance testing confirm that its behavior is unchanged in the new release and that the following additional conditions are met:

1. The same functional behavior is appropriate in the new release, which occurs only if the requirements of the component are unchanged.
2. The same functional behavior is sufficient to meet the requirements only if the requirements do not contain timing constraints. If this is not the case, the timing constraints need to be retested because changes to hardware, systems software, and other components in the system can all affect timing. This can be done by using a kind of invariance testing that measures timing and by the methods described by Qiao, Wang, Luqi, and Berzins (2006, March).
3. Constraints due to shared resources need to be rechecked, which can usually be done via system-level stress testing. Such constraints include:
 - a. Sufficient main memory and disk space



- b. Sufficient I/O resources such as number of files that need to be open at the same time, printers, sensors, actuators, or other peripherals.
 - c. Sufficient network bandwidth to support worst-case communications load.
 - d. Effective access to showed databases and web services, including both timing and freedom from deadlocks.
4. The slicing analysis is only valid under the assumption that the machine code that is actually running corresponds to the source code that was subjected to slicing analysis.
 5. The analysis depends on the assumption that the computer correctly translates the source code into machine language.

The fourth assumption is frequently made without explicit acknowledgement in theoretical studies, but it cannot be adopted without verification in serious risk analysis because of the following plausible failure modes:

1. Memory-corrupting bugs—these include out-of-bounds write operations on arrays and through invalid pointers. Such bugs can cause seemingly innocuous statements to overwrite parts of the program itself at runtime, with unpredictable and potentially catastrophic results.
2. Deliberate cyber-attacks—compromise of system security via network or unauthorized insider access to systems can deliberately modify machine code at run-time.

Memory-corrupting bugs are faults in the code that should be detected by test and evaluation processes, and some types can be prevented. One class of memory-corrupting bugs is caused by premature deallocation of dynamically created objects. Garbage-collection algorithms are supposed to prevent this class of problems so that garbage-collected languages such as Java and Lisp should be immune to this type of problem. Software written in languages without garbage collection, such as C, C++ and Ada, needs special quality-assurance methods to look for premature deallocation. There exist a variety of tools that can be used for this task, including Valgrind (2009, April) (see the system commands Memcheck and Ptrcheck) and Insure ++ (2009, April).

We note that in the absence of perfect computer security, which is not likely to be attainable in the near future, no amount of test and evaluation can detect or prevent failures of the second kind because they are not present in the system while it is being tested; they only appear later—after attacks at run-time. We, therefore, recommend adding a design modification that checks at run-time whether component code is still the same as it was in the test load for all mission-critical systems that do not already have such a capability.

This can be done by packaging the machine code in blocks with secure digital signatures and adding a process that periodically checks the signatures while the system is running. To make this secure, the digital signatures have to be cryptographic checksums with strong encryption so that attackers cannot modify a code module and then forge a signature without knowledge of the secret key. The periodic checking process systematically scans the code modules and checks their digital signatures. If it discovers a modified module, it can repair that module and also report the problem to appropriate authorities. Repair can be accomplished by reloading the module from an uncorruptable source such as read-only memory or CD. Failure due to possible physical damage to media can be mitigated by redundant copies. The repair process checks the digital signature of the new



copy to verify its integrity and goes to alternative backup copies if there are any discrepancies. We note that this mechanism can be used to compensate for faults due to memory corruption regardless of whether they were caused by attacks or by faults in the code. The state of corrupted modules will usually have to be restored to the most recent, valid date after the corrupted code is repaired. Component designs may have to be augmented to provide this service. There is extensive literature on how to perform rollbacks, particularly in the context of database transitions. A discussion of this problem for object-oriented components can be found in Vandewoude and Berbers (2005).

The mechanism proposed above is similar to a scheme used by a telephone company to keep its software operational, despite the presence of memory-corrupting bugs, which were known to exist but whose source could not be located. This technology has been proven effective in practice and has been used for decades.

The mechanism can also repair faults due to corruption of data if the scanning process understands the data structures and has code to check the invariant constraints associated with them. This can be incorporated into the architecture via a standard interface that every data type must implement for a service that checks all associated data constraints and repairs them if needed.

Technology upgrades typically move to new hardware, which implies the use of new compilers and new versions of the operating system. Presumably, these underlying services are reliable, but, if we are to retest only a subset of the components in the new release, these assumptions need to be verified. This can be done using invariance testing, as explained by Berzins (2008). The correct operation of the new version of the compiler can be checked by combining invariance testing with the approach to testing translators described in Berzins, Auguston, and Luqi (2001, December).

Retesting Unchanged Components under New Load Conditions

The previous section discusses situations in which the following conditions hold:

1. The code of the component is unchanged.
2. The requirements and specifications of the component are unchanged.
3. The expected workload of the component is unchanged.

This section examines what should be done if the first two conditions are met, but the third one is not: the code and requirements of a component are unchanged, but the expected workload is different. This situation is expected when a component is reused in a different context. Such situations will be common when one of the stated objectives of open architectures is achieved: extensive reuse of common components across platforms.

In these cases, some retesting is necessary. We would like to do this efficiently by reusing previous test results and focusing additional testing effort on the system behavior that will be exercised more in the new workload than it was in the previous ones. We, therefore, seek a systematic method to generate new test cases that characterize situations expected in the new deployment context that were not expected in the previous deployment contexts. This informal idea can be made precise in the context of automated statistical testing (Berzins, 2008).

Automated statistical testing is characterized by the following properties:



1. Test cases are automatically generated by random sampling from an *operational profile*. An operational profile is a probability distribution that represents the relative frequency of different input values to the system under test in its expected execution environment.
2. Pass/fail decisions for individual test cases are automated and done by a single *test oracle* procedure that applies to all possible inputs to the service or system under test.
3. If the generated set of test cases runs without detecting any failures, a simple formula gives a lower bound on the mean number of executions with a corresponding statistical confidence level.

The significance of the first two conditions is economic: after the fixed initial cost of implementing the operational profile and the test oracle, the marginal cost of running an additional test case is very small. This is because there is no additional human effort associated with additional test cases; only additional computer resources are needed to run more test cases, and computer time costs much less than human effort.

The consequence is that very large numbers of test cases can be run economically, making it affordable to collect sample sets large enough to provide high statistical confidence levels in the results. Methods for determining the sample size needed to support conclusions of the form “the mean number of executions between failures is at least N with confidence $(1 - (1/N))$ ” can be found in Berzins (2008). The significance of this is that it can enable practical testing to specified risk-tolerance levels, rather than testing until budget runs out. The latter does not provide high confidence in system reliability, although it occurs commonly in current practices.

Figure 1 shows an example of the situation described above. The distribution g_1 represents the operational profile for the initial deployment of a hypothetical reusable component and g_2 represents the operational profile characterizing a new environment in which the component is to be reused. Note that a wider range of input values is expected in the new environment. In this example, g_1 and g_2 are normal distributions; g_1 has a standard deviation of 1.0, and g_2 has a standard deviation of 2.0.



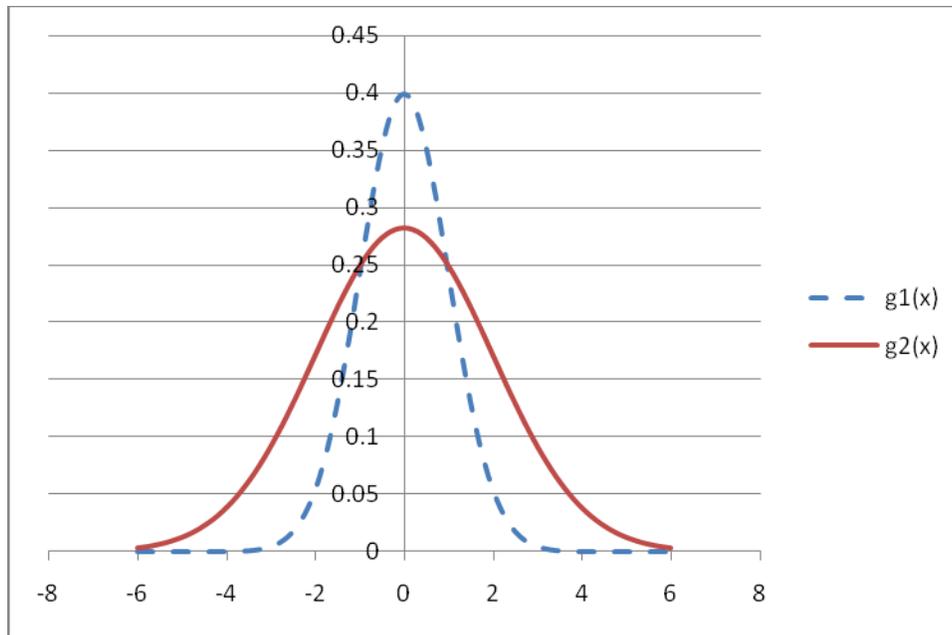


Figure 1. Operational Profiles for Two Different Deployment Environments

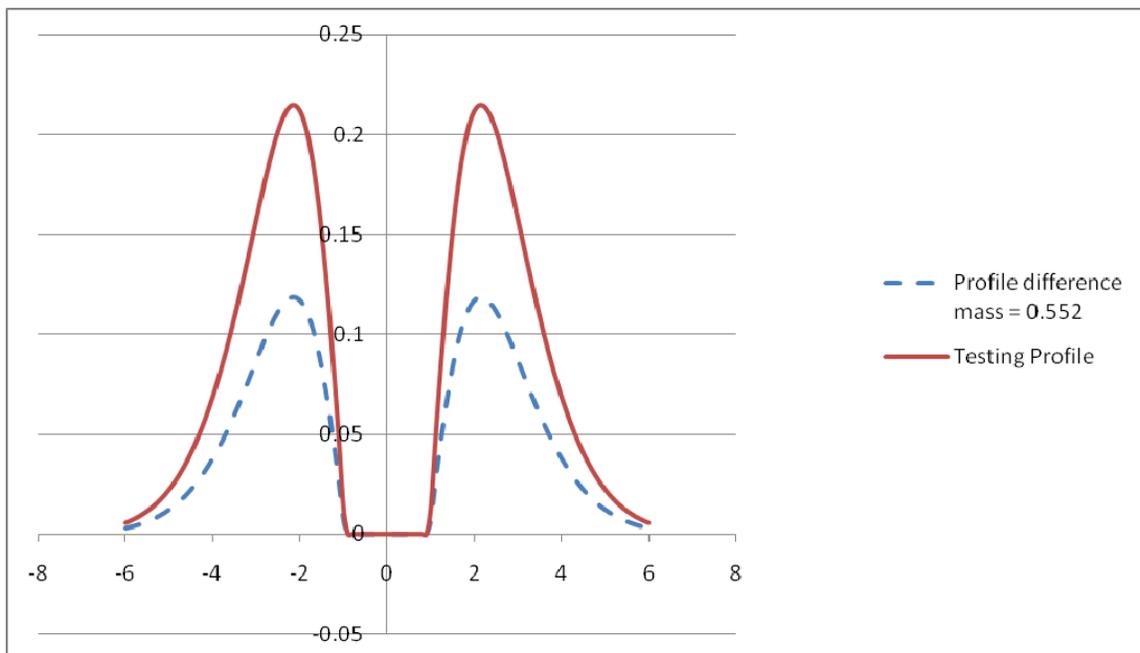


Figure 2. The Derived Testing Profile

Figure 2 shows the profile difference for incremental testing that is derived from the distributions in Figure 1 and the resulting testing profile under the assumption that the number of test cases needed to reach the reliability goals associated with both the previous and the new execution environment are the same.

The profile difference is zero in the region where $g_1 > g_2$, and it is equal to the difference $g_2 - g_1$ everywhere else. The rationale for these choices is the following:

The region where $g_1 > g_2$ has already been adequately tested since the expected number of samples from this region that were checked during prior testing using the profile g_1 exceed the expected number of samples from the same region that would be required in testing under the execution profile g_2 , characterizing the new deployment environment. Therefore, we can avoid this region in the second round of testing, which is accomplished by setting the testing profile to zero in this region.

The remaining region needs more test cases for adequate coverage. If we consider an arbitrary slice of this region, we find that the area under g_1 in this slice represents the expected number of test cases that were run in the previous round of testing governed by the profile g_1 . The area under g_2 in the same slice represents the expected number of test cases from the slice that need to be run in the second round of testing. The total area under the profile difference represents the number of test cases needed for the second round of testing as a fraction of the number of test cases required in the first round of testing. In the example, this fraction is calculated to be .552. The testing profile is proportional to the profile difference, which must be normalized by dividing it by the probability mass under the curve to make all of the probabilities add up to 1.

The more general case—in which the reliability goals in the two execution environments differ—has a similar rationale, but the two distributions have to be scaled to account for the differences in the number of test cases needed in each test.

Let N_1 be the number of test cases that were needed from profile p_1 for the first deployment environment and N_2 be the number of cases from a different profile p_2 , needed for the second environment. Then, in the general case, the profile difference is zero where $N_1 * p_1 > N_2 * p_2$ and is equal to $(N_2 * p_2 - p_1 * N_1) / (N_1 + N_2)$ elsewhere.

The testing profile is again the normalized profile difference, obtained by dividing it by the area under the profile difference curve.

We are currently investigating effective methods for modeling operational profiles and for deriving model parameters from historical measurements of actual system loads. Such measurements can come from instrumenting systems to collect data during training exercises or actual missions.

The inputs to the software module must be analyzed to determine dependencies among them. It is also necessary to look for dependencies between the interfaces and other external environmental factors within the context of the operational profile and testing goals. If dependencies exist, they should be characterized.

Once the inputs and the relationship(s) among them are known, the next step is to estimate or specify the distributions that characterize the probabilistic behavior of the inputs. If there are dependencies, the notion of conditional distributions will be considered as a way to handle them. There also may be multiple possible distributions for each input, depending on the state of the environment. This also applies if the goals can vary from testing the normal range of inputs to testing extreme cases, which may be necessary for checking boundary conditions and checking the robustness of the component with respect to unplanned contingencies.



A histogram can be used to represent the new data resulting from the measurements to provide a visual check of the observations. However, it is advisable to fit a distribution based on a theoretical model of the expected distributions for the following reasons:

1. Smoothing—the histogram will show irregularities due to granularity of the random sampling in the measurements. These are not physically significant and are most effectively mitigated by finding the best fit to a smooth curve that interpolates between the samples and smoothes out the gaps.
2. Extrapolation—realistic probability distributions do not cut off suddenly but rather gradually decrease with long tails. Such tails are impossible to accurately estimate based solely on measured data because the number of observed samples is often too small to provide an accurate measurement near the extremes of the expected range of values. If we use the histogram as measured, it is likely that we will set the probability distribution to zero in places where it is actually small, but nonzero. Since this will result in tests that do not cover the full range of possible parameter values, we propose to use a theoretical model in this region and to do the extrapolation by matching the standard deviation of the actual measurements to the standard deviation of the theoretical model. This will smoothly extrapolate the tails out to or beyond the real limits of the input value range. Details about how to choose an appropriate theoretical model for this purpose are still under investigation.

We are also planning to investigate the effectiveness of Bayesian methods for estimating the distributions based on the actual data. This approach will also need a theoretical model of the probability distribution function, which will be used as the prior distribution.

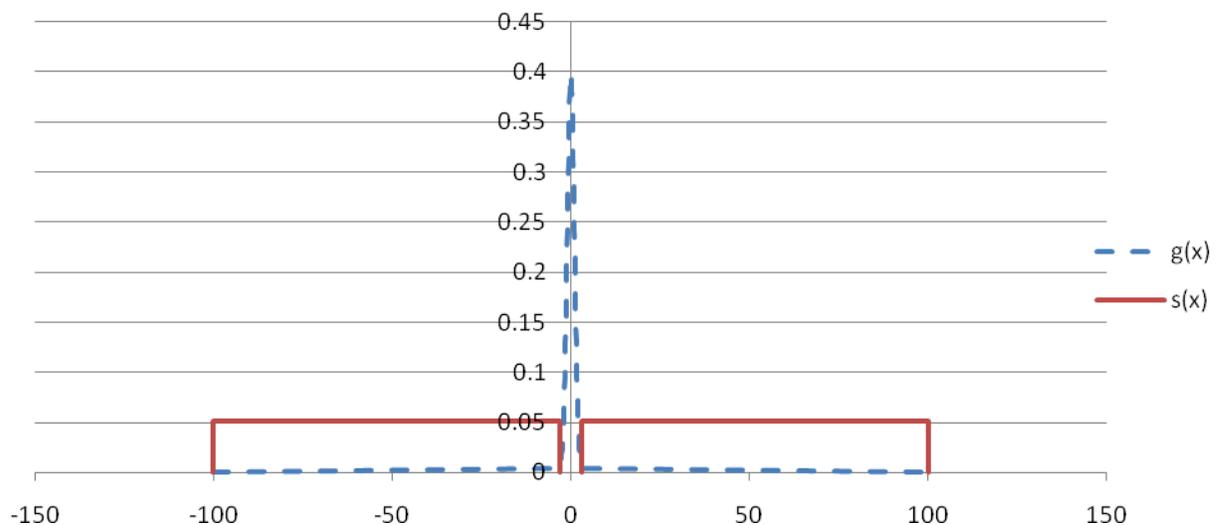


Figure 3. A Stress-testing Profile, $s(x)$, Compared to an Operational Profile, $g(x)$

The methods outlined above should provide a systematic way to deal with the “known unknowns.” However, military environments are characterized by uncertainty and surprises. To hedge against the possibility of “unknown unknowns,” we recommend running additional tests on components to be reused in new environments with a “stress-testing profile” that purposely exaggerates the range of expected input values. This kind of stress testing is difficult to put on a scientific basis because we are trying to hedge against

possibilities that we have no basis for predicting. The following heuristics are proposed as strategies to try:

1. Use a uniform distribution that extends from three to one hundred standard deviations in all directions from the measured mean of the distribution. This is illustrated in Figure 3. The curve *g* shown in blue represents the normal profile, which is the same as the curve *g1* shown in Figure 1, and the curve *s* represents the stress-testing profile. The curve *s* has been scaled up by a factor of 10 to make it easier to see in the figure.
2. Use a uniform distribution that covers the entire valid range of input values. This will include completely unexpected input values.

Recalling that these strategies are intended to be used in the context of completely automated statistical testing, in which the marginal cost of running and analyzing additional test cases is very low, we recommend a mixed strategy that runs tests from all three of the proposed testing profiles, each with a number of samples derived from the risk-tolerance parameter *k*, specified by system stakeholders and the measured execution frequency parameters *e_s* according to the relation $T_s = (k e_s) \log_2(k e_s)$, as explained in Berzins (2008). *T_s* represents the number of the test cases that are needed for testing services to the statistical confidence level implied by the specified risk-tolerance parameter.

Relevant Previous Work

Methods for detecting memory corrupting bugs via static and dynamic program analysis have been studied (Alzamil, 2006; 2008, November). Program slicing (Weiser, 1984, July) has been used in a wide variety of applications, including testing (Binkley, 1998; Gupta, Harrold & Soffa, 1992; Harman & Danicic, 1995; Hierons, Harman & Danicic, 1999; Hierons, Harman, Fox, Ouarbya & Daoudi, 2002), debugging (Agrawal, DeMillo & Spafford, 1993; Lyle & Weiser, 1987), program understanding (De Lucia, Fasolino & Munro, 1996; Harman, Hierons, Danicic, Howroyd & Fox, 2001), reverse engineering (Canfora, Cimitile & Munro, 1994), software maintenance (Gallagher, 1991, August; Cimitile, De Lucia & Munro, 1996; 1994), change merging (Horwitz, Prins & Reps, 1989; Berzins & Dampier, 1996), and software metrics (Lakhotia, 1993; Bieman & Ott, 1994). More detailed surveys of previous work on slicing can be found in Binkley and Harmon (2004).

The problem of state transfers for modules upgraded at run-time is addressed by Vandewoude and Berbers (2005). A method for assessing the impact of timing constraints on proposed system upgrades is described in Qiao, Wang, Luqi, and Berzins (2006, March).

Conclusion

Program slicing and invariance testing are methods that can be used to identify cases in which it is safe not to retest an unchanged component. These methods need to be augmented with other means for establishing the absence of other possible failure modes such as the possibility of memory-corrupting bugs and timing faults. This paper identifies ways to solve these issues.

When components are reused in environments with substantially different load characteristics than previous deployment environments, it is important to test the



components under the new modes of operation. This paper presents systematic and efficient ways to accomplish that.

Further work is needed to explore ways to address other possible failure models, including possible interference due to shared system resources, and to address the longer-term goal of eventually eliminating the need for repeating integration testing after every system change. Specifically, more work is needed on methods for certifying the reliability of architectures independently from the components that they contain and for certifying the conformance of an implementation to a given architecture in order to attain the long-term goals outlined in the introduction.

List of References

- Agrawal, H., DeMillo, R., & Spafford, E. (1993). Debugging with dynamic slicing and backtracking. *Software Practice and Experience*, 23(6), 589–616.
- Alzamil, Z. (2006). Application of computational redundancy in dangling pointers detection. In *Proceedings of the IEEE International Conference on Software Engineering Advances (ICSEA 2006)* (pp. 30–36). Tahiti: IEEE.
- Alzamil, Z. (2008, November). Application of redundant computation in program debugging. *Journal of Systems and Software*, 81(11), 2024-2033.
- Berzins, V., Auguston, M., & Luqi. (2001, December). Generating test cases for system generators. In *Proceedings of the Conference on Dynamic and Complex System Architecture*. Brisbane, Australia.
- Berzins, V., & Dampier, D. (1996). Software merge: Combining changes to decompositions. *Journal of Systems Integration*, 6(1-2, special issue on Computer-aided Prototyping), 135-150.
- Berzins, V., Rodriguez, M., & Wessman, M. (2007). Putting teeth into open architectures: Infrastructure for reducing the need for retesting. In *Proceedings of the Fourth Annual Acquisition Research Symposium* (pp. 285-311). Monterey, CA: Naval Postgraduate School.
- Berzins, V. (2008). Which unchanged components to retest after a technology upgrade. In *Proceedings of the Fifth Annual Acquisition Research Symposium* (pp. 142-153). Monterey, CA: Naval Postgraduate School.
- Bieman, J., & Ott, L. (1994). Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20(8), 644–657.
- Binkley, D. (1998). The application of program slicing to regression testing. In M. Harman & K. Gallagher (Eds.), *Program Slicing, Information and Software Technology*, 40(11-12, special issue), 583-594. San Diego, CA: Elsevier.
- Binkley, D.W., & Harman, M. (2004). A survey of empirical results on program slicing. In M.V. Zelkowitz (Ed.), *Advances in Computers* (pp. 105–178). (Vol. 62). San Diego, CA: Elsevier.
- Canfora, G., Cimitile, A., & Munro, M. (1994). RE²: Reverse engineering and reuse re-engineering. *Journal of Software Maintenance: Research and Practice*, 6(2), 53–72.
- Cimitile, A., De Lucia, A., & Munro, M. (1996). A specification driven slicing process for identifying reusable functions. *Software Maintenance: Research and Practice*, 8(3), 145–178.
- De Lucia, A., Fasolino, A., & Munro, M. (1996). Understanding function behaviours through program slicing. In *Proceedings of the 4th IEEE Workshop on Program Comprehension* (pp. 9–18). Los Alamitos, CA: IEEE Computer Society Press.
- Gallagher, K. (1991, August). Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8), 751-760.



- Gupta, R., Harrold, M., & Soffa, M. (1992). An approach to regression testing using slicing. In *Proceedings of the IEEE Conference on Software Maintenance* (pp. 299–308). Los Alamitos, CA: IEEE Computer Society Press.
- Harman, M., & Danicic, S. (1995). Using program slicing to simplify testing. *Software Testing, Verification and Reliability*, 5(3), 143–162.
- Harman, M., Hierons, R., Danicic, S., Howroyd J., & Fox, C. (2001). Pre/post conditioned slicing. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '01)* (pp. 138–147). Los Alamitos, CA: IEEE Computer Society Press.
- Hierons, R., Harman, M., & Danicic, S. (1999). Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, 9(4), 233–262.
- Hierons, R., Harman, M., Fox, C., Ouarbya, L., & Daoudi, M. (2002). Conditioned slicing supports partition testing. *Software Testing, Verification and Reliability*, 12(1), 23–28.
- Horwitz, S., Prins, J., & Reps, T. (1989). Integrating non-interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3), 345–387.
- Howden, W. (1987). *Functional program testing and analysis*. New York: McGraw-Hill.
- Insure++. (2009). Parasoft. Retrieved April 2009, from <http://www.parasoft.com/jsp/products/home.jsp?product=Insure>
- Jézéquel, J.-M., & Meyer, B. (1997, January). Design by contract: The lessons of Ariane. *Computer*, 30(2), 129-130.
- Lakhotia, A. (1993). Rule-based approach to computing module cohesion. In *Proceedings of the 15th Conference on Software Engineering (ICSE '93)* (pp. 34–44). Los Alamitos, CA: ACM/IEEE.
- Lammel, R. (2008, January). Google's map reduce programming model—Revisited. *Science of Computer Programming*, 70(1), 1-30.
- Lyle, J., & Weiser, M. (1987). Automatic program bug location by program slicing. In *Proceedings of the 2nd International Conference on Computers and Applications* (pp. 877–882). Los Alamitos, CA: IEEE Computer Society Press.
- Marshall, M. (1992). Fatal error: How patriot overlooked a scud. *Science*, 255(5050), 1347.
- Qiao, Y., Wang, H., Luqi, & Berzins, V. (2006, March). An admission control method for dynamic software reconfiguration in complex embedded systems. *International Journal of Computers and Their Applications*, 13(1), 28-38.
- Valgrind. (2009). Valgrind's tool suite. Retrieved April 2009, from <http://valgrind.org/info/tools.html>
- Vandewoude, Y., & Berbers, Y. (2005). Component state mapping for runtime evolution. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers* (pp. 230-236). Las Vegas, NV: PLC.
- Weiser, M. (1984, July). Program slicing. *IEEE Transactions of Software Engineering*, 10(4), 352-357.



2003 - 2009 Sponsored Research Topics

Acquisition Management

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

Contract Management

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

Financial Management

- Acquisitions via leasing: MPS case
- Budget Scoring
- Budgeting for Capabilities-based Planning
- Capital Budgeting for DoD



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

Human Resources

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

Logistics Management

- Analysis of LAV Depot Maintenance
- Army LOG MOD
- ASDS Product Support Analysis
- Cold-chain Logistics
- Contractors Supporting Military Operations
- Diffusion/Variability on Vendor Performance Evaluation
- Evolutionary Acquisition
- Lean Six Sigma to Reduce Costs and Improve Readiness
- Naval Aviation Maintenance and Process Improvement (2)
- Optimizing CIWS Lifecycle Support (LCS)
- Outsourcing the Pearl Harbor MK-48 Intermediate Maintenance Activity
- Pallet Management System
- PBL (4)
- Privatization-NOSL/NAWCI
- RFID (6)
- Risk Analysis for Performance-based Logistics
- R-TOC Aegis Microwave Power Tubes



- Sense-and-Respond Logistics Network
- Strategic Sourcing

Program Management

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

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



THIS PAGE INTENTIONALLY LEFT BLANK





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

www.acquisitionresearch.org



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



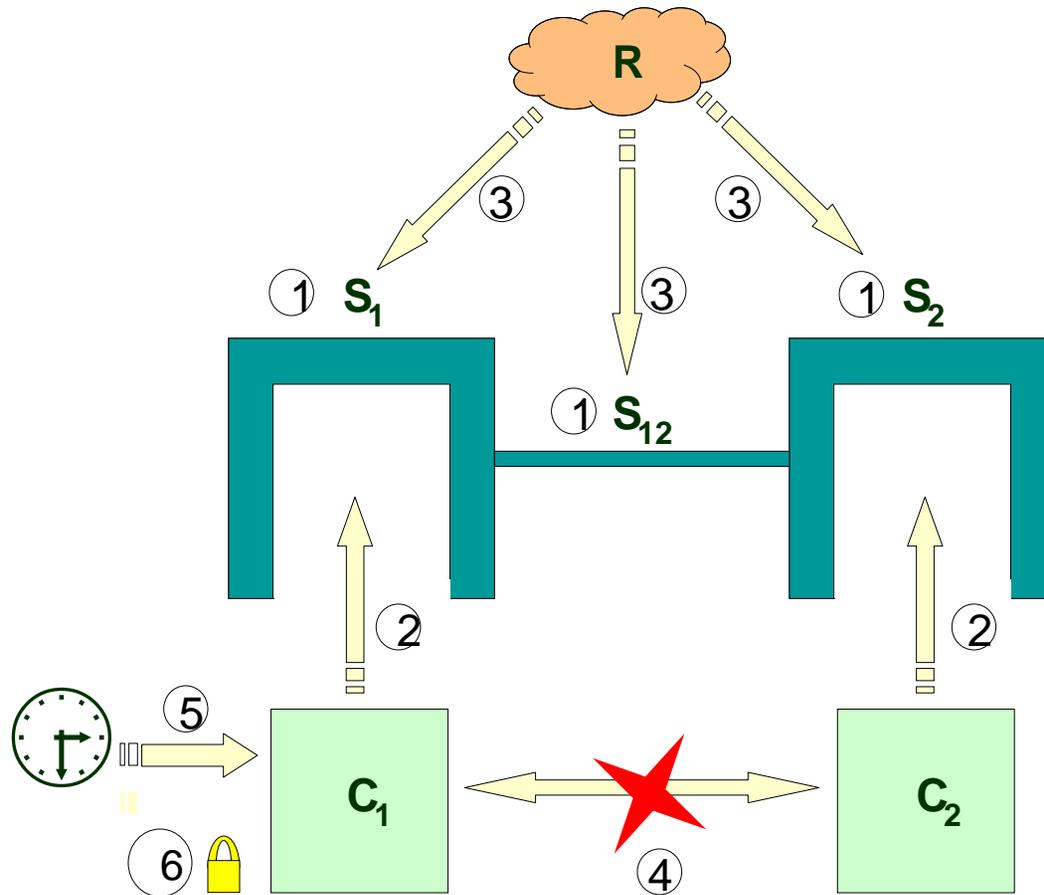
Scientific Roadmap - Approach

- 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
- S₁** Standard for Component 1
- S₂** Standard for Component 2
- S₁₂** Standard for connection between components 1 and 2
- C₁** Component 1
- C₂** 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

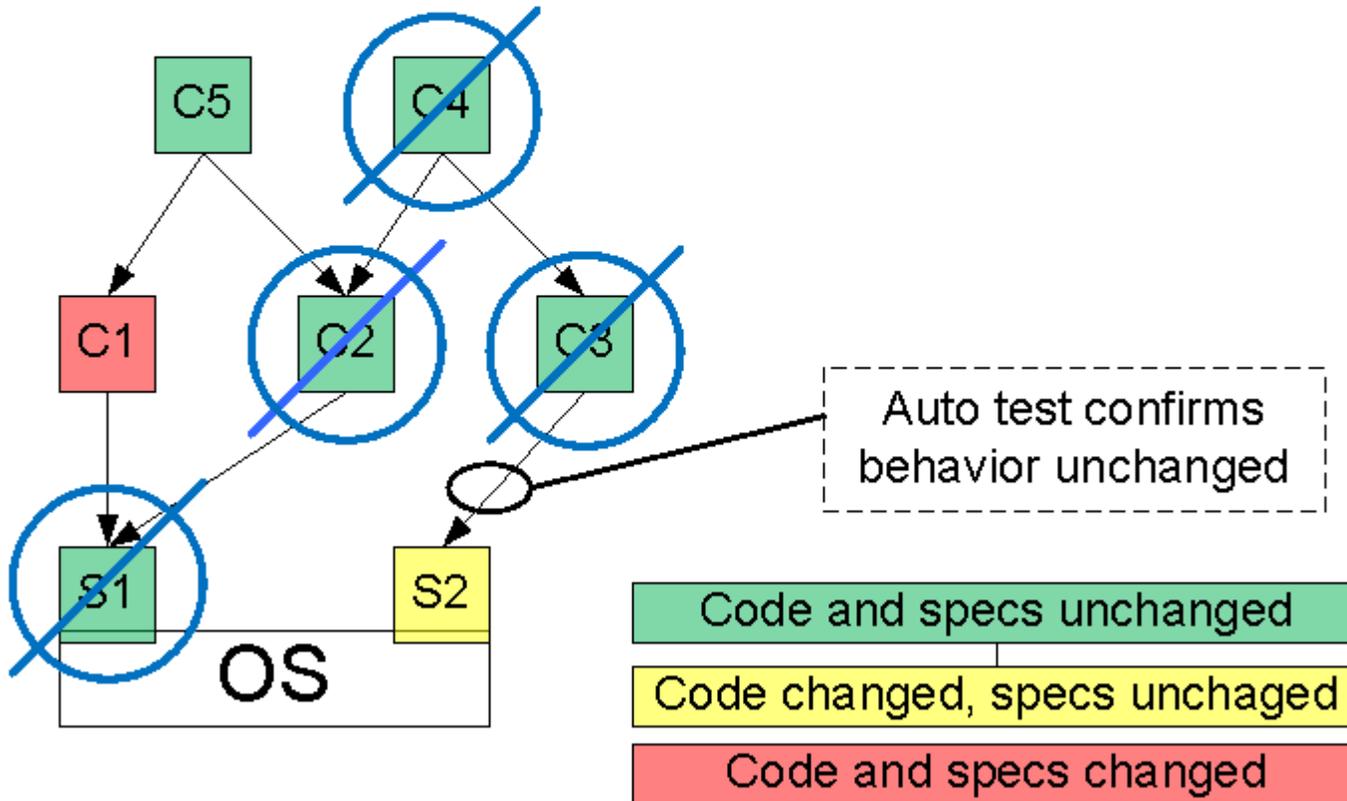
- When its slice or behavior has changed
- When requirements have changed
 - New functionality needs to be tested
 - Test all affected components
- When the *range of expected operating conditions* has expanded
 - Even if there was no other change, new test scenarios are needed
 - Indicated by a modified 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



= No retest due to slicing and invariance testing



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			

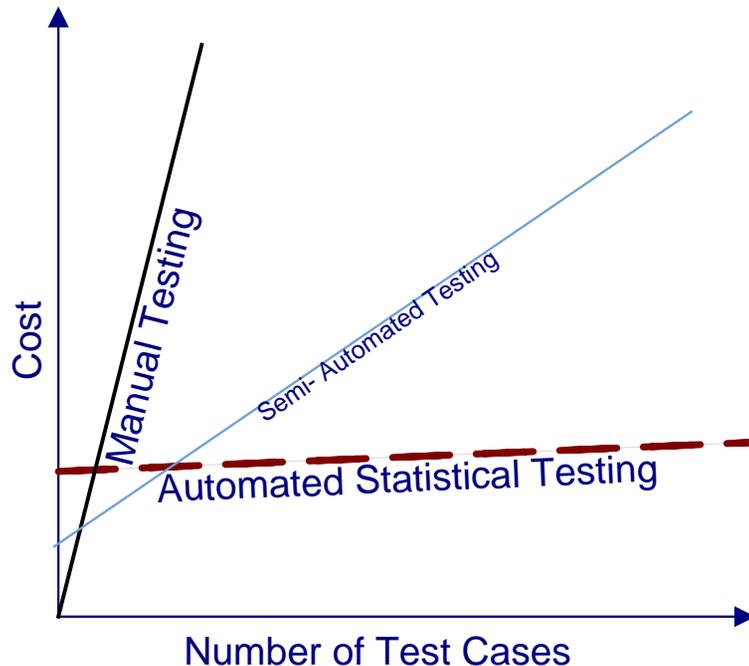
P: Probability of occurrence in the lifetime of an individual system, ranges taken from MIL_STD-882D



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

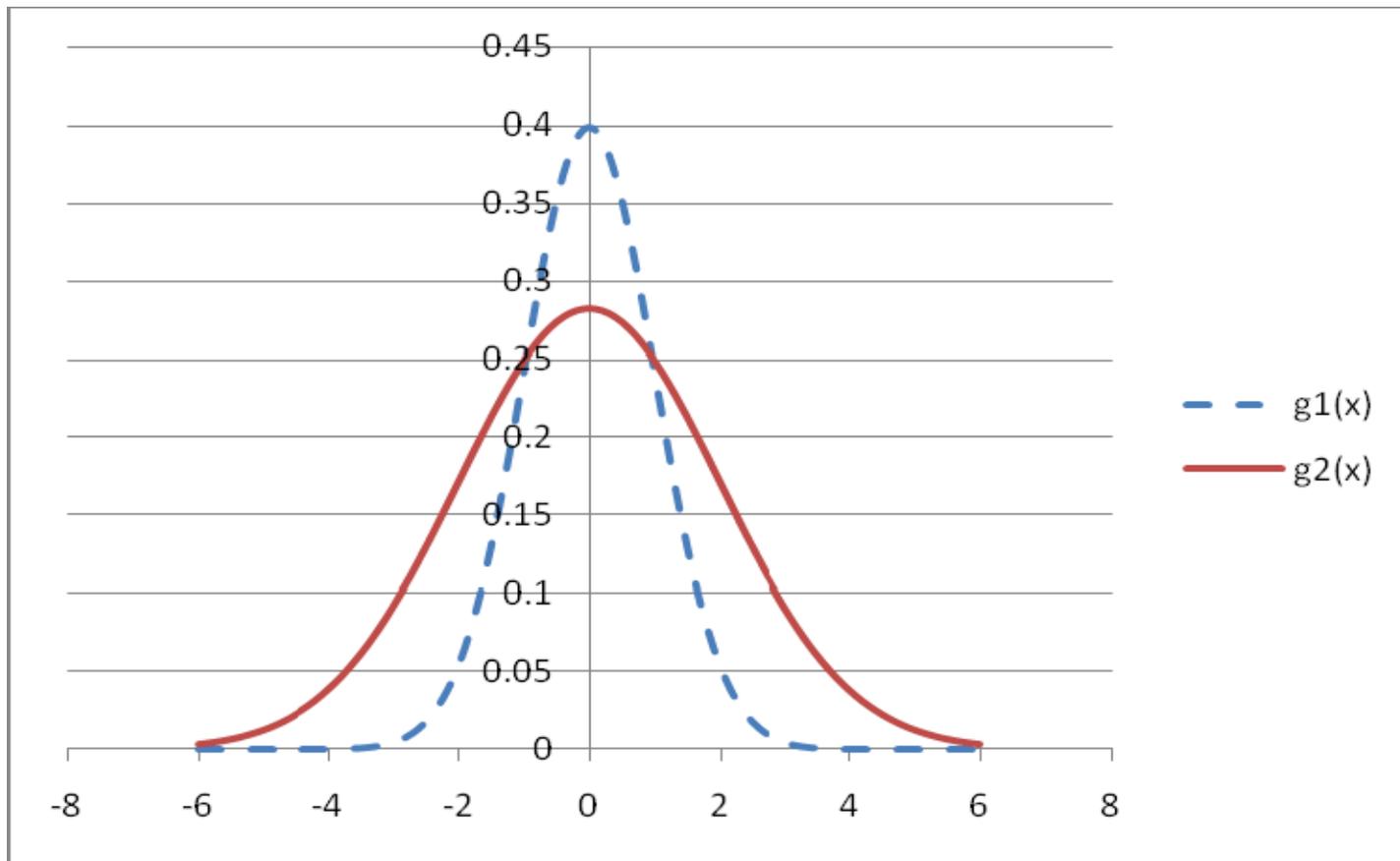


Example of Using an Operational Profile for Reuse Testing

- Currently fielded software has been tested with N samples from operational profile $g_1(x)$ and functions reliably in that environment
- Software is being reused and placed in new environment represented by operational profile $g_2(x)$
- What is the minimum amount of testing required to ensure operability and reliability in the new environment?



Operational Profile for Two Different Environments

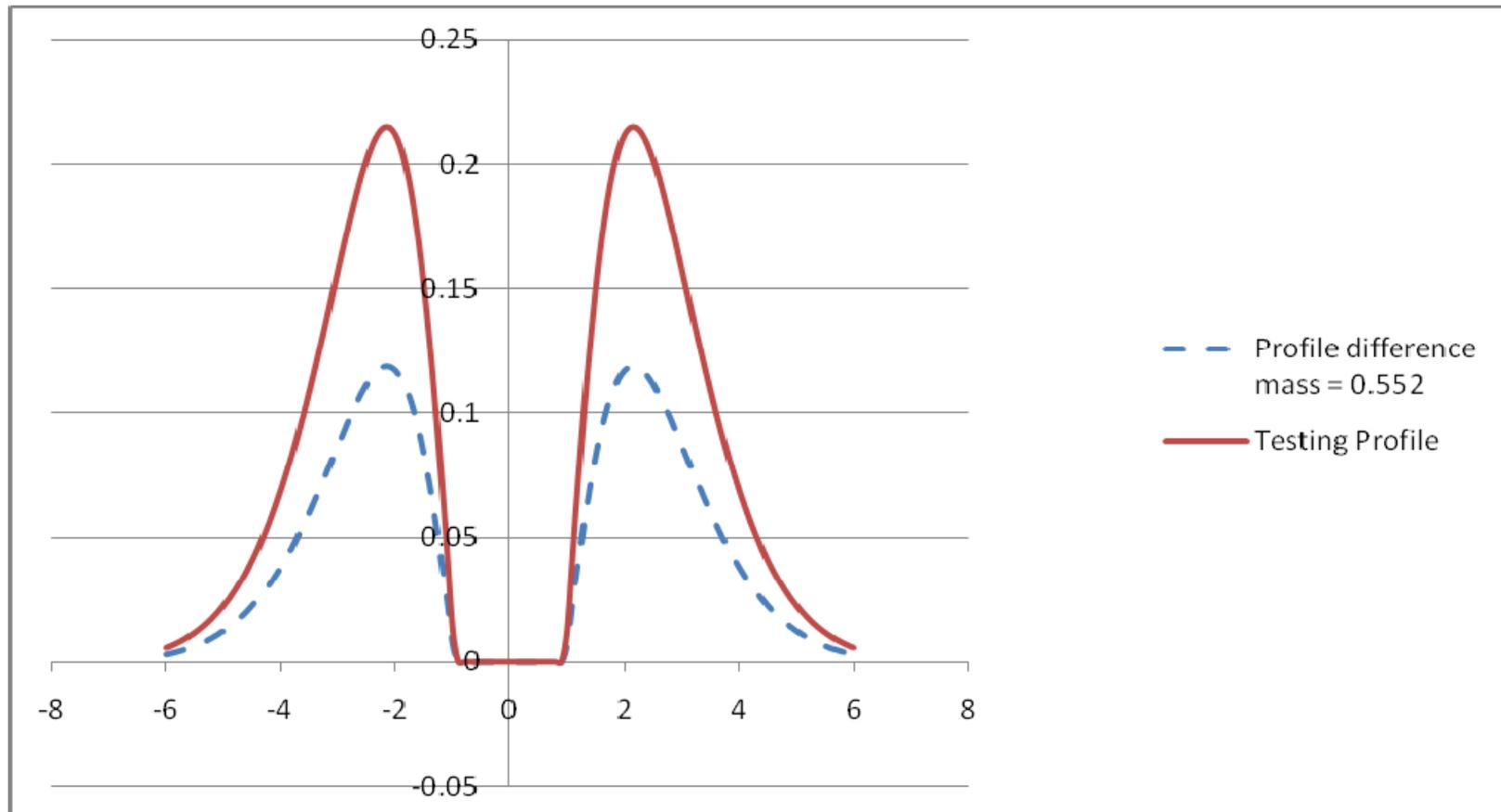


Example of Using an Operational Profile for Reuse Testing (cont)

- Need additional testing in regions more likely in the new profile than in the old one
- The profile difference defines the needed test cases
 - $P_d(x) = \text{if } g_2(x) > g_1(x) \text{ then } g_2(x) - g_1(x) \text{ else } 0$
 - Must be scaled if reliability goals differ in the two environments
 - Must be normalized to become a probability distribution



Derived Testing Profile

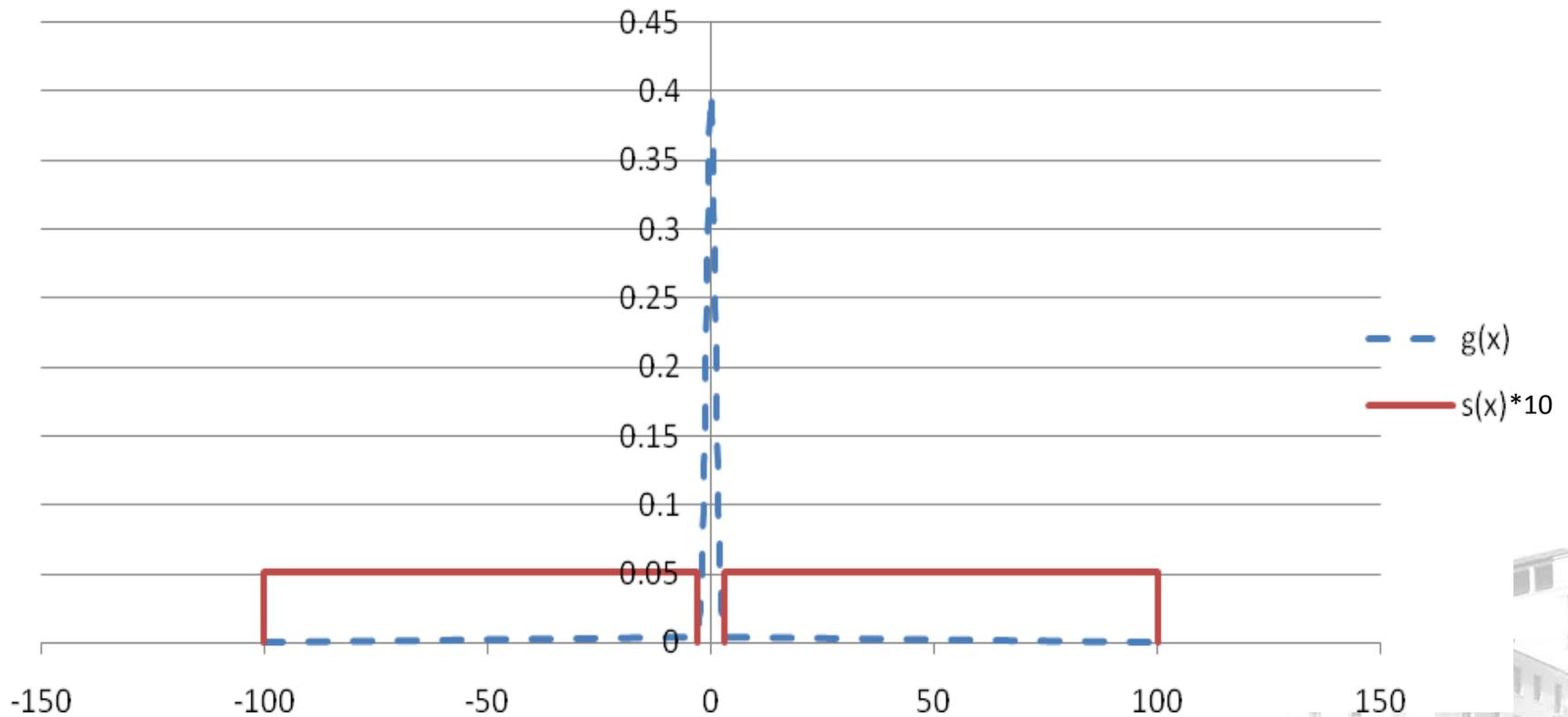


Example of Using an Operational Profile for Reuse Testing (cont)

- 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



Defense Acquisition in Transition
6TH ANNUAL ACQUISITION RESEARCH SYMPOSIUM

May 12-14, 2009
Monterey, CA

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]

