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

Prepared for: Naval Postgraduate School, Monterey, California 93943

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

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.acquistionresearch.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

Abstract[1]

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 g1 represents the operational profile for the initial deployment of a hypothetical reusable component and g2 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, g1 and g2 are normal distributions; g1 has a standard deviation of 1.0, and g2 has a standard deviation of 2.0.

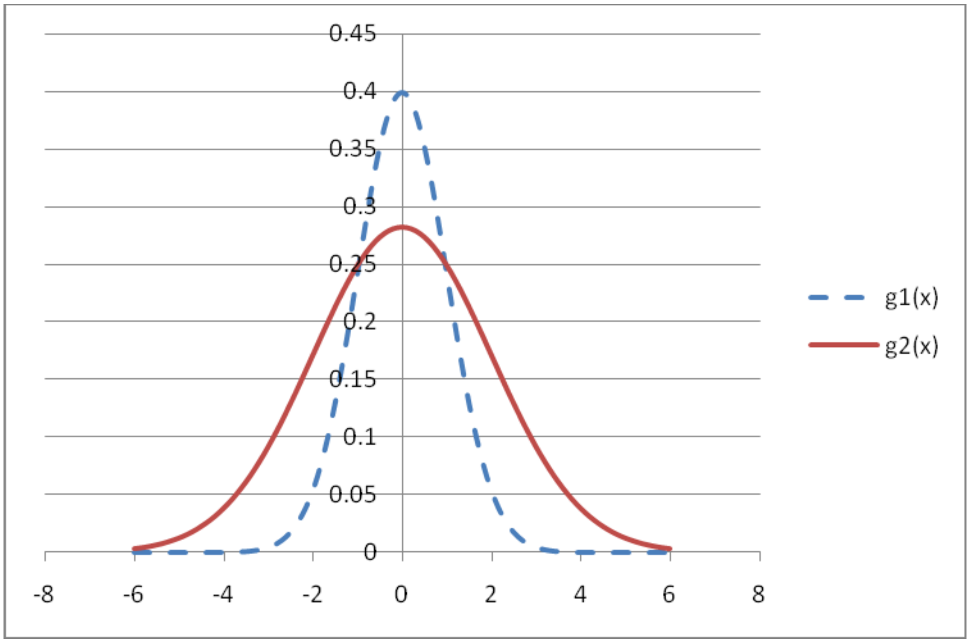

Figure 1. Operational Profiles for Two Different Deployment Environments

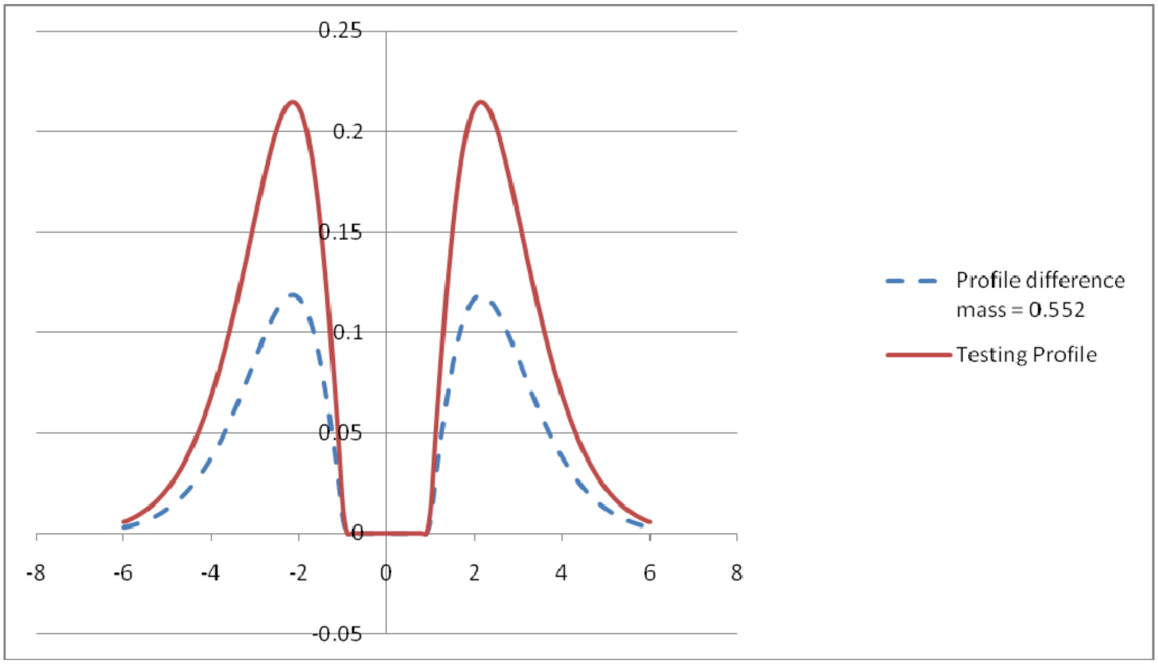

Figure 2. The Derived Testing Profile

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 g1 > g2, and it is equal to the difference g2 – g1 everywhere else. The rationale for these choices is the following:

The region where g1 > g2 has already been adequately tested since the expected number of samples from this region that were checked during prior testing using the profile g1 exceed the expected number of samples from the same region that would be required in testing under the execution profile g2, 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 g1 in this slice represents the expected number of test cases that were run in the previous round of testing governed by the profile g1. The area under g2 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 N1 be the number of test cases that were needed from profile p1 for the first deployment environment and N2 be the number of cases from a different profile p2, needed for the second environment. Then, in the general case, the profile difference is zero where N1*p1 > N2*p2 and is equal to (N2*p2 – p1* N1)/(N1 +N2) 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.

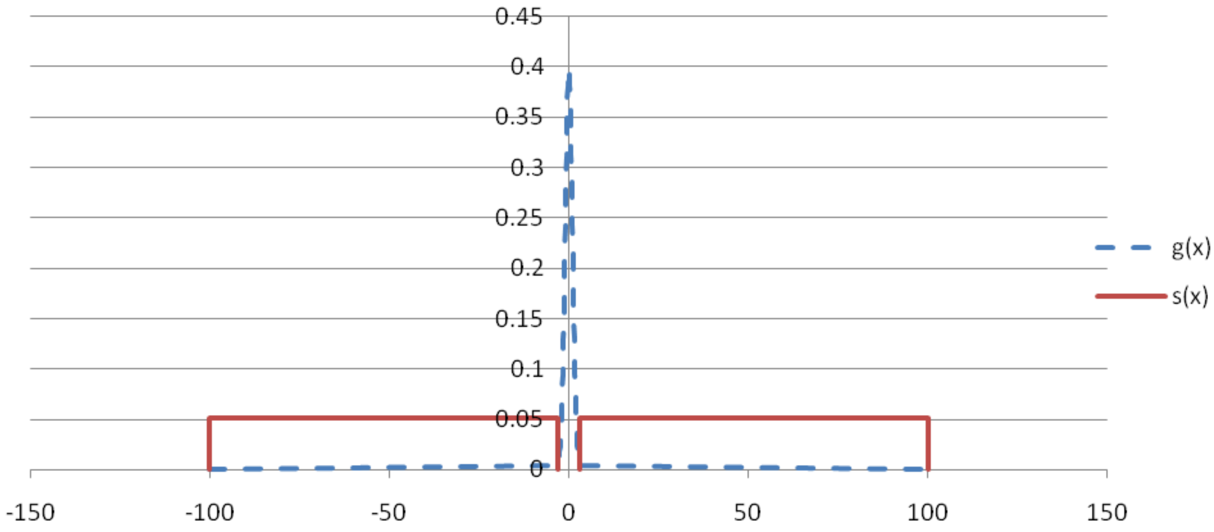

Figure 3. A Stress-testing Profile, s(x), Compared to an Operational Profile, g(x)

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 es according to the relation Ts = (kes)log2(kes), as explained in Berzins (2008). Ts 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). RE^2: 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