NPS-AM-13-104

ACQUISITION RESEARCH PROGRAM SPONSORED REPORT SERIES

Improving Acquisition Process Efficiency via Risk-Based Testing

25 September 2013

Dr. Valdis Berzins, Professor

Naval Postgraduate School

Approved for public release; distribution is unlimited.

Prepared for the Naval Postgraduate School, Monterey, CA 93943.

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

The research presented in this report was supported by the Acquisition Research Program of the Graduate School of Business & Public Policy at the Naval Postgraduate School.

To request defense acquisition research, to become a research sponsor, or to print additional copies of reports, please contact any of the staff listed on the Acquisition Research Program website (www.acquisitionresearch.net).

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

Abstract

This paper proposes the idea that affordable quality assurance can be achieved by focusing testing effort based on severity of risk exposure. This implies that testing effort should be allocated based on results of a suitable risk analysis: Parts of the software that could cause high-severity mishaps should be subjected to more intensive testing to control costs of future software failures, and those that cannot should be subjected to less intensive testing to safely reduce testing cost. We explain the concept of software slicing and detail how it can be used to link risk analysis and testing. We also identify some existing software tools that can be used to do software slicing and evaluate their readiness to support the proposed process improvement.

Keywords: Software Testing, Safe Cost Reduction, Software Slicing, Software Reuse, Open Architecture

THIS PAGE INTENTIONALLY LEFT BLANK

About the Author

Valdis Berzins is a professor of computer science at the Naval Postgraduate School. His research interests include software engineering, software architecture, reliability, 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.

Graduate School of Business & Public Policy
Naval Postgraduate School
Monterey, CA 93943-5000
Tel: (831) 656-2610
Fax: (831) 656-3407
E-mail: berzins@nps.edu

THIS PAGE INTENTIONALLY LEFT BLANK

NPS-AM-13-104

ACQUISITION RESEARCH PROGRAM SPONSORED REPORT SERIES

Improving Acquisition Process Efficiency via Risk-Based Testing

25 September 2013

Dr. Valdis Berzins, Professor

Naval Postgraduate School

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

Improving Acquisition Process Efficiency via Risk-Based Testing

Introduction

Our objective is to affordably achieve acceptable levels of operational risk by setting testing levels based on risk analysis, and to decrease the cost of testing a reusable software component for each new context via improvements in process efficiency.

Previous work on system risk assessment in the context of safety certification has combined severity of potential failures with estimated probabilities of occurrence to gauge risks associated with system operation, but this work is directly applicable only at the whole-system level. This is due to the fact that software-related operational hazards are mostly associated with the physical parts of the system, which are only indirectly affected by the software.

We are investigating methods to determine how much testing and what other risk-mitigation measures, if any, should be applied to each software component in an open architecture. However, the relationship between the individual embedded software components and the associated external effects is currently difficult to determine due to the size and complexity of practical software components and lack of automated decision support. We are investigating how to apply software slicing and related dependency analysis techniques to solve this challenge. To our knowledge, this is the first attempt to apply this new approach to address this system-of-systems challenge.

The intended principles of operation for the proposed software risk-mitigation approach are as follows:

1. Perform a conventional whole-system operational risk analysis, using approaches adapted from safety procedures certification such as those identified in the Department of Defense's (DoD; 2012) *Standard Practice: System Safety*. The result of this step is a list of potential mishap types, associated with and ranked by their risk levels. We extend the definition of mishaps in this context to include various aspects of mission failures that could be induced by system failures, in addition to the types of mishaps traditionally considered in a safety certification.
2. Perform a system-level dependency trace to identify which subsystems affect each type of mishap listed in Step 1 and which software services

affect each of those subsystems. This part of the process involves skilled human effort, using well-established risk assessment methods such as fault tree analysis, supported by relevant historical data when it is available, and otherwise using subjective human judgment guided by risk matrices (see DoD, 2012).

3. Perform a software dependency analysis using software slicing, to identify which software modules affect each of the software services. This part of the process is new, along with Steps 4–6. Our study is aimed at refining these steps and assessing their feasibility and effectiveness in practice. Step 3 is completely automatable in theory, and some tools for slicing currently exist. Assessing suitability of existing tools to carry out this step is one of our goals, and some results in that direction are reported here.
4. Using the mishap list from Step 1 and the dependency relations from Steps 2–3, identify the set of potential mishaps that can be affected by each software module.
5. Associate the maximum risk level of the set of mishaps identified in Step 4 with the corresponding software modules.
6. Use the risk level derived in Step 5 to determine the level of testing and possible levels of additional risk mitigations to be associated with each software module.

Steps 4, 5, and the first half of 6 are readily automatable, although existing software tools do not do so yet. Doing this is relatively easy if the results of Step 3 can be captured in a form suitable for further automated processing. This depends on the nature of the application programming interface (API), if any, provided by the software slicing tools.

The motivating acquisition goal is to reduce total system ownership costs, particularly the cost associated with quality assurance per system update. The overall impact of adopting this approach would be more effective test and evaluation. If the worst-case risks associated with each component are known, then they can provide a principled and systematic basis for determining how much testing each component needs. This approach will put the informal guideline to “test the most critical components the most thoroughly” on a sound quantitative basis, by providing an objective means for calculating how many test cases are needed for a given component, based on the associated risk levels. Some of the building blocks needed for this calculation can be found in Berzins (2008).

The testing approach described previously seeks to limit operational risks by running more test cases on the modules whose failure would lead to the most severe consequences. This is a statistical procedure aimed at reducing the likelihood of sampling errors—that is, reducing the likelihood that the finite set of test cases actually run may pass successfully by pure chance, even though the failure rate of the component is actually unacceptably high. Cost is driven by two components: the effort associated with testing and the cost associated with operational failures due to software faults that remain undiscovered or uncorrected. The second component of the cost is difficult to determine a priori, because it is driven by unknown attributes of the system and its environment. The risk analysis provides an approximate characterization of the second cost component based on best available information. The testing level is set so that we run the minimum number of test cases needed to establish sufficient statistical confidence that the second component of the cost will be acceptably low.

Our hypothesis is that this is the best practical approach for reducing this cost component. Traditional quantitative optimization methods are not applicable in this case because we do not have accurate and tractable models of the second component of the cost, and obtaining such models is unlikely to be technically or economically feasible. The proposed method is expected to produce better results than unguided, subjective human judgment or educated guesswork because it is systematic and takes all available data into account.

Our long-term scientific goal is to enable new acquisition processes that achieve system dependability but require less quality assurance effort per system update or per each potential configuration of the system. This includes avoiding repetition of test procedures when they do not provide new information, as well as seeking quality assurance methods that enable a single analysis to simultaneously certify a set of many different possible system configurations (Berzins, Rodriguez, & Wessman, 2007). When fully developed, such methods will under some conditions eliminate the need for integration testing after reconfiguration and should eventually enable plug-and-fight capabilities if the range of needed configurations can be characterized in advance and the proposed pre-certification procedures can be affordably carried out for a set of plug-compatible components spanning the expected range of potential configurations. The content of this report indirectly contributes to this long-term goal. As outlined in Berzins (2008), software slicing can determine some conditions under which it is safe not to retest a component after a system upgrade. That procedure also depends on the availability of reliable slicing tools with APIs suitable for supporting post-processing of the computed slices.

This report focuses on enabling a principled and computer-aided connection between a system-wide risk analysis, a risk budget for each individual software

module or service, and the test cases needed to establish an appropriate level of confidence that the implementation of the module or service meets its risk budget. A risk budget for a system is the maximum acceptable probability that the system will fail within a given time period. Wall clock time is appropriate in most cases because large-scale systems have physical realizations that work continuously whenever the system is in operation. However, software components are different because they operate in discrete steps, in response to discrete signals that may or may not occur at regular time intervals. The natural formulation of risk budgets for software modules and services is the probability of their failure per each execution of the module or service. The expected number of software component executions per mission, or per system lifetime, must then be estimated to connect software testing considerations to the system-level risk analysis.

This report examines Step 3 of the risk-mitigation approach in detail, because this is the part of the process that benefits the most from automated decision support. Software slicing is a kind of dependency analysis that can be carried out by software tools that operate on the source code of the software components to be tested based on associated system risks. Algorithms to do this are known and in theory should be fast enough to be affordably applied to practical-sized (large) software systems.

The section Software Slicing and Related Previous Work briefly explains software slicing, including relevant previous work and some examples that illustrate what it does. The next section, Existing Slicing Tools, describes some existing tools for software slicing. The section Analysis of the Indus-Kaveri Slicing Tools describes our analysis of one such tool, the Indus-Kaveri slicer for Java and outlines the results of the analysis. Finally, the Conclusions section presents our conclusions regarding the slicing tool and implications with respect to risk-based testing.

Software Slicing and Related Previous Work

Software slicing is a kind of dependency analysis that automatically reduces a program to a smaller form that produces the same behavior with respect to some observation points, which typically show only part of the entire behavior of the program. The concept was originally developed by Mark Weiser in the 1980s.

The original definition of a program slice as introduced by Weiser is based on the deletion of program statements: “A slice is an executable subset of program statements that preserves the original behavior of the program with respect to a subset of variables of interest and at a given program point,” as paraphrased by De Lucia (2001, p. 142).

In order to understand program slicing, we need to understand what a slicing criterion is. According to Weiser (1984), a slicing criterion of a program is a tuple <*p*,

V>, where *p* is a statement in the original program and *V* is a subset of variables in the program.

A slice of a program based on the slicing criterion C = <*p*, *V*> is an executable subset of statements from the original program that reproduces the behavior of the original program at point *p*, with regards to the set of variables in *V* (De Lucia, 2001). In other words, the slice of a program can be obtained by deleting statements from the original program that do not affect the values of the variables in *V* at program point *p*. This way, the original program and its slice will produce the same values for the set of variables in *V* at program point *p*.

We can have more than one slice of a program with respect to a given slicing criterion: In fact, the entire program is considered a slice. The smaller the slice, the better it is. Unfortunately, finding a minimal slice is an unsolvable problem. Weiser (1984) has proven that "there does not exist an algorithm to find statement-minimal slices for arbitrary programs" (p. 353).

A slice contains all of the statements that affect the part of the behavior of the program that is visible from the point of view of the slicing criterion. A minimal slice contains only statements that affect the program's visible behavior, and in a small slice, almost all of the statements in the slice affect its visible behavior (with respect to the chosen slicing criterion). In the context of risk-based testing, we assume that a given low-level software component can affect a top-level software service if and only if any part of the low-level component is contained in the slice of the entire software system with respect to a slicing criterion corresponding to the result computed by the top-level software service. This is precisely the information needed in Step 3 of the risk-mitigation approach outlined in the Introduction.

This is a safe approximation: It is exactly correct if the slice is statement minimal. If the slice is small but not minimal, this criterion will still find all of the low-level components that can affect the top-level service, although it might also find some others that cannot affect the service. Thus the approximation will never miss any critical subcomponents, although it could sometimes call out a subcomponent as critical when in fact it is not. This would subject that subcomponent to extra testing, which would be safe but wasteful. This is why smaller slices are better in our context, and why relative size of slices produced is included in our assessment criteria for slicing tools.

If two different versions of a program have the same slice with respect to a slicing criterion, then they must have the same behavior, for those aspects visible through the slicing criterion. This property is the basis for reduction of regression testing via slicing. Practical application for reduction of regression testing requires computing the slices of each software service in the current and next versions of the system and comparing them. Since slices can be large and many are needed, this

requires reliable slicing tools and automated ability to compare slices to determine whether they represent the same program.

To better illustrate the concept of slicing, consider the sample code in Figure 1.

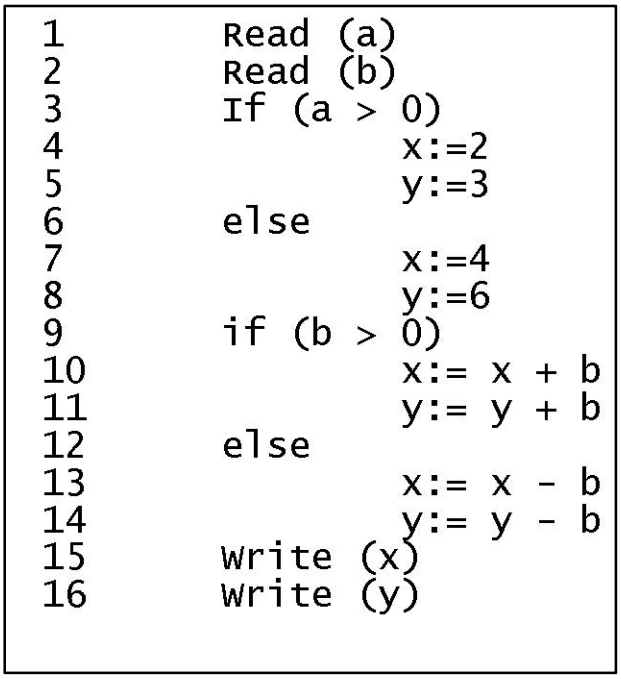

Figure 1. Sample Program P1
(Z'ghidi, 2013)

A static slice of the program in Figure 1 with respect to variable *x* at line 15 would result in all program statements that might affect the value of *x* up to line 15 of the program, which can be obtained by deleting irrelevant statements from the original program that do not influence the value of variable *x* at line 15. Figure 2(a) shows a slice of program P1 in Figure 1 based on the slicing criterion C1 = <line 15, *x*>. Figure 2(b) shows another slice of program P1 based on the slicing criterion C2 = <line 16, *y*>. These slices show the parts of the program that contribute to each of the two output variables, *x* and *y*. Although the examples may appear to depend on common-sense understanding of what the programs are doing, these slices can be computed automatically based on analysis of control flow and data flow. These are processes that have been routinely used in optimizing compilers for decades.

Program slicing 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 (Cimitile, De Lucia, & Munro, 1996; Gallagher, 1991), change merging (Berzins & Dampier, 1996; Horwitz, Prins, & Reps, 1989), and software metrics (Bieman & Ott, 1994; Lakhotia, 1993). More detailed surveys of previous work on slicing can be found in Binkley and Harman (2004). These applications have mostly been demonstrated in research labs using home-grown tools.

Our recent work (Berzins, 2012; Berzins, Lim, & Ben Kahia, 2011) has identified potential applications of software slicing and related dependency analysis methods to setting testing levels of subsystems based on global (system-wide) operational risk. These analyses bridge the gap between system-wide risk analysis and risk exposure levels due to potential failures of lower level subsystems. It has also outlined the initial concepts for risk-based resource allocation in a planned series of system upgrades. The current report is focused on developing the details of these ideas.

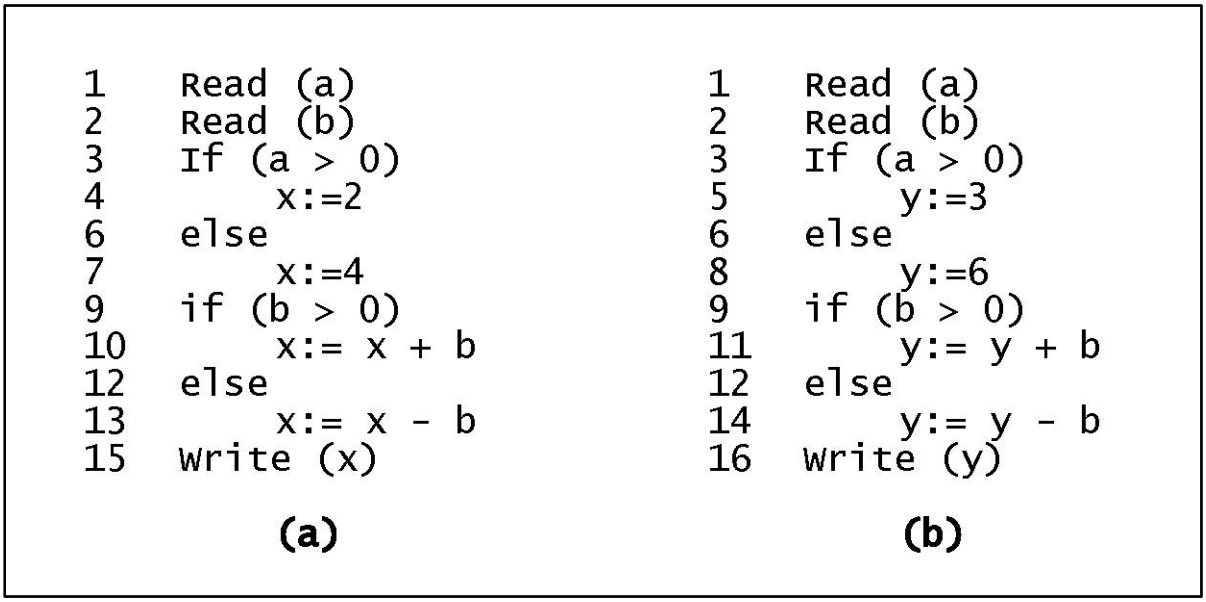

Figure 2. Static Slice of P1 Based on C1 (a) and on C2 (b)
(Z'ghidi, 2013)

We note that manual determination of the software dependencies needed for risk-based testing is very labor intensive and error prone if done manually, especially on the scale of practical military software systems. In such systems, the dependency chains can be long and indirect, involving mixtures of data flow and control flow whose paths may involve code in widely separated parts of the system, some of which may operate at different times than the affected top-level service. In particular, manual inspection of a program call graph is not sufficient to find all of the relevant dependencies, because data flow links through state variables of classes and other

repositories of state information (such as databases) are left out by such a simplified approach. This implies that reliable software slicing tools are a necessary part of practical risk-based testing, if we need the results to be dependable and cost effective.

Existing Slicing Tools

Despite its potential benefits, slicing has not been widely used in industry. This may be partly due to lack of appropriate commercial tools and partly due to lack of familiarity with the possible benefits. Prior work at the Naval Postgraduate School (NPS; Lim & Ben Kahia, 2011) has identified several slicing tools that might be suitable, including Code Surfer, a commercial slicing tool for C/C++ developed by Gramma Tech; Jslice, a slicing tool for Java; and Indus-Kaveri, another slicing tool for Java. Lim and Ben Kahia (2011), doing a prior investigation of these tools, had licensing issues with Code Surfer and documentation problems with Jslice, and decided to analyze Indus-Kaveri. Preliminary results were reported in Berzins et al. (2011) and Berzins (2012). This report completes the assessment of the Indus-Kaveri slicing tool.

Analysis of the Indus-Kaveri Slicing Tool

Indus is a static analysis tool that can be used to perform static slicing of programs written in Java. Kaveri is an Eclipse plug-in that uses the Indus program slicer to compute slices of Java programs and then displays the results visually as a set of highlighted statements in the editor (Jayaraman, 2008). Eclipse is an open-source integrated development environment (IDE) that was designed to be extensible via independently developed plug-ins, with the goal of integrating advances in software analysis tools developed by different research and development teams.

Kaveri acts as a user interface that can be used to simplify program understanding and program debugging. It visually highlights the set of relevant statements with respect to a given slicing criterion, which helps the programmer to focus on statements that may affect the value of a variable of interest, such as the result of a failed test case, and ignore other irrelevant statements.

Despite the usefulness of Kaveri, the tool cannot be used for safe reduction of regression testing without being modified. For risk-based testing, we need to find which software services are included in the computed slice. In order to decide whether a newer version of a program needs to be retested, we need to compare a slice of the original program with the corresponding slice of the new program. In either case, if the output of the slicer is simply a set of highlighted statements on a screen display, these additional steps would be labor intensive, and it would not be

possible to automate the process for large-scale applications. In both cases, we would need to save the output of the slicer into a file for further processing (Lim & Ben Kahia, 2011). Unfortunately Kaveri does not provide such a capability.

Assuming that the output issues for the tool could be fixed, we sought to check whether Indus-Kaveri performs correctly by using simple test cases, each of which focused on particular features of the language being analyzed by the tool, in this case Java. Details of these test cases can be found in Z'ghibi (2013). The results of the testing are summarized in Table 1.

Table 1. Kaveri Evaluation Results
(Z'ghibi, 2013)

Tested Construct	Slice Correctness	Precise	Issues
Assignment Statements	Correct	Yes	None relevant
Loops	Incorrect	Not Applicable	Does not select loop closing brackets
If Conditions	Incorrect	Not Applicable	Does not select *else* statements and condition closing brackets
Switch Conditions	Incorrect	Not Applicable	Does not include *case* conditions
Arrays	Correct	No	Treats all elements of the array as a single object
Pointers	Correct	No	Overestimated slice in the presence of aliasing
Object Attributes	Correct	Yes	None relevant
Inheritance	Correct	Yes	None relevant
Method Overloading	Correct	Yes	None relevant
Method Overriding	Correct	No	Cannot determine which method is being called
Exceptions	Incorrect	Not Applicable	Does not include closing brackets and exception keywords (try/catch)
External Classes	Incorrect	Not Applicable	Does not highlight relevant statements in the external class

Some of the discovered limitations, such as not highlighting variable declarations, can be overlooked in the current context; other limitations, however, make the tool produce an incorrect slice and, therefore, need to be addressed before we can use Kaveri in risk-based testing or the safe reduction of regression testing.

Despite being able to compute correct and precise slices for some test cases, Kaveri has some serious limitations. First, the tool is only able to highlight relevant statements of the slice and does not allow printing the slice to a text file. This makes it hard to automatically process the computed slices. Second, Kaveri does not select the closing brackets that indicate the end of classes, methods, loops, conditional statements, and exception blocks, which can alter the meaning of the computed slice and thus invalidate a risk analysis based on it. Third, Kaveri is not able to slice through external classes and can only highlight relevant statements in the file containing the slicing criterion. This is a serious limitation because almost all practical applications are large enough to occupy multiple files (typically each class is in a separate file).

A method for overcoming these limitations was developed (Z'ghibi, 2013). This method invokes the Indus Java program slicer directly using the command-line interface, rather than going through the graphical interface provided by Kaveri. Indus is the underlying computation engine used by the Kaveri interface. This slicer operates on Jimple, which is an intermediate-level representation of Java programs, between byte code and Java statements. Before the slicer computes the slice of a Java program with respect to a given criterion, the code of the program is converted to Jimple. The slicer computes the slice, then saves the result represented in a Jimple format to a file (Jayaraman, 2008). Our investigation developed some code that processes this output to produce a representation of the Java slice. The results of this investigation are summarized in Table 2.

Table 2. Indus Evaluation Results
(Z'ghibi, 2013)

Tested Construct	Slice Correctness	Precise	Issues
Assignment Statements	Correct	Yes	None relevant
Loops	Correct	No	Includes non-relevant loop statements
If Conditions	Correct	No	Includes some irrelevant statements
Switch Conditions	Incorrect	Not Applicable	Does not include *case* conditions
Arrays	Correct	No	Treats all elements of the array as a single object
Pointers	Correct	No	Overestimated slice in the presence of aliasing
Object Attributes	Correct	Yes	None relevant
Inheritance	Correct	Yes	None relevant
Method Overloading	Correct	Yes	None relevant
Method Overriding	Correct	No	Cannot determine which method is being called
Exceptions	Correct	No	Includes all statements in the exception checking block
External Classes	Correct	Yes	None relevant

This shows that most of the problems with Kaveri can be overcome by using Indus directly. The prototype post-processor developed in the study is not complete, in that it leaves one of the limitations unresolved and in that it depends on some restrictions on the syntax of the source program that were introduced to simplify the implementation.

We conclude that the modified tool developed in this study can be used to demonstrate feasibility of the risk-based testing method outlined in the Introduction for experimental case studies that work around its known limitations. It cannot be used in large-scale applications without additional refinement and development of the post-processing software to remove its remaining limitations.

Conclusions

We found that the combination of Indus and Kaveri is not reliable in the sense that it sometimes produces incorrect slices. Much of the problem can be attributed to weaknesses and faults in the graphical interface provided by Kaveri, as indicated by the experimental direct use of the the Indus Java Program Slicer through its command-line interface.

Remaining difficulties are related to the fact that Indus works on Jimple, an intermediate representation close to Java bytecode, rather than directly on the Java source. The Jimple-level slices produced by Indus appear to be correct, but mapping the output back to the Java source has problems, particularly for the mappings used by Kaveri. Experimental code developed in this study shows that it can be done better. However, there was not enough time in the study to implement a product quality mapping to Java, and further engineering is required to produce a post-processor that would enable Indus to be used for risk-based testing of practical systems. Practical application of the proposed method requires either finding a different and better commercial slicing tool or further development of a product quality post-processing tool.

References

Agrawal, H., DeMillo, R., & Spafford, E. (1993). Debugging with dynamic slicing and backtracking. *Software Practice and Experience*, *23*(6), 589–616.

Berzins, V. (2008). Which unchanged components to retest after a technology upgrade. In *Proceedings of the Fifth Annual Acquisition Research Symposium* (pp.142–153). Retrieved from http://www.acquisitionresearch.net

Berzins, V. (2012). Certifying tools for test reduction in open architecture. In *Proceedings of the Ninth Annual Acquisition Research Symposium* (pp. 185–194). Retrieved from http://www.acquisitionresearch.net/

Berzins, V., & Dampier D. (1996). Software merge: Combining changes to decompositions. *Journal of Systems Integration*, *6*(1–2), 135–150.

Berzins, V., Lim, P., & Ben Kahia, M. (2011). Test reduction in open architecture via dependency analysis. In *Proceedings of the Eighth Annual Acquisition Research Symposium* (pp. 333–344). Retrieved from http://www.acquisitionresearch.net/

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). Retrieved from http://www.acquisitionresearch.net/

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. *Information and Software Technology*, *40*(11–12), 583–594.

Binkley, D. W., & Harman, M. (2004). A survey of empirical results on program slicing. *Advances in Computers*, *62*, 105–178.

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. *Journal of Software Maintenance: Research and Practice*, *8*(3), 145–178.

De Lucia, A. (2001). Program slicing: Methods and applications. In *Proceedings of the First IEEE International Workshop on Source Code Analysis and Manipulation* (pp. 142–149). Florence, Italy: IEEE Computer Society.

De Lucia, A., Fasolino, A., & Munro, M. (1996). Understanding function behaviours through program slicing. In *Proceedings of the Fourth IEEE Workshop on Program Comprehension* (pp. 9–18). Berlin, Germany: IEEE Computer Society Press.

Department of Defense (DoD). (2012, May 11). *Department of Defense standard practice: System safety* (MIL-STD-882E). Retrieved from http://www.acquisitionresearch.net/

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). Orlando, FL: 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* (pp. 138–147). Florence, Italy: 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.

Jayaraman, G. (2008). *Indus-Kaveri*. Retrieved from http://projects.cis.ksu.edu/gf/download/docmanfileversion/100/1458/Kaveri-0.6.0B.pdf

Lakhotia, A. (1993). Rule-based approach to computing module cohesion. In *Proceedings of the 15th International Conference on Software Engineering* (pp. 34–44). Baltimore, MD: ACM/IEEE.

Lim, P., & Ben Kahia, M. (2011, June). *Suitability of commercial slicing tools for safe reduction of the testing effort* (Master's thesis, Naval Postgraduate School). Retrieved from http://www.acquisitionresearch.net/

Lyle J., & Weiser, M. (1987). Automatic program bug location by program slicing. In *Proceedings of the Second International Conference on Computers and Applications* (pp. 877–882). Beijing, China: IEEE Computer Society Press.

Weiser, M. (1984, July). Program slicing. *IEEE Transactions on Software Engineering*, *SE-10*(4), 352–357.

Z'ghidi, A. (2013, June). *Evaluation of existing slicing tools and their usefulness to safely reduce regression testing* (Master's thesis, Naval Postgraduate School). Retrieved from http://www.acquisitionresearch.net/

THIS PAGE INTENTIONALLY LEFT BLANK

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

www.acquisitionresearch.net