NPS-AM-11-C8P10R02-043

EXCERPT FROM THE

PROCEEDINGS

OF THE

EIGHTH ANNUAL ACQUISITION RESEARCH SYMPOSIUM WEDNESDAY SESSIONS VOLUME I

Test Reduction in Open Architecture via Dependency Analysis

Valdis Berzins, Peter Lim, and Mohsen Ben Kahia, NPS

Published: 30 April 2011

Prepared for the Naval Postgraduate School, Monterey, California 93943

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

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.net

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

Preface & Acknowledgements

During his internship with the Graduate School of Business & Public Policy in June 2010, U.S. Air Force Academy Cadet Chase Lane surveyed the activities of the Naval Postgraduate School's Acquisition Research Program in its first seven years. The sheer volume of research products—almost 600 published papers (e.g., technical reports, journal articles, theses)—indicates the extent to which the depth and breadth of acquisition research has increased during these years. Over 300 authors contributed to these works, which means that the pool of those who have had significant intellectual engagement with acquisition issues has increased substantially. The broad range of research topics includes acquisition reform, defense industry, fielding, contracting, interoperability, organizational behavior, risk management, cost estimating, and many others. Approaches range from conceptual and exploratory studies to develop propositions about various aspects of acquisition, to applied and statistical analyses to test specific hypotheses. Methodologies include case studies, modeling, surveys, and experiments. On the whole, such findings make us both grateful for the ARP's progress to date, and hopeful that this progress in research will lead to substantive improvements in the DoD's acquisition outcomes.

As pragmatists, we of course recognize that such change can only occur to the extent that the potential knowledge wrapped up in these products is put to use and tested to determine its value. We take seriously the pernicious effects of the so-called "theory–practice" gap, which would separate the acquisition scholar from the acquisition practitioner, and relegate the scholar's work to mere academic "shelfware." Some design features of our program that we believe help avoid these effects include the following: connecting researchers with practitioners on specific projects; requiring researchers to brief sponsors on project findings as a condition of funding award; "pushing" potentially high-impact research reports (e.g., via overnight shipping) to selected practitioners and policy-makers; and most notably, sponsoring this symposium, which we craft intentionally as an opportunity for fruitful, lasting connections between scholars and practitioners.

A former Defense Acquisition Executive, responding to a comment that academic research was not generally useful in acquisition practice, opined, "That's not their [the academics'] problem—it's ours [the practitioners']. They can only perform research; it's up to us to use it." While we certainly agree with this sentiment, we also recognize that any research, however theoretical, must point to some termination in action; academics have a responsibility to make their work intelligible to practitioners. Thus we continue to seek projects that both comport with solid standards of scholarship, and address relevant acquisition issues. These years of experience have shown us the difficulty in attempting to balance these two objectives, but we are convinced that the attempt is absolutely essential if any real improvement is to be realized.

We gratefully acknowledge the ongoing support and leadership of our sponsors, whose foresight and vision have assured the continuing success of the Acquisition Research Program:

- Office of the Under Secretary of Defense (Acquisition, Technology & Logistics)
- Program Executive Officer SHIPS
- Commander, Naval Sea Systems Command
- Army Contracting Command, U.S. Army Materiel Command
- Program Manager, Airborne, Maritime and Fixed Station Joint Tactical Radio System

- Program Executive Officer Integrated Warfare Systems
- Office of the Assistant Secretary of the Air Force (Acquisition)
- Office of the Assistant Secretary of the Army (Acquisition, Logistics, & Technology)
- Deputy Assistant Secretary of the Navy (Acquisition & Logistics Management)
- Director, Strategic Systems Programs Office
- Deputy Director, Acquisition Career Management, US Army
- Defense Business Systems Acquisition Executive, Business Transformation Agency
- Office of Procurement and Assistance Management Headquarters, Department of Energy

We also thank the Naval Postgraduate School Foundation and acknowledge its generous contributions in support of this Symposium.

James B. Greene, Jr.
Rear Admiral, U.S. Navy (Ret.)

Keith F. Snider, PhD
Associate Professor

Panel 10 – New Testing Protocols for the Open Architecture Era

Wednesday, May 11, 2011	
3:30 p.m. – 5:00 p.m.	**Chair: Captain Brian Gannon, USN**, Program Manager, Naval Open Architecture, PEO IWS ***Modeling Complex System Testing: Characterizing Test Coverage to Improve Information Return*** Karl Pfeiffer, Valery Kanevsky, and Thomas Housel, NPS ***Test Reduction in Open Architecture via Dependency Analysis*** Valdis Berzins, Peter Lim, and Mohsen Ben Kahia, NPS ***Utilizing Statistical Inference to Guide Expectations and Test Structuring During Operational Testing and Evaluation*** Joy Brathwaite, Georgia Institute of Technology, Alton Wallace and Robert Holcomb, Institute for Defense Analyses

Captain Brian Gannon—CAPT Gannon was born in Chicago, Illinois and received a commission in 1985 through the Naval Reserve Officer Training Corps program at the Illinois Institute of Technology. His formal education includes a Bachelor of Science in Mechanical Engineering from the Illinois Institute of Technology, a Master of Science in Astronautical Engineering from the Naval Postgraduate School, and a Master of Business Administration from the University of Phoenix.

His service tours include Electronics Readiness Officer, ASW Officer and CIC Officer onboard *USS Gary* (FFG-51) from 1986 to 1989; Combat Systems Instructor at the Surface Warfare Officer's School in Coronado, CA, from 1989 to 1992; Student in the Space Systems Engineering curriculum at the Naval Postgraduate School from 1992 to 1994; Aegis Project Officer at the Port Hueneme Division, Naval Surface Warfare Center from 1994 to 1998; AEGIS LEAP Intercept (ALI) Project Officer in the Navy Theater Wide Program Office (PMS 452) from 1998 to 2002; TBMD Section Head in the Aegis Combat System Engineering Program Office (PMS 400B) from 2002 to 2003; Combat Systems Officer on the Fleet Maintenance staff for Commander, United States Pacific Fleet from 2003 to 2005; Technical Representative for Surface Naval Weapons (PEO IWS 3.0) and Aegis Ballistic Missile Defense (PD 452) portfolio of programs at Raytheon Missile Systems in Tucson, AZ.

CAPT Gannon assumed his present duties as Major Program Manager Future Combat Systems and Open Architecture (PEO IWS 7.0) in October 2008.

Captain Gannon's personal awards include the Meritorious Service Medal (four awards), Navy Commendation Medal and the Navy Achievement Medal in addition to various service awards. He is married to the former Jean Raup of Alexandria, VA. He has three children: Brittany (18), Timothy (15), and Christopher (13).

Test Reduction in Open Architecture via Dependency Analysis

Valdis Berzins—Professor, Computer Science, 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. [berzins@nps.edu]

Peter Lim—Peter Lim received a BTech in Electronics Engineering and MEng degrees in Electrical and Computing from the National University of Singapore. He is currently pursuing an MS in Software Engineering at the Naval Postgraduate School. His research area is in software engineering processes and testing tools, and his research interests include computer vision and microcontroller implementation. [plim@nps.edu]

Mohsen Ben Kahia—Mohsen Ben Kahia received a BS in computer sciences from the Tunisian Naval Academy. He is currently pursuing an MS in software engineering at the Naval Postgraduate School. He was involved as a member of the software research center of the Tunisian Navy. His research areas in software engineering include software maintenance and testing. In parallel he is pursuing research on improving distributed system efficiency. [mbenkahi@nps.edu]

Abstract

In the Verification and Validation (V&V) phase, whenever there is a newer release of a given program, test engineers need to re-conduct all the tests performed on the previous program release—a costly process known as regression testing. By using the concept of program slicing, this project aims to be more effective in managing costly human effort by selectively retesting the subset of the newer program release that is critical and necessary. Program Slicing is an abstraction and program analysis technique based on the principle of eliminating/deleting parts or subsets of the program statements that are irrelevant to a given slicing criterion (Weiser, 1984). The result, which is known as the program slice, holds those statements that directly or indirectly affect the value computed at a given program point. Based on the behavior invariance theorem, the project team intends to reduce the human effort in testing by performing selective regression testing only on the affected subset of the program that is identified by the slicing analysis algorithm, while maintaining the same test adequacy criteria. The primary objective of this project is to evaluate the various commercial-off-the-shelf (COTS) program slicing tools and assess their suitability for enabling safe reduction of testing effort. Identification of suitable tools is a step on the critical path towards application of program slicing to reduce the time and cost of regression testing in the Navy's technology upgrade process and in many other contexts.

Introduction

Unlike in other domains, frequent and rapid changes in software requirements and systems are not uncommon. The widespread assumption that changes to software are easier and less costly than hardware modification is implicit in current acquisition processes and contributes to the rate of change, along with external factors such as emerging new technologies, capabilities, and threats. However, the assumption may not always be valid, especially in today's System-of-System (SoS) environments, because software systems are very complex, are likely the main controller of the cyber-physical systems, and require

integrating or interfacing with other systems. Therefore, a simple or slight change in the software that is not investigated or retested properly may have a big repercussion on the behavior of the system and result in an "ineffective" or "unsafe" system. That is the motivation for both performing regression testing and the current policy of completely retesting a software system after each modification. This regression testing accounts for a significant part of the time and cost of upgrades to such systems. Our project is investigating the potential feasibility and effectiveness of applying COTS tools for software slicing to safely reduce the time and cost of this regression testing process.

The goal of regression testing is to ensure that a new program release after a code update or correction does not adversely affect the intended behavior relative to the previous certified program release (White & Leung, 1992; Binkley, 1998). Regression testing often requires executing the complete software system on a large number of test cases to validate the behavior. It is an expensive process in terms of both human effort and machine resources. One way to reduce the cost of regression testing is to conduct a majority of the new test cases automatically and off-line. Another way is to retest the critical components only. To mitigate the risk of software failure resulting from program changes on a previous stable program release and to conduct retesting selectively, we aim to effectively minimize the effort of regression testing by using the concept of program slicing to identify the critical components that need to be retested.

Program slicing is a technique for restricting the behavior of a program to a specific subset of interest according to a slicing criterion. This may produce a small subset of the program that still duplicates the same behavior with respect to that criterion. In other words, the base program and its slice will execute and produce the same values for the subset of interest (Weiser, 1984; Korel & Laski, 1988; Gallagher & Harman, 1998). Hence, program slicing is not only useful for testing, but could be used to support program comprehension, debugging, maintenance, reuse, reengineering, merging, teamwork, etc. (Weiser, 1984; Gallagher & Harman, 1998). Our focus is the application of program slicing to safely reduce the testing effort and evaluating the suitability of several COTS slicing tools in support of this goal.

This project will use previous work by Mark Weiser, Susan Horwitz, Keith Gallagher, and others. The project team may use or modify their proposed mathematical models whenever necessary in this research.

The rest of the sections are organized as follows: The next section provides the objectives and motivation for the research. The section titled Slicing Challenges describes the potential challenges that a slicing tool needs to address. The Dependencies section identifies the possible dependencies that can affect a modern program. Adequacy Criteria for Slicing Tools discusses the determination of adequacy criteria for slicing tools, and Assessment Scope and Procedure briefly describes our research approach.

Objectives and Motivation

The primary objective of the project and this paper is to provide criteria for evaluating and applying program slicing tools to safely reduce re-testing of software components in new software releases. The secondary objective is to conduct experimental assessments and compare the suitability of available COTS program slicing tools for safe reduction of testing effort. We aim to determine the suitability of available COTS program slicing tools for practical software application. The experimental assessments are not yet complete, and their results will be reported in a future publication.

The new approach will be applicable to program V&V to assist the developers and test engineers in identification and analysis of critical software components. The intention is to identify the most adequate slicing tools among the evaluated ones based on the required tool assessment criteria.

Slicing Challenges

This section characterizes features of programs that appear in applications of interest, and that must be handled accurately by slicing tools in order to safely reduce regression testing. We also describe previous research results related to slicing of programs that have these features.

Slicing Object-Oriented Programs

The first program slicing algorithms were intended for imperative programs with simple scalar data.

Object-oriented programs have been addressed using the same general approach as imperative programs extended to include specific algorithms that handle typical features of sequential object-oriented programs. For example, Larsen and Harrold (1996) proposed an algorithm for static slicing of sequential object-oriented programs based on an extended representation of system dependence graphs (SDG) to handle some of the object-oriented features such as classes and their instances, objects, inheritance, polymorphism, and dynamic binding.

Pointers and Recursion

For a slicing tool to be useful, its slicing technique must be sufficiently precise and efficient. Preciseness and efficiency are difficult to achieve when pointers and recursion are used in a program. In this context, many researchers have focused their effort on improving slicing tools to improve these aspects of the analysis and minimize this shortcoming (Liang & Harrold, 1999).

The size or length of a slice has a big impact on its usefulness. In our case, for safe reduction of regression testing, every additional statement that is unnecessarily included in a slice may mislead us to include additional test cases, which require re-execution, and may contradict our main purpose of avoiding spending effort on irrelevant tasks. Hence, this parameter will be included in our assessment criteria for the evaluation of the tools.

Difficulties with pointers arise because data is not limited to statically identified variables, as in simpler slicing methods, but rather, locations that are computed so that the same variable name may refer to different locations at different stages of the computation. The presence of pointers enables a phenomenon called aliasing. Aliasing happens when several different variables refer to the same memory location. The problem is that aliasing complicates the computation of slices and requires the use of approximations. This point is addressed differently by each slicing algorithm. The problem of checking two pointer variables can be referred to the same memory location has been proved to be un-decidable, which implies that perfectly precise analysis of aliasing is impossible, and that safe approximations are necessary.

Consequently, the analysis of program dependencies is less specific with pointers, and a slicing tool may overestimate the size of the slice and include irrelevant statements in the computed slice, as illustrated by the example shown in Figures 1 and 2. This is assuming that the algorithm generates correct slices, which is more complicated in this case, introducing more risk of error in the tool implementation. Our tool assessment criteria

will therefore include test cases that probe the soundness and resolution of slices involving pointers.

In C, for example, we can have multiple levels of indirect pointers (double or more). If this is the case, how would a slicer compute the slices correctly, precisely, and efficiently? Binkley and Lyle (1998) addressed this issue and proposed an approach based on Pointer State Sub-graphs (PSS) they used with the Unravel tool, an ANSI-C program slicer developed at NIST to support their research. However, their paper addresses a "relaxed" definition of slicing that does not require slices to be executable, which is not appropriate in our context, because they do not always satisfy the behavior invariance property: for all initial states in which the original program terminates, the slice must also terminate and compute the same results for all quantity included in the slicing criterion that determines the slice.

Mark Weiser's (1984) original definition of slices requires slices to be executable, but does not define what "executable" means. Since he talks about what happens when a slice does not terminate, we can conclude he did not require executable program to terminate for all initial states. The "relaxed" version used by Binkley & Lyle (1998) does not require slices to be executable, and does not define "executable" either. However, the discussion of an example in the paper indicates they do not consider slices that do not terminate cleanly to be executable. The example talks about a runtime exception caused by attempting to dereference an uninitialized pointer, which can happen in their "relaxed" slice, but not in the original program. This situation does not satisfy the behavior invariance property on which our test reduction analysis depends, because their relaxed definition allows cases where the original program terminates cleanly and produces a well-formed result, but the slice terminates abnormally, and therefore, does not produce the same result as the original program.

For the purpose of safe test reduction, in practical situations that include the possibility of abnormal termination, we need a version of slicing that conforms to the following conditions:

1. All slices must be executable if the original program is.
2. Whenever the original program terminates cleanly, the slice must terminate cleanly and produce the same result as the original program for all observable values specified by the slicing criterion.

The first condition is required for the meaning of the slice to be well defined. We define a program to be executable if it is sufficiently well-formed to compile, load, and begin execution. Programs that fail to terminate or that terminate abnormally are still considered to be executable.

Both conditions are consistent with the original definitions in Weiser (1984), but they have been refined to explicitly detail what "executable" means and have been extended to explicitly state how abnormal termination is to be interpreted.

In order to be suitable for test reduction, as well as conformance to the above conceptual refinements, a slicing tool must correctly represent the control dependencies associated with runtime exceptions and exception handling mechanisms. Experimental assessment of such tools should therefore include test cases that are sensitive to these aspects, such as the example in Binkley & Lyle (1998).

In recursion, the problem of overestimation may occur because recursion is a self calling construct. This means that the slicer cannot wait until after it has computed the

dependencies of everything used by the function to compute the dependency of the function itself, and must instead solve a fixed point equation to get a precise result. That process is complicated and may not be computationally traceable in the general case. A practical solution may need a safe approximation that produces an overestimation of the slice. For example, in the HC (Harrold & Ci) algorithm (Harrold & Ci, 1998), the slicer locks the current statement until the determination of its dependents.

> For programs with recursion, the HC algorithm can produce imprecise slices. When the algorithm requests slicing information for a non-local variable at a recursive call, it may find that the slicing information is currently being computed and thus, unavailable. In this case, the algorithm computes an overestimate of the statements that affect the slicing criterion. (Liang & Harrold, 1999)

Test cases suitable for measuring the degree of overestimation of slices of recursive programs should therefore be included in experimental slicing tool assessment.

Slicing Concurrent Programs

Parallel or concurrent imperative programs introduce additional concepts such as inter-process synchronization and communication. These new dependencies require additional features in the slicing algorithms. Zhao, Cheng and Ushijima (1996) addressed this problem by proposing what they called Process Dependence Nets (PDN). They generalized program dependence graphs (PDG) to represent program dependencies in a concurrent imperative program with a single procedure. In their approach, selection, synchronization, and communication dependency edges were added to control any data dependency edges in the traditional PDG. The notions of nondeterministic parallel control flow net and nondeterministic parallel definition-use net were introduced for representing multiple control flows and multiple data flows in concurrent programs. These new constructs are required because control flows and data flows of parallel processes are not independent due to the existence of inter-process synchronization among multiple control flows, and inter-process communication among multiple data flows in the program. For concurrent object-oriented programs, they proposed the System Dependence Net (SDN), which can be used to represent concurrency issues in the program, in addition to the other features used in sequential object-oriented programs.

Another challenge that requires attention is the non-determinism when a process interacts with a number of other concurrent processes while communicating. This introduces an additional challenge to the slicing task and requires special considerations such as intermediate representations that accurately represent sets of possible behaviors rather than a single definite behavior.

Dependencies

In a program, statements affect each other in different ways. Basically, any software language has the following three main constructs:

1. iteration,
2. selection, and
3. repetition.

If we include parallel programming capabilities and the external environment, we add other constructs such as synchronization, interference, and external inter-connections. To

cover these features, a complete program slicing algorithm needs to handle the following dependencies:

Data Dependency

"A statement u is directly data-dependent on a statement v if the value of a variable computed at v has a direct influence on the value of a variable computed at u" (Zhao, Cheng, & Ushijima, 1996).

Control Dependency

"A statement u is directly control-dependent on the control predicate v of a conditional branch statement if whether u is executed or not is directly determined by the evaluation result of v" (Zhao, Cheng, & Ushijima, 1996).

Parallel Dependencies

This type involves dependencies between concurrent processes. In such a case, classical control and data dependencies are insufficient to represent the flow of data and control. Additional dependencies have been defined for the purpose:

Selection Dependencies

“They [Selection dependencies] are similar to control dependencies but involve nondeterministic selection statements. A statement u is directly selection-dependent on a nondeterministic selection statement v if whether u is executed or not is directly determined by the selection result of v” (Zhao, Cheng, & Ushijima, 1996).

Synchronization Dependencies

“A statement u is directly synchronization-dependent on another statement v if the start and/or termination of execution of v directly determines whether or not the execution of u starts and/or terminates” (Zhao, Cheng, & Ushijima, 1996).

Internal-Communication Dependencies

“A statement u in a process is directly internal-communication dependent on another statement v in another process if the value of a variable computed at v has a direct influence on the value of a variable computed at u by an inter-process communication” (Zhao, Cheng, & Ushijima, 1996).

External Dependencies

An important aspect related to programs is their communication with external components and interfaces. This connection brings up another type of dependency that could have a significant impact on the behavior of the program.

Any program that uses external libraries, makes system calls, fetches data from an external database, or calls external application level services has external dependencies. This is usually done through interfaces. From the slicing perspective, the behavior of the slicer toward these components is important to evaluate. For safe reduction of regression testing, we need to know if a slicing tool includes these types of dependencies and to what extent. A program may not use these components as a whole, but rather call a subset of its

functions or use a subset of the service components. In this case, it would be useful if the slicer is able to provide precise indications of dependencies related to these external calls, or means by which users could model such dependencies.

In the same context, programs may execute input/output (I/O) statements directed to certain destinations. These statements may involve hardware modules (an actuator, a sensor, a control module, an I/O stream) and/or external software modules (a web service, another software system) and/or humans (Operator). In this perspective, a change in a software module may require retesting the components that may share the I/O destination, in particular if the changed module updates an I/O destination that is read or sensed by an unmodified component. I/O destinations should be treated as "variables" with respect to data dependencies, even if they are external to the system. Anything with a state that can influence software behavior has to be included as an I/O destination whether or not it is a component entity.

This includes physical systems that are affected by actuator controlled by the software component and sensed by sensors read by the software component. It also includes computational entities such as files, databases, web services, and human operators that interact with the system based on information it displays.

The way the slicer interprets these dependencies may significantly optimize safe reduction of regression testing. Correctly modeling the direction of such dependencies is vital for soundness and efficiency of the proposed approach.

When Retesting Can Be Safely Omitted

Re-testing of an unchanged component can be safely omitted when slicing analysis confirms that program behavior is unchanged and all of the following conditions are met:

1. Requirements of the component are unchanged.
2. Workload of the component is unchanged.
3. Behavior of the deployed machine code correctly corresponds to the source code that was analyzed via slicing.
4. Real-time constraints and other resource constraints related to the component are re-checked in system-level tests.

These assumptions and methods for checking them are discussed in detail in Berzins (2008). Methods for efficiently retesting components when component workload has changed are given in Berzins and Dailey (2009, 2010).

Adequacy Criteria for Slicing Tools

We propose criteria for determining the adequacy of slicing tools according to the assessment below. Slicing algorithms are specific to the programming language and whether it is sequential or concurrent. Other aspects of the environment of the program to be analyzed by slicing, including the hardware platform and the external components, play a significant role on its behavior. Hence, we need to include all possible dependencies present in the program according to its type, its behavior, and its environment. This is required to identify suitable algorithms that handle the possible dependencies and also the full range of features provided by the targeted programming language, specifically those of object-oriented programs.

Size of the Computed Slice

For determining the adequacy of slicing tools, the size of the computed slice is significant in our context, because the bigger the slice, the more likely it will interfere with the components modified in the new release of the system. Hence, the concept of minimal slice is very important in such a perspective. Since finding exact minimal slices is algorithmically unsolvable in the general case (Weiser, 1984), all practical tools will sometimes produce overestimations of slices. Our criteria should be able to compare the size of the slices from different algorithms by statement to statement analysis, if the tools are used on the same language, and by slice-program ratio, if the tools have to be used on different languages but for the similar requirements.

In this same context—the importance of the size of generated slices, Jackson and Rollins (1994) introduced the concept of program chopping. This technique aims at generating smaller and more precise slices with regard to a pair of slicing criteria, a source, and a sink. Chopping produces the subset of the program's statements influencing sink elements and caused by source elements. Program chopping means that a slice is limited between two statements. The chopping criteria are the source statement with a set of variables to a forward slicing and the sink statement with a set of variables for backward slicing.

From regression testing perspective this technique must be used with great care to avoid optimizations that could invalidate the analysis. To be safe, all components that were modified in the new release must be included as source statements. To judge whether a particular service of an unchanged component must be retested, the appropriate sink consists of the return statements for the service, together with all associated return values and output variables.

Programming Languages

Most of the slicing tools are programming language specific and since each language has its specificities, the criteria need to be language specific. For example, pointers can be explicitly manipulated by the programmer in C and C++, but in Java they are implicitly invoked, and operations on them are restricted to equality checks. When slicing C/C++ programs, the slicing algorithm needs to be evaluated specifically on how it analyzes pointers, with attention to computed pointer values.

Another point is related to the type of the language, meaning imperative or object-oriented (OO). This requires specific attention because OO programs include additional features, such as simple and multiple inheritance, polymorphism, and dynamic binding. The slicer of OO programs has to deal with these additional concepts by defining additional dependencies or redefining previously proposed dependencies to accurately capture these aspects of OO programs.

Behavior of the Program

If we are slicing a sequential program, our dependencies are limited to data and control. Specific considerations are to be added for object-oriented programming features, as mentioned above. In terms of dependencies, we only have the two mentioned types.

In sequential programs, slices are also sequential and follow the structure of the program consistently. When concurrent programs are evaluated, slices become more challenging to compute, because new dependencies are required to be evaluated by the algorithm, and are usually located externally to the procedure or method of interest. Hence, if we are evaluating a concurrent program, the slicer must include this capability.

More specifically, another level of analysis needs to be performed that is related to the language and its constructs. To be clear, we will give examples for the following criteria in order to explain the concept.

Pointers and Parameter Passing

The concept of call by value and call by reference needs to be addressed carefully during slicing, because the difference between these two aspects is considerably critical in safety critical systems. If we call by value, the scope of the change is limited within the procedure or the method. In C and Java, the parameter is passed by value or reference, depending on the type, and this is done by regular calls. A slicer needs to identify these aspects efficiently to avoid mishaps or abnormal behavior when the program is changed and is being tested for release.

In Figure 1 we have two different programs that have different behaviors due only to aliasing introduced by pointers in line 4 of (b). If we want to slice both programs with respect to the slicing criterion {8, p1.a}, then the set of statements {1,2,3,4,5,6,8,10,11,12} would be an overestimate for (a) but a precise slice for (b). This is due to the fact that statement (6) in program (a) doesn't affect the object p1 since program (a) creates two separate objects. In (b) this is not the case, because p1 and p2 point to the same object from statement (4) onwards. Statement (6) must be included in a correct and precise slice of (b), because in program (b), statement (6) changes the value of p1.a.

```
1  public class Test {
2  public static void main(String[] args) {
3  point p1 = new point();
4  point p2 = new point();
5  p1.a = 1;
6  p2.a = 2;
7  System.out.println("irrelevant1");
8  System.out.println("P1: a= "+p1.a);
9  System.out.println("irrelevant2");
10 System.out.println("P2: a= "+p2.a);
11 }
12 }
```

(a)

```
1  public class Test {
2  public static void main(String[] args) {
3  point p1 = new point();
4  point p2 = p1;
5  p1.a = 1;
6  p2.a = 2;
7  System.out.println("irrelevant1");
8  System.out.println("P1: a= "+p1.a);
9  System.out.println("irrelevant2");
10 System.out.println("P2: a= "+p2.a);
11 }
12 }
```

(b)

Figure 1. Example of the Preciseness of the Slicing Tool When Pointers Used

Note. (a) p1 and p2 are instantiated independently; (b) p1 and p2 refer to the same object point.

Slicer Output

Since we have planned to use the computed slices to safely reduce regression testing, we will need the capability to compare these slices and analyze them. For this reason, we would like a slicing tool to provide the following:

1. An operation to check if two slices are the same and/or
2. An operation to save a slice into a file.

Note that Capability 1 in this list can be realized using Capability 2, together with an external tool for comparing files.

If the slicer output is merely a set of highlighted statements within a screen display of the original code, the comparison will be very challenging.

In addition, this research could not lead to an efficient procedure if there is no possibility of automation. Automation is crucial in this context, especially for large scale

systems or incremental projects based on agile method. Automation may involve the usage of other tools, especially for textual comparison between different slices. This infers an essential need of an interface between the slicer and any additional tool.

External Components

As discussed in the previous section, external dependencies are important from testing perspective, because an external component may be shared by other software modules. This point of intersection between separate modules scales up the impact of change in the subsystems. If the slicer can reach these external points by any representation or indication, the tester may count on it to identify potential extent of the scope of change. Though these components come outside the program code, they are part of the whole integrated system.

The criteria we proposed so far were set up according to the specified challenges described in the Dependencies section, and the dependencies described in the Adequacy Criteria for Slicing Tools section. This research is still in progress and may not yet address all criteria relevant to the assessment of the suitability of slicing tools. The most important condition that needs to be verified is that the tools generate correct slices with respect to the behavior invariance property.

Assessment Scope and Procedure

Tentatively, the project team plans to use and conduct assessment for the following slicing tools: (1) Indus's static slicing tool for Java programs. This tool was developed by Kansas State University and delivered as an Eclipse plug-in under the product name Kaveri; (2) GrammaTech's CodeSurfer, a static slicing tool for C/C++ programs, formerly developed by Wisconsin Slicing Project; and (3) Jslice static and dynamic slicing tool for Java programs. This tool was developed by the National University of Singapore.

Most of the academic slicing tools developed for research purposes will not be evaluated in this project. Such tools include the following: (1) Unravel static slicing tool for C programs, a prototype tool contracted to the National Institute of Standards and Technology by the United States Nuclear Regulatory Commission and the National Communications System; and (2) Oberon slicing tool for Oberon system, developed by the Johannes Kepler University. For these tools, the lack or limitation of documentation and support represents a serious obstacle for us to include them in this research. We also gave lower priority to the Oberon tool, because the Oberon language (a modern version of Pascal) is not in widespread use, compared to Java and C/C++.

The team is currently testing some of the above tools and will provide a comprehensive test-driven adequacy criteria and test cases in a later publication. Some of the planned criteria identified are in the Dependencies section and the Adequacy Criteria for Slicing Tools section, including pointer handling, coverage of OO features, exceptions, concurrency features, program size limitations, programming language handling, etc; and the test cases will focus on soundness of the analysis of unchanged behavior and effort reduction based on off-line and on-line testing.

Slicing the program in Figure 1 using Indus gave us the following result:

```
public class Test {
public static void main(String[] args) {
point p1 = new point();
point p2 = new point();
p1.a = 1;
p2.a = 2;
System.out.println("irrelevant1");
System.out.println("P1: a= "+p1.a);
System.out.println("irrelevant2");
System.out.println("P2: a= "+p2.a);
}
}
```

(a)

```
public class Test {
public static void main(String[] args) {
point p1 = new point();
point p2 = p1;
p1.a = 1;
p2.a = 2;
System.out.println("irrelevant1");
System.out.println("P1: a= "+p1.a);
System.out.println("irrelevant2");
System.out.println("P2: a= "+p2.a);
}
}
```

(b)

Figure 2. Slicing the Program Using Indus

Note. (a) Sliced on criterion {10, p2.a}; and (b) Sliced on criterion {10, p2.a}.

The slicing tools Indus highlights in green are totally relevant statements, and those in yellow are partially relevant statements to the criterion. As shown in Figure 2, Indus generated the same slices for the two programs which are correct, but for (a), the generated slice includes statement (5) which is irrelevant with regard to the defined criterion, so that we have an overestimated slice in case (a).

Conclusion

The time and cost of software development are important concerns in system acquisition. For systems with long lifetimes, regression testing is a major component of the cost of each new release, including periodic technology upgrades typical of DoD/Navy systems. Slicing has the potential to reduce the time and cost of this regression testing, which is necessary to ensure the safety and effectiveness of each new release. The research reported here will facilitate the practical application of this approach by evaluating existing slicing tools to determine if any of them are currently able to adequately support this process.

This project is in its early stages, and this paper reports some of our preliminary results, which consist of evaluation criteria for slicing tools in the context of their ability to achieve safe reduction of regression testing. We have identified potential difficulties in implementing sound and discriminating slicing tools, together with associated risks with respect to safe reduction of regression testing. These difficulties and risks have been used to derive tool assessment criteria to be used in the experimental phase of this project. These criteria are described in the Dependencies section and in the Adequacy Criteria for Slicing Tools section of this paper.

Experimental assessment is currently in progress and is not yet complete. Assessment results for particular tools will be reported when the effort is complete and measurements related to the evaluation criteria are available.

If the result of the tool assessment is positive, the next step will be to use the chosen slicing tools to identify possible reductions in regression testing for a subset of a real system in a pilot study to check the safety and effectiveness of the theoretically proposed test reduction process in a realistic setting.

If the result of the tool assessment is not positive, our project will identify the candidate tools that are closest to meeting the requirements for supporting safe regression test reduction and the current shortcomings of those tools, thus laying the groundwork for obtaining the needed tool support.

References

Berzins, V. (2008, April). Which unchanged components to retest after a technology upgrade. In *Proceedings of the Fifth Annual Acquisition Research Symposium*. Monterey, CA: Naval Postgraduate School.

Berzins, V., & Dailey, P. (2009, April). How to check if it is safe not to retest a component. In *Proceedings of the Sixth Annual Acquisition Research Symposium*. Monterey, CA: Naval Postgraduate School.

Berzins, V., & Dailey, P. (2010, May). Improved software testing for open architecture. In *Proceedings of the Seventh Annual Research Symposium—Acquisition Research: Creating Synergy for Informed Change* (pp. 385–398). Monterey, CA: Naval Postgraduate School.

Binkley, D. (1998). The application of program slicing to regression testing. *Information and Software Technology, 40*(11–12), 583–594.

Binkley, D. W., & Lyle, J. R. (1998). Application of the pointer state subgraph to static program slicing. *Journal of Systems and Software.*

Gallagher, K. B., & Harman, M. (1998). Program slicing. *Information and Software Technology (Special Issue), 40*(11–12).

Harrold, M. J., & Ci, N. (1998). Reuse-driven inter-procedural slicing. In *Proceedings of the 1998 International Conference on Software Maintenance* (p. 74).

Jackson, D., & Rollins, E. J. (1994). Chopping: A generalization of slicing. In *Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering* (pp. 11–17).

Korel, B., & Laski, J. (1988). Dynamic program slicing. *Information Processing Letters*, 155–163.

Larsen, L., & Harrold, M. J. (1996). Slicing object-oriented software. In *18th International Conference on Software Engineering* (pp. 495–505).

Liang, D., & Harrold, M. J. (1999, August). Reuse-driven inter-procedural slicing in the presence of pointers and recursion. In *Proceedings of the 1999 International Conference on Software Maintenance* (pp. 421–432).

Weiser, M. (1984). Program slicing. *IEEE transactions on software engineering, SE–10*(4), 352–357.

White, L., & Leung, H. (1992). Regression testability. *IEEE Micro, 12*(2), 81–84.

Zhao, J., Cheng, J., & Ushijima, K. (1996). Static slicing of concurrent object-oriented programs. In *Proceedings of the 20th IEEE Annual International Computer Software and Applications Conference*.