Which Unchanged Components to Retest after a Technology Upgrade

Valdis Berzins
Professor – Department of Computer Science (NPS)
E-mail: berzins@nps.edu, Phone: 831-656-2610

Context

- The Navy is moving towards an Open Architecture paradigm
  - Joint interoperable systems that ***adapt*** and are built using open interfaces, open design principles, and open architectures
- Expected long term benefits from Navy Open Architecture
  - Business benefits:
    - Flexible acquisition strategies and contracts that enable ***software reuse, easy systems upgrade***, and ***shared data*** throughout the Navy
  - Technical benefits:
    - Modular open architectures facilitate ***portability***, maintainability, interoperability, ***upgrade-ability*** and ***long-term supportability***
- The Achilles Heel - Test and Evaluation
  - Current practices require ***retesting unchanged components*** in each new deployment context, typically every two years
  - Substantial budget and schedule are currently devoted to retesting
  - ***New technology, processes, and policies*** are needed to ***safely reduce*** this effort and free resources for testing new functionality

Objectives

- ***Safely reduce*** software system testing cost

- Software system testing cost consists of
  - Up-front testing cost

    PLUS

  - Cost attributed to missed errors
    - I.e., cost of future system failures

- We seek to reduce both parts of the cost

Problem Statement

- According to Navy and other experience, traditional approaches to testing are not well suited to open environments
  - They are ***too expensive***, ***take too long*** and ***lack agility*** to react to changes during acquisition or missions
  - Have to be ***repeated after every change***
- Typical testing assumptions are not valid for Open Architectures
  - Conventional testing methods require the ***system environment*** to be ***fixed*** and ***known in detail*** at test and evaluation time
    - o Effectiveness of testing is very sensitive to the expected operating environment, which is ***unknown for reusable components***
  - Current test and evaluation methods check conformance to ***specifications***
    - o The majority of ***failures*** in software systems are due to requirements and ***specification errors***, and commonly show up after a subsystem has been ***moved to a different environment***
    - o Commonly called "system integration problems"

Approaches

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

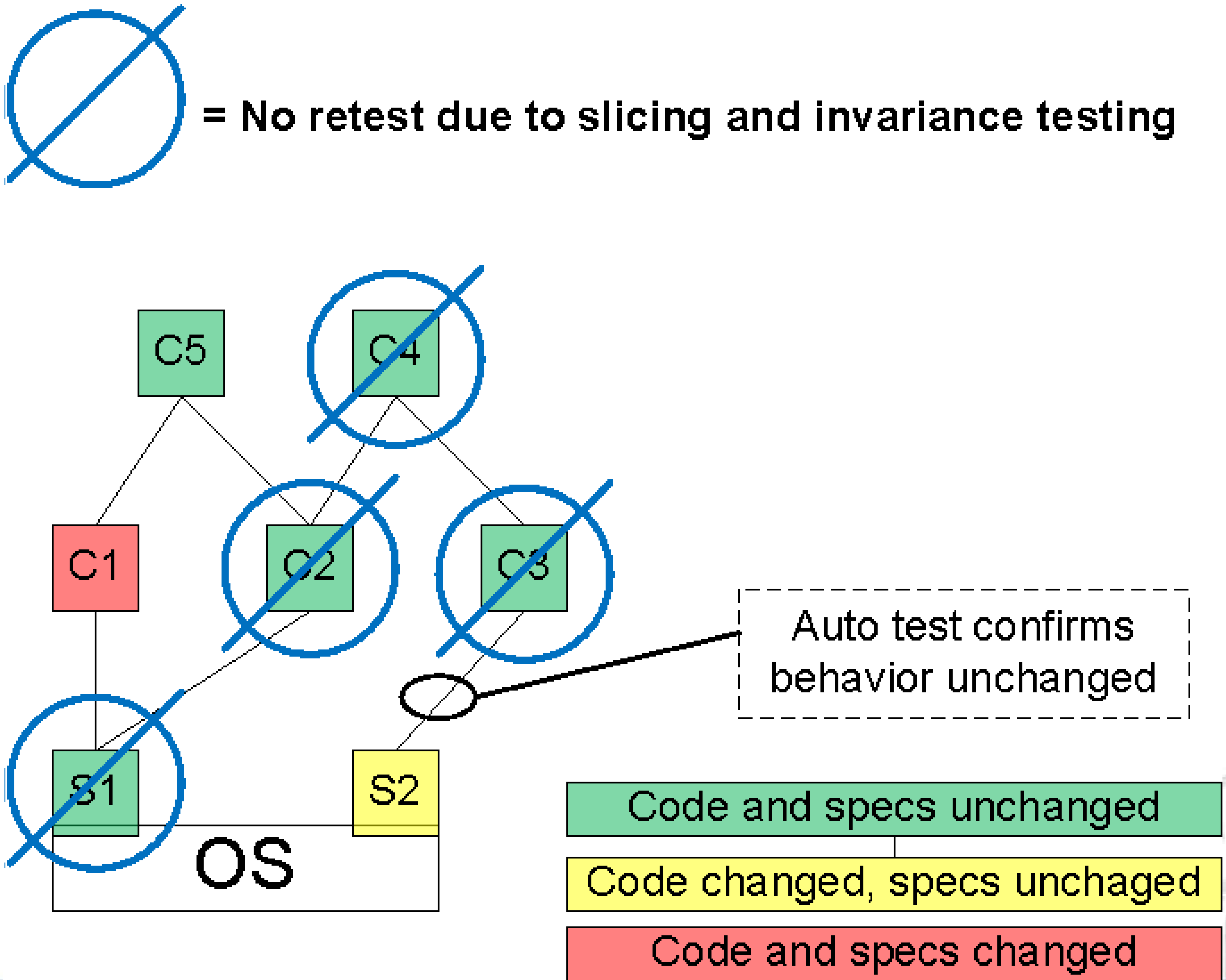
- Maintain safety (this paper)
  - Program slicing to confirm unchanged behavior of unchanged code
  - Automated testing to confirm unchanged behavior of modified code

- Enable Plug-and-Fight (long term vision)
  - Eventually ***eliminate integration test after every reconfiguration***
  - A technology roadmap to accomplish this was presented last year
    - o *Proceedings of the Fourth Annual Research Symposium—Acquisition Research: Creating Synergy for Informed Change* (May 16-17 2007, pp. 285-312).
  - This paper addresses a simplified sub-problem of the vision

Retesting Unchanged Components?

- Retesting is necessary *but not always*
- Did component behavior change?
  - Does *it depend on modified code*?
  - Does *the modified code have different behavior*?
- Did component requirements change?
  - Is the *old behavior still appropriate*?
- Did component workload change?
  - Did the *range of valid inputs change*?
  - Did the *range of expected inputs change*?
  - Did the *set of reachable states change*?
- Did available *resources change*?
  - Memory, processor, network bandwidth,…
  - Do other modified components use more resources?

Example


= No retest due to slicing and invariance testing
C5
C4
C1
C2
C3
S1
S2
OS
Auto test confirms behavior unchanged
Code and specs unchanged
Code changed, specs unchanged
Code and specs changed

Approach: Program Slicing [Weiser 84]

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

Invariance Testing Extends Program Slicing

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

How Much Invariance Testing is Enough?

- How many tests are needed to reach *high confidence*?
  - Stakeholder defines the acceptable risk threshold *k*
    - o *The mean time between observations of a behavioral difference in a given operating system service is k-times longer than a mission*.
- Number of test cases is computed for each service in the middleware interface to the operating system
  - It is determined by the following formula

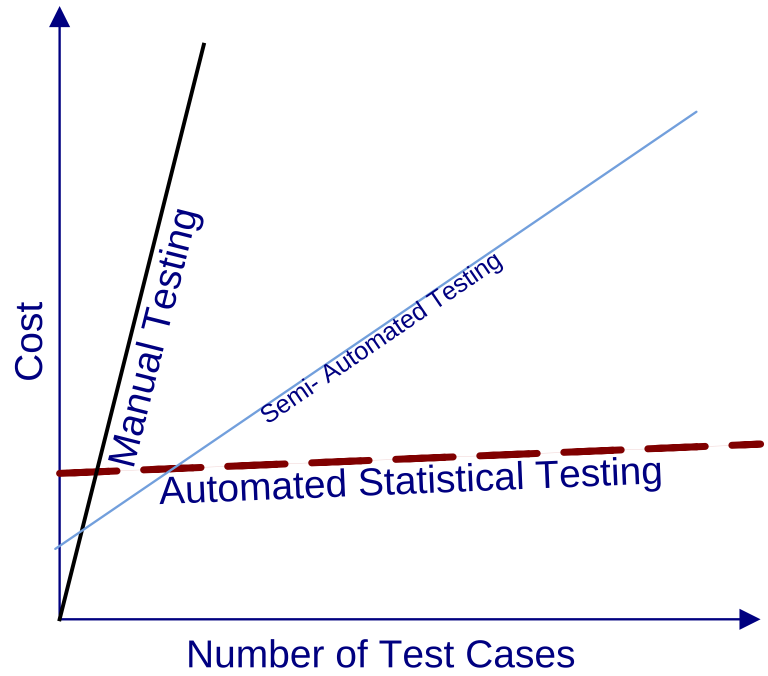
$$T_s = (k\, e_s) \log_2 (k\, e_s)$$

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

Testing Efforts vs. Acceptable Risk

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

Number of test cases required for different levels of risk tolerance



Testing cost characteristics

Why Do We Need Operational Profiles

- Can be used to ***automate selection of test cases***

- Reliability of a system is determined by the operational profile
  - Real systems have bugs, coding errors, requirement omission, etc.
  - System reliability varies from **0** (always fails) to **1** (never fails) in different environments

- Operational profiles have proved useful in practice
  - Example: reliability testing of telephone-switching software

- It takes human effort to produce an operational profile
  - Measure the frequency distributions of operating system calls and associated input parameters
    - o Can be collected on- or off- line

When Retesting a Service is Necessary

- When its slice or behavior has changed

- When requirements have changed
  - New functionality needs to be tested
  - Test all affected components

- When the ***range of expected operating conditions*** has expanded
  - Even if there was no other change, new test scenarios are needed
  - Indicated by a modified operational profile

- When computing speeds or timing constraints have changed
  - Changed hardware processing rates can adversely affect scheduling algorithms and cause missed deadlines

Conclusions

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

Backup Slides

Related Work

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