CROSSTALK

March / April 2015 The Journal of Defense Software Engineering Vol. 28 No. 2

TEST & DIAGNOSTICS

Report Documentation Page

Form Approved
OMB No. 0704-0188

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

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
APR 2015		**00-00-2015 to 00-00-2015**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
CrossTalk, The Journal of Defense Software Engineering. Volume 28, Number 2, March/April 2015	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **517 SMXS/MXDED,6022 Fir Ave,Hill AFB,UT,84056-5820**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

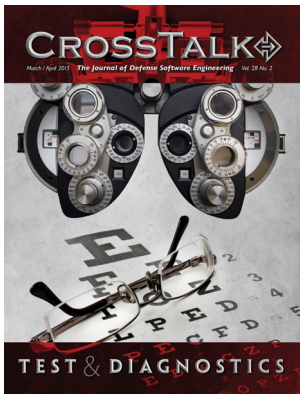
14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**40**	

TABLE OF CONTENTS

Departments



Cover Design by Kent Bingham

Test and Diagnostics

CrossTalk

NAVAIR Jeff Schwalb
DHS Joe Jarzombek
309 SMXG Karl Rogers

Publisher Justin T. Hill
Article Coordinator Heather Giacalone
Managing Director David Erickson
Technical Program Lead Thayne M. Hill
Managing Editor Brandon Ellis
Associate Editor Colin Kelly
Art Director Kevin Kiernan

Phone 801-777-9828
E-mail Crosstalk.Articles@hill.af.mil
CrossTalk Online www.crosstalkonline.org

CrossTalk, The Journal of Defense Software Engineering is co-sponsored by the U.S. Navy (USN); U.S. Air Force (USAF); and the U.S. Department of Homeland Security (DHS). USN co-sponsor: Naval Air Systems Command. USAF co-sponsor: Ogden-ALC 309 SMXG. DHS co-sponsor: Office of Cybersecurity and Communications in the National Protection and Programs Directorate.

The USAF Software Technology Support Center (STSC) is the publisher of CrossTalk providing both editorial oversight and technical review of the journal. CrossTalk's mission is to encourage the engineering development of software to improve the reliability, sustainability, and responsiveness of our warfighting capability.

Subscriptions: Visit <www.crosstalkonline.org/subscribe> to receive an e-mail notification when each new issue is published online or to subscribe to an RSS notification feed.

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the CrossTalk editorial board prior to publication. Please follow the Author Guidelines, available at <www.crosstalkonline.org/submission-guidelines>. CrossTalk does not pay for submissions. Published articles remain the property of the authors and may be submitted to other publications. Security agency releases, clearances, and public affairs office approvals are the sole responsibility of the authors and their organizations.

Reprints: Permission to reprint or post articles must be requested from the author or the copyright holder and coordinated with CrossTalk.

Trademarks and Endorsements: CrossTalk is an authorized publication for members of the DoD. Contents of CrossTalk are not necessarily the official views of, or endorsed by, the U.S. government, the DoD, the co-sponsors, or the STSC. All product names referenced in this issue are trademarks of their companies.

CrossTalk Online Services:
For questions or concerns about crosstalkonline.org web content or functionality contact the CrossTalk webmaster at 801-417-3000 or webmaster@luminpublishing.com.

Back Issues Available: Please phone or e-mail us to see if back issues are available free of charge.

CrossTalk is published six times a year by the U.S. Air Force STSC in concert with Lumin Publishing <luminpublishing.com>.
ISSN 2160-1577 (print); ISSN 2160-1593 (online)

CrossTalk would like to thank DHS for sponsoring this issue.

Success of mission and business functions should be the focus of verification and validation activities. Issues from sloppy manufacturing hygiene and insufficient test and diagnostics can enable exploitation and compromise of operational functionality. Enormous energies are put into assuring safety-critical functions address any source of taint, such as vulnerabilities, weaknesses, and malicious logic. With more functionality being delivered through cloud services the burden of security shifts to development with test being the last line of defense.

Industry has substantially invested in improving the quality of products and systems because of customer/user demand; contributing to a rise of automated software testing capabilities and test services. As the size and complexity of software and logic-bearing devices increase, test and diagnostic capabilities must mature to address the changing environment in which products and systems are deployed. Despite encouraging results with various quality improvement approaches, the software industry is still far from zero defects. Part of that is a cultural issue with developers often making risk decisions for which they are not held accountable, such as disabling compiler warnings and selecting unpatched components from libraries. Testing is further complicated because software-based systems often have additional features, interfaces, and functionality that use third party libraries, general purpose applications, and a multiplicity of features in system libraries and system calls. As witnessed by the myriad of patches needing to be addressed due to residual exploitable vulnerabilities, weaknesses, and malware, software-based systems are inherently susceptible to attack and manipulation. To address the difference between what is conceived and what is delivered, testers need to think about how software-based systems are actually integrated and deployed. If libraries are incorporated and deployed by a compiler, or configuration choices undermine design choices, or someone exposes a weakness, then testers need to factor in means for detecting these before deployment; not after an application or system compromise. Often, more comprehensive test programs are needed. This requires improved security functionality and more rigorous review, testing and inspection. Test coverage for agile continuous testing, automated API testing, metamorphic testing (including runtime checking), fuzz testing, and other techniques and methods need to be part of test organizations' process improvement list of strategic considerations. Multiple techniques and tools, including static and dynamic analysis, should be used for software assurance. Test-driven development is a programmer practices that has been employed by a growing number of software development teams. Despite the fact that testing often accounts for at least 30-40% of total project costs, only limited attention has been given to testing in various software process improvement models, including the Capability Maturity Model Integration (CMMI). As a result, the testing community has developed and used its own improvement models, such as the Test Maturity Model integration (TMMi) for test process improvement that is positioned as being complementary to the CMMI to more comprehensively address those issues important to test managers, test engineers and quality professionals.

Unfortunately, partially because of the lack of adequate due-diligence and due-care in development and integration test activities, vulnerabilities are proliferating rapidly; thus stretching mission capabilities and resources. As we seek to discover and mitigate the root causes of these vulnerabilities, sharing the knowledge we have of them help to mitigate their impact. In order to keep pace with growing threats we must facilitate the automated exchange of information. With that objective the Department of Homeland Security (DHS) sponsors 'free for use' standardized means for sharing information. These include the Common Weakness Enumeration (CWE) that provides standardized means for identifying and mitigating architectural, design and coding flaws introduced during development and detectable in testing, along with the Common Attack Pattern Enumerations and Classification (CAPEC) that enables developers, testers and defenders to discern attack constructs, build software resilient to them, and determine the sufficiency of test regimes focused on checking security concerns. These open specifications for interoperable security automation enable secure, machine-to-machine communication of actionable indicators within and between organizations that want to share this information. These have been developed collaboratively between Federal Government and industry partners working toward information sharing mechanisms and solutions to reduce the risk of tainted components. These standardized means for sharing information are already being used, and they contribute to efforts that enable more stakeholders to secure their part of cyberspace.

CrossTalk again thanks DHS Office of Cybersecurity and Communications for co-sponsoring this issue focused on test and diagnostics. Along with DoD, NIST, and GSA, DHS co-sponsors the Software & Supply Chain Assurance (SSCA) Forum in which Federal, academic, and industry stakeholders address risks and mitigation methods. SSCA Forums are free and open to the public, and resources are available on the SSCA Community Resources and Information Clearinghouse, with many applicable to test and diagnostics -- see <https://buildsecurityin.us-cert.gov/swa/pocket_guide_series.html> for "Software Security Testing" and "Key Practices for Mitigating the Most Egregious Exploitable Software Weaknesses."

Justin T. Hill
Publisher, CrossTalk

Portable Automated Test Station: Using Engineering-Design Partnerships to Replace Obsolete Test Systems

Benjamin Chase, Hill Air Force Base

Abstract. The Air Force A-10C attack aircraft is currently experiencing obsolescence issues with its legacy support equipment. The Portable Automated Test Station Model 70 (PATS-70) replaces more than a dozen pieces of obsolete and irreparable flightline support equipment. The PATS-70 is a robust, flightline qualified test set that has gone through rigorous environmental testing. While it was developed to function as maintenance support equipment for the A-10C aircraft, it has no A-10 specific components so it can be adapted for maintenance on other aircraft, platforms, or systems.

Introduction

The Portable Automated Test Station Model 30 (PATS-30), Organizational (O)-Level test set is comprised of a COTS ruggedized Personal Computer using the Microsoft Windows XP operating system, a transport case, a cable set, and custom software. The PATS-30 functions as maintenance support equipment for the A-10C aircraft by performing maintenance activities and troubleshooting avionics system faults while the aircraft is on the ground.

The core component of the PATS-30, the ruggedized laptop, is no longer sustainable. Since the laptop is no longer available, the PATS-30 will no longer be supportable. Southwest Research Institute (SwRI) was tasked by the Air Force to address the end-of-life issues of the currently fielded PATS-30 and develop a proof of concept unit that performs the functions of the PATS-30. SwRI constructed one prototype, identified as the PATS-50 and field-demonstrated the prototype.

The PATS-50 proof of concept project demonstrated the feasibility of re-hosting the PATS-30 functions onto a different test platform. During the proof of concept project, some functionality was lost, the environmental requirements were not met, and additional requirements were identified. Analysis of the additional requirements relative to the PATS-50 was completed. The results of this analysis showed that the PATS-50 concept could be utilized and re-hosted onto a test station that included the lost functionality from the PATS-30, addressing the environmental requirements, and expanding capabilities to include several separate test systems that have been identified for inclusion into the PATS. This new design concept was designated the Portable Automated Test Station Model 70 (PATS-70) [1].

The PATS-70's modular design has been developed to meet the functionality requirements, environmental requirements, and additional expansion capabilities. The PATS-70 replaces more than a dozen pieces of obsolete and irreparable flightline support equipment. It has been organically developed using COTS components and industry standard software. An engineering-design partnership was formed between the Air Force and Marvin Test Solutions[1] to facilitate this development and to help the program meet its objective on-time, and on-budget. This led to the selection of the Marvin PXI-based, ultra-rugged MTS-207 platform, laying the ground work for PATS-70 instrumentation selections.

In addition, the PATS-70 provides flexibility over other test sets since the software architecture was uniquely designed for ease of adding additional Test Program Sets (TPSs) to support the war fighter's needs. Moreover, the PATS-70 hardware has spare capacity to add additional COTS PXI components to support future TPS development. With thousands of PXI cards available today, this provides the Air Force the necessary flexibility to tackle a multitude of test requirements and applications. The PATS-70 has also been selected as the test platform for a new A-10 weapons systems maintenance capability, merging the capability of eight legacy test sets along with additional flightline test capabilities into one test set.

The PATS-70 is a robust, flightline qualified test set which has gone through rigorous MIL-STD-810G, and MIL-STD-416F environmental testing. While it was developed to function as maintenance support equipment for the A-10C aircraft, it has no A-10 specific components so it can be adapted for maintenance on other aircraft, platforms, or systems. The flexibility and added functionality gives the PATS-70 an advantage in maintaining multiple systems throughout the Department of Defense.

System Overview

The PATS-70 is an automatic test system designed to perform functional tests on the Fairchild Republic A-10C Thunderbolt II's (A-10) Anti-skid, Alpha Mach, Stability Augmentation System (SAS), and Fuel Quantity Indicating System. The PATS-70 provides the logic and hardware control necessary to coordinate and automate control of these system functions. The PATS-70 is an automated, user friendly, state-of-the-art adaptable test set that provides robust system diagnostic capability, significantly reducing the time required to bring an aircraft into mission ready status. The PATS-70 provides a mission ready test set for the A-10 Command Center. The A-10 Aircraft Operational Test System (OTS) consists of a PATS-70, and the Operational Test Program (OTP). The OTS functions as maintenance support equipment for the A-10 aircraft. The OTS performs maintenance activities as well as trouble shooting avionics system faults while the aircraft is on the ground. The PATS-70 utilizes up-to-date, sustainable technology for Operational Flight Program (OFP) software loading and diagnostic avionics system testing and includes additional TPSs to enhance its capability while decreasing the A-10 maintainability footprint. To preserve combat effectiveness, and efficiency, the PATS-70 automates and consolidates multiple test capabilities into one mission ready test set. The PATS-70 is expandable to allow for the addition of future aircraft [2].

Design

The PATS-70 is a portable field-level/O-level test set (see Fig. 1) capable of performing functional tests on aircraft systems. The test set is housed in an enclosure constructed of a durable composite material for protection and an embedded wire frame for electromagnetic interference (EMI) compliance. The enclosure consists of upper and lower sections secured together with eight (8) turn lock latches. A pressure-relief valve is located in the lower half of the enclosure. The PATS-70 is based on the Compact Peripheral Component Interface (cPCI or PCI); PCI eXtensions for Instrumentation (PXI) technology and includes a rugged 14-slot PXI chassis, power supplies, numerous circuit cards, wiring harnesses, cables for interfacing the test set with the Unit Under Test (UUT), heaters, fans, and other electrical and mechanical components [3].

Engineering-Design Partnership

An engineering-design partnership was formed between the Air Force and Marvin Test Solutions. Having extensive background and domain expertise with the development of test systems and test software, the 309th Software Maintenance Group was looking to partner with a company that had a similar level of domain expertise in flightline test and ultra-rugged test platforms. This led to the selection of the Marvin Test Solutions MTS-207 ultra-rugged PXI chassis. Having deployed flightline test sets in 20 countries in the past two decades, Marvin Test Solutions provided the platform and support to help the 309th SMXS deliver the PATS-70 on-time, and on-budget.

Hardware

The PATS-70 is PXI-based instrumentation platform. The circuit cards are COTS PXI products. PXI is an architecture for test and measurement applications that is based on the cPCI bus. This high-performance architecture provides for the throughput and synchronization required for the performance of precision measurements. The following list describes the main hardware components of the PATs-70 hardware.

Portable Automated Test Station, MTS-207-3 chassis - The Internal Chassis Assembly is the main assembly of the MTS-207-3. It "hangs" from the top panel of the MTS-207-3 accommodating connectors, switches, etc. via four (4) shock absorbers designed to protect the internal electronics. The Internal Chassis Assembly accommodates all the MTS-207-3 electronics. Its main assemblies are the PXI card cage, and the Power Board circuit card assembly (CCA). The Power Board provides all required PXI chassis power rails as well as additional supplies required for the operating of the display and peripheral PATS-70 equipment. Additionally, the Power Board provides control over the MTS-207-3 heaters, allowing operation at extreme low temperatures. The EMI Filter protects the MTS-207-3 from power surges and eliminates conducted emissions, to ensure compliance with MIL-STD-461 requirements.

User Interface Display (Tablet) – A modified Miltope RTCU-2 Tablet computer is used as the operator console. The Tablet is powered by a 1.06 GHz Intel Core i7-620UE processor with 4MB L2 Smart Cache and 8 GB of RAM. The Tablet is dock mounted or extended on the supplied user interface cable. The Government modification of the tablet allows for external connection utilizing reliable MIL circular connector technology.

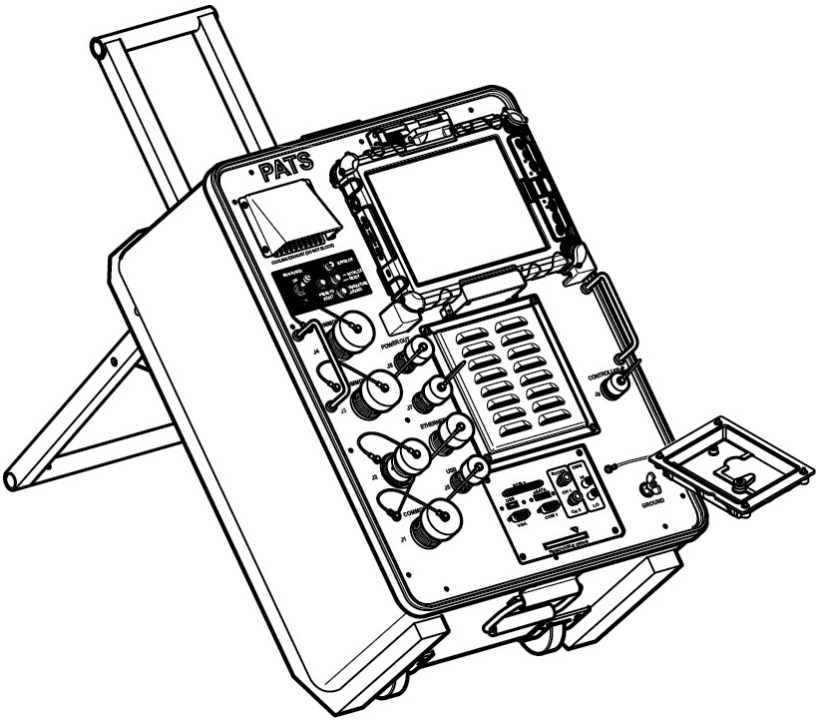


Figure 1. PATS-70 Hardware

Removable Solid State Hard Drives – The removable storage device is a minimum of 120 GB Solid State Drive (SSD); it is mounted inside the case via a drive slot on the face of the chassis, or stored in the engineering panel cover. It is configured with Windows XP or Windows 7 OS and allows integration of classified software when required.

Controller CCA- This controller contains a 2.53 GHz Intel i7 core processor with 4GB Random Access Memory (RAM). Utilizing the rear transition module, this CCA has four Gigabit Ethernet ports, two VGA ports, five USB ports and two RS232 ports, in a 6U cPCI module.

45 Relay Form C CCA - The Form C relay matrix for the high current switching requirements includes 45 single pole double throw Form C relays with 7A contact rating per channel in a 6U PXI module.

8x132 2AMP Relay Matrix CCA – This is a very high density electro-mechanical relay matrix with a 132x8 format and 1 pole switching. The matrix is constructed using high reliability 2A electro-mechanical relays with long life and stable contact resistance and is a single slot, 6U PXI module.

1553 Communications CCA – The communications card supports up to 4 dual redundant 1553 channels. Each channel operates simultaneously as a bus controller, bus monitor and remote terminal in a 3U PXI module.

Digital Multi-Meter (DMM) CCA – The 6.5 digit multi-meter is capable of true AC RMS measurements from 10Hz to 100 KHz, measures 1uV to 330V, frequency counting from 1Hz to 300 KHz in a 3U PXI module. The DMM supports Volts DC, Amp DC, Two-Wire Resistance, Four-Wire Resistance and Frequency measurements, in a 3U PXI module.

Small Computer System Interface (SCSI) CCA – A single channel SCSI interface, supports up to 320MB/s throughput; it is backwards compatible with ultra2 SCSI and is a 3U PXI module.

Differential Oscilloscope CCA – A 2 channel 14 bit resolution, 300MHz bandwidth digitizer features a maximum sample rate of 400MS/s and is a 3U PXI module.

Arbitrary Waveform Generator (ARB) CCA – A 2 channel 14 bit resolution, 50 MHz bandwidth waveform generator features a maximum sample of 200MS/s and is a 3U PXI module.

Software

The software subsystem consists of four software layers; the Operating System, the Device and Instrument Drivers, Test Executive Software, and the Test Programs as shown in Fig. 2. Each layer is a building block for the next layer.

Operating System - comprised of COTS software that is provided on each PATS-70 System. The PATS-70 uses two removable Solid State Hard drives (SSD); one containing the Windows XP Operating System (OS) and one with the Windows 7 OS. Different sets of tests are available depending on the OS currently loaded. TPS test programs, PATS-70 Self-

Test, and System Calibration all run under the Windows 7 OS. The A-10 Operational Test Program (OTP) runs under the Windows XP OS and contains its own Self-Test function built into the OTP software.

Device Drivers and Instrument Wrappers - includes any software that initializes and controls specific system interfaces between the system controller and system devices.

Test Executive Software - includes the requisite COTS software used to create and modify test programs.

Test Programs - specific programs designed to test and manipulate the UUT. Two of these TPSs are the Alpha Mach and Anti-Skid which are software that can be loaded on the Windows 7 System SSD. The Alpha Mach and Anti-Skid TPSs are used to conduct diagnostic tests for the Alpha Mach and Anti-Skid systems on the A-10.

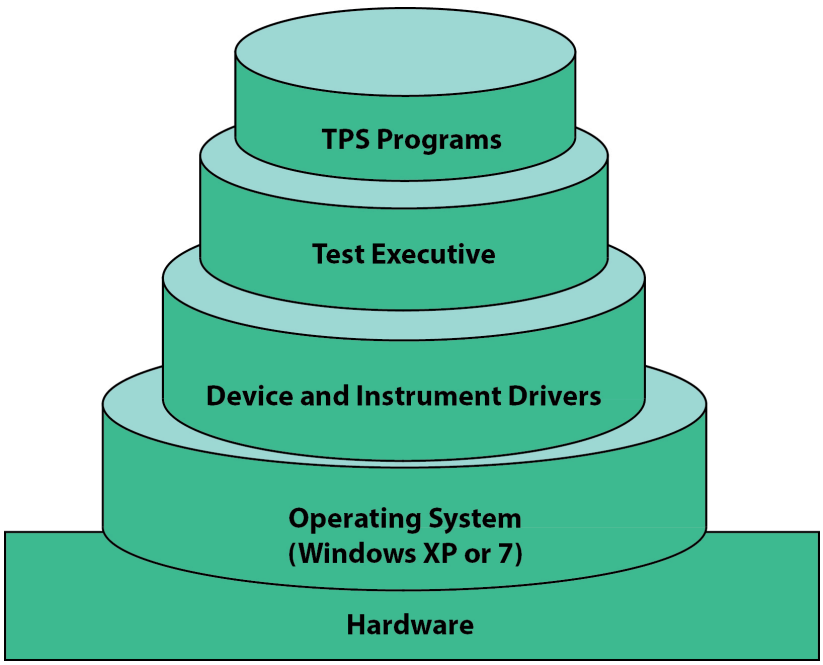


Figure 2. System Software

In addition, there are two other software components incorporated in the PATS-70 system:

Self-Test - tests the PATS-70 and identifies the faulty replaceable subassembly such as the arbitrary waveform generator, digital multimeter, and oscilloscope.

System Calibration - tests the PATS-70 against a defined performance standard and identifies the faulty replaceable subassembly. PATS-70 calibration determination has been approved and listed by Air Force Metrology and Calibration (AFMETCAL). PATS-70 calibration traceability and configuration management is accomplished through an automatic calibration routine, approved Computer Program Identification Number (CPIN), and calibration procedures.

Applications

The PATS-70 attaches to the A-10C aircraft through a variety of TPS connections in which applicable tests are run to track down and identify issues. In addition, the PATS-70 test set has the ability to perform firmware and/or Operational Flight Program (OFP) loads to the Integrated Flight and Fire Control Computer (IFFCC), Central Interface Control Unit (CICU), Improved Electronic Processor Unit (IEPU), and download Non-Volatile Memory (NVM.) The PATS-70 consolidates a wide variety of avionic specific support equipment and software into a single test unit.

The current software available on the PATS-70 to maintain the aircraft consists of the Operational Test Program (OTP), Alpha Mach, and Anti-skid TPSs which are described in the following.

Operational Test Program

The OTP TPS is software that can be loaded onto the Windows XP System SSD. The OTP is used to conduct diagnostic tests for avionics and weapons stations of the aircraft and to load OFPs.

Alpha Mach

The Alpha Mach TPS is used during flightline test of the A-10C aircraft to isolate anomalies in the Alpha Mach computer and its related components.

The Alpha Mach is part of the secondary flight control system. It receives air pressure and lift data to operate the leading edge slats and automatically improve high angle of attack airflow to the engines. The system also provides audible warnings to the pilot for engine peak performance and impending stall situations.

Anti-Skid

The Anti-Skid Control System TPS is used during flightline testing of the A-10C aircraft as the Anti-Skid operational check and to troubleshoot and isolate Anti-Skid Control System anomalies.

The Anti-Skid Control System is a modulating wheel skid control system which proportionately reduces the amount of left hydraulic system pressure supplied to both main landing gear brakes when either main wheel begins to skid.

Enhanced Capabilities and Applications

Stability Augmentation System

The Stability Augmentation System (SAS) TPS is used during flightline test of the A-10C aircraft to isolate anomalies in the SAS computer and its related components.

The purpose of the SAS is to improve the stability and handling quality of the aircraft, especially in low altitude, high angle-of-attack situations. The computer, a part of the stability augmentation system, contains sensors and electronic circuits to amplify and modify sensor signals. The computer interfaces with the Low Altitude Safety and Targeting Enhancement (LASTE) computer for further processing of the pitch, roll, and yaw rate data. The LASTE computer sends its signals to the SAS computer which drives aircraft actuator devices in the pitch and yaw axes. Contained in the test sequence are directives for hardware resource control which are sent to the Test Executive Program. When directed to perform a measurement, the Test Executive Program returns measurement data which is compared with test limits contained in the test sequence to determine a GO or NOGO result. The test description, limits and GO/NOGO results are then passed back to the Test Executive Program for display on the PATS-70 [2].

Fuel Quantity Indicating System Tester

The A-10 Fuel Quantity Indicating System (FQIS) consists of capacitive fuel probes and compensators, a fuel quantity intermediate device (FQID), an indicator, and associated wiring harnesses. The FQID has a pivotal role in the fuel quantity system. Its purpose is to monitor tank probes and compensators to gather fuel information. It generates analog voltage signals proportional to the fuel quantity and sends these to an external indication device (fuel quantity indicator) [4].

The FQIS Tester is designed for use with the PATS-70 tablet. The tester measures fuel probe and compensator capacitances and harness insulation resistance; simulates the tank probe capacitances at empty, full, and unbalanced; and stimulates the FQID and fuel quantity indicator. This allows for full calibration, testing, and troubleshooting of the FQIS.

PATS-70A

The 309th Software Maintenance Group is currently developing the PATS-70A which will consist of a modified PATS-70 Core Unit, PATS-70A Auxiliary Unit, and equipment interface cables. The modification of the Core Unit includes adding several additional PXI cards and the associated wiring harnesses and software changes. Since the PATS-70 is based on a COTS platform and the PXI standard, these modifications require a minimal effort and demonstrate the adaptability of the PATS-70 [5].

The PATS-70A is designed to provide in-depth testing and troubleshooting functions to support a variety of A-10 armament related Line Replaceable Units (LRUs) at both the flightline and Intermediate (Back shop) levels. The testing functions to be provided by the PATS-70A include:

1. PATS-70A System Calibration
2. PATS-70A System Self-Test
3. Digital Data Processing Unit (DDPU)
4. Dual Rail Adaptor (DRA)
5. DRA w/ Launcher Unit (LAU)105's attached
6. Electronic Gun Control Unit (EGCU)
7. Electrical Test Panel (ETP)
8. LAU-105/A Guided Missile Launcher
9. LAU-117A(V)3/A Guided Missile Launcher
10. LAU-131/A Rocket Launcher
11. LAU-88A/A Guided Missile Launcher
12. Munitions Armament Unit (MAU)-40/A Bomb Ejector Rack
13. MAU-50/A Bomb Ejector Rack
14. Modified Triple Ejector Rack (TER)-9A (Digital TER)
15. Triple Ejector Rack (TER)-9A
16. Pylon Wiring-Weapons Station 1/11
17. Pylon Wiring-Weapons Station 2/10
18. Pylon Wiring-Weapons Station 3/9
19. Pylon Wiring-Weapons Station 4/8
20. Pylon Wiring-Weapons Station 5/7
21. Pylon Wiring-Weapons Station 6
22. Station Control Unit A (SCU-A)
23. Station Control Unit B (SCU-B)
24. DRA Wiring Harness
25. Gun, Aircraft Unit (GAU)-8A_Wiring&Sensors
26. Guided Missile Interface Unit (GMIU)
27. LAU-105_Power Supply
28. LAU-105_Wiring Harness
29. LAU-117_Launcher Electronic Assembly (LEA)
30. Modified TER-9A Electronic Control Unit Rack Kit
31. TER-9A Wiring Support Assembly
32. A-10C Armament Wiring

Future Growth

The PATS-70 is a robust, flightline qualified test set which currently functions as maintenance support equipment for the A-10C aircraft. Yet it has no A-10 specific components so it can be adapted for maintenance on other aircraft, platforms, or systems such as helicopters, tanks, or armored vehicles.

The PATS-70 also has vast potential as an intermediate back shop tool. The PATS-70A development will support testing of armament related LRUs. This functionality can be augmented to include a variety of DoD LRUs and Weapons Replaceable Assemblies (WRAs).

Summary

The PATS-70 started deployment earlier this year having successfully completed a rigorous qualification and validation phase in 2013. The PATS-70 program demonstrated that Engineering-Design partnerships and cooperation can help the Government to better support the warfighter. By partnering with Industry and combining the domain expertise of the 309th and Marvin Test Solutions, the Air Force was able to deploy a PATS-70 test set that meets the warfighter's needs on-time and on-budget.

Acknowledgements

This work was made possible by the Portable Test Station program of the 309th Maintenance Wing, 309th Software Maintenance Group, 516th Squadron, MXDED Flight at Hill Air Force Base, Utah.

ABOUT THE AUTHORS

Benjamin Chase is an electrical engineer at the 309th Software Maintenance Group at the Ogden Air Logistics Complex, Hill Air Force Base, Utah. He is part of the Portable Automated Test Station (PATS-70) Training and Production team. Chase holds a Bachelor of Science in electrical engineering from Utah State University.

7278 4th St.
Hill AFB, UT 84056-5205
Phone: 801-586-2255
Fax: 801-586-0444
E-mail: Benjamin.chase.1@us.af.mil

REFERENCES

1. 516th SMXS/MXDED, "Portable Automated Test Station (PATS)-50 Final Independent Validation and Verification (IV&V), PATS-Analysis Proof of Concept," USAF 309th SMXG, Hill AFB, UT, Rep. 516SMXS-A10PATS-10-001, Jan. 20, 2011.
2. 516th SMXS/MXDED, "Software Design Description For Portable Automated Test Station (PATS-70)," USAF 309th SMXG, Hill AFB, UT, Rep. 516SMXS-A10PATS-11-000, May 4, 2011.
3. 516th SMXS/MXDED, "Operation and Maintenance with Illustrated Parts Breakdown for the Portable Automated Test Station Model 70 (PATS-70)," USAF 309th SMXG, Hill AFB, UT, Rep. PATS70-OM_MANUAL-04FEB2014, Feb. 4, 2014.
4. 516th SMXS/MXDED, "Test Requirements Document (TRD) for A-10C Fuel Quantity Indication System (FQIS)," USAF 309th SMXG, Hill AFB, UT, Rep. PATS70A-TRD-FQIS, March 14, 2014.
5. 516th SMXS/MXDED, "System / Subsystem Requirement Document A-10C Aircraft Portable Automated Test Station Model 70A (PATS-70A)," USAF 309th SMXG, Hill AFB, UT, Rep. PATS70A-SSRD-31JULY2013, July 31, 2013.

NOTES

1. All references to Marvin Test Solutions and/or Marvin products are provided for technical purposes only; no commercial endorsement is implied or intended.

Metamorphic Runtime Checking of Applications Without Test Oracles

Jonathan Bell, Columbia University
Christian Murphy, University of Pennsylvania
Gail Kaiser, Columbia University

Abstract. For some applications, it is impossible or impractical to know what the correct output should be for an arbitrary input, making testing difficult. Many machine-learning applications for "big data", bioinformatics and cyberphysical systems fall in this scope: they do not have a test oracle. Metamorphic Testing, a simple testing technique that does not require a test oracle, has been shown to be effective for testing such applications. We present Metamorphic Runtime Checking, a novel approach that conducts metamorphic testing of both the entire application and individual functions during a program's execution. We have applied Metamorphic Runtime Checking to 9 machine-learning applications, finding it to be on average 170% more effective than traditional metamorphic testing at only the full application level.

Introduction

During software testing, a "test oracle" [1] is required to indicate whether the output is correct for the given input. Despite a recent interest in the testing community in creating and evaluating test oracles, still there are a variety of problem domains for which a practical and complete test oracle does not exist.

Many emerging application domains fall into a category of software that Weyuker describes as "Programs which were written in order to determine the answer in the first place. There would be no need to write such programs, if the correct answer were known [2]." Thus, in the general case, it is not possible to know the correct output in advance for arbitrary input. In other domains, such as optimization, determining whether the output is correct is at least as difficult as it is to derive the output in the first place, and creating an efficient, practical oracle may not be feasible.

Although some faults in such programs - such as those that cause the program to crash or produce results that are obviously wrong to someone who knows the domain - are easily found, and partial oracles may exist for a subset of the input domain, subtle errors in performing calculations or in adhering to specifications can be much more difficult to identify without a practical, general oracle.

Much recent research addressing the so-called "oracle problem" has focused on the use of metamorphic testing [3]. In metamorphic testing changes are made to existing test inputs in such a way (based on the program's "metamorphic properties") that it is possible to predict what the change to the output should be without a test oracle.

That is, if program input I produces output O, additional test inputs based on transformations of I are generated in such a manner that the change to O (if any) can be predicted. In cases where the correctness of the original output O cannot be determined, i.e., if there is no test oracle, program defects can still be detected if the new output O is not as expected when using the new input.

For a simple example of metamorphic testing (where we do have a test oracle), consider a function that calculates the standard deviation of a set of numbers. Certain transformations of the set would be expected to produce the same result: for instance, permuting the order of the elements should not affect the calculation, nor should multiplying each value by -1. Furthermore, other transformations should alter the output, but in a predictable way: if each value in the set were multiplied by 2, then the standard deviation should be twice that of the original set.

Through our own past investigations into metamorphic testing [4] [5] [6], we have garnered three key insights. First, the metamorphic properties of individual functions are often different than those of the application as a whole. Thus, by checking for additional and different relationships, we can reveal defects that would not be detected using only the metamorphic properties of the full application. Second, the metamorphic properties of individual functions can be checked in the course of executing metamorphic tests on the full application. This addresses the problem of generating test cases from which to derive new inputs, since we can simply use those inputs with which the functions happened to be invoked within the full application. Third, when conducting tests of individual functions within the full running application in this manner, checking the metamorphic properties of one function can sometimes detect defects in other functions, which may not have any known metamorphic properties, because the functions share application state.

Approach

In order to realize these improvements, we present a solution based on checking the metamorphic properties of the entire program and those of individual functions (methods, procedures, subroutines, etc.) as the full program runs. That is, the program under test is not treated only as a black box, but rather metamorphic testing also occurs within the program, at the function level, in the context of the running program. This will allow for the execution of more tests and also makes it possible to check for subtle faults inside the code that may not cause violations of the full program's metamorphic properties and lead to apparently reasonable output (remember we cannot check whether that output is correct, since there is no test oracle).

In our new approach, additional metamorphic tests are logically attached to the individual functions for which metamorphic properties have been specified. Upon a function's execution when it happens to be invoked within the full program, the corresponding function-level tests are executed as well: the arguments are modified according to the function's metamorphic properties, the function is run again (in a sandbox, not shown) in the same program state as the original, and the output of the function with the original input is compared to that of the function with the modified input. If the result is not as expected according to the metamorphic property, then a fault has been exposed.

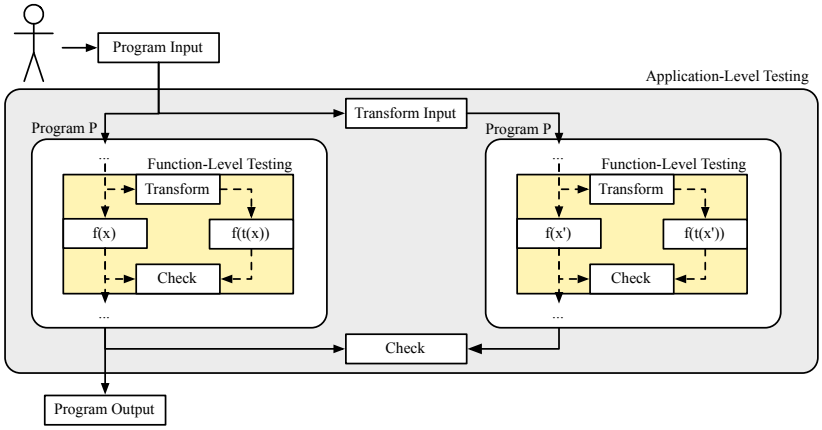


Figure 1: Model of Metamorphic Runtime Checking of program P and one of its constituent functions, f. Metamorphic Runtime Checking combines program-level metamorphic testing with function-level metamorphic checking, performing such checking automatically.

As shown in Figure 1 the tester provides a program input to a Metamorphic Runtime Checking framework, which then transforms it according to the metamorphic property of the program P (for simplicity, this diagram only shows one metamorphic property, but a program may, of course, have many). The framework then invokes P with both the original input and the transformed input; as seen at the bottom of the diagram, when each program invocation is finished, the outputs can be checked according to the property.

While each invocation of P is running, metamorphic properties of individual functions can be checked as well. As shown on the left side of Figure 1, in the invocation of P with the original program input, before a function f is called, its input x can be transformed according to one of the function's metamorphic properties to give t(x). The function is called with each input, and then f(t(x)) is evaluated according to the original value of f(x) to see if the property is violated.

Meanwhile, in the additional invocation of P (right side of the diagram), function-level metamorphic testing still occurs for function f, this time using input x', which results from the transformed program input to P. In this case, f(t(x')) and f(x') are compared.

						# of Metamorphic Properties identified at the level of:	
Application	**Domain**	**Language**	**LOC**	**Functions**	**Invariants**	**Application**	**Function**
C4.5	classification	C	5,285	141	27,603	4	40
GAFFitter	optimization	C++	1,159	19	744	2	11
JSim	simulation	Java	3,024	468	306	2	12
K-means	clustering	Java	717	46	137	4	12
LDA	topic modeling	Java	1,630	103	1,323	4	28
Lucene	information retrieval	Java	661	57	456	4	26
MartiRank	ranking	C	804	19	3,647	4	15
PAYL	anomaly detection	Java	4,199	164	19,730	2	40
SVM	classification	Java	1,213	49	2,182	4	4

Table 1: Listing of applications studied

In this example, if we used only the application-level property of P, we would run just one test. However, by also considering P's function f with one specified metamorphic property, we can now check two properties and run a total of three tests. This combined approach also allows us to reveal subtle faults at the function level that may not violate application-level properties. Our study shows that this sensitivity gain can increase the effectiveness of metamorphic testing by up to 1,350% (on average, 170%).

Evaluation

To evaluate the effectiveness of Metamorphic Runtime Checking at detecting faults in applications without test oracles, we compare it to runtime assertion checking using program invariants (a state-of-the art technique). When used in applications without test oracles, assertions can detect some programming bugs by checking that function input and output values are within a specified range, the relationships between variables are maintained, and a function's effects on the application state are as expected [7]. While satisfying the invariants does not ensure correctness, any violation of them at runtime indicates an error.

The experiments described in this section seek to answer the following research questions:

1. Is Metamorphic Runtime Checking more effective than using runtime assertion checking for detecting faults in applications without test oracles?
2. What contribution do application-level and function-level metamorphic properties make to the effectiveness of Metamorphic Runtime Checking?
3. Is Metamorphic Runtime Checking suitable for practical use?

In these experiments, we applied both runtime assertion checking and Metamorphic Runtime Checking to nine real-world applications that are representative of different domains that have no practical, general test oracles: supervised machine learning, unsupervised machine learning, data mining, discrete event simulation, and NP-hard optimization. The applications are described (along with the number of invariants, function-level and application-level properties) in Table 1.

To create the set of invariants that we could use for runtime assertion checking, we applied the Daikon invariant detector tool [8] to each application. To identify the application-level metamorphic properties for the experiment, we followed the guidelines set forth in [4], which categorizes the types of properties that applications in these domains tend to exhibit.

To identify function-level properties, we inspected the source code and hand-annotated the functions that we expected to exhibit the types of properties described in [4]. To ensure that the properties were not limited to only the ones that we could think of, some of the function-level metamorphic properties used in this experiment are based on those used in other, similar studies such as [9], [10] and [11].

Methodology

To determine the effectiveness of the testing techniques, we used mutation analysis to systematically insert faults into the source code of the applications described above, and then determined whether the mutants could be killed (i.e., whether the faults could be detected) using each approach. Mutations that yielded a fatal runtime error, an infinite loop, or an output that

was clearly wrong (for instance, not conforming to the expected output syntax or simply being blank) were discarded since any reasonable approach would detect such faults.

We also did not consider "equivalent mutants" for which the inputs used in the experiment produced the same program output as the original, unmutated version, e.g., those mutants that were not on the execution path for any test case or that would not have been killed with an oracle for these inputs.

For each mutated version, we conducted runtime assertion checking with the invariants detected by Daikon. If any invariant was violated, the mutant was considered killed. We then performed Metamorphic Runtime Checking on the same mutated versions to determine whether any of the specified metamorphic properties were violated. The inputs used for mutation analysis were the same as those used for detecting invariants and verifying metamorphic properties.

Figure 2 summarize the results of our experiment evaluating the efficacy of Metamorphic Runtime Checking. Overall, Metamorphic Runtime Checking was more effective, killing 1,602 (90.4%) of the mutants in the applications, compared to just 1,498 (84.5%) for assertion checking.

Broadly speaking, Metamorphic Runtime Checking was more effective at killing mutants that related to operations on arrays, sets, collections, etc. However, further analysis could characterize the types of faults each approach is most suitable for detecting, but it follows, that runtime assertion checking and Metamorphic Runtime Checking should be used together for best results. When used in combination in our experiments, they were able to kill 95% of the mutants (totaling across all applications): only 88 of the 1,772 survived.

To understand the factors that impacted the efficacy of Metamorphic Runtime Checking, we performed a deeper analysis of the contribution of the separate mechanisms. We first determined the number of mutants killed only by application-level properties, then the number killed only by function-level properties. Table 2 shows these results.

On average, we saw a 170% improvement in the number of mutants killed when combining application-level properties with function-level properties. The variance in improvement was very large, however, showing a striking improvement of 1,350% in PAYL, while showing smaller improvement in C4.5 and MartiRank. There was no improvement at all in the JSim and LDA applications, because application-level properties had already been able to kill all mutants.

We believe that this improvement is attributed primarily to our increase in: the number of properties identified (scope); the number of tests run (scale); and the likelihood that a fault would be detected (sensitivity).

The improvement in the scope of metamorphic testing was particularly clear in the anomaly-based intrusion detection system PAYL. We were only able to identify two application-level metamorphic properties because it was not possible to create new program inputs based on modifying the values of the bytes inside the payloads, since the application only allowed for syntactically and semantically valid inputs that reflected what it considered to be "real" network traffic.

These two properties were only able to kill two of the 40 mutants. However, once we could use Metamorphic Runtime Checking to run metamorphic tests at the function level, we were able to identify many more properties that involved changing the byte arrays that were passed as function arguments, thus revealing 27 additional faults.

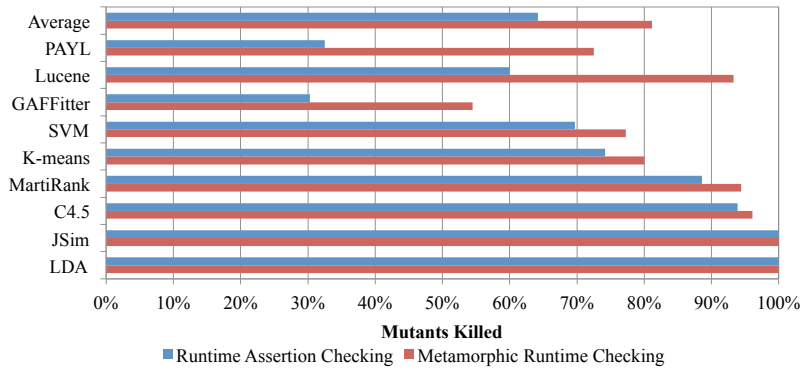


Figure 2: Results of mutation analysis comparing metamorphic runtime checking and runtime assertion checking. Metamorphic runtime checking was on average more effective.

		Mutants Killed By				
Application	**Total Mutants**	**Application-level Properties Only**	**Function-level Properties Only**	**Both Types**	**Not Killed**	**MRC % Improvement**
C4.5	856	133	37	653	33	4.71%
GAFFitter	66	2	14	20	30	63.64%
K-means	35	6	11	11	7	64.71%
JSim	36	14	0	22	0	0.00%
LDA	24	2	0	22	0	0.00%
Lucene	15	5	3	6	1	27.27%
MartiRank	413	298	22	70	23	5.98%
PAYL	40	0	27	2	11	1350.00%
SVM	287	69	23	130	65	11.56%
Average	197	59	15	104	19	169.76%

Table 2: Number of Mutants Killed by Different Types of Metamorphic Properties

Likewise, we were able to increase the scale of metamorphic testing by running many more test cases. For instance, in MartiRank, even though we specified function-level properties for only a handful of functions, many of those are called numerous times per program execution, meaning that there are many opportunities for the property to be violated.

Another reason why function-level properties were able to kill mutants not killed by application-level properties is that we were able to improve the sensitivity in terms of the ability to reveal more subtle faults, as seen in GAFFitter. In the function to calculate the "fitness" of a given candidate solution in the genetic algorithm, i.e., how close to the optimal solution (target) a candidate comes, one of the metamorphic properties is that permuting the elements in the candidate solution should not affect the result, since it is merely taking a sum of all the elements.

If, for instance, there is a mutation such that the last element is omitted from the calculation, then the metamorphic property will be violated since the return value will be different after the second function call. However, at the application level, such a fault is unlikely to be detected, since the metamorphic prop-

erty simply states that the quality of the solutions should be increasing with subsequent generations. Even though the value of the fitness is incorrect, it would still be increasing (unless the omitted element had a very large effect on the result, which is unlikely), and the property would not be violated.

Performance Overhead

Although Metamorphic Runtime Checking using function-level properties is able to detect faults not found by metamorphic testing based on application-level properties alone, this runtime checking of the properties comes at a cost, particularly if the tests are run frequently. In application-level metamorphic testing, the program needs to be run one more time with the transformed input, and then each metamorphic property is checked exactly once (at the end of the program execution). In Metamorphic Runtime Checking, however, each property can be checked numerous times, depending on the number of times each function is called, and the overhead can grow to be much higher.

During the studies discussed above, we measured the performance overhead of our C and Java implementations of the Metamorphic Runtime Checking framework. Tests were conducted on a server with a quad-core 3GHz CPU running Ubuntu 7.10 with 2GB RAM. On average, the performance overhead for the Java applications was around 3.5ms per test; for C, it was only 0.4ms per test. This cost is mostly attributed to the time it takes to create sandboxes (so the side-effects of function-level metamorphic testing do not impact application-level testing).

This impact can be substantial from a percentage overhead point of view if many tests are run in a short-lived program. For instance, for C4.5, the overhead was on the order of 10x, even though in absolute terms it was well under a second. However, for most programs we investigated in our study, the overhead was typically less than a few minutes, which we consider a small price to pay for being able to detect faults in programs with no test oracle.

Future work could investigate techniques for improving the performance of a Metamorphic Runtime Checking framework. Previously we considered an approach whereby tests were only executed in application states that had not previously been encountered, and showed that performance could be improved even when the functions are invoked with new parameters up to 90% of the time [12]. It may be possible to reduce the overhead even more, for instance by running tests probabilistically (our framework already allows the tester to specify a probability for checking each function-level metamorphic property, but we turned that off for the studies presented here).

Limitations

We used Daikon to create the program invariants for runtime assertion checking. Although in practice invariants are typically generated by hand, and some researchers have questioned the usefulness of Daikon-generated invariants compared to those generated by humans [13], we chose to use the tool so that we could eliminate any human bias or human error in creating the invariants.

Additionally, others have independently shown that metamorphic properties are more effective at detecting defects than manually identified invariants [14], though for programs on a smaller scale than those in our experiment (a few hundred lines, as opposed to thousands as in many of the programs we studied).

The ability of metamorphic testing to reveal failures is clearly dependent on the selection of metamorphic properties. However, we have shown that a basic set of metamorphic properties can be used without a particularly strong understanding of the implementation - the authors knew essentially nothing about the target systems or their domains beyond textbook generality; the use of domain-specific properties from the developers of these systems might reveal even more failures [15].

Conclusion

As shown in our empirical studies, Metamorphic Runtime Checking has three distinct advantages over metamorphic testing using application-level properties alone. First, we are able to increase the scope of metamorphic testing, by identifying properties for individual functions in addition to those of the entire application. Second, we increase the scale of metamorphic testing by running more tests for a given input to the program. And third, we can increase the sensitivity of metamorphic testing by checking the properties of individual functions, making it possible to reveal subtle faults that may otherwise go unnoticed.

Acknowledgements

We would like to thank T.Y. Chen, Lori Clarke, Lee Osterweil, Sal Stolfo, and Junfeng Yang for their guidance and assistance. Sahar Hasan, Lifeng Hu, Kuang Shen, and Ian Vo contributed to the implementation of the Metamorphic Runtime Checking framework.

Bell and Kaiser are members of the Programming Systems Laboratory, funded in part by NSF CCF-1302269, NSF CCF-1161079, NSF CNS-0905246, and NIH U54 CA121852.

Homeland Security

The Department of Homeland Security, Office of Cybersecurity and Communications (CS&C) is responsible for enhancing the security, resiliency, and reliability of the Nation's cyber and communications infrastructure and actively engages the public and private sectors as well as international partners to prepare for, prevent, and respond to catastrophic incidents that could degrade or overwhelm these strategic assets. CS&C is seeking dynamic individuals to fill critical positions in:

- Cyber Incident Response
- Cyber Risk and Strategic Analysis
- Networks and Systems Engineering
- Computer and Electronic Engineering
- Digital Forensics
- Telecommunications
- Program Management and Analysis
- Vulnerability Detection and Assessment

To learn more about the DHS, Office of Cybersecurity and Communications, go to www.dhs.gov/cybercareers. To apply for a vacant position please go to www.usajobs.gov or visit us at www.DHS.gov.

ABOUT THE AUTHORS

Jonathan Bell is a Ph.D. student in Software Engineering at Columbia University. His research interests include software testing, program analysis, and fault reproduction. He's received an M Phil, MS and BS in Computer Science from Columbia University.

Dept. of Computer Science
Columbia University
New York, NY 10027
Phone: 212-939-7184
E-mail: jbell@cs.columbia.edu

Christian Murphy is an Associate Professor of Practice and Director of the Master of Computer and Information Technology program at The University of Pennsylvania. His primary interests are software engineering, systems programming, and mobile/embedded computing. He received his Ph.D. in Computer Science from Columbia University.

Dept. of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104
Phone: 215-898-0382
E-mail: cdmurphy@cis.upenn.edu

Gail E. Kaiser is a Professor of Computer Science at Columbia University and a Senior Member of IEEE. Her research interests include software reliability and robustness, information management, social software engineering, and software development environments and tools. She has served as a founding associate editor of ACM TOSEM and as an editorial board member for IEEE Internet Computing. She received her Ph.D. and MS from CMU and her ScB from MIT.

Dept. of Computer Science
Columbia University
New York, NY 10027
Phone: 212-939-7184
E-mail: kaiser@cs.columbia.edu

REFERENCES

1. Pezzé, M. and M. Young, Software Testing and Analysis: Process, Principles and Techniques. 2007: Wiley.
2. Weyuker, E.J., On testing non-testable programs. Computer Journal, 1982. 25(4): p. 465-470.
3. Chen, T.Y., S.C. Cheung, and S.M. Yiu, Metamorphic testing: a new approach for generating next test cases. 1998, Dept. of Computer Science, Hong Kong Univ. of Science and Technology.
4. Murphy, C., et al., Properties of Machine Learning Applications for Use in Metamorphic Testing, in Proc. of the 20th International Conference on Software Engineering and Knowledge Engineering (SEKE). 2008. p. 867-872.
5. Murphy, C., et al., On Effective Testing of Health Care Simulation Software, in Proc. of the 3rd International Workshop on Software Engineering in Health Care. 2011.
6. Murphy, C., K. Shen, and G. Kaiser, Automated System Testing of Programs without Test Oracles, in Proc. of the 2009 ACM International Conference on Software Testing and Analysis (ISSTA). 2009. p. 189-199.
7. Nimmer, J.W. and M.D. Ernst, Automatic generation of program specifications, in Proc. of the 2002 International Symposium on Software Testing and Analysis (ISSTA). 2002. p. 232-242.
8. Ernst, M.D., et al., Dynamically discovering likely programming invariants to support program evolution, in Proc. of the 21st International Conference on Software Engineering (ICSE). 1999. p. 213-224.
9. Barus, A.C., et al., Testing of Heuristic Methods: A Case Study of Greedy Algorithm. Lecture Notes in Computer Science, 2011. 4890: p. 246-260.
10. Kanewala, U. and J.M. Bieman, Techniques for Testing Scientific Programs Without an Oracle, in Proc. of the 2013 International Workshop on Software Engineering for Computational Science and Engineering. 2013.
11. Cheatham, T.J., J.P. Yoo, and N.J. Wahl, Software testing: a machine learning experiment, in Proc. of the ACM 23rd Annual Conference on Computer Science. 1995. p. 135-141.
12. Murphy, C., et al., Automatic Detection of Previously-Unseen Application States for Deployment Environment Testing and Analysis, in Proc. of the 5th International Workshop on Automation of Software Test (AST). 2010.
13. Polikarpova, N., I. Ciupa, and B. Meyer, A comparative study of programmer-written and automatically inferred contracts, in Proc. of the 2009 International Symposium on Software Testing and Analysis (ISSTA). 2009. p. 93-104.
14. Hu, P., et al., An empirical comparison between direct and indirect test result checking approaches, in Proc. of the 3rd International Workshop on Software Quality Assurance. 2006. p. 6-13.
15. Xie, X., et al., Application of Metamorphic Testing to Supervised Classifiers, in Proc. of the 9th International Conference on Quality Software (QSIC). 2009. p. 135-144.

Dealing With the Time Crunch in Software Testing

Randall W. Rice, Rice Consulting Services, Inc.

Abstract. In my research involving thousands of software testers over the last 15 years, several challenges are mentioned as critical pain points in nearly every interview and survey. One of the most common complaints from software testers is that, "There is no way we can do a complete test in the time we are given."
The mismatch of scope and time causes stress for software testers and often leaves them in a no-win position. They might work very hard in the short amount of time they have been allotted; only to be blamed at times for not finding certain defects once the software is released to users.

Another impact of time constraints is that schedule delays in development and other project activities can erode the planned test schedule. In those situations, the overall delivery deadline remains firm even when the testers get access to the software later than planned. This schedule crunch is another reason testers feel stress and may be forced to skip some tests.

The truth of the matter is that no matter how much time you have, it is not enough time to test all functions and combinations of functions in most software. In essence, all testing is sampling, so it becomes very important where we draw the boundaries in testing and where we take our samples.

In this article, I will explore some ways to deal with the time constraints in software testing and still have a reasonable level of test coverage and overall confidence in the quality of the application being delivered.

The Impact of Time Constraints on Testing

The lack of time for testing is a symptom of deeper project, process and organizational issues. However, that doesn't eliminate the impact of the time crunch. The impact is seen in the following areas.

1. Lack of Coverage

Test coverage can be seen in a variety of ways: code coverage, requirements coverage, and test case coverage, to name a few of them. If you can't cover enough of these items, it's hard to give a reasonable level of confidence in the software.

2. Lack of Quality

If you can't test to the levels needed to assess the quality of the application, the test will be incomplete and defects will be missed. Overall software quality suffers.

The reality is that just because testing stops, software problems (i.e., defects) will still continue. It's just that the users will eventually find them. Sometimes the application is mission critical or safety critical, so to miss finding defects due to lack of test time can be a very high risk.

However, even "complete" testing does not necessarily equate to high quality applications. This seems to be contradictory. After all, why do we perform testing if we can't make the assurance of quality?

What we may deem as "complete" may just be an illusion. There is always one more thing (or millions of things) you can test.

3. Poor Morale

It really does kill morale to ask people to do something as challenging as testing, then not give enough time, tools or people to get the job done. Even worse is when the testers get blamed for defects when the time has been cut short, or the amount of work is impossible to complete even in a thousand years.

Test teams that are starved of resources often don't see the possibility of applying what would normally be considered sound testing techniques. The comments are along the lines of, "Oh, we would never have time to document a test plan. We barely have enough time to test as it is."

Never mind the idea that a test plan could actually save time by making sure the resources are used in the most efficient ways, or that a workable scope balanced by risk could be defined – all of which save time.

4. Missed Expectations

Sometimes people have the expectation that exhaustive testing is possible with scarce resources and that testers would just waste time if they had more of it. Or, management believes sometimes just the opposite, "Since we can't test everything, why bother at all?" This is primarily a perception issue.

Many times, project deadlines are established with the expectation that an arbitrary allocation of time will be enough time for testing, even though there is no clarity about which kinds of tests are needed, or even what the scope of testing will be. It is only after the design of the system becomes clear that the scope of the project becomes clear.

So when defects are discovered, the project schedules often slip and so does the management attitude toward the testing effort. These projects are either delivered later than planned, or delivered on-time with high levels of defects.

An Exercise in Priorities

Realizing that you could have all the time you desired and still not have enough time to completely test anything of even moderate complexity, the issue becomes how to define the scope of testing.

Key ways that testing fails are to 1) do the wrong tests, 2) do the right tests in the wrong ways and 3) fail to do enough of the right tests. So, in the third way mentioned, lack of time is a root cause. In the first two ways, it is possible to leverage the time available for testing with efficient test design and test execution, combined with reviews and inspections.

If we recognize the fact we can't test everything, the challenge then becomes to pick the right things to test. Picking what to test is like packing a suitcase. You only have some much room and so much weight allowance to work with. First, you pack the big and important things, like the main clothes you need. Next, you pack the smaller important items, like toiletries. Finally, if you have room and weight, you can pack your fuzzy slippers.

A key question is "How do we prioritize?" Some might say by risk. Others might say "by what we getting paid or pressed the most to deliver." It is important to note that criticality is not the same as risk. Risk is a potential loss while criticality is importance. Sometimes risk and criticality has a direct relationship, such as a function that is safety-critical. However, in other cases, a function may be critical to certain users yet pose no potential negative impact.

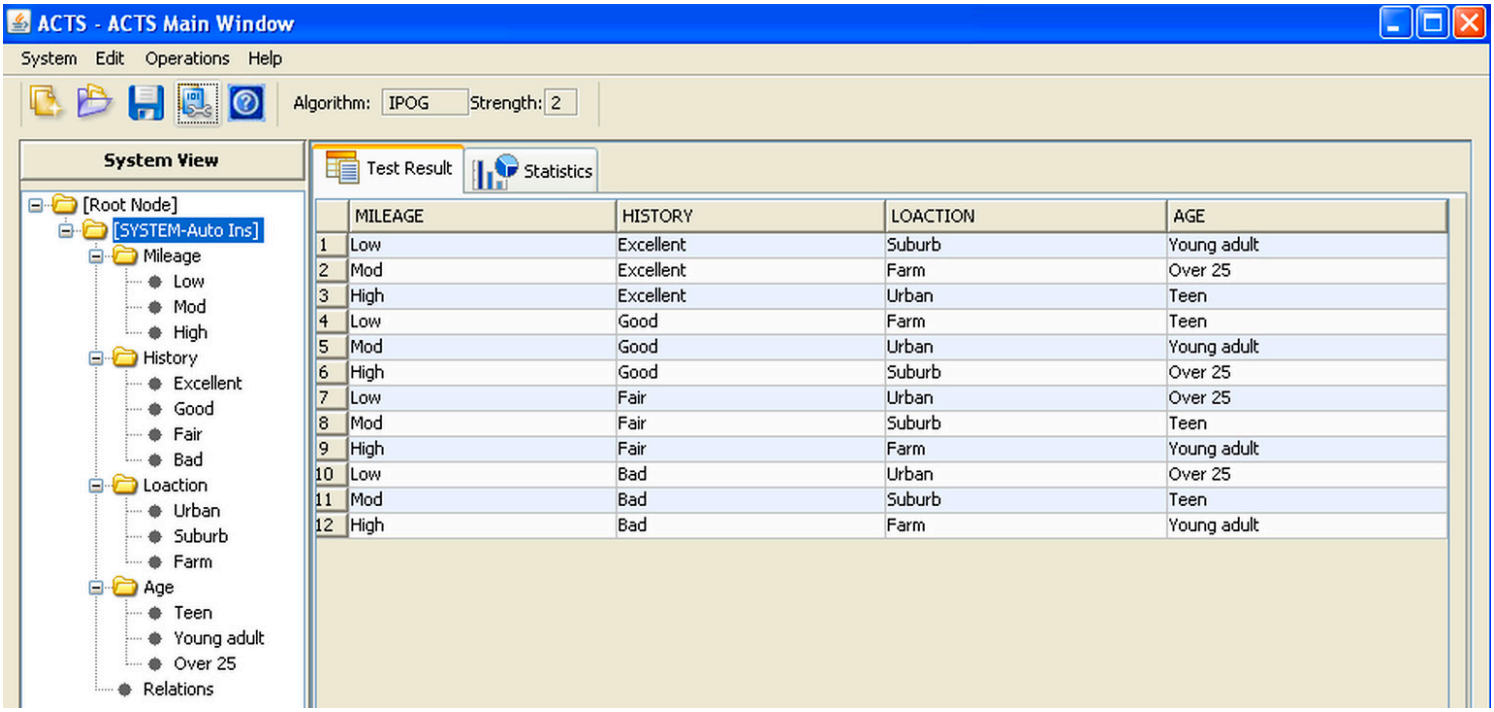


Figure 1: ACTS Combinatorial Test Design Tool

Yet there is more to efficiency than designing and prioritizing the right tests. You also have to design and perform them efficiently.

Going back to the suitcase analogy, how you pack the suitcase makes a big difference in how much you can pack in it. Some people are masters at packing a suitcase. They know how to roll their clothes, how to fill small spaces and so forth.

Likewise, testers need an efficient process for testing. One example of this is using combinatorial techniques to get the most coverage from the fewest number of test cases. NIST has a wonderful free tool called ACTS (Advanced Combinatorial Testing System) which generates test cases based on combinatorial test design [1]. The advantage is that you get high levels of test coverage with a minimal number of tests (Figure 1).

In this example, there are 72 possible condition combinations. After applying the pairwise algorithm, we are left with twelve cases. While this is a great reduction in the total number of test cases, care must be taken to review the tests and supplement them with tests that may be important, but not generated by the tool. You will also need to manually define the expected results, which requires time.

What About Test Estimation?

Test estimation is inherently tricky because it is an approximation. Some people are very challenged in providing estimates of the testing effort due to lack of knowledge about what is to be eventually tested. In some cases, the problem of test estimation boils down to poor estimating methods. By "poor estimating methods" I mean that an organization is not handling test estimates in a way that deals with the facts and risks as they really are. Instead, they become overly optimistic or just pull a number out of thin air for the testing effort.

In those cases where the product is already developed, such as COTS applications, test estimates can still be difficult if you don't have access to the software yet. It is also a surprise to management sometimes that the time needed to test a COTS product may actually be a multiple of the time and cost needed to acquire and install the product.

In estimation, accuracy is not so much the issue as is getting the recipient of the estimate to accept your estimate, which you honestly believe as the true effort and time needed for testing.

Let's say you work hard to create an accurate test estimate for a project. You estimate the test to take five people three weeks to complete. Your manager may assume there is some padding in your estimate, so she "readjusts" your estimate to be two weeks for testing with the same five people. In this example, the accuracy of the original estimate really doesn't matter. When it's all said and done, you and your team are still expected to do three weeks of testing in two weeks time!

However, the next time you have to provide an estimate, you will remember that estimate reduction and will likely inflate your initial estimate because you know it will be cut. This is how the dysfunction around estimation is propagated.

That is the dark side of test estimation. There is a better way but it requires a healthy, eyes-open attitude on everyone's part to re-adjust the estimate when necessary or make adjustments in scope and resources.

Another point of awareness and agreement is that an estimate is not completely accurate – it is a best approximation based on experience and other factors.

The problem with highly precise estimates is that they often assume things will go well and without delays. The reality is seldom that trouble-free.

To deal with the unexpected occurrences that cause actual test times and resources to differ from their estimates, contingencies are needed. A helpful way to think of contingencies is in the form of reserves.

Reserves are simply more people, tools, and other things available in case of unforeseen events. Contingencies are your "Plan B" which describe your options should a certain risk materialize.

Some people might think of reserves as "padding" an estimate, but my experience is that reserves and contingencies are needed as a safety net for when schedule crunch occurs. It is very important not to squander reserves in time-wasting activities. For example, if your team spends forty hours on work that could easily be done in ten hours, that is an inefficient use of time and is not the purpose of reserves.

Dealing With the Time Crunch

Here are a few ways to deal with short test timeframes.

1. Get Good at Prioritizing

Unfortunately, most of the discussion around prioritizing testing is all about risk. While risk is a key concern, it is not the only way to prioritize testing. Risk-based testing is a sound practice in many cases, but there are risks, even to risk-based testing. One such risk is that our risk assessment may be totally wrong due to factors we cannot foresee.

In addition, risk-based testing becomes ineffective when everything is a high risk. You will need another way to prioritize tests in that situation. To prioritize means to determine what is important. But the big question is "Important to whom?"

Priorities for testing are often a reflection of the stakeholders, such as project management, senior organizational management, customers and users. One big trap we fall into is thinking that the priorities given to development and testing groups are the real priorities.

Testers are not in charge of the priorities of a project, and therefore, not really in charge of the priorities of testing. Even though a tester may write the test plan and set the test schedule, they may not reflect the true project and stakeholder priorities. Like the traveller who packs someone else's bag (don't really do that – you can get in trouble for it!), you may include the important things to you, but not to the other person. It is all about who owns the bag.

You can prioritize based on:

Mission need – What solves a key problem or delivers needed functionality right away?

Stakeholder needs – Which products and features do users and other stakeholders value most?

Risk – Where might problems most likely appear, and where might they carry the most impact?

Management directive – Ultimately, management sets project priorities. However, on occasion, someone in management decides his or her pet project or feature is most important regardless of business value or risk. Therefore, it becomes a high priority in testing.

Experience – Where have we seen problems in the past that we don't want to repeat?

Sampling – There are many ways to take samples in testing, such as random samples, customer samples, and so forth. The idea is that if you find a concentration of problems in one area, there will likely be others in that same area. This is known as "defect clustering."

2. Management Needs to Understand The Tradeoffs

The management influence on schedules and priorities drives much of the testing time challenge. Many organizations are schedule-driven in which the software delivery schedule takes precedence over other decisions.

There are four key factors in the time challenge:

1. Resource levels
2. Schedules
3. Workload balance
4. Process efficiency

In many cases, deadlines are necessary and important. Management, however, must allocate the right number of resources to get the job done whether it is in development, testing or anything else. At the time of this writing, we have been through an economy where the theme in organizations has been to learn how to do more with fewer resources. That includes people, tools and hardware.

Gerald Weinberg writes about "The Law of Raspberry Jam." This law says, "The wider you spread something the thinner it gets [2]." Just like spreading jam on toast, you can have such a small amount that after spreading it over a piece of toast, you can hardly taste it at all.

Senior management needs to understand and embrace this concept when defining their expectations of what can be accomplished with given resources. There comes a point when project resources can be spread so thin that you can't even tell any difference was made at all.

3. Optimize the Testing Process

Many of the perceived time problems in testing may be due to poor workload balance and inefficient testing approaches. It is management's job to fix inefficient or ineffective processes because people in the trenches don't have the authority or control to make major process changes.

People at the grassroots level may have the opportunity to work smartly, but sometimes that is thwarted by management's decisions to do things that are inefficient and ineffective.

Here are three ways to optimize the testing process. By doing these things, you perform testing more efficiently, which takes less time and often finds more defects as compared to non-optimized test methods.

Optimization Technique #1 - Apply Sampling Techniques

Going back to the idea stated earlier in this article that all testing is sampling, it can be very helpful to learn how to sample for defects.

Testers face a "needle in the haystack" problem. We are looking for something very small in something very large. I am intrigued at how some gold miners deal with a similar challenge, and how those techniques can be applied to software testing and finding defects.

Like testing, gold mining is also a very expensive undertaking. Even a small operation can cost thousands of dollars per day to move layer upon layer of dirt to get close to finding gold. Unfortunately, there are no signs that read, "Dig here for gold."

One interesting and effective technique to know where to dig for gold is to take soil samples by drilling. That is the best way to see a) if there is any gold in the immediate soil and b) how dense the gold concentration may be, c) how deep the gold is, and d) the quality of the gold. If the soil sample shows gold 20 feet or more below the surface, it will probably cost too much to remove the dirt above it.

Like in sampling for gold, in testing, we can sample in various areas of a system with functional testing, we can sample lines of code with structural testing, we can perform random tests either manually or with tools, or we can sample based on risk, just to name a few possibilities.

Sampling can help us find those pockets or clusters of defects. This can lead us to the best places to focus testing for the little time we have. Finding high concentrations of defects in software is like striking gold in that you get a very high return on investment for testing.

Optimization Technique #2 - Apply the Pareto Principle

The Pareto Principle (the 80/20 rule) that says you can get the majority of value from the minority of people, time or effort. In fact, my experience is that it is possible to get more than 90% of the value of some efforts in a very short time frame.

As an example, once I was asked to perform an eight-week test in two weeks. So, the client and I worked together to focus on key areas and skip minor areas. The test design was based on critical user workflows and modular tests that yielded many more defects than expected.

Optimization Technique #3 – Use Combinatorial Methods

As mentioned earlier, tools such as ACTS and model-based tools can help greatly by intelligently combining test conditions into the most efficient set of tests. If you choose not to use a combinatorial tool, even a basic decision table can help reduce the number of tests without sacrificing coverage.

4. Agile Methods Can Help

I take great care with how I portray agile methods to avoid promoting agile techniques as the solution for all software project problems.

When applied pragmatically, agile methods can be a great help in getting the right things done quickly. When applied haphazardly or in the wrong cultures, agile can lead to disaster faster.

The good aspect of agile is that people learn to minimize the meetings, the documentation and the project overhead to focus on deliverables. The challenging part of agile is to keep the knowledge we really need because of the lesser emphasis on documentation. Agility is all about dealing with change gracefully and focusing on the right things - do that, and you'll be fine.

5. The Right Tools Can Help

Tools can save time when used in the right ways. Test design tools such as ACTS (Figure 1) can save huge amounts of time as compared to designing tests manually. The generated tests can then be imported into test execution tools to automate the mundane tests and help in regression testing – given that you don't have to spend large amounts of time on tool implementation and maintenance issues.

The time savings seen from tools is often the result of an existing test tool implementation effort that is well-designed. Initial tool efforts typically take more time because you are learning as you implement.

6. Know When You Are Over-testing

Whether it is your status report, test plan, or anything else, there comes a time when the finish point needs to be declared and then move on. It's like the old saying that "projects are 99% done forever."

Seth Godin says, "Ship often. Ship lousy stuff, but ship. Ship constantly [3]." I don't concur with *all* of that (like the shipping of lousy stuff), but I do get the sentiment. The main point is to overcome the resistance of things that keep us from making progress and to deliver something of value.

How about this instead? Ship often to the *right* people with the quality they need. Some people are happy to get new features, warts and all. They will give you great feedback and will still be raving fans.

Many times, perception is reality. If a software project is delivered with obvious defects, users don't care who is to blame. They experience frustration and want the problems fixed "now." It's the old maxim that says, "There's never enough time to do the job right, but there's always time to do it again."

As an example, an update to Apple's iOS8 caused major problems for users in that they were unable to make phone calls with an iPhone after the update was installed. Apple recalled the update very quickly, but not without reputation damage. For the first time I can recall, a major news outlet identified the QA Manager of a defective product by name [4].

It's not clear why the iOS8.0.1 defects were not found – whether it was a lack of test time or some other root cause. The result, however, shows what happens when defects go "big time." Blaming the QA Manager and testers is not the solution and is not helpful, since testers usually don't make the decision to release software.

Plan of Action

These are not sequential steps, but rather ways to implement some of the ideas in this article.

Lay a foundation of expectations at the management level that exhaustive testing is impossible and that the thinner testing resources are spread, the less effective they become. If your estimates differ greatly from those already set for you, be able to justify why you need more time and resources for testing. Risk is a good way to balance this discussion.

Get good at identifying where the risk is in the things you test.

Learn how to sample products to find where defects may be clustering.

Define a workable scope of testing. If you set the scope of testing too large, you won't finish in time. If you set it too small, you won't get the confidence levels you need for deployment.

Optimize your tests.

Work with project managers to build reasonable reserves and contingency plans.

Create a risk-based test planning and reporting process.

Summary

Like any project activity, testing takes a certain amount of time and resources to perform if it is to measure an accurate level of confidence and quality. Exhaustive testing is impossible due to the very high numbers of possible tests required to test all combinations of conditions. By combining dynamic testing with reviews and inspections throughout a project, the effect is often a time and cost savings.

However, it makes a big difference in the time crunch as to how testing is prioritized, designed and performed, as to whether time and resource constraints are dealt with in an effective way.

Good estimates are helpful, but even the best efforts at estimation may fall short of being accurate. For those times, you need some reserves in time and resources as a contingency. You also need a "Plan B" for those times when the time crunch gets really bad.

There are no magic solutions to the time crunch challenge. However, with some expectation management and careful test selection and optimization, you may just find enough time to perform the level of testing that gives an acceptable level of confidence to stakeholders.

ABOUT THE AUTHOR

Randall W. Rice is a leading consultant, author, and speaker in software testing and software quality. He is a popular speaker at international software testing conferences and is co-author with William E. Perry of the book, *Surviving the Top Ten Challenges of Software Testing* and *Testing Dirty Systems*. Randall also serves on the board of directors of the American Software Testing Qualifications Board (ASTQB).

Rice Consulting Services, Inc.
P.O. Box 892003
Oklahoma City, OK 73189
Phone: 405-691-8075
E-mail: rrice@riceconsulting.com

REFERENCES

1. http://csrc.nist.gov/groups/SNS/acts/documents/comparison-report.html
2. Weinberg, Gerald, "The Secrets of Consulting", New York: Dorset House Publishing, 1985
3. http://99u.com/tips/6249/Seth-Godin-The-Truth-About-Shipping
4. http://www.bloomberg.com/news/2014-09-25/apple-s-iphone-software-snafu-has-links-to-flawed-maps.htm

ADDITIONAL RESOURCES

1. Jones, Capers and Bonsignour, Olivier, The Economics of Software Quality, New York: Addison-Wesley, 2012
2. Perry, William E. and Rice, Randall W, Surviving the Top Ten Challenges of Software Testing, New York: Dorset House Publishing, 1997
3. Weinberg, Gerald, Perfect Software, New York: Dorset House Publishing, 2008

CALL FOR ARTICLES

If your experience or research has produced information that could be useful to others, CrossTalk can get the word out. We are specifically looking for articles on software-related topics to supplement upcoming theme issues. Below is the submittal schedule for the areas of emphasis we are looking for:

Supply Chain Assurance
Sep/Oct 2015 Issue
Submission Deadline: Apr 10, 2015

Fusing IT and Real-Time Tactical
Nov/Dec 2015 Issue
Submission Deadline: Jun 10, 2015

Please follow the Author Guidelines for CrossTalk, available on the Internet at <www.crosstalkonline.org/submission-guidelines>. We accept article submissions on software-related topics at any time, along with Letters to the Editor and BackTalk. To see a list of themes for upcoming issues or to learn more about the types of articles we're looking for visit <www.crosstalkonline.org/theme-calendar>.

Combinatorial Coverage as an Aspect of Test Quality

D. Richard Kuhn, NIST,
Raghu N. Kacker, NIST,
Yu Lei, University of Texas Arlington

Abstract. There are relatively few good methods for evaluating test set quality, after ensuring basic requirements-traceability. Structural coverage, mutation testing, and related methods can be used if source code is available, but these approaches may entail significant cost in time and resources. This paper introduces an alternative measure of test quality that is directly related to fault detection, simple to compute, and can be applied prior to execution of the system under test. As such, it provides an inexpensive complement to current approaches for evaluating test quality.

Introduction

How thorough are your tests? This is a vitally important question for mission critical systems, but very difficult to answer with confidence, especially if tests were produced by third-party test developers.

Generally it must be shown that tests track to enumerated requirements, but this is a coarse grained metric. Structural coverage criteria such as statement or branch coverage may also be applied, if source code is available. Mutation testing – developing multiple versions of the code with mutations, or seeded faults – may be used to compare the fault detection capacity of alternative test suites, or evolve a test suite that produces a sufficiently high score on detecting differences between mutated versions of the code. Such an approach naturally is dependent on the mutations chosen.

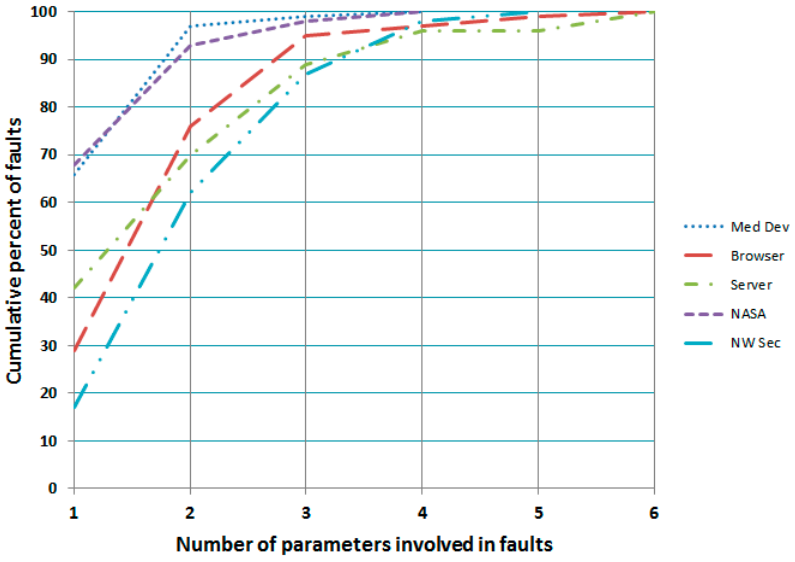


Figure 1. Cumulative fault distribution

Evaluating test quality is a particularly difficult and imprecise process for "black box" testing, where no source code is used. A test goal may be to positively demonstrate a collection of specified features, often by a single test for each feature or option. But simply showing that a particular input can demonstrate the feature does little to prove that an application is adequate for the wide range of inputs likely to be encountered in real-world use. Alternatively, an operational profile may be developed which tests the system according to the statistical distribution of inputs that occur in operational use. This process can provide reasonable confidence for the system's behavior in normal operation, but may miss the rare input configurations that can result in a failure.

A common approach for high assurance is to include tests designed to exercise the system with rare scenarios, based on experience or engineering judgment. This approach is clearly dependent on the skill of testers, and it may leave a large proportion of the possible input space untested. It also provides no quantitative measure of the proportion of significant input combinations that have been tested. Therefore, if test services are to be contracted out, there is little sound basis for developers to specify the level of testing required, or for testers to prove that testing has been adequate for the required assurance level. This paper describes measurement methods derived from combinatorial testing that can be used in analyzing the thoroughness of a test set, based on characteristics of the test set separate from its coverage of executable code.

Distribution of Faults

Empirical data show that most failures are triggered by a single parameter value, or interactions between a small number of parameters, generally two to six [1], a relationship known as the interaction rule. An example of a single-value fault might be a buffer overflow that occurs when the length of an input string exceeds a particular limit. Only a single condition must be true to trigger the fault: input length > buffer size. A 2-way fault is more complex, because two particular input values are needed to trigger the fault. One example is a search/replace function that only fails if both the search string and the replacement string are single characters. If one of the strings is longer than one character, the code does not fail, thus we refer to this as a 2-way fault. More generally, a t-way fault involves t such conditions.

Figure 1 shows the cumulative percentage of faults at t = 1 to 6 for various applications [1]. We refer to the distribution of t-way faults as the fault profile. Figure 1 shows the fault profile for a variety of fielded products in different application domains, and results for initial testing of a NASA database system. As shown in Figure 1, the fault detection rate increases rapidly with interaction strength, up to t=4. With the medical device applications, for example, 66% of the failures were triggered by only a single parameter value, 97% by single values or 2-way combinations, and 99% by single values, 2-way, or 3-way combinations. The detection rate curves for the other applications studied are similar, reaching 100% detection with 4 to 6-way interactions. Studies by other researchers have been consistent with these results. Thus, the impossibility of exhaustive testing of all possible inputs is not a barrier to high assurance testing. That is, even though we cannot test all

possible combinations of input values, failures involving more than six variables are extremely unlikely because they have not been seen in practice, so testing all possible combinations provides very little benefit beyond testing 4 to 6-way combinations.

Matrices known as covering arrays can be computed to cover all t-way combinations of variable values, up to a specified level of t (typically $t \leq 6$), making it possible to efficiently test all such t-way interactions [2]. The effectiveness of any software testing technique depends on whether test settings corresponding to the actual faults are included in the test sets. When test sets do not include settings corresponding to actual faults, the faults will not be detected. Conversely, we can be confident that the software works correctly for t-way combinations contained in passing tests.

As with all testing, it is necessary to select a subset of values for variables with a large number of values, and test effectiveness is also dependent on the values selected, but testing t-way combinations has been shown to be highly effective in practice. This approach is known as combinatorial testing, an extension of the established field of statistical Design of Experiments (DoE), endorsed by the Department of Defense Office of Test and Evaluation in 2009 [3], and used by commercial firms with demonstrated success.

Coverage Implications of Fault Distribution

The distribution of faults reported above suggests that testing which covers a high proportion of 4-way to 6-way combinations can provide strong assurance. If we know that t or fewer variables are involved in failures, and we can test all t-way combinations, then we can have reasonably high confidence that the application will function correctly. As shown above, the distribution of faults varies among applications, but two important facts are apparent: a consistently high level of fault detection has been observed for 4-way and higher strength combinations; and no interaction fault discovered so far, in thousands of failure reports, has involved more than six variables.

Any test set, whether constructed as a covering array or not, contains a large number of combinations. Measuring combinatorial coverage, i.e., the coverage of t-way combinations in a test set, can therefore provide valuable information about test set quality. Combinatorial coverage includes a number of advantages for assessing test quality:

- Computed independently of other evaluations of test quality. Combinatorial coverage provides additional information for decision-makers, and may be used in conjunction with structural coverage, mutation testing, or other approaches.
- Direct relationship with fault detection. The size of the input space spanned by the test set, a significant aspect of fault detection, can be measured by the number of t-way combinations up to an appropriate level of t. The proportion of t-way combinations covered measures the fractional size of the input space that is tested.
- Simple to compute and interpret. Because it is based on the input space of test values, there is no need to run the system under test to compute this measure of test set quality. Freely available tools can be used on any test set expressed as a matrix where rows are tests and columns are parameter values.

Measuring Coverage of Fault-triggering Combinations

Combinatorial testing is based on covering all t-way combinations for some specified level of t, but this form of testing may not always be practical because of established test practices, legal or contractual test requirements, or use of legacy test sets. An alternative to creating a combinatorial test set from scratch is to investigate the combinatorial coverage properties of an existing test set, possibly supplementing it with additional tests to ensure thorough coverage of system variable combinations. Determining the level of input or configuration state-space coverage can help in understanding the degree of risk that remains after testing. If a high level of coverage of state-space variable combinations has been achieved, then presumably the risk is small, but if coverage is much lower, then the risk may be substantial. This section describes some measures of combinatorial coverage that can be helpful in estimating this risk.

Variable-value configuration: For a set of t variables, a variable-value configuration is a set of t valid values, one for each of the variables, i.e., the variable-value configuration is a particular setting of the variables.

Example. Given four binary variables a, b, c, and d, for a selection of three variables a, c, and d the set {a=0, c=1, d=0} is a variable-value configuration, and the set {a=1, c=1, d=0} is a different variable-value configuration.

Simple t-way combination coverage: For a given test set of n variables, simple t-way combination coverage is the proportion of t-way combinations of n variables for which all valid variable-value configurations are fully covered.

Example. Table I shows four binary variables, a, b, c, and d, where each row represents a test. Of the six possible 2-way variable combinations, ab, ac, ad, bc, bd, cd, only bd and cd have all four binary values covered, so simple 2-way coverage for the four tests in Table 1 is $2/6 = 33.3\%$. There are four 3-way variable combinations, abc, abd, acd, bcd, each with eight possible configurations: 000, 001, 010, 011, 100, 101, 110, 111. Of the four combinations, none has all eight configurations covered, so simple 3-way coverage for this test set is 0%. As shown later, test sets may provide strong coverage for some measures even if simple combinatorial coverage is low.

It is also useful to measure the number of t-way combinations covered out of all possible settings of t variables.

Total variable-value configuration coverage: For a given

a	b	c	d
0	0	0	0
0	1	1	0
1	0	0	1
0	1	1	1

Table 1. Test array with four binary components

combination of t variables, total variable-value configuration coverage is the proportion of all t-way variable-value configurations that are covered by at least one test case in a test set. This measure may also be referred to as total t-way coverage.

The number of t-way combinations in an array of n variables is C(n,t) = n!/(n-t)!t!, or "n choose t", the number of combinations of n things taken t at a time without repetition. If each variable has v values, then each set of t variables has v^t configurations, so the total number of possible combination settings is $v^t \times C(n, t)$. Any test set covers at least some fraction of this amount. (Note that there is a natural extension of this formula for the case where variables do not all have the same number of values.) For the array in Table I, there are C(4,2) = 6 possible variable combinations and $2^2 \times C(4,2) = 24$ possible variable-value configurations. Of these, 19 variable-value configurations are covered and the only ones missing are ab=11, ac=11, ad=10, bc=01, bc=10, so the total variable-value configuration coverage is 19/24 = 79%. But only two, bd and cd, out of six, are covered with all 4 value pairs. So for simple t-way coverage, we have only 33% (2/6) coverage, but 79% (19/24) for total variable-value configuration coverage. Although the example in Table 1 uses variables with the same number of values, this is not essential for the coverage measurement, and the same approach can be used to compute coverage for test sets in which parameters have differing numbers of values.

Figure 2 shows a graph of the 2-way (red/solid) and 3-way (blue/dashed) coverage data for the tests in Table 1. Coverage is given as the Y axis, with the percentage of combinations reaching a particular coverage level as the X axis. For example, the 2-way line (red) reaches Y = 1.0 at X = .33, reflecting the fact that 2/6 of the six combinations have all 4 binary values of two variables covered. Similarly, Y = .5 at X = .833 because one out of the six combinations has 2 of the 4 binary values covered. The area under the curve for 2-way combinations is approximately 79% of the total area of the graph, reflecting the total variable-value configuration coverage.

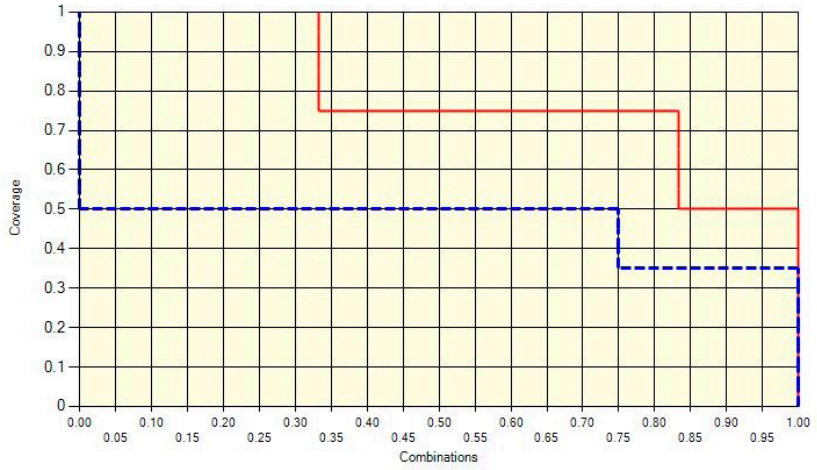


Figure 2. Graph of coverage for Table 1 tests

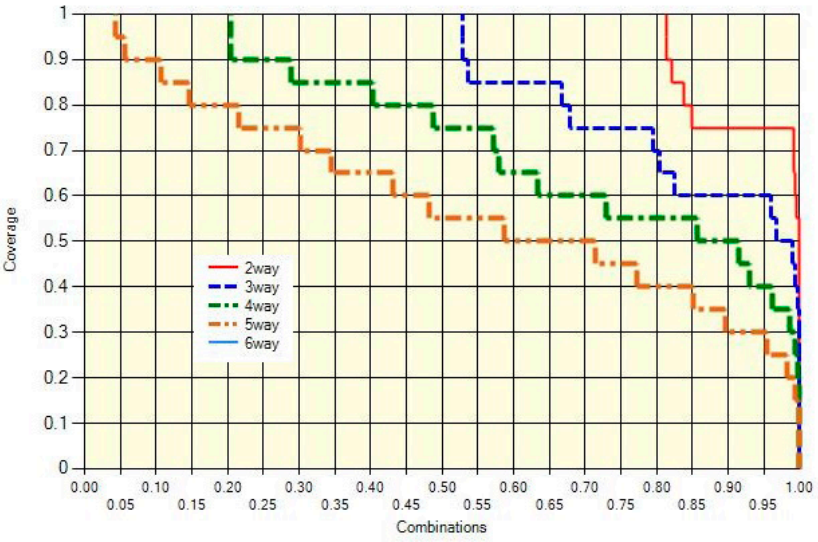


Figure 3. Configuration coverage for spacecraft example.

Practical Examples

The methods described in this paper were originally developed to analyze the input space coverage of tests for spacecraft software [4][5]. A set of 7,489 tests had been developed, although at that time combinatorial coverage was not the goal. With such a large test suite, it seemed likely that a huge number of combinations had been covered, but how many? Did these tests provide 2-way, 3-way, or even higher degree coverage?

The original test suite had been developed to verify correct system behavior in normal operation as well as a variety of fault scenarios, and performance tests were also included. Careful analysis and engineering judgment were used to prepare the original tests, but the test suite was not designed according to criteria such as statement or branch coverage. The system was relatively large, with the 82 variable configuration $1^3 2^{75} 4^2 6^2$ (three 1-value, 75 binary, two 4-value, and two 6-value). Figure 3 shows combinatorial coverage for this system (red = 2-way, blue = 3-way, green = 4-way, orange = 5-way). This particular test set is not a covering array, but pairwise coverage is still relatively good, because 82% of the 2-way combinations have 100% of possible variable-value configurations covered and about 98% of the 2-way combinations have at least 75% of possible variable-value configurations covered.

interaction	combinations	settings	coverage
2-way	3321	14761	94.0
3-way	88560	828135	83.1
4-way	1749060	34364130	68.8
5-way	27285336	603068813	53.6

Table 2. Total t-way coverage for Fig. 3 configuration.

Figure 4 shows a smaller example based on a US Air Force test plan [6] with seven parameters in a $2^4 3^1 4^2$ (four 2-value, one 3-value, and two 4-value) configuration, with 2-way through 6-way coverage for 122 tests. Coverage is remarkably high, with nearly 100% of all 2-way through 4-way combinations

covered. Note that the 2-way and 3-way lines are not visible because with 100% coverage they appear as vertical lines on the right side of the chart.

Figure 5 shows how coverage declines with 25% of the tests removed. Although the smaller test set has less coverage for all but 2-way combinations, coverage is still relatively high, so a test manager might consider this comparison in reviewing the cost/benefit tradeoffs of adding or removing tests.

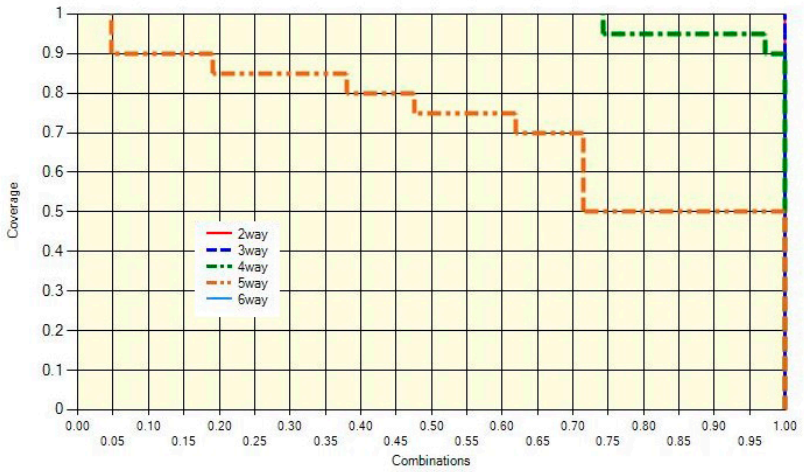


Figure 4. Configuration coverage for USAF test plan.

interaction	combinations	settings	coverage
2-way	21	152	100
3-way	35	664	100
4-way	35	1690	98.7
5-way	21	1818	69.7

Table 3. Coverage for Fig. 4 configuration.

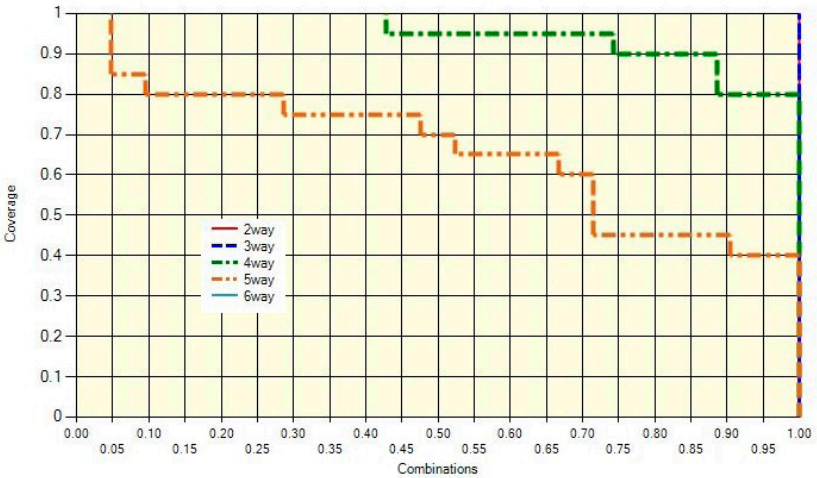


Figure 5. Configuration coverage, 75% of tests in Fig. 4.

interaction	combinations	settings	coverage
2-way	21	152	100
3-way	35	664	99.5
4-way	35	1690	90.0
5-way	21	1818	56.7

Table 4. Coverage for Fig. 5 configuration.

Computing Combinatorial Coverage

Tools are available to compute the measures discussed in this article. Several covering array generators can compute total coverage, and NIST-developed tools that are freely available can compute a variety of additional measures, and produce the reports included in examples above. The tools also include embedded constraint solvers, making it possible to produce counts of covered combinations excluding those that are not possible physically, or should be excluded because of constraints among variables. This is an essential feature for real-world use. It is also possible to generate additional tests to supplement those analyzed, to bring coverage up to any desired level.

The methods and tools introduced above were developed for analysis of NASA software tests, and additional NASA usage has suggested the following areas of utility [7]: 1) as an inline analysis tool for evaluating developer tests, 2) as a planning tool during test development to ensure adequate coverage, 3) as an IV&V audit tool for auditing completed IV&V analysis or multi-project test plans.

Conclusions

Combinatorial coverage provides valuable information for decision-makers because it measures the proportion of the input space that is covered relevant to testing. Because only a small number of variables are involved in failures, testing all settings of 4-way to 6-way combinations can provide strong assurance. For example, if we measure the t-way coverage of tests, and find that all 4-way combinations are covered, 90% of 5-way combinations, and 70% of 6-way combinations are covered, we can reasonably conclude that very few potential failure-triggering combinations have been left untested. Conversely, we can also have confidence that the system has been shown to work correctly for almost all of the relevant input space. Thus, combinatorial coverage can provide significant value in evaluating test quality.

Acknowledgements

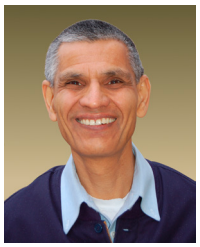
We are grateful to Greg Hutto at Eglin AFB for providing a copy of the 53d Wing tech report on design of experiments in test plan design.

Disclaimer

Certain products may be identified in this document, but such identification does not imply recommendation by the U.S. National Institute of Standards and Technology or other agencies of the U.S. Government, nor does it imply that the products identified are necessarily the best available for the purpose.

ABOUT THE AUTHORS

D. Richard Kuhn is a computer scientist in the Computer Security Division of NIST. His current interests are in information security, empirical studies of software failure, and software assurance, focusing on combinatorial testing. He received an MS in computer science from the University of Maryland College Park.
Phone: 301-975-3337
E-mail: kuhn@nist.gov



Raghu N. Kacker is a researcher in the Applied and Computational Mathematics Division of NIST. His current interests include software testing and evaluation of the uncertainty in outputs of computational models and physical measurements. He has a Ph.D. in statistics and is a Fellow of the American Statistical Association, and American Society for Quality.

Phone: 301-975-2109
E-mail: raghu.kacker@nist.gov

Yu Lei is a professor in Department of Computer Science and Engineering at the University of Texas, Arlington. His current research interests include automated software analysis and testing, with a special focus on combinatorial testing, concurrency testing, and security testing. He received his PhD from North Carolina State University.

Phone: 817-272-2341
E-mail: ylei@uta.edu

REFERENCES

1. D.R. Kuhn, D.R. Wallace, A.J. Gallo, Jr., "Software Fault Interactions and Implications for Software Testing", IEEE Trans. on Software Engineering, vol. 30, no. 6, June, 2004.
2. NIST Special Publication 800-142, Practical Combinatorial Testing, Oct. 2010.
3. C. McQuery, "Design of Experiments in Test and Evaluation". Memo, Office of the Secretary of Defense, May 1, 2009.
4. J.R. Maximoff, M.D. Trela, D.R. Kuhn, R. Kacker, "A Method for Analyzing System State-space Coverage within a t-Wise Testing Framework", IEEE International Systems Conference 2010, Apr. 4-11, 2010, San Diego.
5. D.R. Kuhn, I. Dominguez, R.N. Kacker and Y. Lei. "Combinatorial Coverage Measurement Concepts and Applications", 2nd Intl Workshop on Combinatorial Testing, Luxembourg, IWCT2013, IEEE, Mar. 2013.
6. G. Hutto, "53d Wing Test Plan Examples", Tech. Rpt., Eglin AFB, 2012.
7. C. Price, R. Kuhn, R. Forquer, A. Lagoy, R. Kacker, "Evaluating the t-way Combinatorial Technique for Determining the Thoroughness of a Test Suite", NASA IV&V Workshop, 2013.

Metrics That Matter in Software Integration Testing Labs

Christian Hagen, A.T. Kearney
Steven Hurt, A.T. Kearney
Andrew Williams, A.T. Kearney

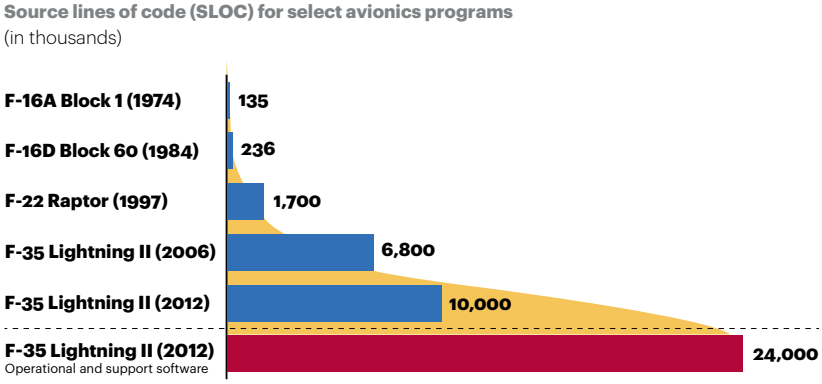
Abstract. Without having in place data-driven metrics that give a holistic business perspective of software integration testing laboratories, leaders of the DoD's weapons programs are unable to optimize the performance of these full-system and subsystem integration labs that test and certify integrated hardware and software during the development, modernization, and sustainment of the U.S. military's integrated and complex systems. Yet these metrics are not available across the DoD lab footprint, even though the labs' significance is growing in parallel with the military's rapid shift from using equipment with capabilities based on advanced hardware to equipment that is dependent on fully integrated, complex systems of both hardware and software.

This game-changing shift is now evident in weapons programs throughout the military. It's seen, for example, in the F-35 program, whose software over the years has increased in size to approximately 24 million source lines of code, which has made the testing more difficult and led, in part, to the program's multiyear delays (see Figure 1).

As this shift intensifies, so too does the program leaders' need to successfully compare the operations of any lab to that of any other lab, each of which today approaches software integration testing with its own particular processes for measuring its progress and success. They simply need to be able to answer several pertinent questions that they cannot answer now, such as the following:

- Is the lab running at the appropriate level of efficiency, effectiveness, and cost, and if not, how can the current level of performance be improved?
- Can the lab handle additional testing and, if so, how much should be moved there and from where?
- Should the lab be updated and, if so, how much money should be spent on the update?
- Should the lab be closed rather than updated and should the testing be transferred to another facility?
- Would the labs see reduced costs and improved performance with the purchase of new, faster testing equipment?

Figure 1
The amount of software in military avionics systems has skyrocketed



Note: SLOC for F-16 and F-22 are at first operational flight. F-35 SLOC figures are from first test flight and current estimates/sources.
Source: Hagen, C., Hurt, S., Sorenson, J. "Effective Approaches for Delivering Affordable Military Software." CrossTalk – The Journal of Defense Software Engineering, Vol. 26 No. 6 (November-December 2013).

The Need for Metrics That Matter

The need to have metrics that matter when making decisions facing software integration labs was recently underscored by Robert Ferguson of Carnegie Mellon University's Software Engineering Institute (SEI). Ferguson's 2012 research shows that using project dashboards and following the right measurements are critical to project management because they provide managers with the information they need to perform different tasks at the correct times—much like the dashboard of a car leads to a successful journey by preventing the driver's running out of gas, going over the speed limit, or arriving late [1].

Ferguson showed a project dashboard—which should be constructed to suggest different decisions about product quality, and about directing and controlling the work—provides measures for the critical areas of project decision making. These measures include scheduling, resource allocation, scope and change, product quality, and effective process performance. In addition, the dashboard should do the following:

- Forecast milestones and delivery of scope
- Provide clear warnings if the plan is not working or an unplanned event has affected some desired outcome
- Support re-estimation and re-planning by showing the magnitude of the problem

Measures like these have never been more important than in today's environment in which software integration labs deliver the software and hardware capability weapon programs must have to win on the battlefield—a critical role the labs can perform only if they have the metrics they need.

Unfortunately, most software and program leaders today are attempting to make decisions without metrics that matter to them. Primarily, they have engineering- and technology-based metrics—metrics that are of value to those who care mostly about being able to test a single piece of equipment, not manage the overall operation of a lab or group of labs. What they need are metrics that are valid to those who must make command-level decisions from a holistic business perspective.

When DoD leaders try to make decisions like moving one lab's testing to other labs, they run headlong into major problems caused by the lack of metrics that really matter. Since each lab measures its progress and success with its own unique processes, decisions across the footprint are made using a nonstandard, and often ad hoc, approach. And with no standard set of metrics, leaders are uncertain about what the available metrics mean, which ones matter, and how they can use them to make fully informed command decisions about the system integration labs. Their confusion is compounded by the lab contractors' belief that since the labs use different technologies, test different equipment, and have completely different workloads, they cannot provide the metrics needed to compare operations—a belief that has been proven groundless in many other industries.

It was this confusion that led the leaders of a major avionics program in the DoD to determine they needed to significantly improve the way they looked across multiple labs to compare

operating costs, performance, and other key metrics. Through research they concluded that the metrics that matter the most for use in the system integration labs would come from examining operations with similar capital-intensive processes.

They found the metrics they needed in manufacturing and operations, which has long been using a standard set of metrics—capacity, efficiency, effectiveness, and capability—to compare the operations of manufacturing plants, regardless of what the plants were producing. While the type of work done in these manufacturing facilities—inputting parts, assembling them, and outputting completed products—differs greatly from that of the integration labs—inputting software code and hardware, running tests against the code, and putting out a report on whether the code is good or bad—the processes are similar. Therefore, the metrics can be similar as well (see Figure 2). Although much has been written about software estimation and quality, [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21], much less has been written about software integration testing.

The metrics found in these plants are derived from the body of work in manufacturing excellence that crosses many industries having similar processes, although a variety of products. They also are perfectly applicable to software integration labs. And now a few decision makers across the labs are starting to discover that these metrics—capacity, efficiency, effectiveness, and capability—enable them to not only measure and improve each software lab's cost and performance but to effectively manage all their labs as they test software systems that are fast becoming the strategic weapons on which the military's future success depends.

These four metrics closely resemble those of the overall equipment effectiveness (OEE) framework that was developed over the years to measure how effectively a process was executed in a manufacturing facility, and is now being used across industries, including the automotive sector. This framework was designed to give leaders the metrics to compare processes across factories and industries—the metrics they simply have to understand if they are to manage effectively their businesses and operations.

OEE, as research shows, is based on a standard set of metrics for understanding the manufacturing process [22]. It is captured through the following formula: OEE = machine availability x machine performance x product quality. The result is presented as a percentage that can be used to understand how the current manufacturing process of a plant is performing, and to determine how one or all three of these factors can be changed to improve this performance. It can also be used to compare performance across manufacturing plants within a company, throughout an industry, or across industries.

Additional research has shown that OEE can be applied more generally to operations, plants, and machinery. An article from 2003, for example, shows how cross-functional teams can apply OEE principles to multiple areas of operations and further shows that OEE principles can be applied to areas beyond manufacturing, pointing the way to its application in software integration labs [23].

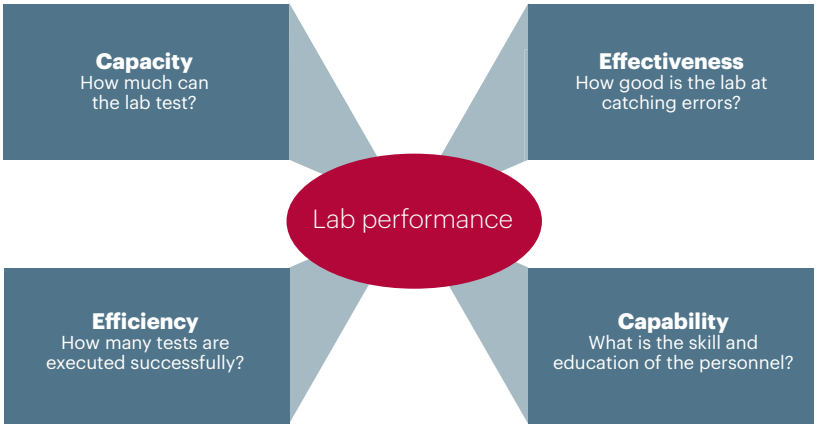
Leaders in the avionics program tweaked this framework for use within the department's software integration laboratories. They have learned that the four metrics—capacity, efficiency, effectiveness, and capability—are easily transferable to the labs because their measurements are directly comparable to those made in the automotive factories (see Figure 3).

Figure 2
The metrics for manufacturing and for software testing labs are similar

	Production metrics	
	Manufacturing	**Software integration lab**
Capacity	• Number of cars produced per hour	• Number of test points executed per hour
Efficiency	• Number of good cars produced per hour	• Number of test points executed per hour
Effectiveness	• Number of quality fixes	• Number of defects found
Capability	• What can the factory produce? (for example, Porsche vs. Yugo)	• What areas and complexity of tests can the lab execute?

Source: A.T. Kearney analysis

Figure 3
Software metrics for laboratory performance



Source: A.T. Kearney analysis

Capacity

While automotive factories count their output of vehicles per hour, software integration labs measure capacity by counting the number of tests executed per hour.

This metric, measured in test points, is the throughput per hour in terms of a lab's ability to execute its raw work. Test points, which at a basic level represent specific criteria to evaluate for validation and successful testing (for example, specific engineering performance values or, for a smoke test, the expected system output to a standard set of inputs), are used as a basis for the starting point for lab capacity. Test points are executed within a variety of test types, such as integration, verification, and regression tests. They are a measure of how much work, in total units, could be accomplished if the lab worked nonstop around the clock.

Capacity is measured in test points, which can easily be converted into derivative metrics like shift, daily, and yearly capacity. And it serves as the best proxy for lab size, showing whether the lab equates to a factory that is big or small. Knowing this capacity will, among other things, help DoD leaders determine whether the work they want to shift to another lab can be handled by that lab or not, vis-à-vis capability or capacity.

Because test points are the basic unit of lab production, comparing dollars per test point is the core indicator of a lab's cost. Using this comparison, decision makers can determine, for example, the cost of running a test or of finding a defect—such

as a major defect that would cause the postponement of an unmanned vehicle's mission or a minor defect that might cause the malfunction of a truck's power steering.

Efficiency

While automotive factories check the number of "lemons" produced each hour, software integration labs measure efficiency by checking the number of tests executed hourly "on condition."

This quality metric indicates how well a lab is doing its work. If it can do 100 units of work each day, but only 50 units, on average, are correct, the labs' efficiency metric would be quite low.

"On-condition" is a test executed successfully—a determination based on the checklist and setup procedures handed down by the system engineers. Since the test is successful, it does not need to be performed again. Efficiency measures the percentage of tests executed correctly—not whether the software being tested passed or failed the test—and is calculated by dividing test points on condition by total test points attempted. "Off-condition" is a test that must be repeated because an error occurred in testing methods or setup. A false "on-condition" test is properly executed on condition, but it must be repeated because further analysis shows the test package was poorly designed.

Tightly linked, lab capacity and efficiency are often measured together to provide a clear understanding of their combined effect. With baselines derived from this combination, leaders can begin making command-level decisions about issues such as how a given action would change the lab's throughput, how a different action would affect the lab's cost per hour or cost per defect, and how some other action would impact the lab's efficiency or capacity.

Effectiveness

While automotive factories count the number of quality assurance fixes, software integration labs measure effectiveness by counting the number of software defects.

This metric points out how good a lab is at discovering errors. If, for example, its primary purpose is to find defects or certify code, the number of work units to defects could be a measure of effectiveness.

Effectiveness is measured by the number of test points executed per defect found, and it is calculated by defect found divided by test points attempted. This measurement of the lab's testing procedure shows how many tests must be run before the lab starts finding errors in the testing procedure. This metric is based on the assumption that labs have the capability to properly test the code. Testing capability means having the subject matter expertise, the appropriate number of personnel, and the right equipment to test the code. Since this metric measures the lab's ability to find defects in an existing code base, the resulting output of the metric is driven by the quality of the code being tested by the lab. Since quality of code can vary greatly across different development types and teams, this metric should not be used as an absolute value to compare labs across different development types. Instead, the trending of this metric within projects of similar size and scope can help a lab administrator track a lab's performance compared to historical testing efforts.

In a white paper on their research into software defects, SEI's Julie Cohen, Robert Ferguson, and William Hayes show that classifying defects appropriately and tracking them differently can increase lab effectiveness [24]. They suggest quantifying the priority of addressing these defects by assigning a Risk Priority Number (RPN) to each defect, a number that is calculated with "three distinct attributes of failure sources": severity, "a rating of the adverse impact of the failure"; occurrence, "how often the source of failure is encountered"; and detection, "how detectable the failure is when it occurs."

While not specifically addressing defects in software integration labs, the SEI authors underscore the need to view the defect data in the appropriate way with the appropriate metrics. This approach is essential to solving the broader issues DoD leaders face as they try to more effectively manage labs across the footprints.

Capability

While automotive factories explore the functionality of their equipment and what each factory can make, software integration labs measure capability by exploring the ability of each lab to meet the overall requirements.

This metric is the skill set of a lab's workforce and the functionality of its equipment. It is used to compare how well each lab can test specific areas of the software and is the function of three factors:

- Knowledge, which is assessed across product, functions, and technology, and is proven through work experience requiring expertise in the product, function, and technology areas
- Competency, which is assessed across current work behaviors and skills required to perform the work and proven by the existence of artifacts, such as current job descriptions and training, which are used to validate managers' and directors' scores for their teams and specific knowledge areas
- Capacity, which is measured by the availability and readiness of the lab's resources (human and infrastructure) to perform an activity

Because capability is also directly affected by a lab's equipment composition, this composition must be analyzed in any lab-to-lab comparison.

Capability plays a major role in leaders' overall management decisions because it has an implicit effect on the other three metrics that matter. Therefore, its impact on each of these metrics must be understood before making changes to the size, experience, or skill set of the workforce.

Approach to Applying the Metrics That Matter

When the leaders of the avionics weapons program began to evaluate the current strategy for software integration labs and to explore alternative models that might deliver better value, they quickly learned they needed metrics on which to base decisions—data-driven metrics that matter. In order to complete the evaluation, the team developed metrics that illuminated each lab's actual performance. These metrics drove the assessment of software integration labs and enabled leaders to accurately measure and compare lab performance across the footprint. They made possible the direct lab comparisons for analysis and enabled the leaders to create a business case to model future-state scenarios and compare cost savings, transition risks, and steady-state capacity risks across scenarios (see Figure 4).

As an example of the power that using the correct metrics gives

to business leaders evaluating testing labs, Figure 5 describes a hypothetical model of additional hours of test time required during a lab transition scenario. By understanding a lab's normal throughput in tests and operational efficiency (first time right execution), the effects of the overall program test hours can be estimated as different areas of the lab are shut down to transition.

In this example, the lab is transitioning in two phases, each of which will reduce the testing capacity of the lab during the transition time. Using the appropriate metrics the leaders can estimate the impact to the program and additional hours required to keep the same level of testing results as before the transition started (revised test hours).

Besides evaluating the labs' current strategy and exploring alternative value models, the assessment's specific objective was to reduce the labs' life cycle costs by moving the program's testing from its current location to potential alternatives and to do so without degrading current performance. The program also set out to answer questions about the attributes of the current lab footprint; about alternatives to the current lab environment; about the costs, benefits, and risks of the current plan and the proposed alternatives; and about the recommended strategy (current plan versus proposed alternatives).

The program met its objective with a thorough analytical review of the current long-term strategy and potential alternatives. In doing so, it determined that the best value alternative would result in the lowest life cycle cost with manageable risk while not degrading lab capabilities or performance.

Results

The metrics developed during the assessment provided the information needed for the leaders to recommend that the avionics program transition the testing to the alternative labs but maintain the current lab's performance and its operator and equipment capability. This result provided less risk during the transition as well as steady state. It also saved more than 30 percent in life cycle costs, for a total net present value savings of hundreds of millions of dollars (see Figure 6).

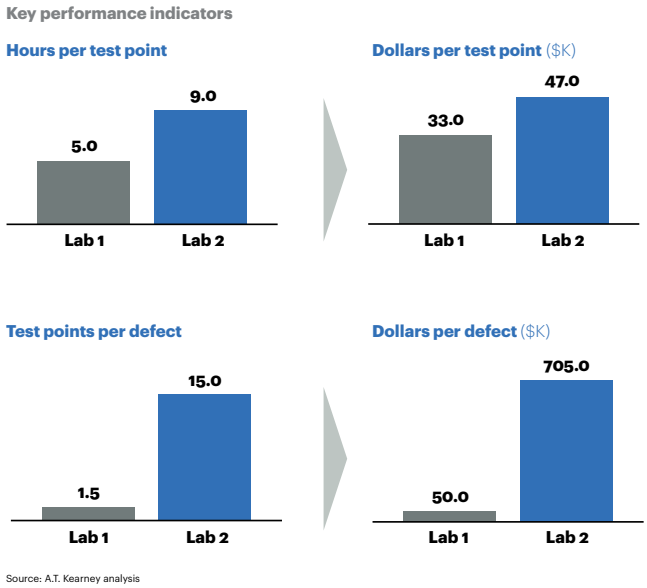
Using the metrics, the team modeled several courses of action through the perceived end-of-life. From these, it recommended a clear course of action for moving the testing, including the expected cost savings, transition risks, and potential risks.

The clear, communicable metrics that were created reflect lab capacity, efficiency, effectiveness, and capability—the four metrics that matter to program leaders and make it possible for them to manage labs more effectively.

Assessing the performance of the system integration labs not with ad hoc metrics valued only by technicians and engineers but with a standard set of metrics that matter to decision makers needing a holistic business perspective can lead to valuable manufacturing-environment benefits, such as the following:

- **Transparency**. With a clear, communicable set of metrics, leaders can quickly and accurately assess performance and capacity. In addition, fact-based, apples-to-apples comparisons will enable them to contrast the performance of one lab to that of others.
- **Cost savings**. Historically, cost advantages between labs have been hidden behind immaterial metrics. Now equal, meaningful metrics highlight current cost-saving opportunities.

Figure 4
Lab comparison across common metrics



Source: A.T. Kearney analysis

Figure 5
Example analytic framework for workload shift across integration labs

Demand calculation — **Illustrative**

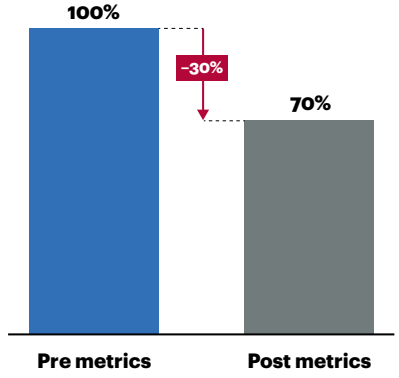
TP = test points
aTP = adjusted test points

				Lab Transitioning	
		Baseline	Period 1	Period 2	Period 3
Nominal demand	Test hours	200	200	200	200
		×	×	×	×
Adjusted capacity	Raw capacity (test points/HR)	2 TP/hr	2 TP/hr	1.8 TP/hr	1.6 TP/hr
		×	×	×	×
	First time right	90%	90%	90%	90%
		=	=	=	=
Adjusted demand	Adjusted test points executed	360 TP	360 TP	324 TP	288 TP
Demand adjustment	Performance variance (test points)	N/A	0 TPs	36 TPs	72 TPs
	Hours surge required	0	0	23 hrs at 1.62 aTPs/hr[1]	50 hrs at 1.44 aTPs/hr[1]
	Revised test hours	200	200	223	250

[1] aTPs/hr: Adjusted test points per hour. Adjusted test points = raw test points / hour * first time right percent.
Source: A.T. Kearney analysis

Figure 6
Focusing on metrics that matter can reduce life cycle costs

Program life cycle costs — **Illustrative**



- Using metrics that matter, an avionics defense program identified lower-cost labs to perform work
- By shifting work to these locations, life cycle costs dropped 30%
- The lower-cost labs are not always driven by just labor costs but are evaluated by total cost to test, which includes:
  —Process
  —Efficiency
  —Lab and test philosophy
  —Equipment requirements

Source: A.T. Kearney analysis

- **Risk mitigation**. The metrics will take into account current and future lab capacity, allowing for more accurate estimates of cost and potential schedule delays.
- **Negotiations support**. The metrics will provide the facts on which the best negotiations are based and enable DoD leadership to accurately size and negotiate requirements for contracting labs.

Moreover, with these metrics that matter, program leaders will have the solid measures they need to develop a full understanding of the labs' current level of efficiency, a starting point on which to base both minor and command-level decisions for the future and for determining the impact of those decisions—whether they are about adding capacity, reducing costs, hiring employees, improving throughput and quality, or similar issues. And as they make these decisions that will drive the effectiveness and savings of labs across the footprint, they will further strengthen labs' role in delivering the most advanced systems to U.S. weapon programs.

ABOUT THE AUTHORS

Christian Hagen is a partner in A.T. Kearney's Strategic Information Technology Practice and is based in Chicago. He advises many of the world's largest organizations across multiple industries, including government and defense contractors. He specializes in helping clients leverage software and information technology to increase efficiencies and gain competitive advantage. Christian has led several global studies for A.T. Kearney and authored over 60 published papers on low-cost competition, software engineering, e-commerce, technology innovation, and strategy.

E-mail: christian.hagen@atkearney.com

Steven Hurt is a partner in A.T. Kearney's Public Sector and Defense Services. Steve has worked with several of the USAF's highest-visibility programs to drive affordability in both software and hardware sustainment. Specifically, Steve has focused on should-cost analyses, business case analyses, contract negotiations, and developing business intelligence aimed at reducing cost.

E-mail: steven.hurt@atkearney.com

Andrew Williams is a principal in the Strategic Information Technology Practice and in Public Sector and Defense Services at A.T. Kearney. Andrew works with both commercial and government clients on some of the most challenging IT issues, including cost optimization, should-cost evaluations, infrastructure services, and large mega-vendor contract negotiations.

E-mail: andrew.williams@atkearney.com

REFERENCES

1. Ferguson, R. (2012). "A Project Dashboard." Unpublished Manuscript, Software Engineering Institute at Carnegie Mellon University.
2. Stark, G. (2011). "A Comparison of Parametric Software Estimation Models Using Real Project Data." CrossTalk – The Journal of Defense Software Engineering, January 2011.
3. Boehm, B. (1981). Software Engineering Economics. Englewood Cliffs, N.J., Prentice Hall.
4. Boehm, B. (2006). "Minimizing Future Guesswork in Estimating," IBM Conference on Estimation, Atlanta, GA, February 2006.
5. Jones, C. (2007). "Software Estimating Rules-of-Thumb." <http://www.compaid.com/caiinternet/ezine/capers-rules.pdf.>, March 2007.
6. Jones, C. (1997). Applied Software Measurement, 2nd Ed. McGraw-Hill, NY.
7. Rone, K. et al. (1994). "The Matrix Method of Software Project Estimation." Proceedings of the Dual-Use Space Technology Conference, NASA Johnson Space Center, Houston, TX, February.
8. J.W. Bailey and V.R. Basili. "A Meta-Model for Software Development and Resource Expenditures." Proceedings of the 5th International Conference on Software Engineering. New York: Institute of Electrical and Electronic Engineers, 1983.
9. ISBSG, International Software Benchmarking Standards Group. <http://www.compaid.com/caiInternet/ezine/ISBSGestimation.pdf>.
10. ISBSG, International Software Benchmarking Standards Group. <http://www.isbsg.org/isbsg.nsf/weben/Project%20Duration>.
11. McConnell, S. (2006). Software Estimation: Demystifying the Black Art, Redmond, WA, Microsoft Press.
12. International Function Point User's Group (IFPUG), Function Point Counting Manual, Release 3.1, 1990.
13. Jones, C. "Achieving Excellence in Software Engineering." Presentation to IBM Software Engineering Group, March 2006.
14. P. Oman. "Automated Software Quality Models in Industry." Proceedings of the Eighth Annual Oregon Workshop on Software Metrics (May 11-13, Coeur d'Alene, ID), 1997.
15. G. Atkinson, J. Hagemeister, P. Oman & A. Baburaj. "Directing Software Development Projects with Product Measures." Proceedings of the Fifth International Software Metrics Symposium (November 20-21, Bethesda, MD), IEEE CS Press, Los Alamitos, CA, 1998, pp. 193-204.
16. T. Pearse, T. Freeman, & P. Oman. "Using Metrics to Manage the End-Game of a Software Project." Proceedings of the Sixth International Software Metrics Symposium (Nov. 4-6, Boca Raton, FL), IEEE CS Press, Los Alamitos, CA, 1999, pp. 207-215.
17. Kemerer, C. F. (1987). "An empirical validation of software cost estimation models." Communications of the ACM, Vol. 30, No. 5, pp. 416-429.
18. Jorgensen, M., and Sheppard, M. (2007). "A Systematic Review of Software Development Cost Estimation Studies." IEEE Transactions on Software Engineering, Vol. 33, No. 1, January, pp 33-53.
19. Fenton N. E., and Pfleeger, S. L. (1997). Software Metrics: A Rigorous & Practical Approach, 2nd Ed., London, PWS Publishing.
20. Jorgensen, M. (2004). "A Review of Studies on Expert Estimation of Software Development Effort." Journal of Systems & Software, Vol. 70, No. 1-2, pp. 37-60.
21. Hagen, C., Hurt, S., Sorenson, J. "Effective Approaches for Delivering Affordable Military Software." CrossTalk – The Journal of Defense Software Engineering, Vol. 26 No. 6 (November-December 2013).
22. OEE Primer, <http://www.oee.com/calculating-oee.html>.
23. C.J. Bamber et al. (2003). "Cross-Functional Team Working for Overall Equipment Effectiveness (OEE)." Journal of Quality in Maintenance Engineering, Vol. 9 Issue 3, pp. 223-238.
24. J. Cohen, R. Ferguson & W. Hayes (2013). "White Paper: A Defect Prioritization Method Based on the Risk Priority Number." Internal White Paper, Software Engineering Institute at Carnegie Mellon University.

Silverlining: A Simulator to Forecast Cost and Performance in the Cloud

Lawrence Chung, The University of Texas at Dallas
Nary Subramanian, The University of Texas at Tyler
Thomas Hill, The University of Texas at Dallas
Grace Park, The University of Texas at Dallas

Abstract. A key question for a Chief Information Officer (CIO) would be the future run-time cost and performance of complex business application software, before deciding to migrate it to a cloud. It is difficult for CIOs to accurately estimate cloud cost and performance in a fast and inexpensive manner. In this article, we describe "Silverlining", a simulator for estimating the cost and performance of a cloud service before migration, to help the CIO not only with go/no-go decisions but also with the budgeting for an appropriate cloud configuration.

Introduction

A software system's operational behavior can be characterized by not only its functional requirements - what the system does – but also its non-functional requirements (NFRs) - how usefully and usably the system executes its functions. To forecast the behavior of a software system in the cloud, we explore two primary run-time NFRs - cost and performance. The objective of the Silverlining Simulator is to predict the operational cost and performance of a system by building a model to imitate the operation of the software system under study. The simulation model needs a description of the basic cloud infrastructure topology (resource capacity) and a step-by-step depiction of the system operation workload (resource usage). The Google cloud infrastructure, employed in the case study, uses the following two primary classes of compute resources (shown in Figure 3): The Google App Engine (GAE) and the CloudSQL Database Engine. The CloudSQL Database Engine was simulated using three cloud configurations (Low-cost/low-power-CPU server - D1, medium-cost/medium-power-CPU server - D16 and high-cost/high-power-CPU server - D32). The low-cost/low-power-CPU server - D1 provided adequate throughput to satisfy the management's goal.

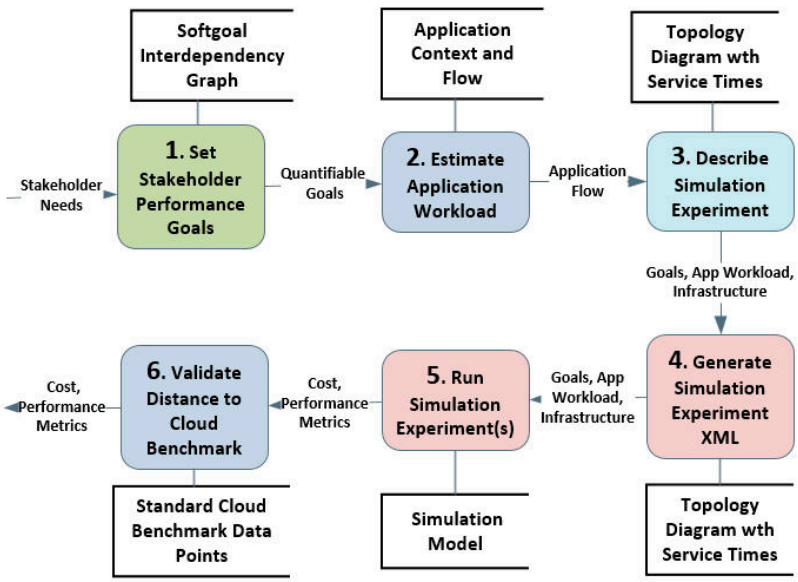


Figure 1. Silverlining simulation modeling framework steps.

Silverlining Simulation Process

In order to provide cloud forecasting capability, Silverlining - our simulator - must first be primed with appropriate base information (examples in Figures 4 and 5). For this purpose, cost and performance goals are obtained from stakeholder requirements for the system, oftentimes in terms of service level agreements (SLAs) (See Figure 1 Step1), and their interdependencies are analyzed by means of a notational convention, called Softgoal Interdependency Graph (SIG), which is intended for representing and reasoning about NFRs.

Next the characteristics of the intended software application is estimated (Step 2), e.g., using varying workloads. The characteristics of the intended software application are then loaded as input into the simulator (Step 3/4), and the simulator would output (Step 5) the cost and performance estimates for executing the software system, with varying workloads and cloud configurations, on the cloud. With proper adjustments for differences from the standard (Step6), the data from the simulator can be used to estimate the cost and performance of the cloud, as well as choosing among the available cloud configurations according to the particular cost and performance business goals that a CIO may have.

Now, an important question is if and how much Silverlining is reliable – i.e., the accuracy of the simulation results. For Silverlining, experiments for a typical application were run on Google cloud (called Google App Engine, or GAE), with varying workloads and cloud configurations (such as platforms and infrastructure characteristics), for a variety of benchmark data, and, using the same workloads and cloud configurations, the results from Silverlining were compared against the benchmark data. The comparison showed the two sets of data were very close for the typical application (or class of applications) that was used for the experiments. Of course, more experiments would be needed, in order for Silverlining to help a CIO and a cloud service provider assess and predict the cost and performance of a variety of (classes) of software applications, as well as choose among alternative cloud platforms and configurations, or, if needed, even adjust cost and performance business goals.

In this paper, a case study of a Vehicle Management System (VMS) Display-Status is presented, which has been in operation for almost three decades and will continue to be for many decades to come. This particular VMS is supposed to manage close to 100,000 vehicles, while carrying out a variety of tasks, such as keeping track of their locations and status (e.g., in normal operation or maintenance, or in emergency repair, moving or stationed), scheduling their routes, allocating crews, dispatching them, compiling statistics, reporting on work progress, etc.

In the following, the process of using Silverlining for the case study is described in a piecemeal manner.

The Vehicle Management CIO is contemplating the migration of the VMS to the cloud. Among other things, two requirements are of critical concerns - the total system operation costs cannot exceed $3,000 per month and the Display-Status application must perform at 300 transactions per minute (tpm). The performance goals and VMS operational characteristics are used as input to the simulation forecaster. The CIO can run the simulation on a local laptop computer and see the cost and performance estimates immediately, i.e., without the time-consuming and costly development, tailoring, installation, configuration and testing the software system, as well as without (the use of) any real cloud or hardware equipment.

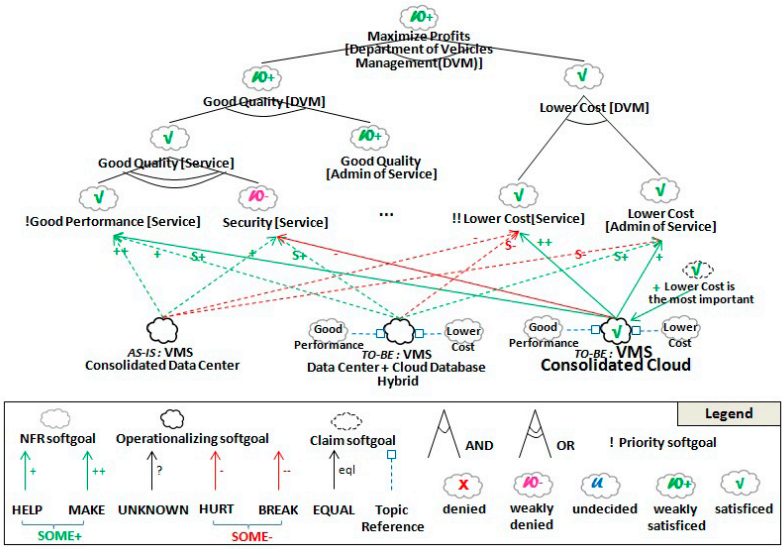


Figure 2-1. The SIG for VMS Business goals.

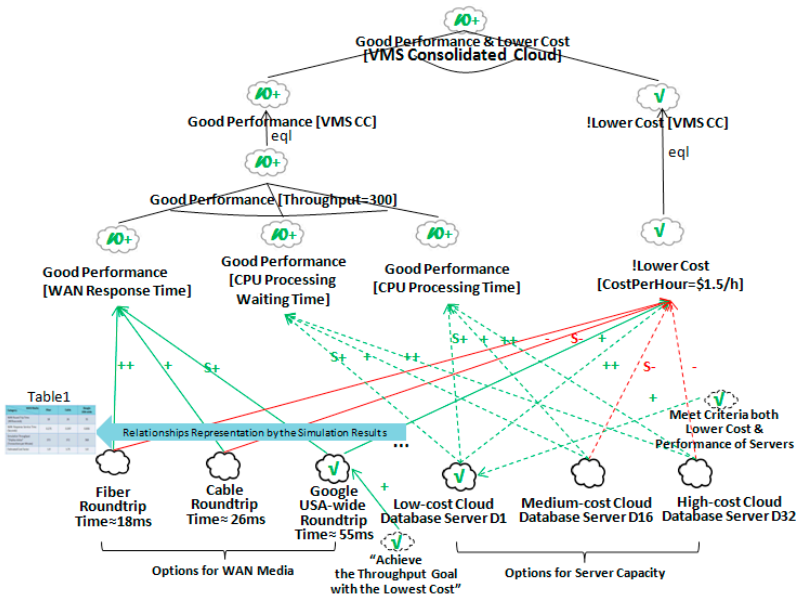


Figure 2-2. The SIG for VMS System goals.

Step 1: Set stakeholder cost and performance goals

First, a VMS Display-Status application SIG (Softgoal Interdependency Graph) [1] is developed to represent a subset of the business and the system non-functional requirements (NFRs). Figure 2-1 shows these NFRs as business (soft-)goals – goals for which there is no absolute criteria for their complete satisfaction - along with their sub-goals, and Figure 2-2 shows the system goals of the VMS, which are traceable to each other and the business (soft-) goals.

As shown in Figure 2-1, the final goal of the CIO of the organization is to maximize profits by lowering cost with good performance of service. To achieve the business goal, the CIO may consider migrating to a cloud, while exploring several options.

The options being considered include a Consolidated Cloud, in which the whole VMS system is operated in the cloud. A Hybrid Cloud is another option, in which the VMS Data center manages important and highly confidential information and the cloud manages less important and less vulnerable information.

Softgoals Each option has its own pros and cons, and the CIO can carry out an estimated tradeoff analysis in terms of potential benefits and risks that each option is likely to bring about. For example, a Consolidated Cloud is estimated to be better for Lower cost[Service] (green ++) than a Hybrid Cloud (red -), but worse for Security[Service] (red -) than a Hybrid Cloud (green S+, i.e., some +). The CIO can decide to choose one of the options that best meets (i.e., satisfices) the particular business goals as a business strategy, after confirming the tradeoff relationships between business goals and strategies by executing the Silverlining Simulator. In this case, we may assume, for the purpose of illustration, that the most important business goal is "Lower Cost", hence consequently the CIO selecting a Consolidated Cloud. By the way, due to the space limitation, we show the simulation results for only the VMS Consolidated Cloud, as indicated by solid lines; other ones whose simulation results are not shown in this paper are indicated by dotted lines.

Softgoals, such as Good Performance and Lower Cost[VMS Consolidated Cloud] in the upper line in Figure 2-2, come from the business strategy on the right-hand side portion on the bottom of Figure 2-1. In consideration of the business strategy, a system analyst can then traceably establish the system's concrete goals, such as Throughput and Cost, which respectively are quantified as 300 transactions per minute and $1.5 per hour, as shown in Figure 2-2. The alignment between business (soft-)goals and system (soft-)goals is represented by "eql" (equal relationship), as in "Good Performance[VMS CC] eql Good Performance[Throughput=300]" and "Lower Cost[VMS CC] eql Lower Cost[CostPerHour=$1.5/h]." Note that, in the piecemeal illustrations, Lower Cost[VMS CC] is considered more important than performance, when selection decisions are made.

To achieve the system goals, a system analyst finds out which configuration elements of a system infrastructure are needed, such as "WAN Media" and "Server Capacity", by estimating the application workload of Step2. There may be several operational options to achieve the system goals for each system element, and a system analyst can also choose one option, using our simulator results. As in Figure 2-2, the

Formula for WANTimeResponse

- WANRTTseconds = WANRTTms / 1000.0 # milliseconds to seconds
- TCPwindowbits = 524288.0 # tcp window size is 64K Bytes
- MaxNetworkThruPut = TCPwindowbits / WANRTTseconds
- MsgResponsebits = MsgResponseBytes * 8.0 #message output bytes to bits
- WANtimeResponse = MsgResponsebits / MaxNetworkThruPut # output WAN seconds

Figure 2-3. Formula for WANTimeResponse.

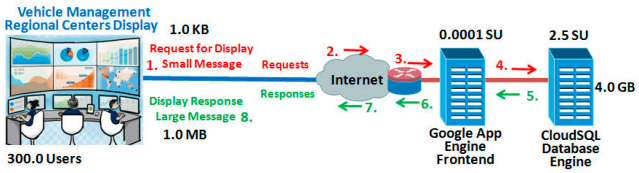


Figure 3. The VMS annotated application workload and workflow.

Category \ WAN Media	Fiber	Cable	Google USA-wide
WAN Round Trip Time (Milliseconds)	18	26	55
WAN Response Service Time (Seconds)	0.275	0.397	0.836
Simulation Throughput "Display-status" (Transactions per Minute)	373	372	368
Estimated Cost Factor	1.9	1.75	1.0

Table 1. Wide Area Network (WAN) Design Alternatives.

"WAN media" has options such as Fiber, Cable, and Google USA-wide [2] which have their own round trip time.

Using the simulator and the WANTimeResponse formula, as in Figure 2-3, a system analyst can obtain results as shown in Table 1. In this case, achieving all three subgoals of Good Performance [Throughput=300] will imply achieving Good Performance [Throughput=300] as well. Moreover, Good Performance [CPU Processing Waiting Time] and Good Performance [CPU Processing Time] are estimated to be achievable, via the use of a Cloud with any of the three different capacities (D1, D16 and D32) being considered (Of course, this can be confirmed, or denied, by simulation results). This would, then, imply that Good Performance [Throughput=300] is estimated to be achievable. In other words, for example, if we estimate that D1 achieves Good Performance [CPU Processing Waiting Time] and Good Performance [CPU Processing Time], then based on the data in Table 1, all WAN media options will achieve the goal Good Performance [Throughput=300] (See all contributions are green). However, their respective WAN Response Service Times (Table 1, second row) are distinguished with ++, +, S+, reflecting the different degrees to which the three different options contribute.

From the perspective of the goal [CostPerHour=$1.5/h], the costs of Fiber, Cable, Google USA-wide, which respectively are 1.9, 1.75, and 1.0, have contributions -, S-, + respectively. Hence, Google USA-wide achieves the system goal of Good Performance, and at the same with the lowest cost among the options considered.

As in the above descriptions, A SIG serves as a notational convention for exploring options for achieving business and system NFR goals, such as cost and performance. At the end, it also serves as a record of the rationale for the various design decisions made, in consideration of tradeoff analysis (pluses and minuses) [3]. In Figure 2-1, for example, the pros (green lines) and cons (the red lines) from different options represent the contributions the different options make towards the NFR (soft-) goals. A Claim, "Lower Cost is the most important", is the rationale for the particular decision made on the VMS Consolidated Cloud. Similarly in Figure 2-2, the selection on Google USA-wide has the rationale with the claim "Achieve the Throughput Goal with the Lowest Cost."

Many uses of SIG can be found in the literature. One such use, dealing with trustworthiness, can be found in [4].

Step 2: Estimate application workload

The workflow for the VMS Display-Status application, together with resource consumption, is shown in Figure 3. The estimated flow and consumption metrics were extracted from runtime transaction monitors [5, 6] and a standard relational database benchmark [7].

The Display-Status application flow is annotated in the following steps: 1. A small request message (1K bytes) is initiated by an average of 300 concurrent users; 2. The message is sent over the internet to a cloud provider's Frontend engine; 3. The Frontend engine executes the Display-Status program (.0001 service units); 4. The cloud Database engine updates and retrieves the status window (2.5 service units, 4 gigabytes); 5,6,7. The response message is returned; and 8. The (1 million bytes) message is displayed in the regional center.

Step 3: Describe simulation experiment

Figure 4 shows an operational Google App Script graphical user interface, which can be used to describe a single simulation/forecaster experiment [8]. The GUI has four sections, which can be used to collect the data elements for the selected simulation experiment:

Create Section - This section groups the XML file actions and establishes the simulation goals for response time (3 seconds) and throughput – i.e., applications transactions per minute (300).

Application Group Section - This section accumulates the operating characteristics of all application workloads - operating hours (24) and primary database size (4 gigabytes).

Application Workload Section - This section estimates the workload characteristics of each application – the total daily requests, workload mix percentage (10), keying time (10), think time, and the number of input and output operations.

Cloud Infrastructure Configuration Section - This section describes the attributes of each component in the cloud infrastructure topology – the number of Clients (300), network elements, frontend compute characteristics (.0001), and the database storage characteristics (2.5).

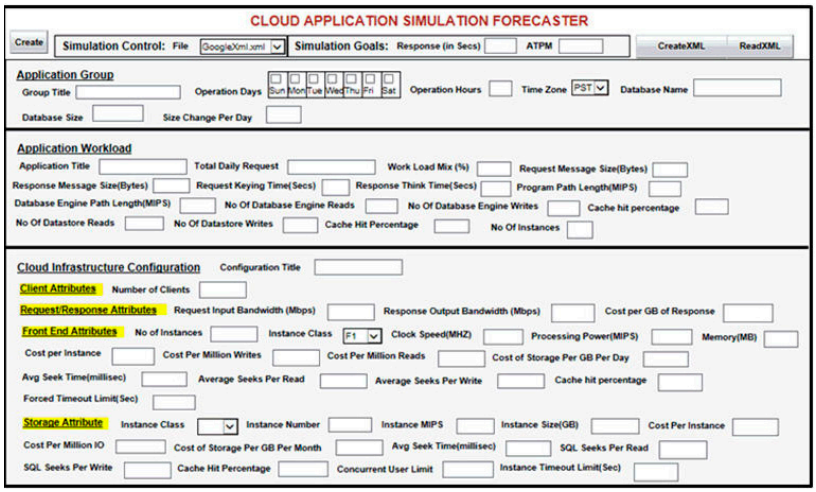


Figure 4. The GUI for the VMS simulation experiment.

```
<runtitle>TEST for VMS 4.0 GB D1 DB Server</runtitle>
<simgoals>
  <responsesecs>3</responsesecs>                    <!-- average response time in seconds -->
  <atpm>300</atpm>                                  <!-- average application txns per minute -->
</simgoals>
<applicationgroup>
  <operationhoursday>24</operationhoursday>         <!-- hours per day operational -->
  <dbsizegbyte>4.0</dbsizegbyte>                    <!-- major database size giga bytes -->

   <applicationworkload>                            <!-- one application workload per transaction
    <apptitle>Display-Status</apptitle>
    <workloadmixp>10</workloadmixp>                 <!-- workload mix percentage for this application
    <requestsmsgbytes>1000</requestsmsgbytes>       <!-- request message size in bytes -->
    <responsemsgbytes>1000000</responsemsgbytes>    <!-- response message size in bytes -->
    <requestkeytimesec>10</requestkeytimesec>

<infrastructureconfig>                              <!-- infrastructure platform configuration
  <configtitle>OLTP Web Database Cloud Model</configtitle>
    <webclient>                                     <!-- web client used to generate app transactionw
      <numberclients>300</numberclients>            <!-- number of concurrent web clients -->
    </webclient>

    <gaefrontend>                                   <!-- Google App Engine Frontend GAE FE description
      <instancenbr>10</instancenbr>                 <!-- GAE FE number of instances default -->
      <instanceclass>F1</instanceclass>             <!-- GAE FE class default F1, F2, F4, F4_1G -->
      <costperinsthour>0.08</costperinsthour>       <!-- cost $00.00 for each GAE FE instance hour usage
    <clouddbengine>                                 <!-- Google CloudSQL database engine description
      <sqlinstanceclass>D1</sqlinstanceclass>       <!-- Google CloudSQL db engine class
      <sqlcostperinsthour>0.10</sqlcostperinsthour> <!-- cost $00.00 for each Google CloudSQL db
      <sqlcostpermio>0.10</sqlcostpermio>           <!-- cost $00.00 for Google CloudSQL db engine
      <sqlcoststoragepergbyteperm>0.24</sqlcoststoragepergbyteperm> <!-- cost $00.00 Google CloudSQL
```

Figure 5. A selected XML description for the VMS simulation experiment.

Step 4: Generate simulation experiment XML

The Google App Script can generate XML to describe a simulation/forecaster experiment. A selected sample of the XML, in Figure 5, is highlighted in red to demonstrate XML's ability to communicate a complete simulation experiment description to multiple discrete event simulation modeling frameworks supporting multiple cloud provider infrastructures. The important additional annotations are explained with the following XML tag descriptions:

<apptitle>Display-Status
<workloadmip>10
<requestsmsgbytes>1000
<responsemsgbytes>1000000
<instancenbr>10
<instanceclass>F1
<costperinsthour>0.08
<sqlinstanceclass>D1
<sqlcostperinsthour>0.10
<sqlcostpermio>0.10
<sqlcoststoragepergbyteperm>0.24

The XML is designed to describe a complete simulation for the following open source discrete event simulation frameworks: SimPy [9], CloudSim [10] and OMNet++ [11].

Step 5: Run Simulation experiment

Simulation models are created for understanding the behavior of a complex system without actually constructing the system. Simulation eliminates the time and expense that are needed to: design, code and test the software system, not to mention the physical hardware equipment either.

Figure 6 presents the results of using an open source discrete event simulation framework, SimPy, for a particular simulation experiment, which corresponds to the Google App Engine cloud characteristics for 300 concurrent users. The report is formatted to closely align with Google's monthly invoice format. An explanation of the significant report data elements follows:

Section I. Latency-goal (3 seconds), Throughput-goal (300 transactions per minute), Op-hours (operation hours per day 24) and #-users (number of concurrent users 300), Display-Status application SIM-THROUGHPUT-PER-MIN (throughput per minute 367.86), and Txn-workload-% (the percent of total transactions dedicated to the Display-Status application 10).

Section II. The estimated daily frontend compute resource usage for 24 hours per day CHARGE is $83.35 daily or $2,500.39 per 30-day month.

Section III. The estimated daily database (D1 CloudSQL) resource usage for one database instance, read-write operations and storage is $12.40 per 30-day month is $372.20. Here, D1 is the cheapest, hence with the lowest computing, infrastructure of Google App Engine, among 32 different infrastructures which enable the use of SQL.

The simulation run of 4-hours-simulation-clock-time, using a laptop computer (1.30GHz), took 2 minutes execution time at a very low cost. The simulation input variables can be modified to describe application workload changes and alternative cloud infrastructure configurations at a minimum cost.

```
I.        Simulation-run-title                 ce-time    Throughput-goal   App-group     smtwtfs  Op-hours #-users
-----------------------------------------   ---------- ---------------- -------------- -------  -------- -------
TEST for VMS 4.0 GB D1 DB Server          ...) 12:3 secs         300 tpm       VMS        yyyyyyy    24        300

  App-txn-title     SIM-MINUTES SIM-AVG-I     SIM-THROUGHPUT-PER-MIN SIM-TXN-COUNT Txn-workload-%
  --------------    ----------- ---------  ... ---------------------- ------------- --------------
  Display-Status         239.51                            367.86          88108             10
  Other                  239.51                           2974.79         712191             90
                                           ...
II. Simulation-forecast-of-GAE-frontend-    ource-usage                                  USED        CHARGE($):
-----------------------------------------   ------------------------------             --------    ---------
   1.  Daily Instance     F1 Hours [avera ...ice 24 hours ea $ 0.08/Hour]                  240         19.20
   2.  Daily Bandwidth Out average Gigab   Gigabyte]                                    534.553       64.15
   3.  30-day Month Total Estimate                                                                   2500.39
                                           ...
III. Simulation-forecast-of-CloudSQL-var    ce-usage                                              DEBITS($):
-----------------------------------------   ------------------------------------                  ---------
   1.  Daily SQL Service     D1 Usage Hou ...24 hours, $ 0.10/Hour]                                   2.40
   2.  Daily SQL Service Read and Write     ge     99.7 million RWs, $0.10/million]                  9.97
   3.  Daily SQL Service Disk Usage [ave    .00 GB for the month, $ 0.24/GB/month]                    0.03
   4.  30-day Month Total Estimate                                                                 372.20
```

Figure 6. VMS SimPy simulation report.

Step 6: Validate distance to cloud benchmark

After a review of the simulation report, one question remains – "Is the simulation throughput of 368 transactions-per-minute (tpm) a reasonable result?" The Silverlining Lab at the University of Texas at Dallas maintains the results of cloud TPC-C [7] benchmarks, ranging from 10 to 6000 concurrent users and throughput ranging from 12.5 tpmC to 7,029 tpmC, for two kinds of platforms – Java and Python – and for a variety of CloudSQL and NoSQL infrastructures [12].

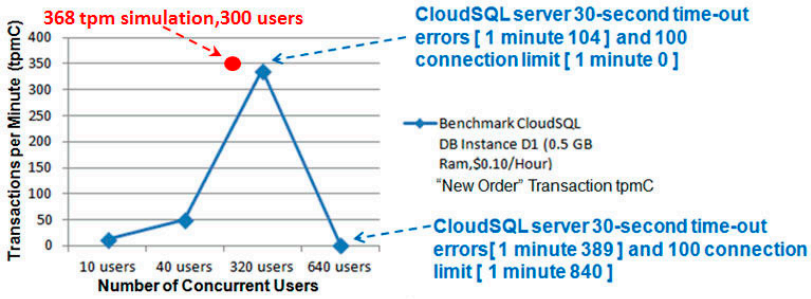
Figure 7 plots (in blue) a subset of the cloud TPC-C benchmark to show 320 users with 338.1 transactions-per-minute throughput, in the table on the bottom. The VMS Display-Status simulation shows 300 users with 368 transactions-per-minute (in red) throughput, with a similar application profile (similar workload, service times, database activity and cloud infrastructure configuration), in the graph on the top.

As seen in Figure 7, the comparison shows that the VMS simulation result is indeed very close (distance 320 to 368 throughput) the corresponding benchmark result. Hence, the answer to the above question, "Is the simulation throughput of 368 transactions-per-minute (tpm) a reasonable result?", would be in the affirmative.

Conclusions: What the Silverlining Simulator Can Do

A common problem confronting just about every CIO is the cost and performance issues, when migration of software to the cloud is considered - a CIO needs to budget the cost and ensure that the performance of the system is not unduly compromised by the migration. This should be true whether the cloud is public, private, or hybrid, since the cost and performance in a cloud-based system is related to the configuration of the resources - different servers and platforms - supporting the intended software system. To aid the IT department and specifically the CIO, we have developed a simulator, Silverlining, that can forecast the cost and performance of a Google GAE and CloudSQL cloud infrastructure.

Our simulator is benchmarked on the GAE with an industry standard benchmark application, TPC-C, which covers a class of online transaction processing (OLTP) relational database-centric applications , and we deployed a configurable Platform-



	10 users	40 users	320 users	640 users
Benchmark CloudSQL DB Instance D1 (0.5 GB Ram,$0.10/Hour) tpmC	12.5	50.4	338.1	3
CloudSQL Maximum Number of FrontEnd Instances (F1 $0.08/Hour/Instance) 128MB MEM, 600MHZ CPU	1.0	3.0	12.0	16.0

Figure 7. VMS simulation data point to TPC-C benchmark curve comparison.

as-a-Service (PaaS) on this cloud with an indication of the class of Infrastructure-as-a-Service (IaaS) to be used. We then ran several experiments on this cloud system to collect cost and performance data. This data seeds the simulator-framework SimPy, which is written in the Python programming language, and the simulator is then able to predict, to a high degree of accuracy, the cost and performance of operating a similar software system to the cloud.

We have described the essentials of our system, Silverlining, in this paper and interested readers are referred to some of the references at the end of this article for further details. Further research in this area relates to identifying adjustment factors when the cloud to be migrated to is not exactly like the GAE, to creating a revised local stand-alone GUI front-end for the simulator (the current Google App Script GUI requires internet connectivity to operate properly), and to porting the simulator to other domains, in order to cover a wide range of software applications and a variety of clouds. The simulator source code can be obtained by a requesting email to any of the authors.

Acknowledgments

This work wouldn't have been possible without the generous Grant from Google and the untiring help of Michael Brauwerman at Google. Five Silverlining research team members also deserve special recognition — Dr. Sam Supakkul, Dr. Rutvij Mehta, Alan Anderson, Yishuai Li and John McCain (NTT Data) — for collaborating, encouraging and leading the way.

REFERENCES

1. S. Supakkul. Softgoal Profile Tool Users Guide, <http://www.utdallas.edu/~supakkul/tools/softgoal-profile/softgoal-profile.html>.
2. Grigorik, Google, Making the Web Faster at Google and Beyond Presentation, November 2012.
3. L. Chung, B. A. Nixon, E. Yu and J. Mylopoulos. Non-Functional Requirements in Software Engineering, Kluwer Academic Publishers, Boston. 2000.
4. N. Subramanian, S. Drager, W. McKeever, "Identifying Trustworthiness Deficit in Legacy Systems Using the NFR Approach", CrossTalk, The Journal of Defense Software Engineering, Special Issue on Legacy System Software Sustainment, Vol. 27, No. 1, January/February, 2014, pp. 4-11.
5. Oracle Enterprise Manager <http://www.oracle.com/us/products/enterprise-manager>, 2014.
6. HP Loadrunner, <http://www8.hp.com/us/en/software-solutions/loadrunner-load-testing>, 2014.
7. Transaction Processing Council, <http://www.tpc.org/tpcc, 2014>.
8. Google App Script Reference, <http://developers.google.com/apps-script>, 2013.
9. Simpy Documentation, <http://simpy.sourceforge.net/old/simpy-manual>, 2013.
10. R. Calheiros, R. Ranjan, A. Beloglazov, C. De Rose and R. Buyya. CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms, SOFTWARE PRACTICE AND EXPERIENCE, John Wiley & Sons, 2010.
11. OMNETT++ Reference, <www.omnetpp.org>.
12. T. L. Hill, Confirming and Reconfirming Architectural Decisions: a Goal-Oriented Simulation Approach, Ph.D. Disserttion, The University of Texas at Dallas, 2014.

ABOUT THE AUTHORS

Prof. Lawrence Chung is in Computer Science at University of Texas at Dallas. Working in Requirements Engineering and Architecture, he was the principal author of "Non-Functional Requirements in Software Engineering", and has been involved in developing "RE-Tools" (a multi-notational tool for RE) with Sam Supakkul, "HOPE" (a smartphone application for disabilities) with Rutvij Mehta, and "Silverlining" (a cloud forecaster) with Tom Hill. He received his Ph.D. in Computer Science in 1993 from University of Toronto.

The University of Texas at Dallas
800 W Campbell Road
Richardson, Texas 75080
Phone: 972-883-2178
E-mail: chung@utdallas.edu

Nary Subramanian is an Associate Professor of Computer Science at The University of Texas at Tyler, Tyler, Texas. Dr. Subramanian received his Ph.D. in Computer Science from The University of Texas at Dallas. He is a Faculty Fellow for Service Learning at UT Tyler's Center for Teaching Excellence and Innovation. He has over fifteen years' experience in industry in engineering, sales, and management. His research interests include software engineering, systems engineering, and security engineering.

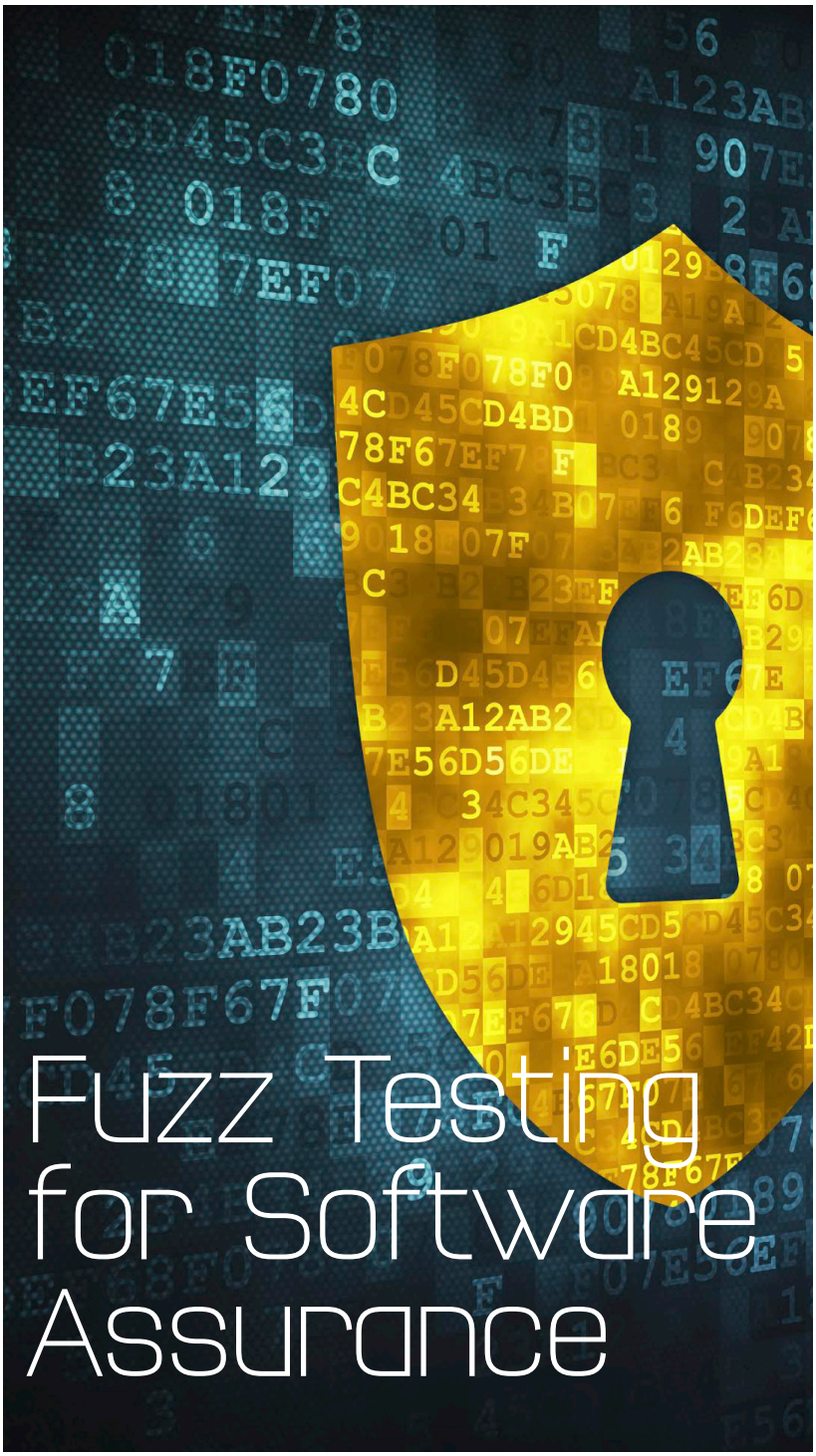
The University of Texas at Tyler
3900 University Blvd.
Tyler, Texas 75799
Phone: 903-566-7309
E-mail: nsubramanian@uttyler.edu

Thomas Hill became an EDS/HP Fellow Emeritus, when he retired as the Director of EDS/HP Fellows and Distinguished Engineering after 40 years of service. He served as a systems engineer, since becoming an Air Force officer in 1963, building the first joint intelligence agency time-sharing system. He joined EDS in 1970, as a systems engineer. He has a BS and MBA in Information Technology and received his Ph.D. in software engineering from The University of Texas at Dallas.

The University of Texas at Dallas
800 W Campbell Road
Richardson, Texas 75080
Phone: 469-767-5011
E-mail: tom.hill.fellow@gmail.com

Grace (Eun-jung) Park is a Ph.D. Student of Computer Science at The University of Texas at Dallas. She has about fifteen years of industrial experience in software engineering, and she is an IT professional engineer in her native country. Her research interests are Requirements Engineering, System/Software Architecture, and Cloud Computing.

The University of Texas at Dallas
800 W Campbell Road
Richardson, Texas 75080
Phone: 469-450-3940
E-mail: exp130530@utdallas.edu



Fuzz Testing for Software Assurance

Vadim Okun, NIST
Elizabeth Fong, NIST

Abstract. Multiple techniques and tools, including static analysis and testing, should be used for software assurance. Fuzz testing is one such technique that can be effective for finding security vulnerabilities. In contrast with traditional testing, fuzz testing only monitors the program for crashes or other undesirable behavior. This makes it feasible to run a very large number of test cases. This article describes fuzz testing, its strengths and limitations, and an example of its application for detecting the Heartbleed bug.

Fuzz Testing and its Role for Software Assurance

Software assurance is level of confidence that software is free from vulnerabilities, either intentionally designed into the software or accidentally inserted at any time during its life cycle and that the software functions in the intended manner [1].

Multiple techniques and tools should be used for software assurance. Static analysis tools examine code for weaknesses without executing it. On the other hand, testing evaluates a program by executing it with test inputs and then compares the outputs with expected outputs. Both static analysis and testing have a place in the software development life cycle.

Positive testing checks whether a program behaves as expected when provided with valid input. On the other hand, negative testing checks program behavior by providing invalid data as input. Due to time constraints, negative testing is often excluded from the software development life cycle. This may allow vulnerabilities to persist long after release and be exploited by hackers. Fuzz testing is a type of negative testing that is conceptually simple and does not have a big learning curve.

Fuzz testing, or fuzzing, is a software testing technique that involves providing invalid, unexpected, or random test inputs to the software system under test. The system is then monitored for crashes and other undesirable behavior [2].

The first fuzzing tool simply provided random inputs to about 90 UNIX utility programs [3]. Surprisingly, this simple approach led to crashes or hangs (never-ending execution) for a substantial proportion of the programs (25 to 33%).

Fuzz testing has been used to find many vulnerabilities in popular real-life software. For example, a significant proportion of recent vulnerabilities in Wireshark (http://www.wireshark.org), a network protocol analyzer, were found by fuzzing. Large organizations are taking note. For example, Microsoft includes fuzz testing as part of its Security Development Lifecycle (http://www.microsoft.com/security/sdl/default.aspx).

A fuzzing tool, or fuzzer, consists of several components and a fuzzing process involves several steps [4]. First, a generator produces test inputs. Second, the test inputs are delivered to the system under test. The delivery mechanism depends on the type of input that the system processes. For example, a delivery mechanism for a command-line application is different from one for a web application. Third, the system under test is monitored for crashes and other basic undesirable behavior.

Strengths and Limitations of Fuzz Testing

Fuzz testing is conceptually simple and may offer a high benefit-to-cost ratio. In traditional testing, each test case consists of an input and the expected output, perhaps supplied by an oracle. The output of the program is compared to the expected output to see whether the test is passed or failed. In the absence of executable specifications or a test oracle (e.g. a reference implementation or checking procedure), finding the expected output for a lot of test cases can be costly. In contrast, fuzz testing only monitors the program for crashes or other undesirable behavior. This makes it feasible to run hundreds of thousands or millions of test cases.

Fuzz testing is effective for finding vulnerabilities because most modern programs have extremely large input spaces, while test coverage of that space is comparatively small [5].

While static source code analysis or manual review are not applicable to systems where source code is not available, fuzz testing may be used. Fuzz testing is a general technique and therefore may be included in other testing tools and techniques such as web application scanners [6].

Fuzz testing has a number of limitations [7]. First, exhaustive testing is infeasible for a reasonably large program. Therefore, typical software testing, including fuzz testing, cannot be used to provide a complete picture of the overall security, quality or effectiveness of a program in any environment. Second, it is hard to exercise the program thoroughly without detailed understanding, so fuzz testing may often be limited to finding shallow weaknesses with few preconditions. Third, finding out what weakness in code caused the crash may be a time-consuming process. Finally, fuzz testing is harder to apply to categories of weaknesses, such as buffer over-reads, that do not cause program crashes.

Fuzzing Approaches

Test input generation can be as simple as creating a sequence of random data [3]. This approach does not work well for programs that expect structured input. No matter how many tests are generated, the vast majority might only exercise a small validation routine that checks for valid input.

In regression testing, valid inputs may be collected, for example, from historical databases of unusual inputs that caused errors in the past versions of the software, and then supplied to the program without modification. Such approach can help uncover a weakness that reoccurs between versions or implementations, but is unlikely to uncover new weaknesses.

Most fuzz generators can be divided into two major categories: mutation based and generation based fuzzers [8]. A mutation based fuzzer produces test inputs by making random changes to valid test input, such as those from regression testing. This approach can be quickly applied to systems, such as protocols or word processors, that accept complex inputs. However, the coverage is only as strong as the set of valid test inputs. If there is no valid test input for a particular system component, the mutation based fuzzer is unlikely to cover this component.

A generation based fuzzer produces test inputs based on some specification of the input format. While implementing the input format in enough detail requires a significant upfront effort, the generation based fuzzer can achieve very high coverage at lower cost.

A relatively recent approach, whitebox fuzz testing, combines symbolic execution with constraint solving to construct new inputs to a program [9]. Whitebox fuzzing has been used by Microsoft to find one third of all the bugs discovered by file fuzzing during the development of Windows 7.

The next step after producing test inputs is providing them to the system under test. Some common delivery mechanisms are files, environment variables, command line and API parameters, and operating system events, such as mouse and keyboard events.

Fuzz testing does not require knowing the expected output, instead there must be monitoring for crashes or other generally undesirable behavior. However, many types of weaknesses do not produce clearly undesirable behavior. Therefore, more sophisticated detection that test input caused a failure can significantly expand the classes of weaknesses uncovered by fuzzing. The following Section describes an example of using dynamic analysis tools to detect a weakness that does not cause a crash under normal operation.

Protocol Testing Experiment

The Heartbleed bug is a widely known vulnerability in OpenSSL, a popular implementation of the cryptographic protocols Secure Sockets Layer (SSL) and Transport Layer Security (TLS). Briefly, under the Heartbeat protocol, the client sends a message and the message length to the server, and the server echoes back the message.

The Heartbleed vulnerability can be exploited to leak confidential information, including passwords and authentication data. It was caused by the failure of OpenSSL to validate the message length, which caused Buffer over-read weakness [10]. For more details, an interested reader can examine Heartbit, an abstracted version of the OpenSSL code demonstrating the Heartbleed vulnerability [11]. Even though buffer overflow, which includes buffer over-read, is a well-known weakness, software assurance tools missed it [12].

Simple fuzz testing, which looks for crashes, would not have detected Heartbleed. The reason is that buffer over-reads rarely lead to program crashes. However, fuzz testing in combination with a memory error detection tool, may have detected Heartbleed, as demonstrated in [13].

Memory error detection tools, such as Valgrind (http://valgrind.org) and AddressSanitizer (http://code.google.com/p/address-sanitizer), are a type of dynamic analysis tools that can be used to instrument code to detect various memory errors, such as buffer overflows and use-after-free errors that may not cause a crash under normal operation.

In the first experiment, [13] ran a vulnerable version of OpenSSL with Valgrind. When the fuzzer sent an exploiting Heartbleed request, Valgrind produced an error trace highlighting the bug. In the second experiment, a vulnerable version of OpenSSL was compiled with the AddressSanitizer compiler option. When an exploiting Heartbleed request was sent to the server, it terminated and an error trace was produced. In both experiments, a programmer could use the error trace to find the Heartbleed bug.

Conclusions

Typical software testing, including fuzz testing, cannot be used alone to produce bug-free software. Since fuzz testing does not require a sophisticated oracle, it can quickly test a very large number of unexpected inputs. When combined with appropriate supplemental tools, this makes it possible to find security vulnerabilities, such as the Heartbleed bug, which may be missed by other tools. As demonstrated by a large number of bugs recently

discovered in production software, fuzz testing can be used to increase software assurance.

Disclaimer:

Any commercial product mentioned is for information only; it does not imply recommendation or endorsement by NIST nor does it imply that the products mentioned are necessarily the best available for the purpose.

ABOUT THE AUTHORS

Vadim Okun is a Computer Scientist at the National Institute of Standards and Technology, where he is leading the SAMATE (http://samate.nist.gov/) team. His current research focuses on software assurance, in particular, the effect of tools on security. He organized Static Analysis Tool Expositions (SATE) – large-scale evaluations to support research in, and improvement of, static analysis tools. Previously, Okun contributed to the development of automated software testing methods: specification-based mutation analysis and combinatorial testing. He received a Ph.D. in Computer Science from University of Maryland Baltimore County.

E-mail: vadim.okun@nist.gov

Elizabeth Fong is a computer scientist currently working in the Software and Systems Division, Information Technology Laboratory of National Institute of Standards and Technology. She performs research, analysis, and evaluation of innovative information technology software and practices with emphasis on new fields of computer technology for Federal Government applications. Recent work involved software assurance, smart card technology, XML registry framework and standards, and interoperability testing technology. Earlier work included technologies and standards development for Object-Oriented, distributed database management and agent-based computing. She has many years of experience in the development of software testing and reference models for information technologies. She received B.SC (Math) New York University, New York, NY, M.S. (Computer Science) Stanford University, CA and Graduate Courses (Computer Science) U. of Maryland, MD.

E-mail: efong@nist.gov

REFERENCES

1. CNSS, National Information Assurance (IA) Glossary CNSSI-4009, 26 April 2010, p. 69, http://www.ncix.gov/publications/policy/docs/CNSSI_4009.pdf.
2. Wheeler, David A. and Rama S. Moorthy, "SOAR for Software Vulnerability Detection, test and Evaluation," IDA paper P-5061, July 2014.
3. Miller, Barton P., Lars Fredriksen, and Bryan So, "An Empirical Study of the Reliability of UNIX Utilities," Communications of ACM 33(12):32-44 (Dec. 1990).
4. McNally, R., Yiu, K., Grove, D., and Gerhardy, D., "Fuzzing: The State of the Art," Technical Note DSTO-TN-1043, 2012.
5. Householder, Allen D., "Why Fuzzing (Still) Works," in Metrics and Standards for Software Testing (MaSST) workshop, p. 39-59, December 2012, http://samate.nist.gov/docs/MaSST_2012_NIST_IR_7920.pdf.
6. Fong, Elizabeth, Romain Gaucher, Vadim Okun, Paul E. Black, and Eric Dalci, "Building a Test Suite for Web Application Scanners," 41st Hawaii Int'l Conf. on System Sciences (HICSS), January 2008.
7. West, Jacob, "How I Learned to Stop Fuzzing and Find More Bugs," DefCon, Las Vegas, August 2007.
8. Neystadt, John, "Automated Penetration Testing with White-Box Fuzzing," http://msdn.microsoft.com/en-us/library/cc162782.aspx, Microsoft, February 2008.
9. Bounimova, Ella, Patrice Godefroid, and David Molnar, "Billions and Billions of Constraints: Whitebox Fuzz Testing in Production," ICSE 2013:122-131.
10. Wheeler, David A., "How to Prevent the next Heartbleed," http://www.dwheeler.com/essays/heartbleed.html, 2014.
11. Bolo – Joseph T. Burger, "Heartbit test case," http://samate.nist.gov/SARD/view_testcase.php?tID=149042.
12. Kupsch, James A. and Miller, Barton P. "Why Do Software Assurance Tools Have Problems Finding Bugs Like Heartbleed?" Continuous Software Assurance Marketplace, 22 Apr. 2014, https://continuousassurance.org/swamp/SWAMP-Heartbleed.pdf.
13. Vassilev, Apostol and Christopher Celi, "Avoiding Cyberspace Catastrophes through Smarter Testing," IEEE Computer, October 2014; 47(10):86-90.

Upcoming Events

Visit ‹http://www.crosstalkonline.org/events› for an up-to-date list of events.

Software and Supply Chain Assurance (SSCA) Forum (Theme: Enterprise Risk Management)
Co-sponsored by organizations in DoD, DHS, NIST & GSA
9-11 March 2015
McLean, VA
Open to public; no cost to attend, but registration is required.
https://register.mitre.org/ssca

International Conference on Software Quality
Long Beach, CA
9-11 Mar, 2015
http://asq-icsq.org/index.html

IEEE International Conference on Cloud Engineering (IC2E 2015)
March 9-13, 2015
Tempe, AZ
http://conferences.computer.org/IC2E/2015

Conference on Systems Engineering Research (CSER 2015)
March 17-19, 2015
Hoboken, NJ
http://www.stevens.edu/sse/CSER2015

The Southwest Cybersecurity Summit
March 25-26, 2015
Phoenix, AZ
http://www.afei.org/events/5A06/Pages/default.aspx

IEEE Mobile Cloud 2015 : The 3rd IEEE International Conference on Mobile Cloud Computing
March 30-April 3, 2015
San Francisco, CA
http://mobile-cloud.net

ETAPS 2015 - 18th European Joint Conferences on Theory and Practice of Software
London, United Kingdom
11-19 April, 2015
http://www.etaps.org/2015

The Fourteenth International Conference on Networks ICN 2015
April 19-24, 2015
Barcelona, Spain
http://www.iaria.org/conferences2015/ICN15.html

WICSA 2015 – 12th IEEE Conference on Software Architecture
Montreal, QC, Canada
20-24 April 2015
http://www.computer.org/portal/web/conferences/calendar

SATURN 2015- Software Engineering Institute (SEI) Architecture Technology User Network 2015
27-30 April 2015
Baltimore, MD
http://www.sei.cmu.edu/saturn/2015

Systems Engineering Test Evaluation Conference
April 27-29 2015
Canberra, Australia
http://sete2015.com.au

11th ACM/IEEE- Symposium on Architectures for Networking and Communications Systems (ANCS'15)
7-8 May 2015
Oakland, CA
http://www.ancsconf.org

The 37th International Conference on Software Engineering
May 16-24, 2015
Firenze, Italy
http://2015.icse-conferences.org

The 39th Annual International Computer, Software & Applications Conference
July 1-5, 2015
Taichung, Taiwan
http://www.computer.org/portal/web/COMPSA

INCOSE 25th Annual Symposium IS 2015
13-16 July 2015
Seattle, WA
http://www.incose.org/newsevents/events/details.aspx?id=255

STC 2015, the 27th Annual IEEE Software Technology Conference
October 12 - 15, 2015
Long Beach, CA
http://ieee-stc.org

The Proof is in the Testing:

After writing BackTalk columns for over 15 years, I'm out of ideas. No clues. Nothing to write about. Zip. Zilch.

I usually have a great stash of ideas – and I am SO used to grabbing my iPhone, and telling Siri to remind me "Write up story of the non-working GPS" or "Maybe hibernating skunks for next column?" Not this time. Not a single idea involving Testing and Diagnostics.

Of course, I have a great excuse. I am teaching a course on high-integrity programming this semester here at the university. As any BackTalk reader for the last 15 years might know, I was (and still am) an Ada advocate. It's a really good language for teaching high-integrity coding practices. We have decided to cover more "secure programming" in our curriculum, and I wanted to test-drive a course as "Special Topics" before we make it a permanent part of the curriculum. It's keeping me pretty busy.

Nice thing about Ada – the language thinks about security for you. It gets real picky about mathematical conversions, and cheerfully lets the user know if there is a overflow or underflow in the conversion. In fact, you have to make a conscious decision to turn off error checking. And, why would you ever do that

Imagine a very simple Java program that inputs an integer from the keyboard, and adds one to it. In Java, if the number is equal to the largest possible integer, then adding one converts is to the smallest integer. It "wraps around." And vice versa – subtracting one from the smallest negative integer will give the largest possible positive integer. Note that Java is one of the most used languages used in introductory computer science – currently it's #2 in popularity. You might think that C++ would be #1, but #1 is Python. However, Java is more of a "software engineering" language than Python, and many educators expect it to move back to #1 next year. All I am saying is that if you can't trust adding 1 to a number – what can you trust? It makes teaching "safe and secure" programming a challenge.

What about other common languages? In Python, the integer is converted to a "long", which has unlimited length (obviously, implemented in software, not hardware). In C and C++, the behavior is undefined. In C#, it is possible to automatically "catch" this – but only if you are in a checked context.

What I am saying is that once you've had the thought, "Gee – I am adding to an integer here – should I worry about overflow?" you can't unthink it. It is no longer an error. It's a condition you considered, and then you made the decision to not worry about it.

And what if the software later fails because of an overflow or underflow? Of course, that is highly unlikely, right? Heard of the Ariane 5 (https://www.ima.umn.edu/~arnold/disasters/ariane.html)?

Brief summary – back in 1996 the European Space Agency launched the Ariane 5 rocket, designed to launch two satellites into orbit. The Ariane 5 was the result of over $7 billion in development, and the rocket and cargo itself cost over $500 million. On its first launch, the guidance system catastrophically failed. How was this possible? After all, the guidance software was based on the well-tested guidance software from the previous rocket version, the Ariane 4.

> *Let me quote from James Gleick (http://www.around.com/ariane.html): "…the guidance system's own computer tried to convert one piece of data -- the sideways velocity of the rocket -- from a 64-bit format to a 16-bit format. The number was too big, and an overflow error resulted. ….. in this case, the programmers had decided that this particular velocity figure would never be large enough to cause trouble. After all, it never had been before. Unluckily, Ariane 5 was a faster rocket than Ariane 4. One extra absurdity: the calculation containing the bug, which shut down the guidance system, which confused the on-board computer, which forced the rocket off course, actually served no purpose once the rocket was in the air. Its only function was to align the system before launch. So it should have been turned off. But engineers chose long ago, in an earlier version of the Ariane, to leave this function running for the first 40 seconds of flight."*

Years of work. $7 billion in development. And "the programmers had decided…" If only they had used Ada, right? As a matter of fact, the Ariane 4 and 5 software was written in Ada, which automatically checks for overflow and underflow, should have triggered a exception that could have been safely detected, handled, and recovered from. However, to quote from (http://en.wikipedia.org/wiki/Ariane_5):

The software was originally written for the Ariane 4 where efficiency considerations (the computer running the software had an 80% maximum workload requirement) led to four variables being protected with a handler while three others, including the horizontal bias variable, were left unprotected because it was thought that they were "physically limited or that there was a large margin of error."

In other words – due to hardware considerations (the Ariane 4 CPU was overloaded) – consciously ignoring automatic overflow and underflow checking saved a few machine cycles. Don't get me wrong – the Ariane 4 software was perfect for the given requirements (hardware and software). The processor in the Ariane 4 was heavily loaded – and the decision to turn off checking was probably well analyzed. It was only much, much later, when the code was reused for a different set of requirements, that the decision should have been revisited. But the code was "rock(et) solid" and well-tested – so why test it again? Copy and Paste. Control-C, Control-V, and go. Nothing to worry about, right?

It's even sadder that there was an exact backup copy of the guidance software running on the Ariane 5 – an exact copy. So when the primary guidance software failed, control was immediately transferred to the backup copy. Running the same code, and encountering the exact same unhandled error.

Software is like that. As Marin David Condic pointed out in his excellent write up found at (http://www.adapower.com/index.php?Command=Class&ClassID=FAQ&CID=328), it would be like reusing Corvette tires on a large 18 wheeler. Just because they are "tires" – the requirements and assumptions made initial development are no longer valid. What's "good" for Corvette tires might not be "good" for a fully loaded semi. Likewise, code needs to be analyzed and tested, even if it's been trusted for years. It's just not safe to reuse otherwise.

But then, we already know that testing and diagnostics are important, right?

David A. Cook
Stephen F. Austin State University

P.S. Come to think of it, maybe I do have an idea for this BackTalk than might work.

Homeland Security

Software and Supply Chain Assurance

Software and Supply Chain Assurance are essential to enabling the Nation's critical infrastructure.

To ensure the integrity of that infrastructure, the software and the IT supply chain must be secure and resilient.

The Software Assurance Community Resources and Information Clearinghouse provides corroboratively developed resources. Visit https://buildsecurityin.us-cert.gov/swa/index.html to learn more about relevant programs and how you can become involved.

Software and Supply Chain Assurance must be "built in" at every stage of development and supported throughout the lifecycle.

Visit https://buildsecurityin.us-cert.gov/bsi/home.html to learn about the practices for developing and delivering software to provide the requisite assurance. Sign up to become a free subscriber and receive notices of updates.

The Department of Homeland Security provides the public-private collaboration framework for shifting the paradigm to software and supply chain assurance.

https://buildsecurityin.us-cert.gov/swa

NAVAIR

309th SOFTWARE MAINTENANCE GROUP

CROSSTALK thanks the above organizations for providing their support.