CrossTalk

July/August 2012 The Journal of Defense Software Engineering Vol. 25 No. 4

The End of the PC

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
AUG 2012		**00-07-2012 to 00-08-2012**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
CrossTalk. The Journal of Defense Software Engineering. Volume 25, Number 4. July/August 2012	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 MXDEA,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

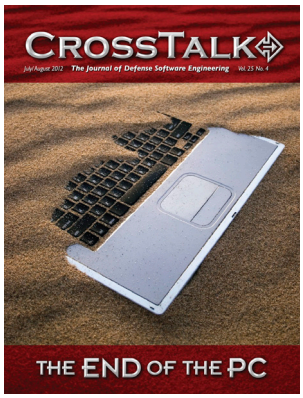
14. ABSTRACT

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)**	**32**	

THE END OF THE PC

Departments



Cover Design by Kent Bingham

The End of the PC

CrossTalk

NAVAIR Jeff Schwalb
DHS Joe Jarzombek
309 SMXG Karl Rogers

Publisher Justin T. Hill
Advisor Kasey Thompson
Article Coordinator Lynne Wade
Managing Director Tracy Stauder
Managing Editor Brandon Ellis
Associate Editor Colin Kelly
Art Director Kevin Kiernan

Phone 801-775-5555
E-mail stsc.customerservice@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 Defense (DHS). USN co-sponsor: Naval Air Systems Command. USAF co-sponsor: Ogden-ALC 309 SMXG. DHS co-sponsor: National Cyber Security Division in the National Protection and Program 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
309 SMXG for sponsoring this issue.

The End of the PC

I find solace in the knowledge that technological innovations continue to develop at ever-increasing speeds in a realm where increasing complexity and intricacy also come along as a certainty. This gives me faith that, technologically, we are only at the mere beginning of what we will achieve in the future. Processing speed has been exponentially increasing for decades, while the space and power needed to harness that speed has been decreasing dramatically. It is no wonder, then, that innovation would eventually lead us to technological innovations such as the mobile world we live in today.

Whether or not we are living in a "post-PC" world is often disputed, but there is little question that our lives and computing needs are becoming increasingly mobile. Our computing needs have been trending towards the cloud, with computing increasingly being migrated server side, and raw computing power becoming less and less important. What, then, is the fate of the personal computer as we understand it? To answer this question, we turn to Dean Klein's analysis of this shifting trend towards mobile and cloud computing solutions in *The PC is Dead – Long Live the PC: Making Computing More Personal*. Here we see that it may not be the end of the personal computer, but perhaps a re-envisioning of it. Perhaps the "personal computer" may need to evolve with technical innovation; insofar as that today's mobile solutions are much more of a personal computing device than the traditional PC.

With technology becoming increasingly complex, compounded with new platforms and environments that are on the forefront of technology, there has been an increasingly large call to evaluate a strategic shift to more fixed-price contracting to reduce acquisition costs and program risks for the DoD. Don O'Neill champions this view, advocating large-scale fixed-priced contracts for software-intensive system development in *Software Doctrine for Fixed-Price Contracting*. Mr. O'Neill tackles the common concerns of fixed-price contracting with viable solutions as well as advocates his vision of affordability and innovation through a doctrine of tenets.

With an increase in hardware complexity also brings the caveat of software complexity as well. To move forward with mission-critical software and still maintain the exacting quality and security requirements needed for aerospace and defense projects, we see an increasing reliance on the realm of testing and software assurance. In *Uncovering Weaknesses in Code With Cyclomatic Path Analysis,* the authors argue for a tighter integration of development and testing as a way to reduce security vulnerabilities, with a fascinating analysis between code coverage software testing methodologies for detecting vulnerabilities early in development.

With increasing intricacy of code necessitating complex sequence of events tests, the authors of *Efficient Methods for Interoperability Testing Using Event Sequences* provide a framework for using combinatorial methods, such as sequence covering arrays, with significantly fewer tests than previous procedures. The authors in *Building Confidence in the Quality and Reliability of Critical Software* tackle the issue of the imperative quality standards in aerospace and defense projects by evaluating the formal methodologies in software verification and testing to meet the high quality standards. Finally, we conclude the issue with Dale Childs and Paul Kimmerly's enlightening comments on process improvement by drawing connections to famous quotes in *Process Performance, Words of Wisdom*.

I think you will agree this issue's collection of articles provides helpful insight into current computing trends and the future of the PC.

Justin T. Hill
Publisher, CrossTalk

The PC is Dead—Long Live the PC: Making Computing More Personal

Dean A. Klein, Micron Technology, Inc.

Abstract. It has been more than 30 years since Steve Wozniak and Steve Jobs built the first Apple I computer in Job's garage and since IBM introduced the Model 5150 personal computer. In that time, the PC has become an integral part of our lives; more than 80% of all households own at least one, and rare is the business or workplace that can function without one.

Yet today, the dominance of the PC as our primary computing device is being threatened by new, emerging platforms like tablet computers, cell phones and ultrabook platforms. The rapid success of these new models, such as Apple's iPad, is leading many to surmise that the end of the PC era has arrived and that the PC is dead. But is the PC really dead, or are advances in technology simply allowing the PC to morph into something new and even more exciting?

Advances in Semiconductor Process Technology

In part, the tremendous advances brought about by the semiconductor industry have not only fueled the growth of personal computers, but also enabled these impressive new computing platforms. To put some scale on the advances of this industry, consider the clock rates and memory densities put forth by the leading manufacturers in the processor and memory arenas. When the IBM PC was introduced, the CPU ran at a whopping 4.77 MHz. Today's CPUs, running at 3.5 GHz with hyper-threaded, multicore CPUs on one chip, provide an improvement of almost 8,000 times. Similarly, consider that at the introduction of the Apple I computer, a state-of-the-art memory device could store 16K bits of information. Today's state-of-the-art NAND Flash chips store up to 128Gb of information on a single piece of silicon—an 8-million-fold increase.

Along with semiconductor speed and density improvements, we have seen a corresponding improvement in the power required to support the chips' circuits. There is also a tradeoff that can be made between performance and power so that if one is willing to accept a reduced level of performance, they can realize tremendous savings in power. For example, the same advances that can deliver a 3.5 GHz quad-core CPU that consumes 65W of power can also deliver a 1.5 GHz dual-core CPU that sips a miserly 5W of power.

Device-level Advancements

While Dynamic Random Access Memory (DRAM) devices have benefitted from the advances in semiconductor processes, DRAM power and performance are increasingly coming under pressure as areas that need major improvements. Fortunately, the memory industry has not been silent; it has delivered a stunning new technology into the hands of system designers in the form of the Hybrid Memory Cube (HMC). HMC makes a dramatic change to the architecture of the CPU and memory interface and provides a 15-fold increase in data bandwidth while lowering power consumption by 70%.

The HMC relies on a 3-D stack of semiconductor chips interconnected with a new technology called Through-silicon Vias (TSVs). The short length of these TSV interconnects gets much of the credit for reducing the energy-sapping inductance and capacitance of standard 2-D interconnects. However, equal credit must be given to new device architecture and I/O. The device architecture ensures an efficient use of the memory cells that get fetched by the system using the HMC device. In addition, unlike any previous memory device, the HMC uses high-speed serializer/deserializer channels for its I/O.

Perhaps the biggest advancement in memory over the past 20 years has been the development of NAND Flash memory. NAND Flash, which gets its name from the logical organization of its memory cells (in a "Not-AND" formation), offers two major advantages over DRAM: cost and power.

Although memory processes are generally the most cost-effective semiconductor processes on the planet, NAND Flash

is the king of cost effectiveness. This is due to the advanced process technology employed to build these miniature marvels and the relative simplicity of the process compared to other semiconductor processes. Today's most advanced NAND devices are now in production with geometries below 20nm (1nm = 1/1-billionth of a meter) and pack more than 128Gb (1Gb = 1 billion bits = 128 billion bytes) of storage on a single semiconductor die. Making NAND even more cost effective, consider the cell size of a NAND cell is about two-thirds that of a DRAM cell and one-tenth that of a Static Random Access Memory cell (like that used in a CPU cache), and each NAND cell stores two or three bits of digital data.

The power advantage of NAND Flash stems from the fact NAND Flash is nonvolatile, meaning that the NAND memory cells retain their value even when power is removed from the chip. This nonvolatility has enabled NAND Flash to become the predominant form of storage for cell phones and tablets. NAND Flash is also making tremendous inroads into computing in the form of Solid State Drives (SSDs) for both notebook and server applications. The advantage of having no moving parts allows SSDs to achieve extreme levels of reliability, performance, and low power consumption in these applications.

The New Computing Paradigm

The advances in CPUs and memory have enabled a wide variety of innovative and successful computing platforms, including tablets and smartphones. In the U.S. alone, smartphones accounted for an estimated six out of every 10 mobile phones sold in 2011 [1]. The successful iPad, its competitors and e-readers accounted for an estimated $64 million to $66 million in tablet computer sales in 2011 [2]. At the 2012 Consumer Electronics Show, the buzz about computers centered on a new class of mobile computing platform—the ultrabook. Ultrabooks are a new class of notebook computers that have the advantage of being thin and light while also offering long battery life. Smartphones, tablets, and ultrabooks represent a true paradigm shift in computing, from traditional personal computers to mobile and cloud-based solutions. Yet this shift is really making computing more personal.

Much of computing today requires connectivity. Whether it is e-mail, surfing the web, shopping, watching video, or gaming, the network is a required component of the computing experience. The growth in mobile network traffic has been phenomenal. Mobile network traffic alone in 2010 was more than three times the entire global Internet traffic in 2000 (237 petabytes per month versus 75 petabytes per month [3]). The increase in mobile traffic volume is also scaling in speed with the average smartphone network connection speed climbing from 625 kb/s in 2009 to 1040 kb/s in 2010 [3]. This increase in mobile network bandwidth and the overall increase of mobile network availability are two of the forces behind the enablement of the smartphone and tablet computer. The other significant force has become known as "the cloud."

Cloud Computing

If you happen to be fortunate enough to own an Apple iPhone 4S, you have probably used Siri, the voice-activated personal digital assistant. For Siri to work, many components are required. The most obvious component is the voice recognition used to interact with Siri. You might think this voice recognition is a function of the iPhone, but in reality very little processing is actually performed by the handset. Instead, the phone uses its connection to the Internet to send highly compressed code to servers that are set up to process the encoded data. These servers pick out the context and meaning of what was spoken to the phone. Once the context of the spoken commands is determined, the cloud is again used to provide the data being requested, in much the same way that you might perform an Internet search. If you asked Siri a question about the weather, the Siri servers would turn this into a web service request from a weather site accessing the site's data to provide your answer, which it would then deliver in text and speech.

There are countless other examples that might be given, but the important effect of the cloud is that it can provide both computing resources and data storage resources to computing devices. It means your personal computing device does not need the amount of processing power or storage that it would need if it were trying to perform all of its tasks locally. A cell phone can use the cloud to access data it does not have room to store in its limited amount of local memory; and a tablet computer can tap into cloud-based services to augment the compute power of its lower-powered processing chip.

The Effect of Usage Models

Human interface factors of non-PC computing platforms often dictate their usage. For the personal viewing of a YouTube video, the large screen of a tablet computer may be just about ideal. Tablets are also popular for watching movies on a plane, writing e-mail, surfing the web and playing games. The screen size is excellent for one-on-one viewing or sharing with another. On the other hand, the small screen size of most cell phones limits viewing to one user, and most web content is not optimized for the limited viewing area.

Screen size is not the only factor that determines usage. For authoring content, it is hard to beat the keyboard of a personal computer. The touch screen keyboards of tablets and smartphones are great for short e-mails and brief notes, but lack the tactile feedback needed for extended typing. Even simple functions such as cut and paste are made difficult by touch-based user interfaces.

Perhaps keyboard input will give way to speech input, allowing tablets to fully take the place of PCs, but it is hard to imagine an office full of people talking to their tablets in a productive manner. At the 2012 Consumer Electronics Show, one Chinese company was demonstrating a "thought-controlled" computer interface. If only it worked reliably!

Some have said the differentiator between PCs and smartphones/tablets is that the latter are content consumption devices, while PCs are content creation devices. This is true in some cases, but dead wrong in one very big case: pictures and video. All smartphones and most tablet computers sold today have at least one camera sensor built in. Mobile video consumption already accounts for more than 52% of mobile video traffic. In fact, in October 2011 alone, 201.4 billion online videos were watched around the world, reaching 1.2 billion unique viewers [4]. But where is all this video data coming from? Increasingly, smartphones and tablet computers are capturing video. At the end of 2011, users were uploading more than 60 hours of content to YouTube every minute—a number that is only expected to grow.

Secure Computing

One major concern for mobile devices of all types is data security. Mobile devices carry an increased risk of data loss, a risk that will restrict access to certain data by fixed (PC) computing resources. Technologies are entering the mobile space to help mitigate this concern. Micron's own C400 SED SSD is a solid state drive that incorporates 256-bit hardware encryption yet delivers the same performance, power advantages, and reliability of Micron's non- encrypted drives. Unlike software-based encryption, which is vulnerable to attack through the memory, operating system, and BIOS, the C400 SED's hardware-based encryption is performed in the SSD hardware, requiring user authentication to be performed by the drive before it will unlock, independent of the operating system. While this encryption technology is only shipping in personal computer drives today, it will find its way into tomorrow's mobile computing solutions.

One Size Does Not Fit All

The personal computer has been the driver and beneficiary of tremendous advances in semiconductor processes and products. Probably more than any other technology, the PC is responsible for huge gains in productivity in most developed parts of the world, and it will continue to be the vehicle for productivity advances in emerging economies around the globe. At the same time, the ease of use and ubiquity of mobile networks will make smartphone and tablet computing attractive for many users.

Is the PC dead? Hardly! But the definition of "personal computer" must not be too narrow. Today's smartphone or tablet is a much more personal computing device than the traditional PC. These platforms possess the processing and storage capability of the desktop personal computer of only a few years ago, yet we carry them in our pocket daily, and we talk to them and touch them.

The overall market for computing platforms has taken a leap as these new mobile computing platforms have gained popularity. While the growth of PC computing has stopped, the smartphone and tablet have combined to fuel continued growth in the semiconductor market for CPUs and memory—key components for all computing platforms. Innovative computing platforms will drive new innovations around the supporting circuitry for these platforms, fueling continued development of CPU, DRAM, and NAND Flash technologies.

Conclusion

Computing solutions, both mobile and fixed, are placing increased demands on cloud computing and storage infrastructure—demands for better access to ever-increasing amounts of data and unprecedented levels of server computing resources and storage resources. Increased use of multicore CPUs, software virtualization, high-speed DRAM, and SSDs are the key building blocks for tomorrow's cloud computing and storage environment.

So is the PC dead? Far from it! But the PC is changing. Our smartphones and tablets have more computing power and storage than our PCs did only a few years ago, enabling these platforms to be much more personal forms of computing than the PCs we have been used to. For many users (and many applications), smartphones and tablets will be the preferred computing device. For others, the PC will be irreplaceable until user interface technology makes the leap forward to enable content creation on new computing models—models that may be far different from even smartphones or tablets.

ABOUT THE AUTHOR

Dean Klein joined Micron Technology in 1999 and is Vice President of Memory System Development. Mr. Klein earned Bachelor of Science and Master of Electrical Engineering degrees from the University of Minnesota and holds more than 220 patents in the areas of computer architecture and electrical engineering. He has a passion for math and science education and is a mentor to the FIRST Robotics team <http://www.USFIRST.org> in the Meridian, Idaho, school district.

Micron Technology, Inc.
8000 S. Federal Way
MS 1-407
Boise, ID 83716
Phone: 208-368-4000

REFERENCES

1. "The NPD Group: Apple Leads Mobile Handsets in Q4 2011, But Android Attracts More First-Time Smartphone Buyers." npd.com. 06 Feb. 2012. Web. 07 Feb. 2012.
2. Jakhanwal, Vinita. "Ebook Reader Display Market to Double in." isuppli.com. 15 Dec. 2011. Web. 07 Feb. 2012.
3. "Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2010–2015." cisco.com. 1 Feb. 2011. Web. 07 Feb. 2012.
4. "More than 200 Billion Online Videos Viewed Globally in October." comscore.com. 14 Dec. 2011. Web. 07 Feb. 2012.

Homeland Security

The Department of Homeland Security, Office of Cybersecurity and Communications, is seeking dynamic individuals to fill several positions in the areas of software assurance, information technology, network engineering, telecommunications, electrical engineering, program management and analysis, budget and finance, research and development, and public affairs.

To learn more about the DHS Office of Cybersecurity and Communications and to find out how to apply for a vacant position, please go to USAJOBS at www.usajobs.gov or visit us at www.DHS.GOV; follow the link Find Career Opportunities, and then select Cybersecurity under Featured Mission Areas.

Software Doctrine for Fixed-Price Contracting

Don O'Neill, Independent Consultant

Abstract. The DoD faces austerity challenges and needs to ensure that defense industry senior executives are committed to meeting these challenges. Consequently, there is a need for a software doctrine for large-scale, software-intensive systems development on fixed-price contracts.

The Challenge of Fixed Price

The DoD faces austerity challenges. The government understands what it needs. These needs were best stated by the challenges outlined by Dr. Ashton Carter, Undersecretary of Defense for Acquisition, Technology, and Logistics (AT&L) [1]. The message here is to deliver "more without more" and to accomplish this through "better buying power." These may sound like slogans, but they are backed by well-conceived guideposts designed to rely on normal market forces in focusing the defense industry on competitiveness, innovation, program management, incentives, efficiency, profitability, and productivity.

The DoD needs to ensure that defense industry senior executives are committed to meeting the AT&L challenges and are accountable for demonstrating game changing progress toward solving these challenges.

For example, the most significant game changer a defense industry senior executive can deliver is an "all in" commitment to accept fixed-price contracts on large software-intensive programs along with a convincing capability to deliver that reflecting an understanding of the cultural changes required. Both the DoD and the defense industry need to populate a tool kit of capabilities for successfully engaging in fixed-price contracts and for evaluating the challenges and benefits of doing so.

Reluctance to accept fixed-price contracts within the defense industry community is based on risk and fear of failure in cost, schedule, and quality performance. This reluctance can be offset by DoD incentives based on technical performance measures designed to tilt the risk calculation in favor of fixed price for those capable of delivering.

Meeting the Challenge on GPS

An example of how a fixed-price contract results in a win/win outcome was turned in by IBM's Federal Systems Division (FSD) [2] performance on the GPS Ground Station, a $150 million fixed-price program [3, 4, 5, 6]. GPS is a high-assurance, real-time system that provides continuous and accurate positioning information to properly equipped users. So, naturally incentives were tied to achieving accuracy of results and a high availability operation.

A team of IBM FSD and software engineers produced the system of 500,000 source lines of code and experienced first hand the challenges and benefits that come with a fixed-price contract. The challenges and how they were met are highlighted as follows:

1. The first challenge was to convince John Akers, the president of IBM, that we could successfully perform a sizable fixed-price contract. A comprehensive set of technical performance measurement incentives organized around the accuracy of results was instrumental in securing that approval.

2. The second challenge was the commitment to systems engineering and software engineering collaboration needed to obtain the deepest possible user domain awareness. This was done through early operations analysis and simulation in order to integrate the needs of the systems, software, and user in the best possible way. Every eyeball was trained on accuracy and high availability incentives.

3. The third challenge was to structure the software development plan as an incremental development with four-well specified design levels each with fine grained cost accounts, formal software inspections of design level artifacts, careful management and visibility of systems engineering to-be-determined items, and a relentless focus on the innovation needed to meet or exceed the accuracy incentives. Designs were recorded in a program design language and by the end of design level 4 represented a 1:4 ratio of design language to estimated sources lines of code. Design levels 1 and 2 supported the systems engineering preliminary design review with intended functions of components, interface specifications, and software architecture rules of construction; design levels 3 and 4 comprised the basis for the software engineering critical design review with provably correct, stepwise refined elaborations of functionality.

4. The fourth challenge was to apply strict accountability and control of cost accounts and work packages based on a work breakdown structure and work responsibility matrix. Cross charging was prohibited, that is, systems engineers were prohibited from charging software engineering work packages. Work packages were opened only when the entry gates had been either met or waived by explicit decision. Work packages were closed only when and as soon as the work package had achieved 100% earned value so that unexpended funds in completed work packages were not used to offset work packages that were over budget. An Estimate to Complete (ETC) was made for each work package each month. Where actuals to date combined with the ETC for a work package exceeded the budget at completion, a corrective action plan was initiated where possible.

In addition to the challenges of fixed-price contract performance, the benefits that result from an improved culture of performance where no one is outstanding until everyone meets the minimum include the following:

1. The value of the IBM FSD contract for GPS was $150 million. The actuals at completion were $165 million. The additional fee paid based on earned incentives was $25 million. This project was a success.

2. Performing on a fixed-price contract disciplines the mind on things that matter most and provides management the will to align the best organizational capabilities to perform on the essentials. It promotes a sense of priority. It promotes a sense of urgency. It discourages waste of any kind.

3. Of real importance, performing on a fixed-price contract had the effect of elevating the software engineering function to a heightened level of importance because it is traditionally the major source of program risk as the tall pole in the tent. As a practical matter, the software development function held the systems engineering function feet to the fire in insisting on completed requirements and specifications documents delivered on time with few to-be-determined items. This program tension resulted in forging a cooperative peer relationship between systems engineers and software engineers where the only rule was, "The person with superior knowledge dominates."

4. With the onus of cost management shifted to IBM FSD, the Air Force acquisition focus was concentrated on accuracy and high availability along with schedule and quality, not sparring over cost and scope issues. Constructive changes were accommodated through value engineering.

Software Doctrine

The vision is to achieve affordability through fixed-price contracting with the defense industrial base whereby the onus for cost management and risk is transferred to the defense industry, which is in turn accorded leeway intended to unleash the forces of competitiveness and innovation. The preferred organization software doctrine for large-scale, software-intensive systems development on fixed-price contracts features the following tenets:

1. Requirements and the technical performance incentives for their achievements are fully known at the beginning and managed and controlled throughout the program life cycle.

2. The software engineering organization reports directly to the program manager.

3. Both the systems engineering and software engineering functions are jointly committed to obtain the deepest possible user domain awareness.

4. Project goals for schedule, cost, and quality are explicitly stated and matched by both the readiness to perform and actual performance.

5. Strict accountability and control of cost accounts and work packages are applied based on a work breakdown structure and work responsibility matrix.

6. Software development planning is based on multiple design levels and staged incremental deliveries [7].

7. The frequency of software product releases is planned, managed, and controlled.

8. Joint systems engineering and software engineering team innovation management results in new ideas that are generated, selected, and used in new product releases.

Conclusion

The market-driven transformation of the defense industry must be fueled by the expectation of the DoD. The government knows what it needs. It now needs to communicate that expectation in practical terms.

Accomplishing this requires a cultural shift away from commoditized software engineering to a more tightly coupled integration of software engineering and systems engineering operating as peer functions reporting directly to the acquisition program management function.

Program risk is directly proportional to the organizational distance among these functions. Being highly competitive by anticipating and leading in the application domain requires understanding the deep needs of the customer and delivering transforming intersectional innovation. This is not achieved by tiers of subcontractors and extended global supply chains. Instead it requires closely-knit, well-integrated management and engineering functions with extended time in market spurred on by the challenge to succeed and not frozen by the fear of failure.

The DoD will know that the defense industry is hearing the message and knows what is expected when prime contractors begin to compete for fixed-price contracts.

ABOUT THE AUTHOR

Don O'Neill served as the President of the Center for National Software Studies from 2005 to 2008. Following 27 years with IBM's Federal Systems Division, he completed a three-year residency at Carnegie Mellon University's SEI under IBM's Technical Academic Career Program and has served as an SEI Visiting Scientist. A seasoned software engineering manager, technologist, and independent consultant, he has a Bachelor of Science degree in mathematics from Dickinson College in Carlisle, Pennsylvania.

REFERENCES

1. Dr. Ashton Carter, Undersecretary of Defense for Acquisition, Technology, and Logistics (AT&L), in his comments at TACOM on March 31, 2011
2. Robinson, William Louis, "IBM's Shadow Force: The Untold Story of Federal Systems, The Secretive Giant That Safeguarded America", ThomasMax Publishing, 2008 ISBN-13: 978-0-9799950-3-5, 214 pages
3. O'Neill, Don, "Integration Engineering Perspective", IBM FSD Software Engineering Exchange, Vol 2 No 2, January 1980
4. O'Neill, Don, "An Overview of Global Positioning System (GPS) Software Design", IBM FSD Software Engineering Exchange, Vol 3 No 1, October 1980
5. O'Neill, Don, "Confident Software Estimation for the Global Positioning System (GPS)", IBM FSD Software Engineering Exchange, Vol 4 No 2, October 1982
6. O'Neill, Don, "GPS Adaptation of Modern Software Design: An Update", IBM FSD Software Engineering Exchange, Vol 4 No 2, October 1982
7. Larman, Craig, "Agile & Iterative Development: A Manager's Guide", Pearson Education, Inc., ISBN 0-13-111155-8, 2008, 342 pages

Uncovering Weaknesses in Code With Cyclomatic Path Analysis

Thomas J. McCabe Sr., McCabe Technologies
Thomas J. McCabe Jr., McCabe Software
Lance Fiondella, University of Connecticut

Abstract. Software flaws represent a serious threat to system integrity. Today, software plays an increasingly important role in the infrastructure of government agencies. These entities outsource and use open-source software within their critical infrastructure; however, the origins and security characteristics of this code are rarely certified. We compare the relative effectiveness of the statement, branch, and cyclomatic code coverage software testing methodologies for detecting flaws in software.

Foreign influence on DoD software is a major security concern [1]. A programmer can insert a flaw into code that looks like an honest mistake, but when triggered leads to unexpected behavior in the system on which the software resides. The consequences could be anything from system unavailability to outright hijacking of the system and all of its functionality. Given the potentially catastrophic consequences of allowing exploitable software flaws to reside in operational systems, software testing is now being acknowledged as a critical step to mitigate software supply chain risks [2].

Protecting against the "inside job" is not the only concern for those wishing to protect software systems from attack. Foreign adversaries persistently attempt to break into the networks of defense facilities and their contractors. A successful intruder would steal anything that could provide economic or strategic advantage. The speculated compromise of the Joint Strike Fighter [3] is a high profile example, with tens of thousands of hours of programming feared lost. Not only can code be copied, it can be studied intensively for weaknesses. By interfacing operational systems running the software and injecting attacks to trigger exploitable weaknesses, the range of consequences mentioned above could be realized. Any unprotected statements in code that could lead to failure become fair game. The only way to ensure compromised software can withstand external attacks is to subject it to rigorous testing and identify weaknesses for removal before they can ever be targeted for attack. A software testing methodology that can eliminate the majority of flaws, both intentional and unintentional, is essential for producing and preserving software dependability.

Software Weaknesses

The Common Weakness Enumeration (CWE) [4] has emerged as a knowledge base of software weaknesses and vulnerabilities. This repository categorizes software flaws across multiple dimensions, describing major properties. For each kind of weakness, the CWE enumerates when it is introduced, common consequences, how likely it is to be exploited, and some examples of code containing the weakness. The CWE was designed to serve as "a standard measuring stick for software security tools targeting these weaknesses" [4]. As such, the CWE may be likened to a medical compendium that focuses only on pathology, describing the conditions, processes, and results of a disease. Treatment methodologies and medications are beyond the scope of the CWE itself. Like medicine, diagnosis and prevention of software vulnerabilities will be critical to limit the harm that can be done by those wishing to do damage. Moreover, software testing will be a key tool for conducting this vulnerability analysis.

Specific software testing methodologies identify some weaknesses, but can fail to identify others. In the absence of a single panacea to the software vulnerability epidemic, a remedy against the majority of common software ailments will prove highly effective. The most widely studied set of software testing strategies are those that study various forms of code coverage [5]. Code coverage is a part of the DO-178B [6] software verification process, which provides guidelines for certifying software in airborne systems and equipment. Code coverage approaches characterize the static control flow paths of an application as a graph with vertex nodes representing code statements, and edges representing possible branches within the code like the if and else statements.

This article is the first to compare the relative effectiveness of the statement, branch, and cyclomatic code coverage software testing methodologies for targeting weaknesses. Statement coverage seeks to test all of the nodes, while the goal of branch coverage is to traverse every edge of the graph. Statement and branch testing have limitations because interactions between decision outcomes can mask errors during testing. As a result, neither statement nor branch testing is adequate to detect vulnerabilities and verify control flow integrity. Cyclomatic Path Analysis [7], on the other hand, detects more CWE vulnerabilities. The fundamental idea behind Cyclomatic Path Analysis, also known as Basis Path or Structured Testing, is that decision outcomes within a software function should be tested independently [8]. By identifying software vulnerabilities with standard testing, a majority of attack opportunities will be eliminated before they can ever be exploited.

Detecting Security Flaws With Cyclomatic Complexity-based Testing

A critical comparison of software testing methodologies is essential to illustrate how competing approaches can fail to identify particular weaknesses. The following three examples consider this additional aspect and demonstrate that cyclomatic complexity-based testing can successfully detect several common weaknesses.

Divide By Zero

CWE-369: Dividing by zero is a commonly occurring problem. In mathematics, dividing a number by zero is not permitted because the result is defined to be infinity. This poses a challenge for computers, which cannot work with such a large number. Attempting to divide by zero on a computer leads to a condition known as overflow. Though one may think this exception should be simple to eliminate, overflows happen quite frequently because many programming languages set a variable to zero before it is ever assigned a value. All too often, programmers neglect to initialize a variable before using it as the denominator of a statement that performs division. This frequent occurrence makes the divide by zero weakness a widespread problem. Dividing by zero can lead to a variety of unpredictable behavior in software. Potential outcomes include unintended branching to error handling routines, software crashes, and similar undesirable behaviors. A programmer who intentionally or unwittingly introduces a divide by zero flaw can induce system crashes, rendering a system unavailable to perform its appointed tasks.

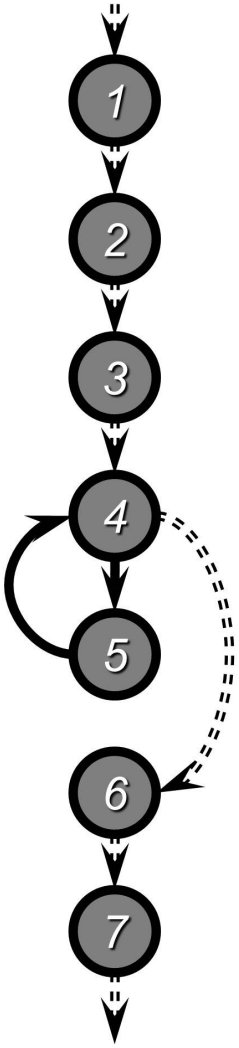


Figure 1: Statement graph of simple average routine.

Algorithm 1: Simple average routine.

```
1: void simpleAvg(int array[], int n)
2: int total = 0;
3: int count = 0;
4: for ( count = 0; count < n; count++ ) do
5:         total += array[count];
6: end for
7: return total / count;
```

The SimpleAvg routine computes an arithmetic average by adding up the first *n* numbers in the array and then divides their total by *n*.

Figure 1 shows the statement graph of the simple average routine.

The nodes in the graph correspond to the seven lines of code and the edges represent the possible transfer of control between these lines. This statement graph is used to measure the coverage with respect to each of the three testing methodologies under consideration. Passing an array with one or more elements and a positive value for the second parameter *n* will lead to successful loop entry and exit, exercising 100% of the statements and branches. For example, the following two lines of code achieve complete statement and branch coverage.

```
1: int array[]= { 1, 2, 3, 4, 5, 6, 7 };
2: int avg = simpleAvg(array, 5);
```

This test invokes the simple average routine, passing it an array with seven values and requests that the average of the first five values be calculated. This test will execute successfully and the tester who would be satisfied with statement or branch coverage could consider their job complete. Note that with this single test case, the condition (count < n) evaluates to true five times, repeating the loop multiple times, and also evaluates to false once to exit the loop. This test exercises both branches into and out of the loop, but fails to consider the case where the loop never runs.

Basis path testing requires a test that does not enter the loop. The following additional test accomplishes this.

```
1: int array[]= { 1, 2, 3, 4, 5, 6, 7 };
2: int avg = simpleAvg(array, 0);
```

This test requests the average of the first zero elements of the array. This will induce a divide by zero exception that could produce a system crash because the loop on line four never increments the variable count. As a result, count still contains the value zero when line seven is reached, where it will generate an overflow exception. Clearly, a statement to ensure that the second parameter of the simple average routine is not zero would eliminate this vulnerability. However, failure to add this guard exposes the code to an otherwise preventable attack. Unlike cyclomatic path testing, path and branch coverage could fail to detect the apparent weakness and subsequently fail to identify the need for this additional check.

Memory Leaks

CWE-401 describes the failure to release memory before removing the last reference. This type of weakness is most commonly known as a "memory leak." Memory leaks occur when an application does not properly track allocated memory so that it may be released after it is no longer needed. Leaking memory slowly eats away at this finite resource. If no scheduled restart of the system occurs [9] undesirable outcomes like an operating system freeze can result. Memory leaks contribute to the unreliability of software. A programmer who intentionally conceals a memory leak provides a digital beachhead from which an attacker can easily launch a denial of service attack that whittles down the memory, crashing the program and unleashing the unexpected consequences of system failure.

Algorithm 2 provides an instance of code containing an exploitable memory leak.

Algorithm 2: Fill arrays routine.

```
1: void fillArrays(void **s1, void **s2, int size1, int size2)
2: if ((*s1=malloc(size1)) && (*s2=malloc(size2)) then
3:         memset(*s1, 0, size1);
4:         memset(*s2, 0, size2);
5: else
6:         *s1 = *s2 = NULL;
7: end if
```

The purpose of the fill arrays function is to allocate memory for two pointers and set the pointers to these newly allocated areas. The pointers are assigned if memory allocation succeeds, but are set to NULL otherwise. At first blush, the implementation appears to be a harmless decision with two possible outcomes. Figure 2 shows the statement graph corresponding to the fillArrays routine.

One may think that the two tests given in the following code fragment should be sufficient to achieve statement and branch coverage.

```
1: void* ptr1 = 0;
2: void* ptr2 = 0;
3: fillArrays(&ptr1, &ptr2, 10, 100);
4: fillArrays(&ptr1, &ptr2, 0xFFFFFFFF, 2);
```

The first test, on line three, will cause the if statement to run, while the test on line four will cover the else statement because the attempt to allocate 0xFFFFFFFF memory will fail on machines with less than four gigabytes of available memory. These tests achieve statement coverage and appear to attain branch coverage. Note, however, that the if statement is actually composed of two conditions. When the first memory allocation (malloc) statement for string pointer s1 succeeds, but the second memory allocation statement fails, the code will still execute the else statement and set both pointers to NULL. The memory from the first successful allocation should be freed, but the reference to this memory is lost when line six is run and the memory is "leaked."

Thorough coverage must also account for scenarios where only the first condition evaluates to true. Figure 3 shows this more detailed cyclomatic graph, where line two is divided into nodes 2a and 2b to represent the two malloc statements embedded in the if statement.

The following code fragment provides the additional test needed to exercise this basis path introduced by the compound logic in the if statement.

```
1: void* ptr1 = 0;
2: void* ptr2 = 0;
3: fillArrays(&ptr1, &ptr2, 2, 0xFFFFFFFF);
```

The flaw lies on the edge between nodes 2b and 5 of Figure 3. This last test will trigger the memory leak because the first amount of memory requested is very small, but the second will fail. Repetitive execution of this last test could quickly chisel away at the memory resources. This is yet another instance where statement and code coverage can prove inadequate, but cyclomatic basis path testing detects the weakness.

Out-of-bounds Read

CWE-125 is an out-of-bounds read. This type of behavior occurs when software reads data before the beginning or past the end of the intended buffer. This can happen when a pointer or its index is increased or decreased to a position beyond the bounds of the buffer or by pointer arithmetic that results in a location outside of the appropriate memory location. Potential outcomes include: software crashing, unintended execution of code, and data corruption. A programmer who devises an out-of-bounds read can do potentially unlimited damage. In the worst case, they could hijack control of the system, turning it against its owners.

Figure 2: Statement graph of fill arrays routine.

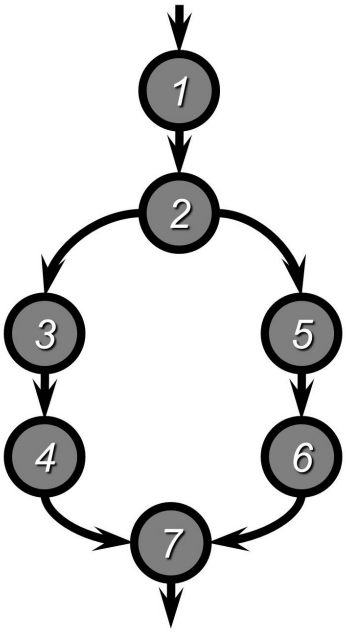
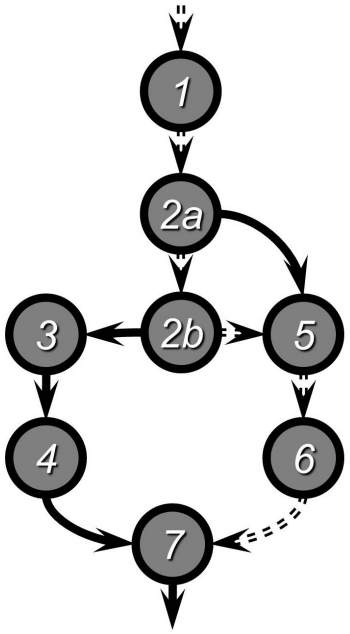


Figure 3: Cyclomatic graph of fill arrays routine.



WHAT IS CYCLOMATIC COMPLEXITY?

Important facts about the cyclomatic complexity metric include:

- **Cyclomatic complexity enables defensive coding procedures such as code flattening, which simplifies understanding the structural characteristics of software.**
- **Cyclomatic complexity models information flow control and can help discover sneak paths within source code.**

Algorithm 3 contains an exploitable out-of-bounds read.

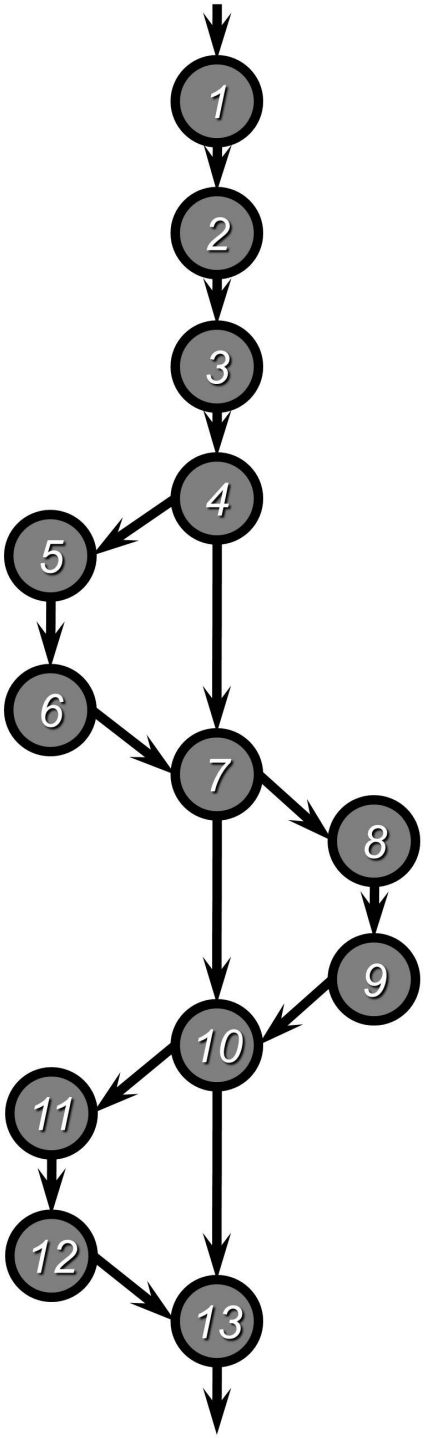
Algorithm 3: Character copying routine.

```
1: void copyChars(char** dest, char** src, int start, int end)
2: int charsToCopy = 1;
3: int lastPos = strlen(*src) - 1;
4: if ( end > lastPos ) then
5:       end = lastPos;
6: end if
7: if ( start < 0 ) then
8:       start = 0;
9: end if
10: if ( end > start ) then
11:      copyToChars += (end - start);
12: end if
13: strncpy(*dest, (*src) + start, charsToCopy);
```

The copyChars routine is intended to copy a range of characters from the source to destination array. There are three sequential checks that occur prior to this copying of characters. The first, on line four, validates that the end position is within the bounds of the source string. The second conducts a similar check to ensure the start position is within bounds, and the third ensures the end position is after the start.

Figure 4 shows the routine's statement graph.

Figure 4: Statement graph of character copying routine.



Note how the three consecutive if tests on lines four, seven, and 10 create three separate branches in the statement graph. A test case that makes each of the three if tests true achieves statement coverage. A second test that makes each of the if statements false attains complete branch coverage. The two tests listed on lines three and four of the following code fragment achieve complete statement and branch coverage.

```
1: char* original = "Hello My World!";
2: char* copy = (char*) malloc(80);
3: copyChars(&copy, &original, -500, 500);
4: copyChars(&copy, &original, 0, 0);
```

Both statement and branch coverage, however, fail to account for the effect that a given decision may have on subsequent decisions. The two test cases that provide statement and branch coverage are insufficient to detect the vulnerability. This routine contains a defect that is only realizable with a specific sequence of decision outcomes. The cyclomatic complexity of the routine is four, meaning that four basis paths must be exercised. The previous two test cases that achieved complete branch coverage exercised only half of these paths. A software tool that supports basis path testing can indicate the sequence of decision outcomes that need to be exercised to test the remaining basis paths.

The following code fragment shows the two additional tests to exercise these other two basis paths in order to provide complete cyclomatic path coverage.

```
1: char* original = "Hello My World!";
2: char* copy = (char*) malloc(80);
3: copyChars(&copy, &original, -10, 500);
4: copyChars(&copy, &original, 1000, 100);
```

The test case on line four initiates the opportunity for an out-of-bounds read. The first test on line four evaluates to true because the end variable equals 100, which is longer than the, "Hello My World!" string. As a result, line five of the character copying routine sets the end variable to the length of the source string. The second test on line seven, however, evaluates to false because the start variable equals 1,000, which is greater than zero. Thus, line eight is skipped. Finally, the test on line 10 evaluates to false because 100 is not less than 1,000, so line 11 is not executed. Line 13 copies a byte from a location 1,000 positions beyond the start of the source string to the destination because charsToCopy=1. This type of out-of-bounds read can be used to feed an application the address of instructions to execute, introducing the potential to commit serious violations of system security. Cyclomatic path testing exposes this vulnerability, but statement and branch coverage do not.

Managing the Attack Map

Up until now, the article has focused on testing simple modules for vulnerabilities. In large-scale software testing, this search is not a mere hunt for vulnerable routines. Instead, it is a more comprehensive examination of relationships to explore control-flow graphs, routine reachability, and the attack map, attack surface, and attack target, which are defined in the following discussion. The attack surface of software is the code within a computer system that can be run by unauthenticated users. Recent research [10] proposed an I/O automata model of a system and its environment to formalize the notion of the attack surface. A concrete implementation of this formalism is cyclomatic path analysis.

The attack surface is the set of functions S that allow user inputs affecting system behavior. Examples include operations that read from configuration files, receive network data, and keyboard inputs. Library functions of potential interest might be input functions such as gets(), recv(), and scanf(). The attack target is the set of routines T that can cause critical impacts when exploits are attempted. Code that might trigger reformat of the hard drive or shutdown certain services are specific instances of attack targets. Calls that can perpetrate these abuses include system functions like exec(), which starts new processes, LoadLibrary(), which can load shared objects, and dynamically linked libraries are all potential threats. Finally, the attack map M is the application subgraph connecting the attack surface and attack target. This structural context promotes the joint analysis of routines that connect the surface and target, which will prove more revealing than study of the two in isolation. Identifying the control flow relationships between the surface and target provides the opportunity to apply a path-oriented approach to focus the review and testing on these connected components. This addresses a major challenge associated with vulnerability isolation, namely the overwhelming amount of source code that must be analyzed.

Many times flaws reside within millions of lines of code and are introduced somewhere along the software supply chain. By accounting for the connectedness of components, cyclomatic path analysis simplifies graph complexity to the routes by which an attacker can reach particular software vulnerabilities.

An additional advantage of structural security analysis is the ability to define lists of functions that must be considered in the performance of attack map analysis, modularizing the process. An example is the list of Microsoft Secure Development Lifecycle (SDL) [11] banned functions. Microsoft recommended processes on secure development specify a list of standard C functions. Microsoft discourages programmers from invoking these routines because they are prone to vulnerabilities like memory leaks and buffer overruns. This list of C functions is ideal for conducting security analysis on legacy applications to bring them into conformance with the Microsoft SDL. The scanf() and printf() functions are banned members of the attack surface and target respectively. Structural simplifications can effectively constrain analysis by aggregating the modules containing the surface and target into two "supercomponents", simplifying the view of the potential paths from entry points to flaw exploitation. This grouping facilitates test specification, providing a global perspective on the analysis task at hand within the context of the application.

Summary

Software vulnerabilities are a consequence of multiple factors. Attackers can disrupt program operation by exercising a specific sequence of interdependent decisions that result in unforeseen behavior. To ensure program behavior is correct, these paths must be identified and exercised as part of secure software development. Software testing techniques that utilize complete line and branch coverage are insufficient and leave too many gaps. Cyclomatic complexity enables more comprehensive scrutiny of the structure and control flow of code, providing significantly higher vulnerability detection capabilities.

Static analysis for code review has been suggested as a valuable aid for critical software assurance [12]. The future of software engineering would benefit from tight integration of development with testing. Automatically warning developers of the security vulnerabilities present in their code will be a first step toward eradicating common weaknesses.

ABOUT THE AUTHORS

Thomas J. McCabe Sr. is a mathematician, author, and founder of McCabe Technologies <http://www.mccabe.com>, which he grew to 175 people. In 1998, he testified before Congress in the hearing entitled, "Year 2000: Biggest Problems and Proposed Solutions." He has authored three books, is highly published in computer science, and lectures internationally on software development and entrepreneurship. In 2009, the ACM-SIGSOFT retrospectively selected his publication on Software Complexity [8] as one of the 23 highest impact papers in computer science.

McCabe Technologies
5163 Harpers Farm Road
Columbia, MD, 21044
E-mail: tom@mccabetech.com

Thomas J. McCabe Jr. has worked with clients of McCabe Technologies for more than 20 years to improve their quality assurance (QA) and test processes. He has delivered numerous presentations on software testing and QA to organizations and conferences including IEEE, QAI, Sticky Minds, DAU, NDU, and DHS. His recent Sticky Minds Webinar, "Uncovering Security Vulnerabilities Off the Beaten Path," presented his experiences applying complexity metrics, verifying control flow integrity, and leveraging path analysis to uncover security vulnerabilities.

McCabe Technologies
3806 Spring Meadow Drive
Ellicott City, MD 21042
E-mail: skillssss@hotmail.com

Lance Fiondella is a Ph.D. student at the University of Connecticut (UConn). In 2007, he received a scholarship from the IEEE Reliability Society for his research on system and software reliability. He also conducts network vulnerability research for UConn's Department of Homeland Security National Transportation Security Center of Excellence (NTSCOE). He was an invited speaker at the 2011 DHS Student Day, where he presented his research on the optimal deployment and protection of high-speed rail.

University of Connecticut
371 Fairfield Road
Storrs, CT, 06269-4155
Phone: 860-486-3665
E-mail: lfiondella@engr.uconn.edu

REFERENCES

1. Defense Science Board Task Force. "Mission impact of foreign influence on DoD software." The Journal of Defense Software Engineering (May 2008): 4-7.
2. Ellison, Robert and Woody, Carol. "Considering software supply chain risks." The Journal of Defense Software Engineering (September-October 2010): 9-12.
3. The Wall Street Journal. "Computer spies breach fighter-jet project." Last accessed: February, 21 2012, <http://online.wsj.com/article/SB124027491029837401.html, April 21, 2009>.
4. The MITRE Corporation. "Common weakness enumeration." Last accessed: February, 21 2012, <http://cwe.mitre.org/>.
5. Miller, Joan and Maloney, Clifford. "Systematic mistake analysis of digital computer programs." 6.2 Communications of the ACM (Feb 1963): 58-63.
6. RTCA, Inc. and EUROCAE, "Software considerations in airborne systems and equipment certification." Technical Report DO-178B (1992).
7. Watson, Authur and McCabe Sr., Thomas. "Structured testing: A testing methodology using the cyclomatic complexity metric." special publication 500-235, National Institute of Standards and Technology, Gaithersburg, MD, (August 1996).
8. McCabe Sr., Thomas. "A complexity measure." 2.4 IEEE Transactions on Software Engineering (December 1976): 308-320.
9. Garg, Sachin, et al. Analysis of software rejuvenation using markov regenerative stochastic petri nets. Proc. of the 6th International Symposium on Software Reliability Engineering. Toulouse, France, 1995.
10. Manadhata, Pratyusa and Wing, Jeannette. "An attack surface metric." 37.3 IEEE Transactions on Software Engineering (May-June 2011): 371-386.
11. Microsoft Corporation, "Microsoft security development." Last accessed: February, 21 2012 <http://www.microsoft.com/security/sdl/default.aspx>.
12. Moy, Yannick. "Static analysis is not just for finding bugs." The Journal of Defense Software Engineering (September-October 2010): 5-8.

Efficient Methods for Interoperability Testing Using Event Sequences

D. Richard Kuhn, NIST
James M. Higdon, Eglin AFB
James F. Lawrence, NIST
Raghu N. Kacker, NIST
Yu Lei, University of Texas at Arlington

Abstract. Many software testing problems involve sequences of events. The methods described in this paper were motivated by testing needs of mission critical systems that may accept multiple communication or sensor inputs and generate output to several communication links and other interfaces, where it is important to test the order in which events occur. Using combinatorial methods makes it possible to test sequences of events using significantly fewer tests than previous procedures.

Introduction

For many types of software, the sequence of events is an important consideration [1, 2]. For example, graphical user interfaces may present the user with a large number of options that include both order-independent (e.g., choosing items) and order-dependent selections (such as final selection of items, quantity, and payment information). The software should work correctly, or issue an appropriate error message, regardless of the order of events selected by the user. A number of test approaches have been devised for these problems, including graph-covering, syntax-based, and finite-state machine methods [3, 4, 5].

In testing such software, the critical condition for triggering failures often is whether or not a particular event has occurred prior to a second one, not necessarily if they are back to back. This situation reflects the fact that in many cases, a particular state must be reached before a particular failure can be triggered. For example, a failure might occur when connecting device A only if device B is already connected, or only if devices B and C were both already connected. The methods described in this paper were developed to address testing problems of this nature, using combinatorial methods to provide efficient testing. Sequence covering arrays, as defined here, ensure that every t events from a set of n ($n > t$) will be tested in every possible t-way order, possibly with interleaving events among each subset of t events.

Definition

We define a sequence covering array, SCA(N, S, t) as an N x S matrix where entries are from a finite set S of s symbols, such that every t-way permutation of symbols from S occurs in at least one row and each row is a permutation of the s symbols [6]. The t symbols in the permutation are not required to be adjacent. That is, for every t-way arrangement of symbols $x1$, $x2$, ..., xt, the regular expression *.*x1.*x2.*xt.** matches at least one row in the array.

Example 1

We may have a component of a factory automation system that uses certain devices interacting with a control program. We want to test the events defined in Table 1. There are 6! = 720 possible sequences for these six events, and the system should respond correctly and safely no matter the order in which they occur. Operators may be instructed to use a particular order, but mistakes are inevitable, and should not result in injury to users or compromise the operation. Because setup, connections, and operation of this component are manual, each test can take a considerable amount of time. It is not uncommon for system-level tests such as this to take hours to execute, monitor, and complete. We want to test this system as thoroughly as possible, but time and budget constraints do not allow for testing all possible sequences, so we will test all 3-event sequences.

Table 1. Example system events.

Event	Description
a	connect air flow meter
b	connect pressure gauge
c	connect satellite link
d	connect pressure readout
e	engage drive motor
f	engage steering control

With six events, ***a, b, c, d, e,*** and ***f,*** one subset of three is {***b, d, e***}, which can be arranged in six permutations: [***bde***], [***bed***], [***dbe***], [***deb***], [***ebd***], [***edb***]. A test that covers the permutation [***dbe***] is: [***adcfbe***]; another is [***adcbef***]. With only 10 tests, we can test all 3-event sequences, shown in Table 2. In other words, any sequence of three events taken from ***a..f*** arranged in any order can be found in at least one test in Table 2 (possibly with interleaved events).

Table 2. All 3-event sequences of six events.

Test	Sequence
1	*a b c d e f*
2	*f e d c b a*
3	*d e f a b c*
4	*c b a f e d*
5	*b f a d c e*
6	*e c d a f b*
7	*a e f c b d*
8	*d b c f e a*
9	*c e a d b f*
10	*f b d a e c*

Returning to the example set of events {*b, d, e*}, with six permutations: [*bde*] is in Test 5, [*bed*] is in Test 4, [*dbe*] is in Test 8, [*deb*] is in Test 3, [*ebd*] is in Test 7, and [*edb*] is in Test 2.

A larger example system may have 10 devices to connect, in which case the number of permutations is 10!, or 3,628,800 tests for exhaustive testing. In that case, a 3-way sequence covering array with 14 tests covering all 3-way sequences is a dramatic improvement, as is 72 tests for all 4-way sequences (see Table 4).

Example 2

A 2-way sequence covering array can be constructed by listing the events in some order for one test and in reverse order for the second test, as shown in Table 3:

Table 3. 2-way sequence covering array.

Test	Sequence
1	a b c d
2	d c b a

Table 4. Number of tests for combinatorial 3-way and 4-way sequences.

Events	3-seq Tests	4-seq Tests
5	8	26
6	10	36
7	12	46
8	12	50
9	14	58
10	14	66
11	14	70
12	16	78
13	16	86
14	16	90
15	18	96
16	18	100
17	20	108
18	20	112
19	22	114
20	22	120
21	22	126
22	22	128
23	24	134
24	24	136
25	24	140
26	24	142
27	26	148
28	26	150
29	26	154
30	26	156
40	32	182
50	34	204
60	38	222
70	40	238
80	42	250

Generating Sequence Covering Arrays

Sequence covering arrays, as the name implies, are analogous to standard covering arrays [7], which include at least one of every *t*-way combination of any *n* variables, where $t < n$. We have developed several methods of generating SCAs, but the most efficient approach is a simple greedy algorithm that iteratively generates multiple candidate tests, then selects the one that covers the largest number of previously uncovered sequences, repeating until all sequences have been covered. This algorithm produces more compact arrays than others developed so far.

Table 4 shows the number of 3-way and 4-way sequence tests for event sets of varying sizes generated using the algorithm. In another paper [6], we have shown the number of tests generated is proportional to log *n*, for *n* events, making it practical to test complex systems with a large number of events using a reasonable number of tests. Logarithmic growth in number of tests can also be seen in Table 4.

Using Sequence Covering Arrays

The motivation for this work was a USAF mission-critical system that uses multiple devices with inputs and outputs to a laptop computer. (Confidentiality rules do not permit a detailed description of this system.) System functionality depends on the order in which events occur, though it does not matter whether events are adjacent to one another (in any sub-sequence), nor which step an event falls under, without regard to the other events. The test procedure for this system has eight steps: boot system, open application, run scan, and connect peripherals P-1 through P-5. It is anticipated that because of dependencies between peripherals, the system may not function properly for some sequences. That is, correct operation requires cooperation among multiple peripherals, but experience has shown that some may fail if their partner devices were not present during startup. Thus the order of connecting peripherals is critical. In addition, there are constraints on the sequence of events: cannot scan until the app is open; cannot open app until the system is booted. There are 40,320 permutations of eight steps, but some are redundant (e.g., changing the order of peripherals connected before boot), and some are invalid (violates a constraint). Around 7,000 are valid, and non-redundant, but this is far too many to test for a system that requires manual, physical connections of devices.

The system was tested using a seven-step sequence covering array, removing boot-up from test sequence generation. The initial test configuration for 3-way sequences was generated using the algorithm given in Sect. 2. Covering all 3-way sequences allowed testing a much larger set of states than using 2-way sequences, but could be accomplished at a reasonable cost. Some changes were made to the pre-computed sequences based on unique requirements of the system test. If 6='Open App' and 5='Run Scan', then cases 1, 4, 6, 8, 10, and 12 are invalid, because the scan cannot be run before the application is started. This was handled by swapping items when they are adjacent (1 and 4), and out of order. For the other cases, several were generated from each that were valid permutations of the invalid case. A test was also embedded to see whether it mattered where each of three USB connec-

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 three areas of emphasis we are looking for:

Software Project Management: Lessons Learned
Jan/Feb 2013 Issue
Submission Deadline: Aug 10, 2012

Supply Chain Risk Management
Mar/Apr 2013 Issue
Submission Deadline: Oct 10, 2012

Large Scale Agile
May/Jun 2013 Issue
Submission Deadline: Dec 10, 2012

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>.

tions were placed. The last test case ensures at least strength 2 (sequence of length 2) for all peripheral connections and 'Boot', i.e., that each peripheral connection occurs prior to boot. The final test array is shown in Table 5. Errors detected in testing included several that could not be attributed to 2-way sub-sequences. These errors would not have been detected using a simple 2-way sequence covering array (which could consist of only two tests, as in Example 2), and may not have been caught with more conventional tests.

Conclusions

Sequence covering arrays can have significant practical value in testing. Because the number of tests required grows only logarithmically with the number of events, *t*-way sequence coverage is tractable for a wide range of testing problems. Using a sequence covering array for system testing described here made it possible to provide greater confidence that the system would function correctly regardless of possible dependencies among peripherals. Because of extensive human involvement, the time required for a single test is significant, and a small number of random tests or scenario-based ad hoc testing would be unlikely to provide *t*-way sequence coverage to a satisfactory degree.

Acknowledgments:

We are very grateful to Tim Grance for support of this work within the NIST Cybersecurity program, and to Paul E. Black for suggestions that helped clarify and strengthen the paper.

Disclaimer:

We identify certain software products in this document, but such identification does not imply recommendation by NIST, nor does it imply that the products identified are necessarily the best available for the purpose.

Table 5. Final test array.

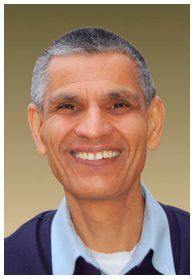
Original Case	Case	Step1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7	Step 8
1	1	Boot	P-1 (USB-RIGHT)	P-2 (USB-BACK)	P-3 (USB-LEFT)	P-4	P-5	Application	Scan
2	2	Boot	Application	Scan	P-5	P-4	P-3 (USB-RIGHT)	P-2 (USB-BACK)	P-1 (USB-LEFT)
3	3	Boot	P-3 (USB-RIGHT)	P-2 (USB-LEFT)	P-1 (USB-BACK)	Application	Scan	P-5	P-4
4	4	Boot	P-4	P-5	Application	Scan	P-1 (USB-RIGHT)	P-2 (USB-LEFT)	P-3 (USB-BACK)
5	5	Boot	P-5	P-2 (USB-RIGHT)	Application	P-1 (USB-BACK)	P-4	P-3 (USB-LEFT)	Scan
6A	6	Boot	Application	P-3 (USB-BACK)	P-4	P-1 (USB-LEFT)	Scan	P-2 (USB-RIGHT)	P-5
6B	7	Boot	Application	Scan	P-3 (USB-LEFT)	P-4	P-1 (USB-RIGHT)	P-2 (USB-BACK)	P-5
6C	8	Boot	P-3 (USB-RIGHT)	P-4	P-1 (USB-LEFT)	Application	Scan	P-2 (USB-BACK)	P-5
6D	9	Boot	P-3 (USB-RIGHT)	Application	P-4	Scan	P-1 (USB-BACK)	P-2 (USB-LEFT)	P-5
7	10	Boot	P-1 (USB-RIGHT)	Application	P-5	Scan	P-3 (USB-BACK)	P-2 (USB-LEFT)	P-4
8A	11	Boot	P-4	P-2 (USB-RIGHT)	P-3 (USB-LEFT)	Application	Scan	P-5	P-1 (USB-BACK)
8B	12	Boot	P-4	P-2 (USB-RIGHT)	P-3 (USB-BACK)	P-5	Application	Scan	P-1 (USB-LEFT)
9	13	Boot	Application	P-3 (USB-LEFT)	Scan	P-1 (USB-RIGHT)	P-4	P-5	P-2 (USB-BACK)
10A	14	Boot	P-2 (USB-BACK)	P-5	P-4	P-1 (USB-LEFT)	P-3 (USB-RIGHT)	Application	Scan
10B	15	Boot	P-2 (USB-LEFT)	P-5	P-4	P-1 (USB-BACK)	Application	Scan	P-3 (USB-RIGHT)
11	16	Boot	P-3 (USB-BACK)	P-1 (USB-RIGHT)	P-4	P-5	Application	P-2 (USB-LEFT)	Scan
12A	17	Boot	Application	Scan	P-2 (USB-RIGHT)	P-5	P-4	P-1 (USB-BACK)	P-3 (USB-LEFT)
12B	18	Boot	P 2 (USB RIGHT)	Appli ti	Sc	P 5	P 4	P 1 (USB LEFT)	P 3 (USB BACK)

ABOUT THE AUTHORS

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.

James Higdon is a senior analyst in Technical Engineering and Acquisition Support with Jacobs Engineering, at the 46th Test Squadron, Eglin Air Force Base, Florida. His current interests are in experimental design and combinatorial testing of hardware/software systems. He received an MS from the Air Force Institute of Technology.

James Lawrence is a Professor in the Department of Mathematics at George Mason University, Fairfax, VA, and a faculty associate at NIST. His current interests are in convexity and combinatorics, including applications in software testing. He received a Ph.D. from the University of Washington.



Raghu 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.

Yu Lei is an Associate 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 Ph.D. from North Carolina State University.

REFERENCES

1. D.L. Parnas, "On the Use of Transition Diagrams in the Design of User Interface for an Interactive Computer System," Proc. 24th ACM Nat'l Conf., pp. 379-385, 1969.
2. W. E. Howden, G. M. Shi: Linear and Structural Event Sequence Analysis. ISSTA 1996: pp. 98-106, 1996.
3. S. Chow, "Testing Software Design Modeled by Finite-State Machines," IEEE Trans. Softw. Eng., vol. 4, no. 3, pp. 178187, 1978.
4. J. Offutt, L. Shaoying, A. Abdurazik, and P. Ammann, "Generating Test Data From State-Based Specifications," J. Software Testing, Verification and Reliability, vol. 13, no. 1, pp. 25-53, March, 2003.
5. B. Sarikaya, "Conformance Testing: Architectures and Test Sequences," Computer Networks and ISDN Systems, vol.17, no. 2, North-Holland, pp. 111-126, 1989.
6. D.R. Kuhn, J.M. Higdon, J.F. Lawrence, R.N. Kacker, Y. Lei, "Combinatorial Methods for Event Sequence Testing", 8 Oct 2010 (submitted for publication). <http://csrc.nist.gov/groups/SNS/acts/documents/event-seq101008.pdf>
7. X. Yuan, M.B. Cohen, A. Memon, "Covering Array Sampling of Input Event Sequences for Automated GUI Testing", November 2007 ASE '07: Proc. 22nd IEEE/ACM Intl. Conf. Automated Software Engineering, pp. 405-408.

Building Confidence in the Quality and Reliability of Critical Software

Jay Abraham, MathWorks
Jon Friedman, MathWorks

Abstract. Software in critical civilian and military aerospace applications, including avionics and other systems in which quality and reliability are imperative, continues to become both more common and more complex. The embedded software development organizations that build these systems must meet stringent quality objectives that are mandated by their organizations or required by customers or governments. For engineering teams to meet these objectives, and to ideally deliver high quality software, state of the art testing and verification solutions are needed. This article examines formal methods based software verification and testing approaches that have been applied to critical software projects in civil and military aerospace and defense projects. Examples are provided to illustrate how these verification techniques can be deployed in practice to improve the quality and reliability of complex avionics systems.

1. The Components of Avionics Embedded Software

Avionics software implemented in critical aerospace applications consists of special purpose embedded software. This software often operates in real time and is responsible for critical operations. Examples include digital flight control systems, full authority digital engine control, guidance navigation control, and similar systems. The embedded software responsible for these systems will consist of multiple components, including automatically generated, handwritten, and third-party code as well as libraries (see Figure 1).

Generated code: Generated code is synthesized from models that are used to describe and analyze the behavior of complex systems and algorithms.

Handwritten code: Handwritten code may include interfaces to hardware (for example, driver software for a cockpit display system, airspeed sensor, or another hardware subsystem), or it may be translated manually from specification documents or models.

Third-party code: Third-party code may be delivered by suppliers or it may be required as part of larger software system (for example, to interface with the real-time operating system).

Libraries: Object code is part of the application code that exists as a library or as compiled legacy code. By definition this software is delivered or is only available in the form of object code (binary files).

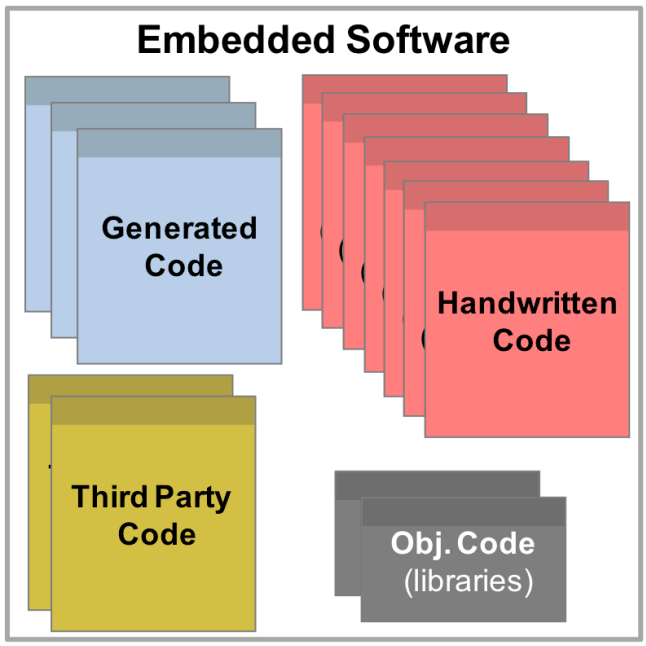


Figure 1: Components of embedded software.

2. Embedded Software Design, Implementation, and Verification

Typically, embedded software design starts by gathering system and software requirements. The code is then written or generated to implement the software. Verification processes focus on confirming the software requirements are implemented correctly and completely, and that they are traceable to the system requirements. The software must be tested and analyzed to ensure that it not only performs as required, but does not include any unintended operations. Additional tests are performed as the software is integrated with the hardware and validated at the system level. The design and verification process described above is often referred to as the V diagram process (see Figure 2).

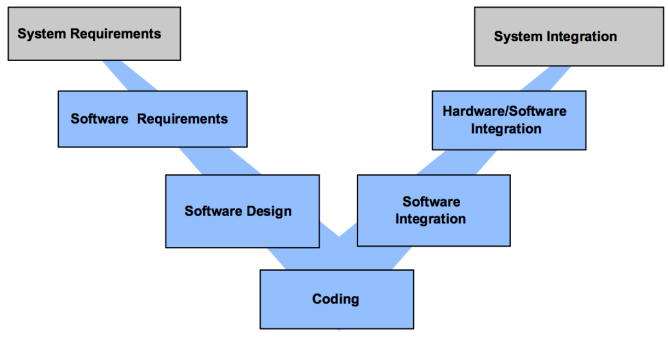


Figure 2: Embedded software design, implementation, and verification (V Diagram).

Even with robust verification processes, complex systems can fail. Causes of failure include insufficient specification, design errors, software coding errors or defects, and other issues unrelated to software. Ideally design and coding errors should be detected on the right-hand side of the V diagram during software, hardware, and aircraft testing and integration processes.

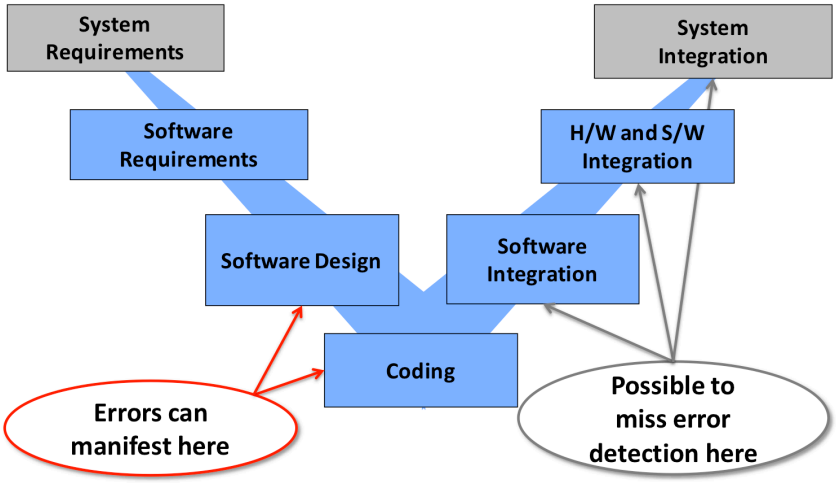


Figure 3: Errors often manifest in the design and coding phases.

Unfortunately, testing phases may fail to detect some errors unless exhaustive testing is employed. A study by the SEI found that for every 100 defects injected in the left-hand side of the V diagram, on average 21 latent defects remain in the system after the testing and verification processes were completed (see Figure 3) [1]. These bugs remain because exhaustive testing is generally not practical. Other techniques must be used to eliminate remaining defects; however, in a report on the use of static analysis to improve quality of code, the authors found that early detection of defects was important, but challenging to accomplish [2]. The difficulty was primarily due to human factors such as the inability to triage results from the tool to identify where code is safe and where it may fail.

A complete discussion on improving the quality of complex systems by addressing every failure point is beyond the scope of this article. Instead, this article focuses on two points: design errors and software coding errors. These errors will manifest in the software design and coding phases of the V diagram.

Examples of design errors include:

- Dead logic (for software combinatorial logic involving AND, OR, and NOT)
- Unreachable states or modes in state machines
- Deadlock conditions
- Nondeterministic behavior
- Overflow or divide-by-zero conditions in arithmetic operations

There are many different types of coding errors. This article covers those that are classified as run-time errors, that is, errors that only express themselves under particular conditions when the software is running. These errors are particularly troublesome because the code may appear to function normally under general test conditions, but may later cause unexpected system failures under other conditions. Some causes of run-time errors include:

- Uninitialized data. When variables are not initialized, they may be set to an unknown value.
- Out of bounds array access. This occurs when data is written or read beyond the boundary of allocated memory.
- Null pointer dereference. This occurs when attempting to reference memory with a pointer that is NULL.
- Incorrect computation. This is caused by an arithmetic error due to an overflow, underflow, or divide-by-zero operation, or when taking a square root of a negative number.
- Concurrent access to shared data. This occurs when two or more different threads try to access the same memory location.
- Dead code. Although dead code (code that will never execute) may not directly cause a run-time failure, it is important to understand why the code will not execute.

3. Traditional Methods of Verifying and Testing Software

Typical software verification processes include manual reviews and dynamic testing. Code review involves line-by-line manual inspection of the source code with the goal of finding errors in the code. The process comprises a team that will perform the review (moderator, designer, coder, and tester), the preparation process (including the creation of a checklist), and the inspection activity itself. Based on the outcome, the development team may need to address errors found and others in the organization will follow up to ensure that issues and concerns raised during inspection are resolved. With this process, detecting subtle run-time errors can be difficult. For example, an overflow due to complex mathematical operations that involve programmatic control can easily be missed. Additionally, the code review process can be inconsistent; since it is highly dependent on human interpretation, results can vary based on the team and context of the review process.

Complementing code reviews, dynamic testing is used to verify the execution flow of software, that is, to verify decision paths, inputs, and outputs. This process involves creation of test cases and test vectors and the execution of the software using these tests. Dynamic testing is well suited to the goal of finding design errors, in which the test cases often match functional requirements. Test teams then compare the results to the expected behavior of the software. Because of the complexity of today's software and tight project deadline requirements, dynamic testing is often not exhaustive. Although many test cases can be generated automatically to supplement those created manually, it is not feasible to expect dynamic testing to exhaustively verify every aspect of embedded software. This kind of testing can show the presence of errors, but not their absence.

In theory, performing code review and executing the right set of test cases can catch every defect no matter the type. In practice, however, the challenge is the amount of time spent reviewing code and applying enough of the right tests to find all the errors in today's complex systems. Even for the simplest operations, such as adding two 32-bit integer inputs, one would have to spend hundreds of years to complete exhaustive testing, which is not realistic [3]. Viewed from this perspective, code review and dynamic testing are bug detection techniques more than proving techniques because they cannot in practice exhaustively show that design errors and code defects have been eliminated.

4. Employing Formal Methods for Verification

To address the shortcomings of code reviews and dynamic testing, which are not exhaustive and can miss design or coding errors, engineers are turning to tools that implement formal methods to prove the absence of certain design and run-time errors, and ultimately to gain greater confidence. Formal methods refers to the application of theoretical computer science fundamentals to solve difficult problems in software and hardware specification and verification. Applying formal methods to models and code gives engineers insight about their design or code and confidence that they are robust when exhaustive testing is not practical.

To better understand formal methods, consider the following example. Without the aid of a calculator, compute the result of the following multiplication problem within three seconds:

$$-4586 \times 34985 \times 2389 = ?$$

Although computing the answer to the problem by hand will likely take you longer than three seconds, you can quickly apply the rules of multiplication to determine that the result will be a negative number. Determining the sign of this computation is an application of a specific branch of formal methods known as abstract interpretation. The technique enables you to know precisely some properties of the final result, such as the sign, without having to fully multiply the integers. You also know from applying the rules of multiplication that the result will never be a positive number or zero for this computation.

Now, consider the following simplified application of the formal mathematics of abstract interpretation to software programs. The semantics of a programming language can be represented by concrete and abstract domains. Certain proof properties of the software can be performed on the abstract domain. In fact, it is simpler to perform the proof on the abstract domain than on the concrete domain.

The concept of soundness is important in the context of a discussion on abstract interpretation. Soundness means when assertions are made about a property, those assertions are proven to be correct. The results from abstract interpretation are considered sound because it can be mathematically proven with structural induction that abstraction will predict the correct outcome. When applied to software programs, abstract interpretation can be used to prove certain properties of software, for example, that the software will not exhibit certain run-time errors [4].

Cousot and Cousot [5] describe the application and success of abstract interpretation to static program analysis. Deutsch describes the application of this technique to a commercial software tool [6]. The application of abstract interpretation involves computing approximate semantics of the software code with the abstraction function, which maps from the concrete domain to the abstract domain such that it can be verified in the abstract domain. This produces equations or constraints whose solution is a computer representation of the program's abstract semantics.

Lattices are used to represent variable values. For the sign example described earlier, the lattice shown in Figure 4 can be used to propagate abstract values in a program (starting at the bottom and working to the top for conditions such as <0, =0, and so forth). Arriving at any given node in the lattice proves a certain property. Arriving at the top of the lattice indicates that a certain property is unproven.

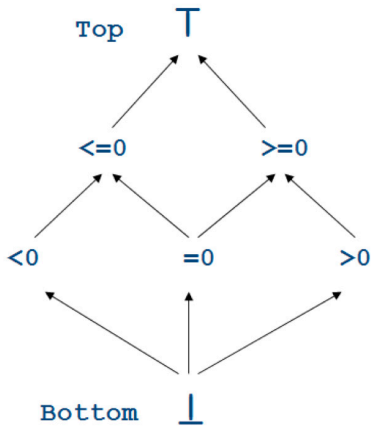


Figure 4: Lattice representation of variables.

Over approximation is applied to all possible execution paths in a program. Analysis techniques can identify variable ranges. That information is used to prove either the existence or the absence of run-time errors in source code.

To better understand the application of abstract interpretation to code verification, consider the following operation:

```
X := X / (X – Y);
```

If X is equal to Y, then a divide by zero will occur. In order to conclusively determine that a divide by zero cannot occur, the range of X and Y must be known. If the ranges overlap, then a divide-by-zero condition is possible.

In a plot of X and Y values (see Figure 5), any points that fall on the line representing X=Y would result in a run-time error. The scatter plot shows all possible values of X and Y when the program executes the line of code above (designated with +). Dynamic testing would execute this line of code using various combinations of X and Y to determine if there will be a failure. However, given the large number of tests needed to be run, this type of testing may not detect or prove the absence of the divide-by-zero run-time error.

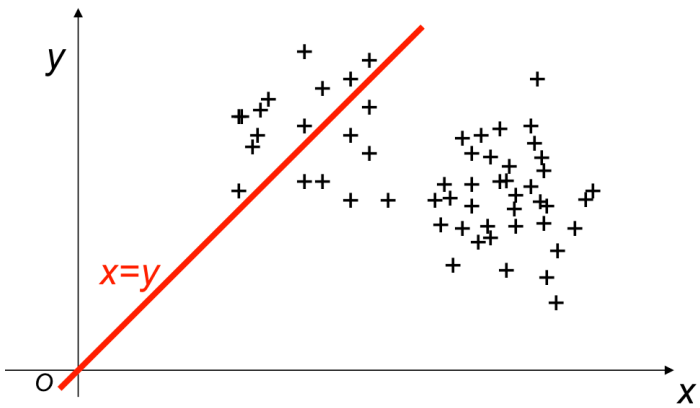


Figure 5: Plot of data for X and Y.

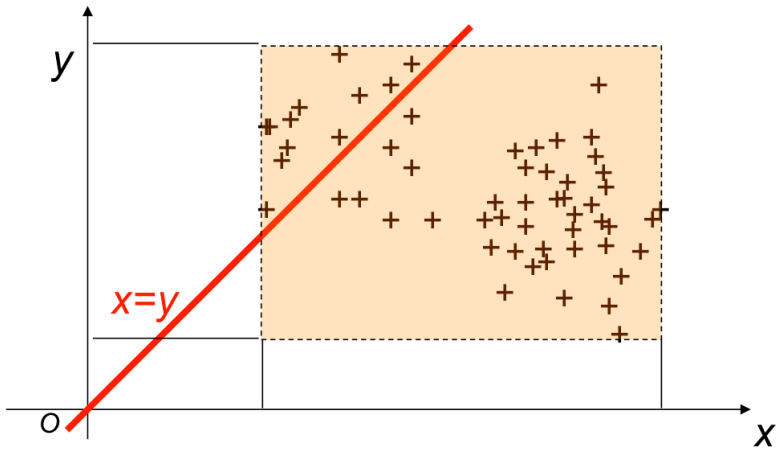


Figure 6: Creating a bounding box to identify potential errors.

Another methodology would be to approximate the range of X and Y in the context of the run-time error condition (that is, X=Y). In Figure 6, note the bounding box created by this method. If the bounding box intersects X=Y, then there is a potential for failure. Some static analysis tools apply this technique. However, approximation of this type is too pessimistic, since it includes unrealistic values for X and Y.

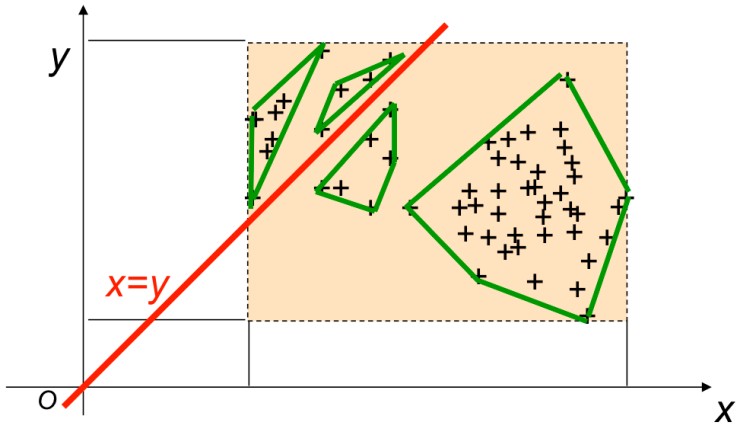


Figure 7: Abstract interpretation.

With abstract interpretation, a more accurate representation of the data ranges of X and Y are created. Since various programming constructs could influence the values of X and Y (for example, arithmetic operations, loops, if-then-else, and concurrency) an abstract lattice is created. A simplified representation of this concept is to consider the grouping of the data as polygons as shown in Figure 7. Since the polygons do not intersect X=Y we can conclusively say that a division by zero will not occur.

The abstract interpretation concept can be generalized as a tool set that can be used to determine variable ranges and to detect a wide range of run-time errors in software. Abstract interpretation investigates all possible behaviors of a program—that is, all possible combinations of values—in a single pass to determine how and under what conditions the program may exhibit certain classes of defects. The results from abstract interpretation are considered complete because it can be mathematically proven that the technique predicts the outcome as it relates to the operation under consideration.

The application of formal methods enables engineers to apply automation to software verification tasks. Unlike manual code reviews, automated application of formal methods is consistent. Formal methods will also provide a complete answer when it can be applied to the problem. Verification based on formal methods can be applied in the software design and coding phases of the V diagram.

5. Application of Formal Methods to Model and Code Verification

Model Verification

During the software design phase, which today typically involves creating models of the advanced control algorithms, engineers need to verify that the design they produce is robust. For example, they need to be certain that their design will not contain overflow errors or state machines with unreachable states. Because engineering teams develop and work with models in this phase, the application of formal methods for verification in this phase is termed *model verification.* The purpose is to produce a robust software design by ideally detecting all design errors or proving their absence.

However, use of models alone does not ensure a robust design. As an example, consider an algorithm that contains an addition operation. The two inputs to the addition operation are generated by other complex mathematical operations. Both inputs are 8-bit signed integers and the output is of the same type. In this scenario, it is possible that the addition operation may result in an overflow. For example, an overflow will occur if the first input has a maximum value of 2^7-1 and the other input is greater than 0. Using the traditional methods of design reviews and dynamic testing, the exact condition that results in the overflow might be missed. In contrast, using formal methods tools, engineers can determine the minimum and maximum ranges of the input to the addition. Furthermore, formal methods tools can determine that it is possible for an overflow to occur and can produce a test case or counter example to show how this overflow design error can occur.

Code Verification

During the coding phase, engineering teams either manually or automatically translate the design documents or models into code. The application of formal methods for verification in this phase is termed *code verification* and the purpose is to produce robust code by identifying and proving the absence of code defects such as run-time errors. This can be accomplished with formal methods coupled with *static code analysis*—the analysis of software without dynamic execution. This technique identifies the absence, presence, and possible presence of a certain class of run-time errors in the code. As a result, engineers can use this technique to prove that the code is free of detectable run-time errors.

During code development and integration, it is important to thoroughly understand the interface between various code components. For example, consider a situation in which handwritten code produced by one team generates an index value that is used for an array access in generated code produced by a second team. The first team believes that the index range can be 0 to 599. The second team believes the maximum index value is 399 and has developed the software with that understanding. Unless there is a test case that causes the index value to exceed

399, this run-time error may not be detected during the integration test. It is even possible that if this illegal array access were to occur during the execution of a test case, the error may not be detected. For example, writing to out-of-bounds memory may not cause a program to fail, unless the data at that location were to be used in some fashion.

The application of formal methods coupled with static code analysis does not require execution of the source code, so it can be used as soon as code is available. Using formal methods and static code analysis tools, engineers can validate software at the component level or as an integrated application. Because these tools propagate variable range values, they can detect or prove that the illegal array access in the example described above may or may not occur.

6. Summary and Conclusion

Today's sophisticated civilian and military aerospace applications often include a complex combination of handwritten and automatically generated code. Even after formal code reviews and dynamic testing is performed on the right-hand side of the V diagram, latent errors can still remain in a system because traditional verification and test methods are often incomplete. Formal methods enable teams to prove that aspects of their models and code are free of a specific type of error, enabling them to focus their verification efforts on the model components or code that require further attention. Applying formal methods for model and code verification instills more confidence in the engineers building modern embedded systems.

REFERENCES

1. SEI, How Good Is the Software: A Review of Defect Prediction Techniques. Brad Clark, Dave Zubrow, 2001
2. Challenges in deploying static analysis; Jain, Rao, Balan, CrossTalk – August 2011
3. Dependable Embedded Systems, Software Testing. Jiantao Pan 1999.
4. Cousot, "Abstract Interpretation", ACM Computing Surveys, 1996
5. Cousot and Cousot "Abstract Interpretation Based Formal Methods and Future Challenges", Informatics. 10 Years Back. 10 Years Ahead, 2001)
6. Deutsch, "Static Verification of Dynamic Properties, SIGAda, 2003
7. Regehr, J., Reid, A., Webb, K.: Eliminating stack overflow by abstract interpretation. In Proc. of the 3rd International Conf. on Embedded Software (EMSOFT), Philadelphia, PA, October 2003
8. <http://www.di.ens.fr/~cousot/projects/DAEDALUS>
9. Spoto A., "JULIA: A Generic Static Analyser for the Java Bytecode", 1982
10. <http://www.mathworks.com/products/polyspace>

ABOUT THE AUTHORS

Jay Abraham is the Product Marketing Manager of Polyspace products at MathWorks. Jay is part of the team leading product strategy and business development of Polyspace products, specializing in the American marketplace. Prior to joining MathWorks, Jay was a Product Marketing Manager at companies such as Wind River and Magma Design Automation, starting his career with IBM. Jay has a B.S. degree from Boston University, a Master's from Syracuse, and is a graduate of the Institute of Managerial Leadership from The Red McCombs School of Business at The University of Texas at Austin.

E-mail: Jay.Abraham@mathworks.com

Dr. Jon Friedman is the Aerospace & Defense and Automotive Industry Marketing Manager at MathWorks. Jon leads the marketing effort to foster industry adoption the MATLAB and Simulink product families and Model-Based Design. Prior to joining MathWorks, he worked at Ford Motor Company where he held positions ranging from software development research to electrical systems product development. Jon has also worked as an Independent Consultant on projects for Delphi, General Motors, Chrysler and the US Tank-Automotive and Armaments Command. Jon holds a B.S.E., M.S.E. and Ph.D. in Aerospace Engineering as well as a Master's in Business Administration, all from the University of Michigan.

E-mail: Jon.Friedman@mathworks.com

Process Performance Words of Wisdom

Dale Childs, Double Play Process Diagnostics
Paul Kimmerly, USMC-TSO Kansas City

Abstract. Process performance forms the cornerstone of the high-maturity concepts in the CMMI®. High maturity generates great discussion in the CMMI-based process improvement world. However, understanding process performance provides benefits to an organization whether or not it adopts the CMMI. The CMMI provides a framework for an organization's process improvement efforts. At its highest levels, the CMMI describes how an organization can use process performance measures to understand and improve its business processes. While this article will mention the high-maturity process areas from the CMMI, it will primarily focus on the analysis of process performance to help an organization. Comments from wise men and women throughout history illustrate ideas related to process performance. In one of his songs, Jimmy Buffett said, "Chasing illusions can get quite confusing.[1]"

This is true of the use of process performance measures. To understand how the use of process performance measures affect an organization, it is good to look back at some words of wisdom related to the concepts behind the use of performance measures and results. This article will illustrate the practical meaning and benefits of understanding process performance by drawing connections to famous quotes.

Setting the Foundation

Managers often struggle for a clear understanding of what is happening on their projects. They find themselves in the same situation as Alexandre Ledru-Rollin when he said, "There go my people. I must find out where they are going so that I can lead them.[2]"

Good use of process performance data depends on establishing a foundation of measurement collection. In the CMMI, this starts with the Measurement and Analysis (MA) process area. In MA, an organization identifies its information needs and measurement objectives. Managers are always looking for information to help them answer questions like Mr. Ledru-Rollin. By starting with information needs, an organization can specify what is needed to answer some of those management questions. It is important to define those measures clearly so everyone is collecting the same data, the same way. Operational definitions of measures are critical to measurement success. For example, as a measure, a work hour can represent many things. An organization should define what it needs to collect. Is it a direct hour, an indirect hour, a billable hour or a support hour? By clearly defining each measure an organization sets itself up for more accurate and meaningful reporting. Care should be taken to ensure that the data is collected accurately and analyzed appropriately. Managers should also communicate the results of the measurement activities back to the people collecting the measures. By doing so, the managers give the practitioners a stake in the measurements. The measures will mean more to the practitioners, which will lead to more accurate reporting. Without this communication, an organization ends up boxed in as Rowan D. Williams stated, "Bad human communication leaves us less room to grow.[3]"

Measurement establishes the foundation that grows into the ability to use process performance data to help an organization improve. Inaccurate reporting stifles that growth.

Dwight D. Eisenhower said, "Things are more like they are now than they have ever been before.[4]"

While that may seem obvious, in the world of process performance, it cannot be taken for granted. In order to know how things are now, an organization must measure the current state of its process performance and compare it to historical performance. Each of the high-maturity process areas in the CMMI contains practices that look at historical results, measure current performance, forecast future performance, and look to make improvements. As Philip Crosby said, "Making a wrong decision is understandable. Refusing to continually search for learning is not.[5]"

Organizational Process Performance

As mentioned above, an organization establishes measurement goals based on information needs. As the organization accumulates historical measurement data, it can begin to predict process performance based on past results. In the Organizational Process Performance (OPP) process area in the CMMI, the organization refines those goals based on a statistical analysis of historical data and business needs for quality and process performance. This is important because as Douglass Lurtan pointed out, "When you determine what you want, you have made the most important decision of your life. You have to know what you want in order to attain it.[6]"

A statistical analysis of historical data is necessary to validate these goals as attainable. Organizations should avoid setting goals like, "We want to be a world-class provider of choice." No one knows what that means, but it sounds cool. People relate to goals like, "We want to reduce customer found defects by 25% in the next year." Organizations should set goals that are clear, measureable, realistic, and easy to understand. After the organization sets its goals, it needs to identify which processes contribute to achieving those goals. Organizations should not reach for too much in analyzing processes. It takes time and money to perform quantitative analysis. Concentrate on those processes that are of concern or that provide the most insight into the achievement of business needs. There must be business reasons for choosing the processes for analysis.

To determine if they can attain what they want, an organization establishes process performance baselines to understand past performance and process performance models to predict future behavior. Keep in mind that these are just tools because as H. Thiel said, "Models are to be used, not believed.[7]"

These tools give insight into process performance. A process performance baseline shows an organization its expected range

of performance based on past performance. By knowing the expected range of performance, an organization understands whether or not a given process can meet its performance goals. Bertrand Russell said, "The degree of one's emotion varies inversely with one's knowledge of the facts—the less you know, the hotter you get.[8]"

Without the facts, managers can make reactive, emotional decisions. Such decisions often lead an organization down the wrong path. Understanding expected performance reduces emotional decisions by giving managers an objective view and reasonable performance expectations. Emotional reaction goes away and objective decision making becomes possible.

Process performance models allow an organization to explore the relationships between different pieces of their process. By using past performance to understand how the different parts of the process relate to one another, organizations can begin to predict what will happen in later parts of the process based on what happens in an earlier part of the process. This gives an organization understanding of what it can do, not just what it has done. John Wooden stressed, "Do not measure yourself by what you have accomplished, but by what you should have accomplished with your ability.[9]"

Process performance models enable managers to understand their ability, and understand when actual results vary from that ability. In the CMMI, process performance models start with a controllable factor, like project size, and create predictive models based on the understanding of the effects of changes to that factor. For example, an organization knows that a size increase of more than 10% during the design phase causes increases in test defect rates. Such knowledge can be used to determine if additional peer reviews or testers are needed to accommodate the size change and prevent a significant increase in test defects. Other models, while they may not be considered process performance models in CMMI terms can also help organizations understand and manage their projects. For example, if an organization knows that finding a higher than predicted rate of requirements review defects historically means a reduction in test and customer-found defects, it can anticipate performance results and make decisions related to those future lifecycle phases.

Quantitative Project Management (QPM)

In QPM, the project managers within the organization select the measures and techniques they will use to manage process performance. Sharon Salzberg stated, "Each decision we make, each action we take, is born out of intention.[10]"

In QPM, the measures and techniques used in the project are selected based on the objectives established in OPP and any unique aspects of the project. If there is no connection to the organization's objectives, the organization goes back to chasing illusions or, as Bob Seger offered, "Working on mysteries without clues.[11]"

Organizations should consider which items to include and which to leave out. Joshua Schachter said it well when he made the point, "Every decision has a cost. Do I make this decision at all or can I move on to the next thing? What we decided to leave out is almost as important as what we put in.[12]"

The baselines and models that an organization creates give insight into the quality and performance objectives set by the organization. As Confucius said, "The expectations of life depend on diligence; the mechanic that would perfect his work must first sharpen his tools.[13]"

Process performance baselines and models provide the tools, which an organization sharpens over time as it gains an understanding of its process performance. But, tools must be used. As Debra Wilson explains, "People who do not use the tools given to them only injure themselves.[14]"

If an organization does not make use of QPM tools, it loses an opportunity to meet business goals and improve performance. QPM is where projects use the models and baselines that are established in OPP to help manage their projects.

Process Performance and Decisions

Not every process is ripe for process performance measurement. An organization should concentrate on those that directly address business and performance objectives. Start with a small set and build from there. Once an organization understands its past results, other areas of opportunity present themselves. Organizations must start somewhere, because as Washington Irving said, "One of the greatest and simplest tools for learning more and growing is doing more.[15]"

When projects use the tools available to them, they gain insight and make better management decisions. When the actual performance, or prediction of performance, does not match expectations set by the baselines and models, managers should ask questions and take action.

Lee Iococca said, "If I had to sum up in one word what makes a good manager, I would say decisiveness. You can use the fanciest computers to gather the numbers, but in the end you have to set a timetable and act.[16]"

Iococca correctly contends that numbers are not answers. Numbers represent indicators that managers should use to ask questions that lead to better decisions. By establishing baselines and models, an organization sets its managers up to make decisions based on an understanding of process performance. Using our size example from earlier, if an organization knows that an increase in project size of more than 10% causes a corresponding increase in test defects, a manager can adjust staff levels or increase test time to allow for what it expects based on past performance.

Causal Analysis and Resolution

As Crosby pointed out in the earlier quote, organizations must continually learn by looking at their past mistakes and problems. That concept forms the basis for Causal Analysis and Resolution (CAR). Catherine Aird stated, "If you cannot be a good example, then you will just have to be a horrible warning.[17]"

Both good examples and horrible warnings should be looked at when selecting outcomes for analysis in CAR. When analyzing process performance, the organization should look at what has worked well in addition to what needs improvement. Successes should be leveraged across the organization and the root causes of problems should be resolved to prevent the recurrence of the problem. As Chuck Berry told us, "Do not let the same dog bite you twice.[18]"

Often organizations focus on symptoms rather than root

causes. Getting the right people in the room, which means those involved in the process, helps identify root causes of problems or successes. Organizations leverage their successes by analyzing the causes behind them just as they fix problems by analyzing the causes behind them. It may be true as Mark Twain said, "Few things are harder to put up with than the annoyance of a good example.[19]"

However, successes can create peer pressure for others in an organization to improve. By using CAR, an organization can identify which annoying good examples are worth promulgating. On the flip side, it is also true that, "The best way to escape from a problem is to solve it,[20]" as Alan Saporta pointed out. CAR allows an organization to find root causes and prevent problems from recurring again and again and again and …

Organizational Performance Management

Organizational Performance Management (OPM) asks an organization to select the improvements it wants to make and to put structure in place to deploy and analyze improvement proposals. The potential improvements can come from a variety of sources. One source is when a project's results historically show that they cannot reach performance goals. For example, if the goal is to be within 10% of estimates and the project is always 25% to 40% off, the project is unlikely to ever meet the goal without making a process change. Winston Churchill pointed to this when he said, "Success consists of going from failure to failure without loss of enthusiasm.[21]"

However, that success only comes from making change. The results of a CAR discussion can also be the source for potential improvements. For CAR groups to be successful, an organization must provide feedback that shows the results are considered important and that the results are being used. As Colin Powell said, "The day soldiers stop bringing you their problems is the day you have stopped leading them. They have either lost confidence that you can help them or concluded that you do not care. Either case is a failure of leadership.[22]"

Another source comes from looking outside the organization for innovations. Organizations often become enamored with their own ideas and refuse to look outside of themselves. This is the trap Friedrich Nietzsche spoke of when he said, "Many are stubborn in pursuit of the path they have chosen, few in pursuit of the goal.[23]"

All available information and sources, internal and external to the organization should be used to support improvement initiatives. Jimmy Buffett summed this up when he told us, "I have read dozens of books about heroes and crooks, and I have learned much from both of their styles.[24]"

Business goals should drive organizational improvements. Outside ideas can be just as valid as those that come from within.

Validation plays an important role in OPM. There are several ways to validate if an improvement is successful. These include piloting changes, modeling behavior and simulating results. To the extent possible, improvements driven by process changes should be validated statistically to ensure that observed changes are not random. In other words, a quantitative look should be taken to ensure that a significant change has occurred. This prevents the pitfall that Mr. Spock addressed when he said, "A difference that makes no difference is no difference.[25]"

Whatever method is chosen, organizations must find project managers willing to take the first steps in trying out new improvements. They should be willing to follow Frank Zappa's words, "I will do the stupid thing first and then you shy people follow.[26]"

Improvement proposals do not always work. An organization should not try to force an idea because it seems like it should work. Use the validation results to determine if the change is worth adopting. W.C. Fields explained that by saying, "If at first you do not succeed, try, try again. Then quit. There is no use being a damned fool about it.[27]"

However, if the organization determines that the improvement was a success, then a plan should be put in place to deploy it.

Using Process Performance Measures to Manage Change

All improvements require change. Ideas may be easily understood and accepted, but change always comes hard. Charles Kettering explained, "The world hates change, yet it is the only thing that has brought progress.[28]"

While change is critical for improvement, change must be managed. Changes deployed to the organization should be as timely as possible, but accomplished in an orderly fashion. As John Wooden told his teams, "Be quick, but do not hurry.[29]"

Unmanaged change creates chaos, but managed change brings benefits. As Francis Bacon pointed out, "Things alter for the worse spontaneously, if they are not altered for the best designedly.[30]"

Organizations have to deal with unplanned, spontaneous change. Managed change is easier to accept and sets the foundation for future improvement.

Using process performance measures greatly aids an organization in making improvements. However, organizations should not blindly follow the numbers. Numbers can be manipulated as Mark Twain said, "Get your facts first, and then you can distort them as much as you please.[31]"

As stated previously, numbers are just indicators. To create useful indicators, organizations should clearly define the analysis techniques that will be used and the rationale for using them. Organizations must never lose sight of the fact that the perception of the staff is just as important as the numbers. George Santayana explained, "Those who speak most of progress measure it by quantity and not by quality.[32]"

Both the hard numbers and soft perceptions determine the success of any improvement effort. If the numbers look good, but the people have legitimate reasons for objection, the organization must consider their viewpoint. Malcolm Gladwell said it well when he explained the need for balance, "Truly successful decision making relies on a balance between deliberate and instinctive thinking.[33]"

Organizations collect a lot of numbers, but real value comes when they are used. Establishing a measurement foundation enables the use of process performance measures once an organization builds some historical data. The high-maturity process areas in the CMMI provide guidance on how quantitative information and process performance measures can be used to help an organization meet its business goals. Remember as Hesiod stated around 800 BC, "Observe due measure, for right timing is in all things the most important.[34]"

The time is now for building a measurement program with the vision for how performance measures will be used. Understanding process performance can be perplexing. Others weathered the storms of change in the past. In order to plan for the future of process improvements and make meaningful change, organizations should consider words of wisdom from those who came before us.

Disclaimer:

CMMI® is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

ABOUT THE AUTHORS

Dale Childs is a certified CMMI High Maturity Lead Appraiser, a certified CMMI instructor, a certified Lean Six Sigma Green Belt, and a trained Malcolm Baldrige National Quality Award examiner. Dale is an SEI affiliate and the CEO of Double Play Process Diagnostics, Inc. Mr. Childs retired form the DoD in 2008. While at the DoD he was responsible for coordinating his agency's CMMI efforts. Dale is the 2011 recipient of the SEI's Member Representative Award. Mr. Childs' efforts are currently focused on working with organizations pursuing CMMI high maturity and business growth.

Double Play Process Diagnostics Inc.
P.O. Box 17015
Pensacola, FL 32522
Phone: 850-450-5626
E-mail: dale.childs@doubleplayconsulting.com

Paul Kimmerly has nearly 25 years experience in software development for the different incarnations of the United States Marine Corps Technology Services Organization. He recently retired from full-time employment, but remains in a consulting role. He was a member of the organization's Software Engineering Process Group for 18 years, serving as the group's lead for more than 15 years. Paul is a certified HMLA and CMMI instructor for the CMMI for Deveopment and the CMMI for Acquisition. He is an SEI affiliate and a member of CROSSTALK's editorial board. He has presented at past SEPG conferences and contributed several articles on process improvement to CrossTalk. In addition to his part-time duties with the USMC TSO, he works with private clients through Double Play Process Diagnsotics Inc.

4921 W. 72nd Street
Prairie Village, KS 66208
E-mail: paul.kimmerly@mcw.usmc.mil
or pjkimmerly@kcnet.com

REFERENCES

1. Written by Jimmy Buffett, The Legend of Norman Paperman, Don't Stop the Carnival, Island, 1998
2. Alexandre Ledru-Rollin, Suzy Platt, ed. Respectfully quoted: a dictionary of quotations (Barnes & Noble, 1993), p. 194
3. Rowan D. Williams. (n.d.). BrainyQuote.com. Retrieved September 28, 2011, from BrainyQuote.com
4. Dwight D. Eisenhower. (n.d.). BrainyQuote.com. Retrieved September 28, 2011, from BrainyQuote.com
5. Philip Crosby, Philip Crosby's Reflections on Quality: 295 Inspirations from the World's Foremost Quality Guru, McGraw-Hill, September 1, 1995
6. Douglass Lurtan, Quotationsbook.com, retrieved September 28, 2011, from quotationsbook.com
7. Henri Thiel, Famousquotes.com, retrieved September 28, 2011, from famousquotes.com
8. Russell, Bertrand, BrainyQuote.com. Retrieved September 28, 2011, from BrainyQuote.com
9. Wooden, John, BrainyQuote.com. Retrieved September 28, 2011, from BrainyQuote.com
10. Sharon Salzberg, O Magazine, The Power of Intention, January 2004
11. Written by Bob Seger, Night Moves, Night Moves, Capitol, 1976
12. Schachter. quotationspage.com. Retrieved September 28, 2011, from quotationspage.com
13. Confucius. BrainyQuote.com. Retrieved September 28, 2011, from BrainyQuote.com
14. Wilson, Debra. BrainyQuote.com. Retrieved September 28, 2011, from BrainyQuote.com
15. Irving, Washington, BrainyQuote.com. Retrieved September 28, 2011, from BrainyQuote.com
16. Iococca, Lee, quotationspage.com. Retrieved September 28, 2011, from quotationspage.com
17. Aird, Catherine, quotesdaddy.com. Retrieved September 28, 2011, from quotesdaddy.com
18. Berry, Chuck. searchquotes.com. Retrieved September 28, 2011, from searchquotes.com
19. Twain, Mark. BrainyQuote.com. Retrieved September 28, 2011, from BrainyQuote.com
20. Saporta, Alan. Quotationspage.com. Retrieved September 28, 2011, from quotationspage.com
21. Churchill, Winston. BrainyQuote.com. Retrieved September 28, 2011, from BrainyQuote.com
22. Powell, Colin. . Quotationspage.com. Retrieved September 28, 2011, from quotationspage.com
23. Nietzsche, Friedrich. BrainyQuote.com. Retrieved September 28, 2011, from BrainyQuote.com
24. Written by Jimmy Buffett. Son of a Son of a Sailor, Son of a Son of a Sailor, MCA, 1978
25. Blish, James. Spock Must Die, Spectra, March 1, 1985
26. Written by Frank Zappa. Don't Eat the Yellow Snow, Apostrophe, Zappa Records, 1974
27. Fields, W.C., BrainyQuote.com. Retrieved September 28, 2011, from BrainyQuote.com
28. Kettering, Charles. BrainyQuote.com. Retrieved September 28, 2011, from BrainyQuote.com
29. Wooden, John. goodreads.com. Retrieved September 28, 2011, from goodreads.com
30. Bacon, Francis. BrainyQuote.com. Retrieved September 28, 2011, from BrainyQuote.com
31. Twain, Mark. BrainyQuote.com. Retrieved September 28, 2011, from BrainyQuote.com
32. Santayana, George. Quotationspage.com. Retrieved September 28, 2011, from quotationspage.com
33. Gladwell, Malcolm. Quotationspage.com. Retrieved September 28, 2011, from quotationspage.com
34. Hesiod. . BrainyQuote.com. Retrieved September 28, 2011, from BrainyQuote.com

Upcoming Events

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

INCOSE International Symposium 2012
9-12 July 2012
Roma, Italy
http://www.incose.org/newsevents/events/details.aspx?id=142

Practical Software and Systems Measurement (PSM)
16 July 2012
Mystic, CT.
http://psmsc.com/Events.asp

COMPSEC 2012
16-20 July 2012
Izmir, Turkey
http://compsac.cs.iastate.edu/

GFIRST8
19-24 August 2012
Atlanta, GA
http://www.us-cert.gov/GFIRST

26th International Biometrics Conference
26-28 August 2012
Kobe, Japan
http://www.ourglocal.com/event/?eventid=11988

Diminishing Manufacturing Sources and Material Shortages & Standardization
27-30 August 2012
New Orleans, LA
http://www.dmsms2012.com/

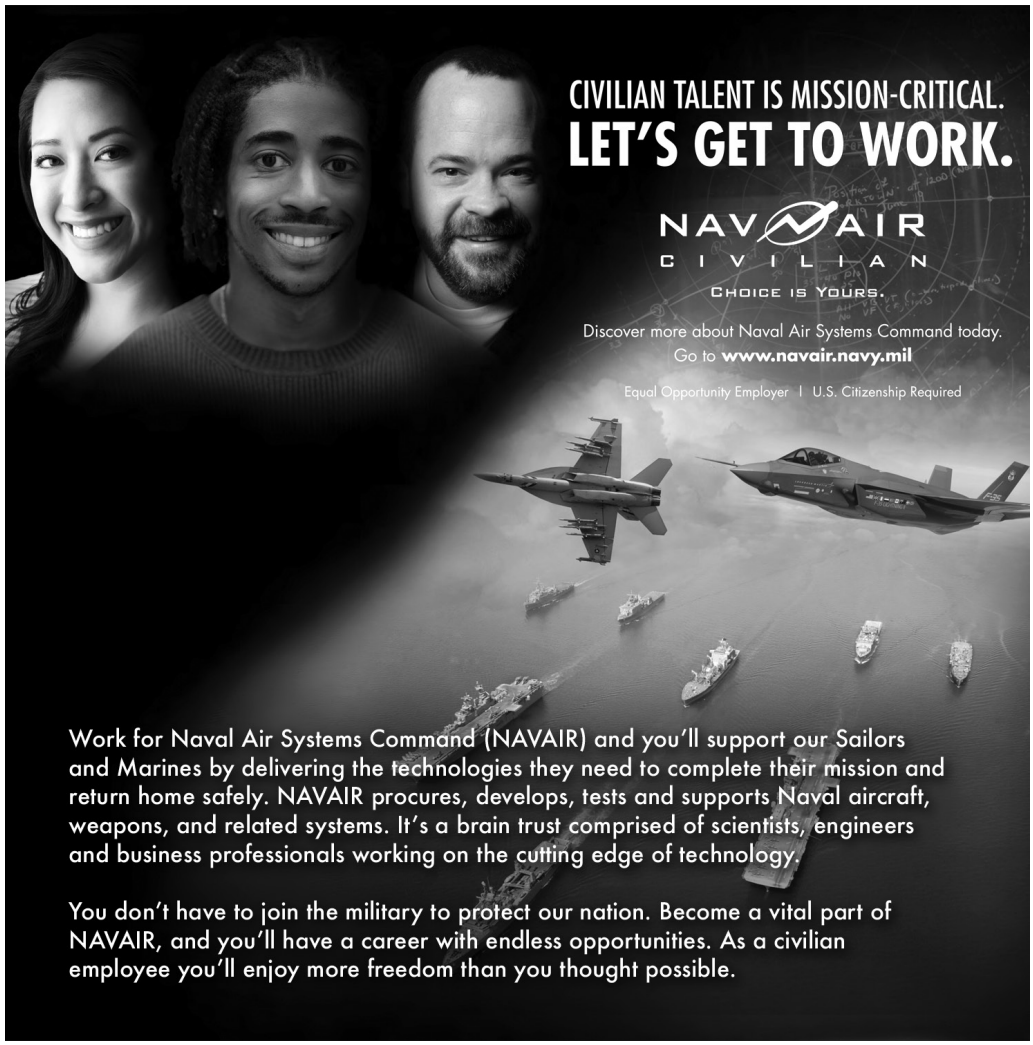
AUTOTESTCON 2012
10-13 September 2012
Anaheim, CA
http://www.autotestcon.com/general/autotestcon-2012

ASIS/(ISC)2 Security Congress
10-13 September 2012
Philadelphia, PA
https://www.isc2.org/congress2012/default.aspx

15th Annual Systems Engineering Conference
22-25 October 2012
San Diego, CA
http://www.ndia.org/meetings/3870/Pages/default.aspx

OWASP AppSec USA 2012
22-26 October 2012
Austin, TX
https://www.owasp.org/index.php/Category:OWASP_AppSec_Conference

12th Annual CMMI Technology Conference
5-8 November 2012
Denver, CO
http://www.ndia.org/meetings/3110/Pages/default.aspx



Thank You!

See you next year!

24th Annual

SSTC Systems & Software Technology Conference

Thank you to all of the sponsors, exhibitors, and attendees that helped make the 24th Annual Systems & Software Technology Conference a big success in April 2012! Visit www.sstc-online.org or follow our Facebook page for information about next year's event.

WWW.SSTC-ONLINE.ORG

Follow Us On Facebook
http://www.facebook.com/TheSSTC

WAR FIGHTING TECHNOLOGIES

ENHANCE ADVANCE MODERNIZE

Luddites of the World, Unite!

During the early 1800s in England, there was a movement protesting progress. In particular, a young man named either Ludham or Ludd (history is murky on this point) gave his name to a movement dedicated to smashing technology to protest the industrial revolution. While their reasons might be considered sound (the new technology allowed the hiring of less-skilled and cheaper labor to replace skilled artisans), their methodology was certainly illegal. The Luddite movement lasted only a few years, but for a brief time the British had more soldiers fighting the Luddites than they had fighting Napoleon. Nowadays, Luddite is used to describe one who is opposed to industrialization, automation, computerization, or new technologies in general.

All I can say about the Luddites is…sometimes I feel their pain; like when buying gas.

I recall that in the good old days I would fill up, walk in, and give the attendant some cash.

When charge cards became popular, you walked into the station, the attendant zipped your card on a paper receipt, and you signed it and were done. Within a few years, the process improved so that you simply inserted your card at the pump, and after filling up, a small receipt emerged from the pump, and you drove off. The system was almost perfect. However, the last time I was at the gas station, I was forced to go through the following process:

• Upon inserting my credit card I was asked, "David, are you a member of the rewards program? If so, scan your rewards card!" (The pump knew my name, surely it could keep tract of the fact that I am a rewards member, and get a $0.03 discount).

• After scanning my rewards card. It asked, "Do you want to apply your $0.03 discount?" (Why, would I not want my discount?)

• After pressing yes, the pump replied, "Do you want a car wash?"

• No. The pump then replied, "Is this a credit or debit card?" (Well, legitimate question, other than I was using my American Express, which really is always a credit card.)

• I pressed credit, then I was asked to enter my zip code. (I am going to figure that if a thief has my credit card and name, he can probably figure out where I live—probably because he has my wallet. Mind you, had I pressed debit I would have been asked for a PIN.)

• Now I am asked, "Do you want a receipt?" (Do they realize that the printer on the pump has been broken for two years now?)

• Finally, am prompted with the words, "Please select grade".

• Nope—not done yet. After I select 87 octane, it responds, "Please hit start to begin". (Please note that this button will either be hidden among many other keys, and/or the word, start, will have long worn off the button, and I am guessing. Heaven forbid I hit cancel instead and start over. Can we please make the start button large, bright red, and extremely well labeled? For that matter, can we assume that once I select the grade of gas and remove the pump nozzle, I am pretty sure I am going to use the fuel. Just turn the pump on!)

I often find myself talking to the pump, explaining that I just want gas—not a hand/eye coordination and reading test before I can start the pump. By the way, once the entire above process is complete, I forgot the final step:

• As soon as the gas starts flowing, the pump now responds with a blaringly loud obnoxious advertisement for the weekly store specials, usually along the lines of, "Now on sale this week for only $4.99—EZSprinkle Shoe Deodorizer." Which, of course, makes me press blindly for the mute button. Heaven forbid I accidentally hit cancel.

Things I used to do on a full-sized computer I now do on a tablet or a smart phone. I find myself using full-sized computers less and less, and other devices such as inter-connected cable boxes and DVD players more and more. My personal smartphone is now my mailbox, contact list, and Google search interface. There will always be a need for personal computers—but, now, instead of a "personal computer," I use devices that are "more personal." The software that runs it all, however, continues to increase in size and complexity. And sometimes decreases in end-user simplicity.

We cannot neglect the human element. The need for software that is simple and understandable remains. Maybe I am a Luddite. I am deeply opposed to progress that makes my life harder. I want things to evolve towards simple and easy to use. The need for end-user buy-in and reliable and understandable software is constant—regardless of the size or shape or evolution of its processor. As it should be.

David A. Cook
Stephen F. Austin State University
cookda@sfasu.edu

Hill Air Force Base is hiring
SOFTWARE ENGINEERS AND COMPUTER SCIENTISTS

Exciting and Stable Workloads:

- ★ Joint Mission Planning System
- ★ Battle Control System-Fixed
- ★ Satellite Technology
- ★ Expeditionary Fighting Vehicle
- ★ F-16, F-22, F-35
- ★ Ground Theater Air Control System
- ★ Human Engineering Development

Employee Benefits:

- ★ Health Care Packages
- ★ 10 Paid Holidays
- ★ Paid Sick Leave
- ★ Exercise Time
- ★ Career Coaching
- ★ Tuition Assistance
- ★ Retirement Savings Plans
- ★ Leadership Training

Location, Location, Location:

- ★ 25 minutes from Salt Lake City
- ★ Utah Jazz Basketball
- ★ Three Minor League Baseball Teams
- ★ One Hour from 12 Ski Resorts
- ★ Minutes from Hunting, Fishing, Water Skiing, ATV Trails, Hiking

Visit us at *www.309SMXG.hill.af.mil*. Send resumes to *shanae.headley@hill.af.mil*.
Also apply for our openings at *USAjobs.gov*

U.S. Department of Homeland Security

NAVAIR

309th Software Maintenance Group

CrossTalk thanks the above organizations for providing their support.