CrossTalk

February 2009 **The Journal of Defense Software Engineering** Vol. 22 No. 2

Software & Systems Integration

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **FEB 2009**	2. REPORT TYPE	3. DATES COVERED **00-00-2009 to 00-00-2009**

4. TITLE AND SUBTITLE **Crosstalk, The Journal of Defense Software Engineering. Volume 22, Number 2, February 2009**	5a. CONTRACT NUMBER
	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 Avenue,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

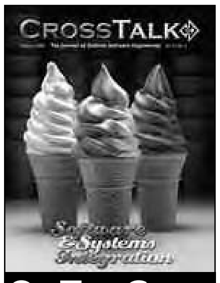
14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**32**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

Software and Systems Integration



On The Cover
Cover Design by Kent Bingham

Additional art services provided by Janna Jensen

Departments

CrossTalk

OSD (AT&L)	*Kristin Baldwin*
NAVAIR	*Joan Johnson*
309 SMXG	*Karl Rogers*
DHS	*Joe Jarzombek*
Managing Director	*Brent Baxter*
Publisher	*Kasey Thompson*
Managing Editor	*Drew Brown*
Associate Editor	*Chelene Fortier-Lozancich*
Publishing Coordinator	*Nicole Kentta*
Phone	(801) 775-5555
E-mail	stsc.customerservice@hill.af.mil
CrossTalk Online	www.stsc.hill.af.mil/crosstalk

CrossTalk, The Journal of Defense Software Engineering is co-sponsored by the Office of the Secretary of Defense (OSD) Acquisition, Technology and Logistics (AT&L); U.S. Navy (USN); U.S. Air Force (USAF); and the U.S. Department of Homeland Security (DHS). OSD (AT&L) co-sponsor: Software Engineering and System Assurance. 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 Programs Directorate.

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

S T S C

Subscriptions: Send correspondence concerning subscriptions and changes of address to the following address. You may e-mail us or use the form on p. 17.

517 SMXS/MXDEA
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

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.stsc.hill.af.mil/crosstalk/xtlkguid.pdf>. 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 author 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: This Department of Defense (DoD) journal 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: See <www.stsc.hill.af.mil/crosstalk>, call (801) 777-0857 or e-mail <stsc.webmaster@hill.af.mil>.

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

Software and Systems Integration: A Historical Perspective

During the Cold War era, military strategy was predicated on the belief that deterrence was assured through arms superiority—the ability to impose overwhelming force and mass on a global scale. These capabilities were achieved through scientific management principles imposed on a capital-intensive industrial base. This approach achieved significant economies of scale. But industrial management principles reinforced a functionally segregated "stovepipe" perspective within the defense establishment.

With the end of the Cold War, industrial-strength deterrence was no longer sufficient. Operational advantage now derives from speed, agility, and precision, which encourages adaptive planning, accelerated cycle times, and a collaborative approach to problem solving. Over the past decade, the DoD acquisition process has been realigning itself from a "system centric" to an "enterprise-wide" or "capabilities-based" paradigm. This strategic redirection reflects fundamental changes in national defense, driven by historic changes in the geopolitical landscape. Individual systems are not only becoming larger and more complex, but most are now expected to be integrated across a complex enterprise. The problem is further exacerbated by the fact that many programs lack a full appreciation of the extent of these integration efforts and therefore underestimate the work effort.

With more than four years of systemic analysis data gathered by the Systems and Software Engineering Directorate on program reviews, our engineers see that a lack of systems and software integration has caused many programs to perform in a suboptimal manner, contributing to cost, schedule, and performance issues. Functional specialization has its role, but successful system development could benefit from increasing multidisciplinary and collaborative approaches across engineering specialties and throughout the life cycle—in particular, attention to engineering early in the development life cycle.

This month's CrossTalk features excellent articles on the topic of software and systems integration. In *Leveraging Federal IT Investment With Service-Oriented Architecture* (SOA), Geoffrey Raines examines how an SOA offers federal senior leadership teams an incremental and focused path forward in utilizing decades of IT investment in existing systems. Five authors take an insider look at the Lockheed Martin Aeronautics Company's *Requirement Modeling for the C-5 Modernization Program*, detailing the process and showing its benefits. Paul E. McMahon explains how Agile techniques can help with the evolutionary acquisition of defense systems in *Defense Acquisition Performance: Could Some Agility Help*?

There are also two informative supporting articles this month that outline exactly what their titles suggest, with *A Model to Quantify the Return on Investment Assurance* by Ron Greenfield and Dr. Charley Tichenor, and Dr. Kelvin Nilsen's *Enforcing Static Program Properties in Safety-Critical Java Software Components*.

As you consider the thoughts in these articles, remember that the software and system engineer's role in combating global emergent threats is no less compelling than that of the front-line warriors whose lives often rest on how well we do our job. We should undertake our tasks with a dedication worthy of the challenge, and take a corresponding amount of pride in our accomplishments. As such, we should strive to coordinate and cooperate in order to accomplish the development of new systems better, faster, and at a reasonable cost.

Bruce Amato

Bruce Amato
Assistant Deputy Director, Software Engineering and Systems Assurance
Office of the Deputy Under Secretary of Defense, Acquisition and Technology

Leveraging Federal IT Investment With Service-Oriented Architecture©

Geoffrey Raines
The MITRE Corporation

Service-oriented architecture (SOA) builds on computer engineering approaches of the past to offer an architectural approach for enterprise systems, oriented around offering services on a network of consumers. For federal senior leadership teams, it offers a path forward, allowing for incremental and focused improvement of their IT support systems. With thoughtful engineering and an enterprise point of view, SOA offers positive benefits such as language neutral integration, component reuse, organizational agility, and the ability to leverage past investment in existing systems.

Similar to the nation's *Fortune* 500 leadership, today's federal leadership teams often find themselves facing significant IT investment and portfolio challenges. They have inherited a computing infrastructure that is often not uniform, and whose technologies span the recent history of computing. The IT infrastructures tend to have diverse environments, complex business logic, inconsistent interfaces, and limited sustainment budgets:

- **Diverse environments.** Mainframe systems, client/server systems, and multi-tier Web-based systems sit side-by-side, demanding operations and maintenance resources from a technology marketplace in which the cost of niche legacy technical skills continues to rise. The portfolio of systems are generally written in a number of different software development languages such as COBOL, Java, assembly, and C, requiring heterogeneous staff skill sets and experience in a variety of commercial products, some of which are so old that they no longer offer support licenses.
- **Complex business logic.** The systems often conform to a set of complex business logic that has developed over a number of years in response to evolving legal requirements, congressional reporting mandates, changes in contractor teams, and refinement of business processes. While some systems are new and robust, many are brittle and hard to modify, relying on technical skills not common in the marketplace that become increasingly more expensive. The maintenance tail on these systems is surprisingly high and competes for resources with required new functionality.
- **Inconsistent interfaces.** Interfaces between systems have grown up spontaneously without enterprise planning over many years. The interfaces are the result of one-off negotiations between various parts of the organization, and have been designed using many varied technologies during the organization's IT history, following no consistent design pattern. Recent enterprise architecture efforts have documented the enterprise interfaces in diagrams that resemble a Rorschach inkblot test.
- **Limited sustainment budgets.** Even without the continuous downward pressure on IT budgets brought by competing national requirements and the view that IT should be increasingly viewed as a commodity, there are not enough budget resources or human resources to recast the portfolio of systems to be modern and robust in one action. David Longworth writes:

> According to analysts at Forrester Research, there are some 200 billion lines of COBOL—the most popular legacy programming language—still in use. Nor is it going away: maintenance and modifications to installed software increase that number by 5 billion lines a year. IBM, meanwhile, claims its CICS [Customer Information Control System] mainframe transaction software handles more than 30 billion transactions per day, processes $1 trillion in transaction values, and is used by 30 million people. [1]

Given budget constraints, an incremental approach seems to be required.

A Path Forward

SOA, as implemented through the common Web services standards, offers federal senior leadership teams a path forward given the diverse and complex IT portfolio that they have inherited, allowing for incremental and focused improvement of their IT support systems. With thoughtful engineering and an enterprise point of view, SOA offers several positive benefits.

Language Neutral Integration

Web-enabling applications with a common browser interface became a powerful tool during the '90s. In the same way that HTML defined a simple user browser interface that almost all software applications could create, Web services defined a programming interface available in almost all environments. The HTML interface at the presentation layer became ubiquitous because it was easy to create, as it was composed of text characters. Similarly, the foundational contemporary Web services standards use XML, which again is focused on the creation and consumption of delimited text. The bottom line is that regardless of the development language your systems use, your systems can offer and invoke services through a common mechanism.

Component Reuse

Given current Web services technology, once an organization has built a software component and offered it as a service, the rest of the organization can then utilize that service. Given proper service governance—including items such as service provider trust, service security, and reliability—Web services offer the potential for aiding the more effective management of an enterprise portfolio, allowing a capability to be built well once and shared, in contrast to sustaining redundant systems with many of the same capabilities (e.g., multiple payroll, trouble ticket, or mapping systems in one organization). Reuse, through the implementation of enterprise service offerings, is further discussed later in this article.

Organizational Agility

SOA defines building blocks of software capability in terms of offered services that meet some portion of the organization's requirements. These building blocks, once defined and reliably operated, can be recombined and integrated rapidly. Peter

Fingar stated, "Classes, systems, or subsystems [that] can be designed as reusable pieces. These pieces can then be assembled to create various new applications" [2]. Agility—the ability to more rapidly adapt a federal organization's tools to meet their current requirements—can be enhanced by having well-documented and understood interfaces and enterprise-accessible software capabilities.

Leveraging Existing Systems

One common use of SOA is to encapsulate elements or functions of existing application systems and make them available to the enterprise in a standard agreed-upon way, leveraging the substantial investment already made. The most compelling business case for SOA is often made regarding leveraging this legacy investment, enabling integration between new and old system components. When new capabilities are built, they are also designed to work within the chosen component model. Given the size and complexity of the installed federal application system base, being able to get more value from these systems is a key driver for SOA adoption. David Litwack writes:

> The movement toward Web Services will be rooted not in the invention of radical new technology, but rather in the Internet-enabling and re-purposing of the cumulative technology of more than 40 years. Organizations will continue to use Java, mainframe and midrange systems, and Microsoft technologies as a foundation for solutions of the future. [3]

Of course, SOA as a concept has existed for many years, and communications between service consumers and providers have been implemented with a number of protocols and approaches before Web services. Web services standards have brought renewed contemporary interest in SOA because of its use of textual XML and its ability to be generated and consumed in diverse computing platforms.

The benefits mentioned, however, accrue only as the result of comprehensive engineering and a meaningful architecture at the enterprise level. SOA as a service concept in no way eliminates the need for strong software development practices, requirements-based life cycles, and an effective enterprise architecture. Done right, SOA offers valuable benefits; however, SOA without structured processes and governance will lead to traditional software system problems.

The Increasing Span of Integration

SOA and its implementing standards, such as the Web services standards, come to us at a particular point in computing history. While several key improvements (such as language neutrality) differentiate today's Web services technologies, there has been a long history of integrating technologies with qualities analogous to Web services, including a field of study often referred to as Enterprise Application Integration (EAI). One of the key trends driving the adoption of Web services is the increasing span of integration being attempted in organizations today. Systems integration is increasing both in complexity within organizations and across external organizations. We can expect this trend to continue as we combine greater numbers of data sources to provide higher value information. Ronan Bradley writes:

> CIOs often have difficulty in justifying the substantial costs associated with integration but, nevertheless, in order to deliver compelling solutions to customers or improve operational efficiency, sooner or later an organization is faced with an integration challenge. [4]

Drawing Parallels: "*Past Is Prologue*" [5]

During the '70s, electronics engineers experienced an architectural and design revolution with the introduction of practical, inexpensive, and ubiquitous integrated circuits (ICs). This revolution in the design of complex hardware systems is informative for contemporary software professionals now charged with building enterprise software systems using the latest technologies of Web services in the context of SOAs.

Like SOA, the IC revolution was fundamentally a distributed, multi-team, component-based approach to building larger systems. Through the commercial marketplace, corporations built components for use by engineering teams around the world. Teams of engineers created building blocks in the form of IC components that could then be described, procured, and reused.

Like software services, every IC chip has a defined interface. The IC interface is described in several ways. First, the chip has a defined function—a predictable behavior that can be described and provides some value for the consumer. Next, the physical dimensions of the chip are enumerated. For example, the number and shape of pins is specified, as are the electronic signaling, timing, and voltages across the pins. All of these characteristics make up the total interface definition for the IC. Of course, software services do not have an identical physical definition, but an analogous concept of a comprehensive interface definition is still viable. Effective software components also possess a predictable and definable behavior.

Introducing and using ICs includes the following considerations:

- **Who Pays?** Building an IC chip the first time requires a large expenditure of resources and capital. The team who builds the IC spends considerable resources. The teams who reuse an IC, instead of rebuilding them, save considerable time and expense. A chip might take $500,000 to build the first time, and might be available for reuse in a commercial catalog for $3.99. The creation of the chip the first time involves many time-consuming steps including requirements analysis, behavior definition, design, layout, photolithography, testing, packaging, manufacturing, and marketing [6]. The team who gets to reuse the chip instead of rebuilding it saves both time and dollars. At the time, designs of over 100,000 transistors were reported as requiring hundreds of staff-years to produce manually [7].
- **Generic or Specialty Components?** Given the amount of investment required to build a chip, designs were purposely scoped to be generic or specific with particular market segments and consumer audiences in mind. Some chips only worked for very specific problem domains, such as audio analysis. Some were very generic and were intended to be used broadly, like a logic multiplexer. The bigger the market and the greater the potential for reuse, the easier it was for a manufacturer to amortize costs against a broader base, resulting in lower costs per instance.
- **Increased Potential Design Scope.** By combining existing chips into larger assemblies, an engineer could quickly leverage the power of hundreds of thousands of transistors. In this way, IC reuse expanded the reach of average engineers, allowing them to leverage resources and dollars spent far in excess of the local project budget.
- **Design Granularity.** The designer of an IC had to decide how much logic to place in a chip to make it most effective in the marketplace. Should the designer create many smaller-function chips, or fewer larger-function chips? Families of chips were often built with the inten-

tion of their functions being used as a set, not unlike a library of software functions. Often, these families of chips had similar interface designs such as consistent signal voltages.

- **Speed of integration.** As designers became familiar with the details of component offerings and leveraged pre-built functions, the speed at which an *integrated* product (built of many components) could come to market was substantially increased.
- **Catalogs.** When the collection of potential ICs offered became large, catalogs of components were then created and classification systems for components were established. Catalogs often had a combination of sales and definitive technical information. The catalogs often had to point to more detailed resources for the technical audiences purchasing the components.
- **Testing.** Technical documents defined the expected behavior of ICs. Components were tested by both the manufacturer and the marketplace. Anomalous behavior by ICs became noted as errata in technical specifications.
- **Engineering support.** IC vendors offered advanced technical labor support to customers in the form of application engineers and other technical staff. Helping customers use the products fundamentally supported product sales.
- **Value chains.** Value chains consume raw components and produce more complex, value-added offerings. ICs enabled value chains to be created as collections of chips became circuit boards, and collections of circuit boards became products.
- **Innovation.** ICs were put together in ways not anticipated by their designers. Teams who designed chips could not foretell all the possible uses of the chips over the years. Componentized logic allowed engineers to create innovative solutions beyond the original vision of component builders.

One might ask: "Were electrical engineers successful with this component-based approach?" Certainly the marketplace was populated by a very large number of offerings based in some part on ICs. Certainly many fortunes and value chains were created. The cost-effectiveness of the reuse approach was validated by the fact that it became the predominant approach of the electronics industry. In short, electronic offerings of the time could not be built to market prices if each chip, specification, module, or component had to be refabricated on each project. Reuse, through component-based methods enabled by new technologies, led this revolution. Yet, the transformation took a decade to occur.

An SOA Analogy

In many ways, the described IC chip revolution is analogous to the effort under way with Web services today. Clearly, Web services components have analogous interfaces definitions as well as defined and documented behaviors that provide some benefit to a potential consumer. One can also reasonably expect that the team producing the Web service will incur substantial expenses that consumers of the service will not. For example, high reliability requirements for the operation of a service and its server and network infrastructure can be a new cost driver for the provider. To continue the analogy, collections of service offerings are becoming sufficiently large enough to require some librarian function to organize, catalog, and describe the components. Many SOA projects use a service registry, such as Universal Description, Discovery, and Integration for this purpose. Enterprise integration engineers are realizing the ability to more rapidly combine network-based service offerings and a new paradigm, sometimes referred to as a *mashup*, is demonstrating the speed at which integration can now occur [8]. Value chains of data integration are already occurring in the marketplace. A data integrator can ingest the product of multiple services and produce a service with correlated data of greater value. Finally, it is also safe to say that service providers may be surprised at how their services get integrated over time and they may be part of larger integration that they could not have foreseen during the original design[1] [9]. In summary, many aspects of the current SOA efforts follow similar component-based patterns, and many of the benefits realized historically by the IC revolution could be potentially realized by SOA efforts.

"The enterprise ... saves resources every time a project reuses a current software service rather than creating redundant services based on similar underlying requirements ..."

Reuse

Historic Source Code Reuse

During the '80s, many organizations, including the DoD, attempted to reuse source code modules with little success. For example, during the DoD's focus on the Ada language, programs were established to reuse Ada language functions and procedures across projects [10]. The basic reuse premise outlines a process where a producer of a source code module would post the source code to a common shared area along with a description of its purpose and its input and output data [11]. At that point, staff from another project would find the code module, download it, and decide to invoke it locally in their source code and actually compile it into their local libraries and system executables. As an example, the DoD states that:

> One of the design goals of Ada was to facilitate the creation and use of reusable parts to improve productivity. To this end, Ada provides features to develop reusable parts and to adapt them once they are available. [12]

For example, Project A might create a high-quality sorting function, and Project B could then compile that function into its own software application.

Though well-intentioned, the actual discovery and reuse of the source code modules did not happen on a large scale in practice. Reasons given for the lack of reuse (at the time) included: lack of trust of mission-central requirements to an external producer of the source code, failure to show a benefit to the contractor *reuser* implementing later systems, inadequate descriptions of the behavior of a module to be reused, and inadequate testing of all the possible outcomes of the module to be reused [13]. All in all, the barriers to reuse were high.

Service Reuse

The danger in describing the use of services as *reuse* is that the reader will assume I mean the source code reuse model just described. In fact, the nature of service reuse is closer to the model of the reuse of ICs by electrical engineers (as outlined in the *Drawing Parallels* section), though still having common issues of trust, defined behavior, and expected performance. In plain terms, reuse in the service context does not mean rebuilding a service, but rather the using again or invoking of a service built by someone else.

The enterprise as a whole saves resources every time a project reuses a current software service rather than creating redundant services based on similar underlying requirements and adding to an agency's maintenance portfolio. Since a system's maintenance costs (over their lifetime) often exceed the cost to build them, the enterprise saves not only in the development and establishment cost of a new service but also in the 20-plus year maintenance cost over the service's life cycle. As one Web vendor stated:

> Web Services reuse is everything: on top of the major cost savings ... reuse means there are fewer services to maintain and triage. So reuse generates savings—and frequency of use drives value in the organization. [14]

However, we should not assume a straight-line savings, where running one service is exactly half as costly as running two services: as the cost of running a service is also impacted by the number of service consumers. Consolidation can make the remaining service more popular, with a greater demand on resources.

Reuse of a service differs from source code reuse in that the external service is called from across the network and is not compiled into local system libraries or local executables. The provider of the service continues to operate, monitor, and upgrade the service (as appropriate). Thanks to the benefits of contemporary Web service technologies, the external reused service can be in another software language, use a completely different multi-tiered or single-tiered machine architecture, be updated at any time with a logic or patch modification by the service provider, represent five lines of Java or 5 million lines of COBOL, or be mostly composed of a legacy system written 20 years ago. In these ways, service reuse is very different from source code reuse of the past.

Some aspects of reuse remain unchanged. The consumer of the service still needs to trust the reliability and correctness of the producer's service. The consumer must be able to find the service and have adequate documentation accurately describing the behavior and interface of the service. Performance of the service is still key. As ZDNet's Joe McKendrick stated:

> Converging trends and business necessity—above and beyond the SOA 'vision' itself—may help drive, or even force, reuse. SOA is not springing from a vacuum, or even from the minds of starry-eyed idealists. It's becoming a necessary way of doing business, of dispersing technology solutions as cost-effectively as possible. And, ultimately, providing businesses new avenues for agility, freeing up processes from rigid systems. [15]

Mature SOAs should measure reuse as part of a periodic portfolio management assessment [16]. The Progress Actional Web site stated that reuse is not only a key benefit of SOA, but also something quantifiable:

> You can measure how many times a service is being used and how many processes it is supporting, thus the number of items being reused. This enables you to measure the value of the service. [14]

The assessment of reuse can be effectively integrated into the information repository used for service discovery in the organization—the enterprise catalog.

SOA as an Enterprise Integration Technology

EAI is a field of study in computer science that focuses on the integration of systems of systems and enterprise applications. With the span of attempted systems integration and data sharing continually increasing in large organizations, the EAI engineering discipline has become increasingly central to senior leadership teams managing portfolios of applications.

The fundamental EAI tenets are based on traditional software engineering methods, though the scale is often considerably larger. While the traditional software coder focused on the parameters that would be sent to, and received from, a function or procedure, the EAI engineer focuses on the parameters that are exchanged with an entire system. The traditional coder might have been writing one hundred source lines of code (SLOC) for a function, while the EAI engineer might be invoking a system with a million SLOC and several tiers of hardware for operational implementation. However, the overall request/response pattern is the same, and the logic issues (such as error recovery) must still be handled gracefully.

SOA can be considered another important step in a 30-year history of EAI technologies. As Chris Harding stated: "SOA eliminates the traditional spaghetti architecture that requires many interconnected systems to solve a single problem" [17].

An SOA's ability to run logic and functions from across a network is not new. Recent examples include Enterprise JavaBeans by Sun Microsystems, Inc., Common Object Request Broker Architecture by the Object Management Group, as well as the Component Object Model, Distributed Component Object Model, and .NET from the Microsoft Corporation. The various methods have differed in the ease with which integration could occur from a programmer's point of view, the methods for conveying run-time errors, the ports required to be open on a network, the quantity of enterprise equipment to operate, and the general design approaches to fault tolerance when failures occur.

Like owners of many other systems of systems environments, decision makers for command and control systems and intelligence systems have an opportunity to leverage SOA to better enable more rapid integration and reconnection of system components. Services can be developed from legacy data sources and existing investment in procedural logic. Aggregation and correlation services can combine the output of more fundamental services to add value for consumers. Finally, registries can detail the ensemble of IT services that an organization will maintain as a portfolio.

Conclusion

SOA offers federal leadership teams a means to effectively leverage decades of IT investment while providing a growth path for new capabilities. SOA provides a technical underpinning for structuring portfolios as a collection of discrete services, each with a definable customer base, an acquisition strategy, performance levels, and a measurable operational cost.

A key current challenge for many federal organizations is the structuring of IT portfolios around a component-based service model and enforcing sufficient standards within their own organizational boundaries, which can be quite large. As the span of attempted integration continues to grow, the challenge of the next 10 years will be enabling that integration model to bridge multiple external organizations that undoubtedly will be using disparate standards and tools.◆

References

1. Longworth, David. "Service Reuse Un-

locks Hidden Value." Loosely Coupled. 29 Sept. 2003 <www.looselycoupled.com/stories/2003/reuse-ca0929.html>.
2. Fingar, Peter, et. al. Next Generation Computing: Distributed Objects for Business. New York: SIGS Books & Multimedia, 1996.
3. Litwack, David, and Peter Fingar. "In the Fast Lane." Internet World Magazine. 1 June 2002 <http://iw.com/magazine.php?inc=060102/06.01.02ebusiness1.html>.
4. Bradley, Ronan. "Agile Infrastructures." GDS InfoCentre. 2008 <http://gdsinternational.com/infocentre/artsum.asp?mag=184&iss=150&art=25901&lang=en>.
5. Shakespeare, William. The Tempest.
6. Intel. "How Chips are Made." 2008 <www.intel.com/education/makingchips/preparation.htm>.
7. Panasuk, Curtis. "Silicon Compilers Make Sweeping Changes in the VLSI." Design World, Electronic Design. 20 Sept. 1984: 67-74.
8. "Mashup Dashboard." Programmable Web. 13 Nov. 2008 <www.programmableweb.com/mashups>.
9. International Genetically Engineered Machine Competition. "Registry of Standard Biological Parts." 2008 <http://partsregistry.org/Main_Page>.
10. DoD. Ada Joint Program Office. Ada 95 Quality and Style Guide Online. Chapter 8. Oct. 1995 <www.adaic.com/docs/95style/html/sec_8/>.
11. Boehm, B.W., et al. "An Environment for Improving Software Productivity." Computer. June 1984.
12. DoD. Ada Joint Program Office. Ada Quality and Style: Guidelines for Professional Programmers. Oct. 1995 <www.adaic.org/docs/95style/95style.pdf>.
13. Traez, Will. Software Reuse: Motivators and Inhibitors. Proc. of COMPCON. Spring 1987.
14. Progress Actional. "Web Services Use and Reuse." <www.actional.com/resources/whitepapers/SOA-Worst-Practices-Vol-I/Web-Services-Reuse.html>.
15. McKendrick, Joe. "Pouring Cold Water on SOA 'Reuse' Mantra." ZDNet. 30 Aug. 2006 <http://blogs.zdnet.com/service-oriented/?p=699>.
16. Roch, Eric. "SOA Service Reuse." 23 Feb. 2007 <http://blogs.ittoolbox.com/eai/business/archives/SOA-Service-Reuse-14699>.
17. Harding, Chris. "Achieving Business Agility Through Model-Driven SOA." ebiz. 29 Jan. 2006 <www.ebizq.net/topics/soa/features/6639.html>.

Note

1. This same component-based approach is also being examined for genetics work. The same interface definition, behavior, cataloging, and reuse discussions are currently occurring, creating a new genetic sub-field known as *synthetic genetics*.

About the Author

Geoffrey Raines is a principal software systems engineer for The MITRE Corporation's Command and Control Center, supporting a variety of government sponsors. Previously, he was the vice president and chief technical officer of Electronic Consulting Services, Inc.—an information technology and engineering consulting professional services firm, where he developed engineering solutions for federal clients. He has a bachelor's degree in computer science from George Mason University.

The MITRE Corporation
7525 Colshire DR
McLean, VA 22102-7539
E-mail: soa-list@lists.mitre.org

Web Sites

The Agile/Waterfall Cooperative

www.rallydev.com/documents/AgileWaterfallCoop-Sliger.pdf

Agile and Waterfall methodologies have different ways of measuring progress, determining success, managing teams, organizing, and communicating. How can they be managed as part of a cohesive project portfolio? Can they coexist and still make the company successful? Software development expert Michele Sliger looks at how continuous improvement through time-boxed iterative deliveries and reviews, implementation of the most important items first, and constant collaborative communication lead to success. Sliger also provides transitional techniques (for Waterfall up-front, at-end, and in-tandem processes) and 10 keys to success.

IBM Federal Service-Oriented Architecture (SOA) Institute

www-03.ibm.com/industries/government/us/detail/resource/N586710B88615G50.html

The Federal SOA Institute's mission is to help the government adopt and benefit from SOA by providing a robust educational environment, advanced solution development capabilities, and opportunities for innovation and collaboration. Along with helping serve that mission, this Web site assists federal agencies in identifying new ways to quickly build and utilize IT systems, integrate and reuse legacy systems, and reduce overall development, systems integration, and operations costs.

Examples of C++ in Safety-Critical Systems

www.cpptalk.net/examples-of-c-in-safety-critical-systems-vt13505.html

Many have heaped praise on Ada and Java for safety-critical systems, and CrossTalk is guilty as charged. Still, there are several examples of the tried-and-true C++ language serving as a perfect—and secure—alternative. As part of the C++TalkNet Forum, this site is an open discussion of C++ in safety-critical scenarios: how it's being used successfully, personal experiences in usage and implementation, published research and conference proceedings, and overall support and encouragement for users of the lesser-known safety-critical systems alternative.

Joint Strike Fighter (JSF) Air Vehicle – C++ Coding Standards

www.research.att.com/~bs/JSF-AV-rules.pdf

If you're a C++ programmer looking for a good set of rules for safety-critical and performance critical code, Lockheed Martin shares the tools its team successfully used for the DoD's JSF program. This site provides direction and guidance that will enable C++ programmers to employ good programming style and proven programming practices leading to safe, reliable, testable, and maintainable code. As well, this document will help programmers develop code that conforms to safety-critical software principles.

Requirement Modeling for the C-5 Modernization Program©

Steven D. Allen, Mark B. Hall, Verlin Kelly, and Mark D. Mansfield
Lockheed Martin Aeronautics Company

Dr. Mark R. Blackburn
Systems and Software Consortium

This article outlines the approach and benefits of a requirement-based modeling effort for the Lockheed Martin Aeronautics Company C-5 Modernization (C-5M) Program to upgrade the aircraft's engines and improve the overall reliability of the aircraft. Requirement-based modeling resulted in more consistent, complete, and precise requirements and interface information to support the design and implementation process. Systems engineers used simulations to validate requirement models and detected a large number of requirement defects that were corrected well before software implementation.

The Lockheed Martin Aeronautics Company C-5M Program is involved in the upgrade of aircraft engines and associated software systems to improve the overall aircraft reliability. Several C-5M project personnel had successfully used the software cost reduction (SCR) requirement modeling method to develop requirements for the C-130J Avionics System [1]. Requirement-based modeling develops precise behavioral requirements and formalizes interface information earlier in the development life cycle to support the design and implementation process. The C-5M Program implemented several related strategies for improving the specification, design, and implementation of the avionics software. These strategies supported on-schedule releases of major functional blocks with a significant reduction in post-release problem reporting and correction. This article focuses on the requirement specification process improvements realized through the use of requirement modeling and early requirement validation.

A *requirement model* is a formal specification of the required functional behavior of a component specified in terms of the interfaces to the component. The modeling process, supported by automated analysis provided in the modeling tool, helps detect requirement and interface problems. In addition, system engineers use a requirement simulator to validate modeled requirements prior to transfer to the designers and implementers. A *requirement simulator* is a tool that loads requirement models and supports scenario execution against the functional behavior captured in the model through a graphical user interface (GUI). A *scenario* is a sequence of input events that result in a corresponding sequence of internal state and output changes. A scenario can represent high-level system use cases or low-level component interactions. Unexpected state or outputs observed during the scenario simulation are often results of requirement defects.

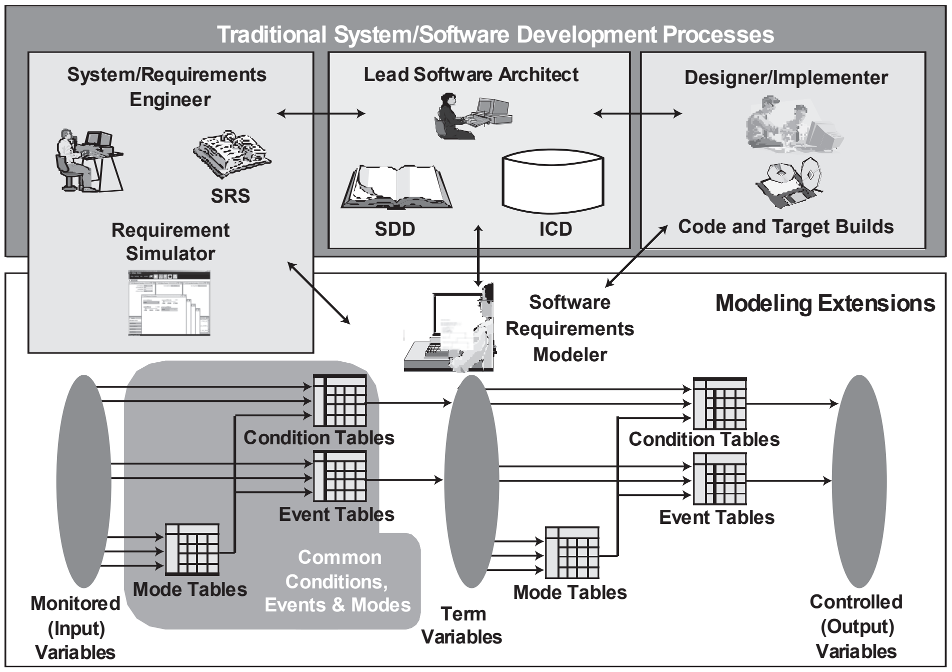
The modeling process and requirement simulation exposed a large number of requirement defects. Fortunately, these defects were identified early in the project and corrected before the software implementation process. Integration problem reports (IPRs) provided a key measure of the defects against requirements, design, and implementation. The number of IPRs was reduced significantly when compared to a prior and similar project, the C-5 Avionics Modernization Program (AMP). This article provides IPR measurement and tracking data that substantiates the claimed process improvements and program benefits. This data supports the conclusion that the C-5 Program process—when compared to the C-5 AMP—detected defects earlier, had about half of the total number of defects, and (on average) corrected the defects twice as fast.

The C-5M Program will continue new development releases for the next several years. The C-5 AMP, now in sustainment, plans to apply the improved processes successfully applied by the C-5M Program. Investigations by the C-5 AMP suggest that large or complex upgrades can be developed and maintained more cost-effectively using the improved processes described in this article.

Process Overview

The C-5M project operates as a physically co-located integrated product team. Figure 1 provides a conceptual overview of the roles and flow of the artifacts that ultimately result in the target software; it also represents both the traditional process steps and roles (top of the figure), and modeling extensions (bottom of the figure) used to develop the C-5M software. The system engineer develops textual requirements as well as any other type of analytical model that is captured in a software requirement specification (SRS). The SRS requirements are developed

Figure 1: *Process Roles and Flow*



using a requirement specification language (RSL) that has a structured syntax and restricted set of verbs (e.g., acquire, validate, provide, and derive). The RSL was developed to complement the SCR modeling method. In parallel, there is a continuous requirement flow-down process. The lead software architect identifies the components of the software architecture and works with the software requirements modelers to formalize the requirements and associated interfaces into models.

The software requirements modeler develops requirement models from the SRS and interface control document (ICD) using a modeling tool that supports the SCR method. Models capture behavioral requirement and interface information (e.g., inputs, outputs, types, and ranges) extracted from an ICD. The modeling process often identifies requirement or interface problems that must be resolved through interaction between the system engineer or software architect. For example, interface specifications were captured in a database that is shared by the project team, including subcontractors, but they were not always complete or consistent during the early part of the program (e.g., the first 100 days). The requirement modeling process and associated tools force the interface information to be complete and consistent. Additional problems or anomalies are identified by the system engineers through requirement simulation of the models. Validated requirement models are linked to the software design document (SDD). The designers and implementers work directly from the SDD, requirement models, and interfaces to implement the code. These modeling-related extensions to the process help to improve the overall performance of the team. Better requirements and interface documentation allow software designers to focus on the detailed design and implementation of the code rather than chasing requirement issues or making assumptions that can result in costly rework.

Interface-Driven Requirement Modeling

SCR is a table-based modeling method that has been effective and easy to learn for most engineers [2]. The SCR modeling language has a well-defined syntax and semantics allowing for a precise and tool-analyzable specification of the required behavior. Models represent the required functionality of a component using tables to relate monitored variables (inputs) to controlled variables (outputs), as reflected in Figure 1. There are three basic types of tables: 1) mode transition tables, 2) event tables, and 3) condition tables. A *mode transition table* is a state machine, where related system states are called system modes, and the transitions of the state machine are characterized by events. An *event* occurs when any system entity changes value. A *condition* is a predicate characterizing a system state. A *term* is any function defined in terms of input variables, modes, or other terms. The SCR tables can be combined to specify complex relationships between monitored and controlled variables using mode or terms variables. This allows common conditions, events, and modes to be defined once and referenced multiple times.

"Better requirements and interface documentation allow software designers to focus on the detailed design and implementation of the code rather than chasing requirement issues or making assumptions that can result in costly rework."

Model developers should employ a goal-oriented approach and work backwards to specify functions and constraints for each output (controlled variable or term) of the component, using the following general guidelines:

- Create a table that assigns each computed value for the table output. The value can be specified using a simple assignment or complex function. This corresponds with the RSL action verb *provide*.
- Use a condition table to describe relationships between an output (or term) if the relationships are continuous over time. Identify all conditions that must be TRUE for each output assignment. Conditions relate to the RSL verb *validate*, because they are constraints on the inputs.
- Use an event table to describe relationships between an output (or term) if the relationships are defined at a specific point in time. Define the events and optional guard conditions that trigger the event for each output assignment.
- Use a mode transition table to describe relationships between an object if the relationship for a mode is defined for a specific interval of time (set of related system states). Identify the set of modes and define the event associated with each source-to-destination transition of the model transition table.
- If there are common conditions that are related to constraints (i.e., conditions or events) of functions of two or more outputs (or terms), then define a term table that can be referenced by other tables. Term reference can be performed in a table assignment, condition, or event. Term variables correspond to the RSL verb *derive*, as terms are intermediate variables that can be referenced by other tables.

See [3] for more details on the interface-driven modeling approach.

Requirement Validation

The modeling process forces the customer requirements to be translated into a language understood by the engineers. This formalization ensures common understanding between the system requirements engineer, lead software architect, and requirements modelers across the system and software interface boundary. Despite this, the system engineers responsible for developing textual requirements were initially reluctant to use the requirement simulator that was specifically designed to support the SCR method. However, after developing a few scenarios, they realized that their understanding of the requirements could be incomplete or incorrect, often due to the complexity of the system. This revised perspective was necessary since they had the domain knowledge to judge the correctness of the modeled requirements, making their input into the validation process critical.

The typical process for validating a scenario requires the system engineer to enter input values associated with a sequence of events using the GUI simulator. After each input, the system engineering observes and validates system outputs and internal states (e.g., terms). Model issues are exposed when sequences of events do not result in outputs that the system engineer expects for a particular scenario. The most common types of problems identified through requirement validation result from inconsistencies in related model condition, mode, or event tables. Similar inconsistencies exist within textual requirements specifications, but the inconsistencies are often difficult to identify through manual inspection and reviews as the related requirement can often be separated in an SRS document by tens or hundreds of pages. Requirements models formally link the related requirements (as reflected by Figure 1), and tools can help detect inconsistencies or issues (e.g., logical contradictions, potential divide-by-zero situation) through simulation and automated analysis.

Measurement Data

The C-5M avionics software was developed in incremental blocks with each release of a software block occurring on-schedule and with full functionality. These fully functional, on-time software releases provide evidence justifying the improved process. However, IPR-related measurement data tracked on both the C-5M Program and the C-5 AMP provides an objective way to compare the program processes. Ideally, a program should have objectives such as to: 1) minimize the total number of IPRs, 2) detect IPR-related defects early, and 3) reduce the time to correct any IPR-related defect. The following IPR measurement data provides comparative data that relates to the stated objectives. Four base measures and one derived measure are used to substantiate the process improvements of the C-5M Program:

- Base measure: Number of days since first program IPR (i.e., referred to as program start).
- Base measure: Number of IPRs.
- Base measure: Date the IPR opened.
- Base measure: Date the IPR closed.
- Derived measure: Average days to approval = Date the IPR closed - Date the IPR opened.

The C-5 measurement analyst noted that a C-5M IPR was often more detailed and specific than a C-5 AMP IPR, due to the more detailed and precise requirements defined for C-5M using the improved processes. For example, a C-5M IPR might state very specific details such as:

```
the color of the pressure
readout should be red, not
yellow, when the pressure
exceeds the pressure
threshold limit.
```

For the C-5 AMP, a similar IPR might state:

```
the warning messages have
invalid color settings.
```

As a result, the IPRs on C-5 AMP often had a broader scope relative to C-5M IPRs and required more time to investigate and implement the necessary corrective action, sometimes to the point where a single C-5 AMP IPR was equivalent in scope to two or more C-5M IPRs.

The next section provides measurement data that supports this claim. For example, the number of days to approve an IPR for the C-5 AMP was at some times almost three times longer than the number of days for an IPR-approval for the C-5M Program. Note that the numbers of Total IPRs and Days To Approve IPRs are not included in the measurement data in order to protect program-privileged information.

"The modeling process forces the customer requirements to be translated into a language understood by the engineers. This formalization ensures common understanding ... across the system and software interface boundary."

Measurement Analysis

Figures 2, 3, and 4 (on the following page) compare C-5M data against C-5 AMP measurement data in 100-day increments from the start of each respective program[1]. Figure 2 shows the total number of accumulated IPRs. The number of IPRs for the C-5M Program was slightly higher than the C-5 AMP through the first 400 days of the program. The number of C-5 AMP IPRs increases significantly at about the 500th day of the program; by the 800th day, the number of C-5 AMP IPRs is about double the number of C-5M IPRs. Figure 2 shows a linear prediction that the expected number of IPRs for the C-5M Program over the next 200 days will total about a third of the number of IPRs for the C-5 AMP; this prediction is substantiated by the measurement data shown in Figure 3.

Figure 3 shows the number of IPRs added during each 100-day period (e.g., IPRs from 300th day to the 400th day). This view of the data shows that more IPRs were detected early in the C-5M Program. This suggests that requirement modeling and validation helped to detect defects early. Figure 3 also shows that starting at day 500, the IPRs for the C-5 AMP increased significantly. The number of C-5 AMP IPRs was nearly double the number of IPRs for the C-5M Program at the same point in time. The C-5M IPRs discovery rate continues to remain significantly lower than the C-5 AMP defects from day 500 through day 800.

Figure 4 provides data on the number of days required to approve a system change that corrects an IPR-related defect. The IPR approval process for the C-5M Program was longer at the beginning. The process defined in this article was not completely refined during C-5M's first 100 days, and additional improvements were being put into place during the early days of program execution. However, the data indicates that the average approval period continued to decline through day 800 of the program. The combined data indicates that at 800 days, the number of detected IPRs by the C-5M Program is less but the time to correct the defect is significantly shorter than that for the C-5 AMP. The combined data supports the conclusion that the requirement modeling and validation provides a significant improvement in early defect identification, faster defect removal, and correction.

Leveraging Models

Approximately 90 percent of the detailed software design descriptions rely on the requirement models. The requirement model is linked to the SDD rather than having the design specified in-text. Models represent both high-level and low-level requirements (i.e., derived requirements). Unusual or complex designs are documented in the SDD using text, flow diagrams, or other engineering drawings (as needed). This is another process efficiency gained through leveraging the requirement modeling process. The model provided a formal, precise statement of the requirements that could be referenced directly in the SDD.

Common modeling patterns—such as data retrieval, data validation, and filtering—were identified and evolved for the different software components. The system

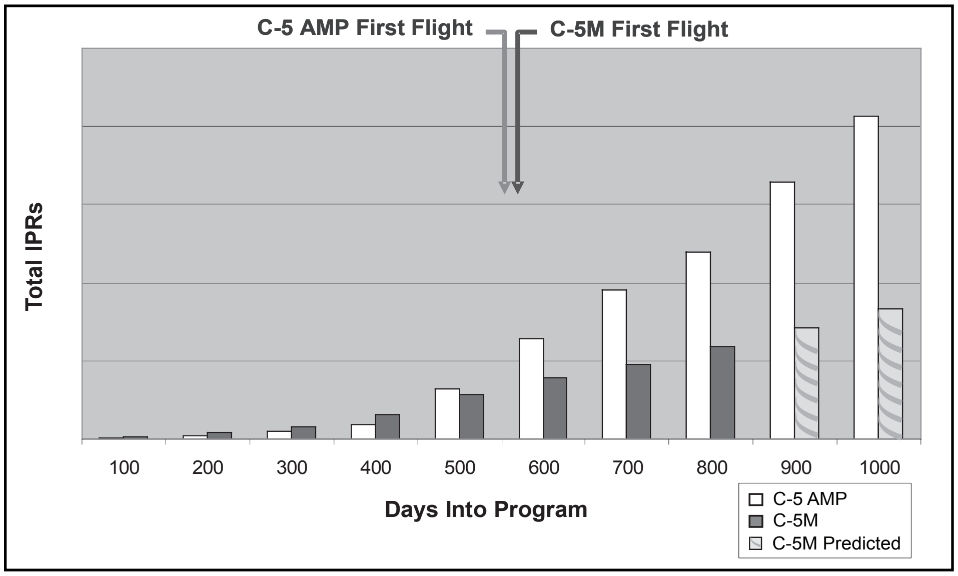


Figure 2: *IPRs vs. Days Into Program*

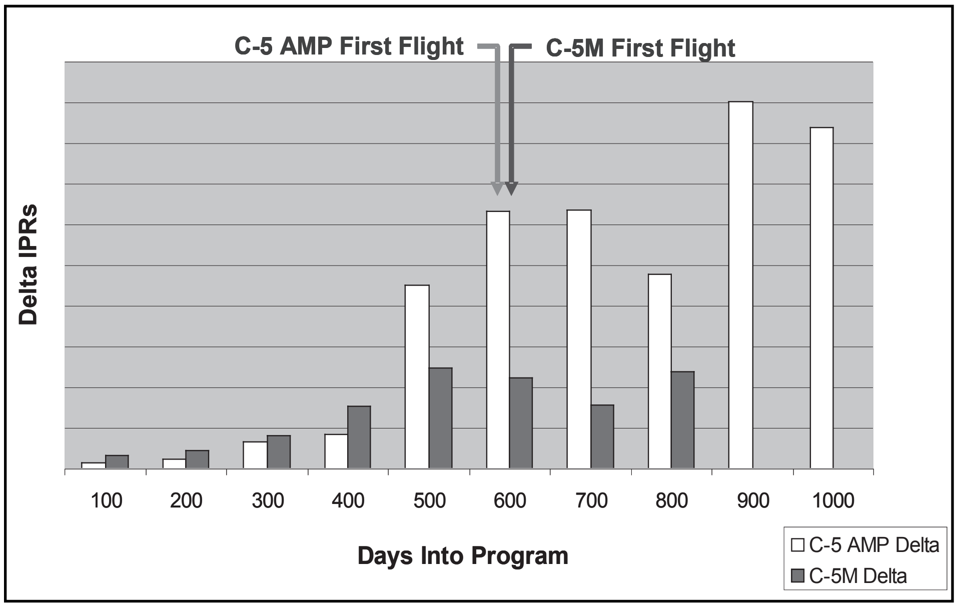


Figure 3: *Delta IPRs vs. Days Into Program*

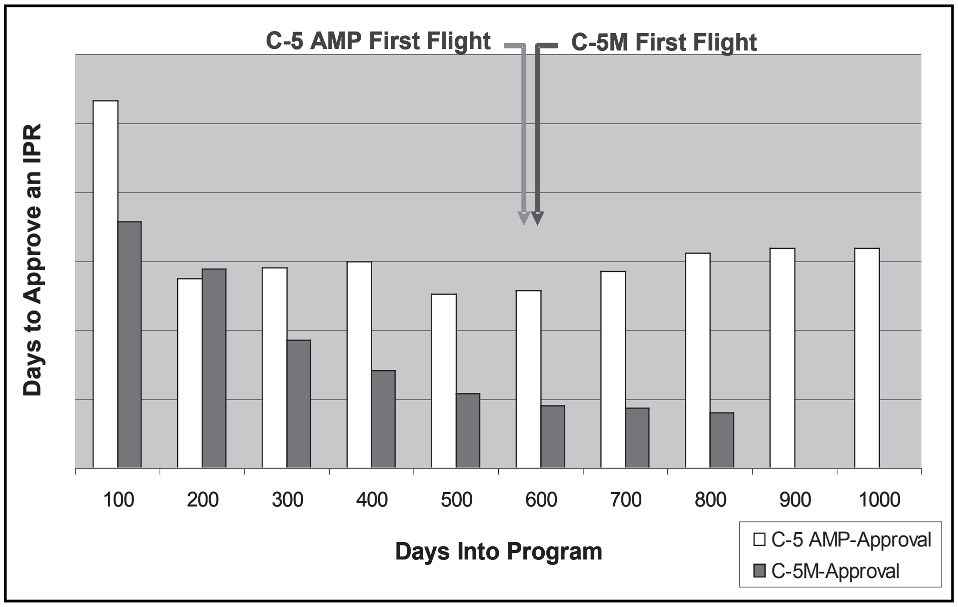


Figure 4: *Average Days to IPR Approval*

engineer and lead software modeler validated the patterns that were captured as model templates. The novice team members began development of requirement models using model templates. The model templates helped to promote additional consistency into the modeling process.

Summary

This article describes the approach and benefits derived through the use of requirement modeling for the C-5M Program. Requirement modeling helped to develop better requirements and interface information to support the design and implementation process. The systems engineers used requirement simulations of the models to validate the correctness and consistency of the requirements. The requirement modeling and simulation processes uncovered a large number of requirement defects prior to software implementation. In addition, measurement data substantiates the claimed process improvements and program benefits. Measurement data supports the conclusion that the C-5M Program process detected defects earlier, had about one-half of the total number of defects, and on average corrected the defects twice as fast as the C-5 AMP. Also described were the substantial benefits seen by the Lockheed Martin Aeronautics Company and its customer from early validation of the systems and software requirements. The significant benefits realized on the C-5M Program have resulted in plans to incorporate this requirement modeling process into the C-5 AMP sustainment efforts for large and complex software-system upgrades.◆

References

1. Faulk, Stuart, et al. Experience Applying the CoRE Method to the Lockheed C130J. Proc. of the Ninth Annual Conference on Computer Assurance, IEEE 94CH34157. Gaithersburg, MD, June 1994: 3-8.
2. Kelly, V., et al. Requirements Testability and Test Automation. Proc. of the Lockheed Martin Joint Symposium. June 2001.
3. Blackburn, Mark R., Robert D. Busser, and Aaron M. Nauman. "Interface-Driven, Model-Based Test Automation." CrossTalk May 2003 <www.stsc.hill.af.mil/crosstalk/2003/05/blackburn.html>.

Note

1. At the request of Lockheed Martin management, the horizontal line values for Figures 2, 3, and 4 have been removed.

About the Authors

Steven D. Allen is a staff engineer with the Lockheed Martin Aeronautics Company. He has more than 24 years of experience in the analysis, design, and development of software for integrated software subsystems. In his role over the last few years as a requirements engineer for the mission processing subsystem of the C-5M Program, Allen has been active in the process, procedures, and use of current software tools to clearly define and validate the software system requirements through the use of requirement modeling, simulation, and verification techniques.

Mark B. Hall is a senior staff engineer with the Lockheed Martin Aeronautics Company. He has more than 20 years of experience in the analysis, design, and development of software for integrated avionics. As the software architect for the mission processing subsystem of the C-5M Program, Hall has been active in process and procedures to clearly define the software requirements and validate them through the use of modeling and simulation. He has a bachelor's degree electrical engineering from Southern Polytechnic State University and an MBA from Kennesaw State University.

Mark D. Mansfield is a staff engineer with the Lockheed Martin Aeronautics Company. He has more than 10 years of experience in the analysis, design, and development of software for integrated avionics. In his role over the last few years as the systems engineer for the mission processing subsystem of the C-5M Program, Mansfield has been active in the process and procedures to clearly define the system and software requirements and to validate them through the use of modeling and simulation. He has a bachelor's degree in aeronautics from Embry-Riddle Aeronautical University.

Verlin Kelly is a staff specialist for the Lockheed Martin Aeronautics Company and has 41 years of experience in embedded systems and software and automated support systems. He has developed software engineering training curriculum and courses as well as co-authored presentations to the Systems and Software Technology Conference on "Quantitatively Managing Multi-Company Software Teams" and "Requirement Testability and Test Automation." Kelly has a master's degree in operational mathematics from the University of Texas at Arlington (UTA) and a bachelor's degree in physics and mathematics from Baylor University. He is involved in system/software Unified Modeling Language development and automated testing and has served on the industry advisory council for the computer science and engineering departments at the UTA and Texas Christian University.

Mark R. Blackburn, Ph.D., is a Systems and Software Consortium fellow and co-inventor of the T-VEC (also known as test-vector) system. He has more than 20 years of software systems engineering experience, spending most of his time helping companies adopt model-driven engineering tools and methods. Blackburn is a frequent speaker at conferences and symposia, and has authored more than 70 papers covering a broad spectrum of topics such as modeling, verification, software safety, security, reliability, and measurement. He has a bachelor's degree in mathematics from Arizona State University, a master's degree in mathematics from Florida Atlantic University, and a doctorate in information technology from George Mason University.

Systems and Software Consortium
2214 Rock Hill RD
Herndon, VA 20170
Phone: (703) 742-7136
Fax: (703) 742-7350
E-mail: blackburn@software.org

Coming Events

March 31-April 2
Spring 2009 Software Test & Performance
San Mateo, CA
www.stpcon.com

April 1-4
2nd International Conference on Software Testing, Verification, and Validation
Denver, CO
http://bitterroot.vancouver.wsu.edu/icst2009

April 6-8
DTIC 2009
Alexandria, VA
www.dtic.mil/dtic/announcements/conference.html

April 6-9
2009 NanoTechnology for Defense Conference
Burlingame, CA
www.usasymposium.com/nano/default.htm

April 20-23
21st Annual Systems and Software Technology Conference
Salt Lake City, UT
www.sstc-online.org

April 20-24
DISA Customer Partnership Conference
Anaheim, CA
www.afcea.org/events/disa/landing.asp

Coming Events: **Please submit coming events that are of interest to our readers at least 90 days before registration. E-mail announcements to: nicole.kentta@hill.af.mil.**

Defense Acquisition Performance: Could Some Agility Help?

Paul E. McMahon
PEM Systems

The problems with the Waterfall Model and the challenges involved with evolutionary acquisition of defense systems have been well-chronicled. This article describes how Agile techniques can be applied to address these challenges under today's defense acquisition regulations. The article employs a previously published hypothetical space system acquisition case study, along with a case study of a legacy modernization project that employed Agile techniques to aid the acquisition process.

In the May 2007 CrossTalk article, "Defense Acquisition Performance Assessment – The Life-Cycle Perspective of Selected Recommendations," Dr. Peter Hantos analyzes the *conceptual integrity* of recent defense acquisition recommendations made in a Defense Acquisition Performance Assessment (DAPA) report. Hantos states that:

> The acquisition life-cycle models of the DoD/National Security Space Acquisition Policy 03-01 policies are inherently Waterfall, and as such, inadequate for the acquisition of large-scale, software-intensive systems even if they are used with the intent of Evolutionary Acquisition. [1]

Much has been written over the past few years about the problems with the Waterfall Model, so the issues raised by Hantos are not surprising. Through the use of a hypothetical space system acquisition case study employing an evolutionary acquisition strategy, the following three key obstacles are identified and discussed in the primary sections of this article:

1. Funding for Risk Mitigation Alternatives.
2. Deferring Non-Key Performance Parameter Requirements.
3. The Inability to Define Measures of Technology Readiness.

Over the past few years, Agile practices have also received attention mostly on small commercial projects in the software arena, although interest has recently been growing in the defense industry and on larger efforts [2]. This article describes how Agile techniques can help with the evolutionary acquisition of defense systems. In particular, this article explains how an Agile approach could be used to effectively tackle each of the three obstacles identified in the referenced article.

1. Funding for Risk Mitigation Alternatives

Traditional risk management approaches identify, categorize, and prioritize risks, along with establishing risk mitigation approaches. If the risk is high enough, alternatives may be pursued through distinct parallel activities (used as a backup in case the current course fails). Often, however, risk mitigation is handled through the normal engineering management activities without the need for added funding for alternatives.

Hantos also states that:

> Having risk mitigation plans in the conventional sense is different from spiral planning. It involves the creation of additional plans to eliminate or gradually reduce the risk by having alternative course(s) of action lined up in case the risk materializes or its likelihood drastically increases. A key element of such risk planning is that funding for alternative actions needs to be provided in addition to the allocated, regular cost of development. [1]

How Do Agile Teams Mitigate Risk?

Successful Agile teams that I have observed deal with risk mitigation differently [2]. Rather than expend valuable resources on alternative approaches that may never get used, resources are focused on breaking the known problem down into manageable chunks and tackling each to systematically gain knowledge that reduces uncertainty. With the new knowledge gained, course corrections are acted upon in a timely fashion.

Key to Agile planning is not setting the plan in stone for the subsequent increment prior to assessing the new knowledge gained from the previous increment. Agile approaches provide risk mitigation by reducing the risk through a series of small course corrections rather than spending extra resources on distinct alternatives that may never be employed. The Agile approach does, however, require an understanding of how to continuously identify the current high-priority work that needs to be done now based on the latest knowledge of the risk, and what work to *defer* given the current available resources [3]. This leads to the second obstacle.

2. Deferring Non-Key Performance Parameter Requirements

In the case study described in [1], difficulties encountered when attempting to defer work on a space system legacy modernization project are analyzed:

> Allowing program managers to defer non-key performance parameter requirements to later upgrades is an attractive proposition from the program manager's view. It provides an effective risk management tool by greatly expanding their decision-making authority and flexibility. In the context of our case study, how could the program manager using this newly acquired freedom reduce the scope of the first acquisition increment? Unfortunately, analysis shows there are not many opportunities after all. [1]

The challenges encountered when planning and executing the modernization of a critical legacy application are not insignificant. For example, Hantos alludes to the challenges related to convincing an end-user to incrementally accept partially modernized functionality when they need all of their current legacy functionality to do their job today. This leads to a question: Does the evolutionary acquisition concept fall apart when applied to modernizing tightly coupled critical legacy systems?

Deferring work is fundamental to Agile risk management practices. But the decision to defer work—and deciding what work to defer—isn't made by the program manager alone. As the following different case study demonstrates, the key value of work deferment isn't in providing "freedom" to the program manager to "reduce the scope" of any particular increment of the project.

A Different Case Study Using Agile Practices to Modernize a Business-Critical Legacy System

Over the past few years, I was on a team that helped a client modernize a legacy system that is critical to their business. Using Agile practices, we attempted to plan an incremental approach to deploy the modernized functionality.

We started with the user interface. It was a chunk that could be broken off and integrated with the remaining legacy applications supporting incremental deployment. Some argued against this approach because the user interface was not viewed as the highest technical risk. With Agile practices, we attack high risks first—but technical risks are not necessarily the highest risks. In fact, we soon discovered that end-user buy-in was a higher risk, especially when modernizing a legacy system that is critical to an organization's daily functions and has been in place for 40 years. We also discovered that by providing a modern user interface first—and by not only demonstrating this capability early, but also training end-users early and incrementally while listening intently to their feedback—we were able to build key relationships and a key support group that became crucial to keeping the project on firm ground and moving forward.

On the legacy modernization case study, similar to the space system case study article, the majority of the legacy application subsystems were too tightly coupled to be deployed in separate increments. But we also knew that due to the criticality of the full system (to the business) that it needed to be fully tested and accepted by the end-users prior to deployment. To meet this challenge, it was decided to continue with an evolutionary life cycle but deploy it incrementally to a lab environment where the new system could be tested and demonstrated, and where end-users could both provide feedback to developers and be given training.

Each increment of work culminated in a week-long lab session where previously agreed-to user scenarios were demonstrated and end-users were given hands-on operational training. It was during the hands-on lab time when end-users provided their most valuable feedback to the development team. These collaborative sessions proved invaluable as end-users became familiar with features that would simplify their current job, while also finding *acceptable alternative approaches* to non-key requirements that had been *deferred* to later increments. It is important to note that by providing this *incremental training* before all requirements had been implemented that many deferred *non-key requirements* were mutually agreed to be *no longer needed.*

As the end-user's knowledge and confidence with the new system grew through each lab session, they became the most powerful voice and driving factor in the eventual full acceptance and deployment of the modernized system. Formal training was deployed to all end-users prior to the system go-live date, and an aggressive maintenance support program was put in place that included an around-the-clock help desk and on-site, immediate technical support.

"As the end-user's knowledge and confidence with the new system grew ... they became the most powerful voice and driving factor in the eventual full acceptance and deployment of the modernized system."

The legacy modernization case study was similar to the hypothetical space system study in that there were not many opportunities to deploy *incrementally to-the-field*, but this fact did not reduce the importance of employing an incremental evolutionary life cycle. This approach was critical to our ultimate success. Also critical was the strategy of integrating end-users into the engineering cycle early where their voice, along with the voice of the development team, became a driving force in decisions on what work to defer and what work to pull forward. These decisions were coordinated through the program manager.

Also critical to the project's success was management's commitment of the time required by end-users to fully participate in the multiple lab sessions. The time it takes to learn a new system cannot be underestimated nor its importance from the end-user buy-in perspective. This may well have been the most important risk-mitigation strategy employed. Without the strong commitment of the end-users, who truly felt they were a part of the project development team, this effort could not have succeeded.

It is worth noting that while the legacy modernization case study described may not technically fit the definition of evolutionary acquisition, it demonstrates the criticality of an *evolutionary development strategy* (that includes integrated end-user collaboration) to the success of tightly coupled legacy modernization projects.

A Key Value of Requirements Deferral

It is also important to note that a key value in deferring work when using Agile practices is to provide the opportunity to the development team to maximize value to the end-users, rather than providing the program manager with freedom to reduce incremental scope.

This key value is often misunderstood. Some view the primary value of requirements deferral as a method to allow program managers to defer facing difficult cost and schedule challenges. Actually, its greatest value is the opposite. Requirements deferral, when used appropriately, can help address critical risks sooner (e.g., the risk of end-user buy-in) while at the same time increasing the opportunity to meet user needs more cost-effectively through alternative approaches (as shown in the legacy modernization case study). Nevertheless, applying requirements deferral appropriately can be tricky, as the following examples demonstrate.

An example of an appropriate use of requirements deferral is to free up the needed human resources from a less important task, pulling a more valuable task forward in the schedule to help in demonstrating a key system feature sooner to an end-user. When this is done appropriately, earned value (EV) and schedule commitments are maintained since we are trading the value of one task for another task of equal or greater importance. The overall project risk is also reduced.

With respect to the space system study, Hantos states:

> There are other considerations that would make the deferral of requirements difficult. For example, complex graphics and elaborate

> display designs are important in any ground system. As a requirements-pacing strategy, one might consider releasing the first version of the ground software with simplified user interfaces. This is an effective engineering approach, but it may backfire with end-users of the system. In similar situations, satellite operators forced to work with intermediate systems having limited capabilities created resentment and blocked buy-in when the final system became available. [1]

If the lab approach (as discussed in the legacy modernization case study) was used in the space system case study, this resentment might have been avoided since the operators would not be forced to use the limited capabilities in an operational environment. By training the operators early in the lab environment, there is a good possibility they could find alternative approaches to non-key deferred requirements—as was the case in our legacy modernization project. The feedback from the operators could also lead to decisions to pull critical display requirements forward, while deferring less critical user interface functions and thereby aiding overall end-user buy-in.

But what if you are already behind schedule and don't have the resources to pull any work forward in the schedule? In this situation, it can be appropriate to defer requirements to meet a critical milestone. Nevertheless, it is important to understand when using this technique that there is a difference between meeting a milestone and being on schedule from an EV perspective.

For example, it is inappropriate to defer requirements and then take EV for work that was never completed, thereby creating the false impression of being on schedule from an EV perspective. When less work is accomplished than was planned, you are behind schedule from an EV perspective. Unfortunately, too often today, program managers inappropriately utilize requirements deferral in this manner to mask the real project status.

Decisions related to pushing work out and pulling other work in should be made by key knowledgeable technical team members together with management and the customer. These decisions should be made at the start of each increment to ensure the best decision is made with the best available information and to ensure that all stakeholders understand and are on board with the rationale for a requirement deferral decision.

3. The Inability to Define Measures of Technology Readiness

According to Hantos, the DAPA report's findings state that:

> ... there are no clearly definable measures of technology readiness, and the inability to define and measure technology readiness during Technology Readiness Assessments is the reason that immature technology is incorporated into plans prior to milestone B. [1]

Hantos responds by stating: "On the contrary, numerous sources are available to help with technology readiness assessments" [1]. Hantos also provides a number of sources currently available to support this position.

"By training the operators early in the lab environment, there is a good possibility they could find alternative approaches to non-key deferred requirements—as was the case in our legacy modernization project."

While I don't disagree with the existence of these sources, I believe the underlying intent of the DAPA report was to raise awareness to the lack of effectiveness of these sources in achieving their ultimate goal of providing an accurate assessment of technology readiness.

How Do Successful Agile Teams Measure Technology Readiness?

Agile practices employ a simple approach to technology readiness. The following criteria give a sense for this approach [3, 4]:

- The work associated with the technology can be estimated by those who are going to do the work.
- Those who are going to do the work have established at least one implementation approach.
- The planned work can be partitioned into reasonably small chunks (one to three months).

This approach is not meant to imply that the best estimates come only from practitioners. To the contrary, especially on large efforts; accurate estimates require both a bottom-up and top-down approach. Many Agile development estimation techniques out-of-the-box don't scale well to large projects because they focus primarily on a bottom-up estimation approach, which is insufficient for large complex efforts [4]. However, a hybrid approach composed of *Agile bottom-up* together with a more traditional top-down approach as a cross-check has been proven to work well for large-scale efforts [2, 3].

Some believe that on complex efforts with long development cycles that it is impractical to perform detailed (daily/weekly) cost and schedule estimates. I agree that it doesn't make sense to formally update the project master plan and schedule daily or even weekly on large complex efforts. It has been my experience, however, that the most successful large-scale efforts manage the work daily at the grassroots level and continuously update their estimates using informal daily or weekly meetings and team task lists [2]. With the successful projects, these meetings are short and directly involve the people who are actually doing the real work—with management primarily in a listening and supporting role.

It is important to recognize that my recommendations are not meant to imply that we should wait until a technology is mature, or within months of being mature, before incorporating it. This would lead us back to a Waterfall approach where we need to know everything before beginning. This is why I recommend a hybrid Agile-traditional approach for large complex efforts.

Unfortunately, I am finding that many are missing the fact that Agile approaches are not in conflict with high-risk technologies. In fact, they are best suited to address high risks because, when applied appropriately, Agile approaches drive the true cost and schedule to the surface sooner.

Some may also believe this approach is too simplistic for the complex work involved with space systems. But does the intent of measuring technology readiness change based on complexity? If its intent relates to establishing a confidence factor in hitting cost and schedule targets when implementing the technology, then I believe the right balance of Agile practices is the best choice.

The technology readiness criteria used on Agile projects go beyond just the need for granular work packages. Those who are going to do the work must have an approach and must have completed adequate *analysis* to commit to the *collaboratively agreed to* cost and schedule targets to complete that work.

To accurately estimate effort, regardless of complexity, the problem needs to be broken down into manageable chunks that can be estimated by those who are actually going to do the work—not their managers. Agile practices utilize the knowledge that people are the most important factor in hitting cost and schedule commitments, and these practices recognize that different people perform in different ways.

By requiring that at least one implementation approach be known, I am saying that sufficient *analysis* needs to have been completed during the initial planning stage to ensure that the problem has been thought through enough to see a potential reasonable solution.

On large complex efforts, however, bottom-up and top-down estimates rarely get to the same point. This is not necessarily bad. Having a *challenge* schedule can be good to help stretch the team's productivity. But when the difference between the published schedule and the developer's real work gets too wide, this strategy can backfire, leading to the wrong decisions by the wrong people at the wrong time.

Final Thoughts

We should not wait to have all the answers before utilizing new technologies. On the other hand, experiences on both Agile and non-Agile efforts of the past indicate that the lack of adequate analysis may lie at the root of immature technologies being prematurely incorporated into project plans [2]. It is an appropriate balance we seek.

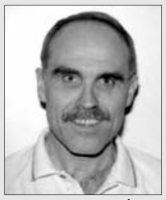
Contractors have more options today under the current defense acquisition regulations than many realize. Agile methods can be successfully used in DoD acquisitions today; but to succeed, it is important to have management buy-in and an understanding of the implications of using the right balance of Agile techniques on both the government and contractor side.

Agile approaches will not solve all the current acquisition challenges faced today, nor will they make a complex problem simple. However, the greatest benefit of Agile approaches may not be in solving small, well-understood problems, but in helping solve those that are least understood. This is because they help us avoid making decisions too early (e.g., leaning toward the Waterfall approach) and too late (e.g., deferring the wrong work for the wrong reasons). Agile practices—balanced appropriately with proven traditional practices—can help us understand sooner how big a problem is, and can help us leverage today's available options and human resources more effectively, regardless of the problem's complexity.◆

References

1. Hantos, Peter. "Defense Acquisition Performance Assessment – The Life-Cycle Perspective of Selected Recommendations." CrossTalk May 2007.
2. McMahon, Paul E. "Lessons Learned Using Agile Methods on Large Defense Contracts." CrossTalk May 2006.
3. McMahon, Paul E. "Are Management Basics Affected When Using Agile Methods?" CrossTalk Nov. 2006.
4. Cohn, Mike. Agile Estimating and Planning. Pearson Education, Inc., 2006.

About the Author



Paul E. McMahon, principal of PEM Systems, helps large and small organizations as they move toward increased agility and process maturity. He has taught software engineering, conducted workshops on engineering process and management, is a frequent speaker at industry conferences, and is a certified ScrumMaster. McMahon has published articles on Agile development and lessons using the CMMI® framework as well as the book, "Virtual Project Management: Software Solutions for Today and the Future." He has more than 25 years of engineering and management experience and 10 years of independent consulting experience.

PEM Systems
118 Matthews ST
Binghamton, NY 13905
Phone: (607) 798-7740
E-mail: pemcmahon@acm.org

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

A Model to Quantify the Return On Information Assurance

Ron Greenfield and Dr. Charley Tichenor
Defense Security Cooperation Agency

Forecasting—and subsequently measuring—a program's financial return is an indicator of how well it supports its parent organization's strategic plan. This can help prioritize investments and help forecast and subsequently measure an individual's or team's job performance. This article presents a model to either forecast the financial Return on Information Assurance (ROIA) for Information Assurance (IA) countermeasure(s), or measure the financial impact of actual costs and the benefits of their use[1].

This article explains and demonstrates the structure of a model for forecasting, and subsequently measuring, the ROIA, or the ROIA model[2]. This includes IA initiatives such as firewalls, antispyware software, antivirus software, etc. Also, it can be used to determine the actual return of those countermeasures at the end of a given time period. Organizations are encouraged to either use this structure *as is* or modify it, and then populate it with their local variables[3].

Review of the Related Literature

Two important references apply to this research.

The first is the book "The Balanced Scorecard: Translating Strategy Into Action" [1], which measures Return on Investment using four categories:

1. Financial.
2. Customer satisfaction.
3. Improvement of internal processes.
4. Investment in learning and growth.

The currently formulated ROIA model only considers the financial category. This is not to downplay any other facet of IA, such as unintentional disclosure of information, loss of reputation, etc., which locally may be of equal or greater importance. This only means that there is room for future research to improve the ROIA model to address the Return on Investment of non-financial benefits.

The second reference is from Australia, specifically the New South Wales (NSW) Department of Commerce's "Return on Investment for Information Security" model [2]. The ROIA model is based on the NSW approach, although there are particular modifications. For example, Table 1 shows a modified version of the corresponding NSW table[4], and Table 2 is borrowed with little change although it is used somewhat differently here.

Theory

We define the term *return* as a measure of the degree to which a program is beneficial to the organization. Conceptually, it can be calculated as follows:

$$\frac{\$\text{ Benefits}}{\$\text{ Costs}}$$

For example, suppose a program costs $1,000, and brings in $1,500. The financial return could be then calculated as:

$$\frac{\$1{,}500\text{ gain}}{\$1{,}000\text{ cost}}$$

or, 50 percent. All other things being equal, the organization's balance sheet shows an increased bottom line of $500.

Using another example, suppose a program costs $1,000, but instead results in a cost avoidance of $1,500.

The financial return could be then calculated as:

$$\frac{\$1{,}500\text{ cost avoidance}}{\$1{,}000\text{ cost}}$$

or, 50 percent return. All other things being equal, the organization's balance sheet also shows an increased bottom line of $500.

The ROIA model generally views *return* in this second sense, as long as the organization's bottom line improves as measured using the U.S. Federal Accounting Standard Advisory Board's Generally Accepted Accounting Principles.

One IA goal is to either prevent or reduce future incidents of *successful* malicious attacks. Installing countermeasures can help achieve this goal. The ROIA model is currently based on how well the countermeasures reduce the *repair or replace* costs of forecasted future attacks. Countermeasures could include special software, such as antispyware software, security-related hardware, or IA training.

Therefore, we incorporate the following general concepts into the model:

- Current probabilities of successful attacks.
- Costs to repair or replace materiel as a result of successful attacks occurring before countermeasures are installed.
- Costs to repair or replace materiel as a result of successful attacks occurring after countermeasures are installed.
- Costs of countermeasures to prevent or reduce successful future attacks.
- Return on Investment and financial present values.

Table 1: *Probability of Vulnerability. Potential Number of Threats per Individual Computer per Year*

Likelihood	How Often per Individual Computer?	# Occurrences per 365-Day Year per Individual Computer. *At Least —*	Statistical — Mean	Statistical — Distribution
Negligible	Unlikely to occur	0	0.25	Poisson
Very Low	Between 12 and 24 months	0.5	1.42	Poisson
Low	Between 6-12 months	1	1.93	Poisson
Medium	Between 1-6 months	2	7.04	Poisson
High	Between 1 week and 1 month	12	32.00	Poisson
Very High	Between 1 day and one week	52	155.00	Poisson
Extreme	From 1 to 20 per day, or more	365	500.00	Poisson

More specifically:

- The **financial benefits** are defined here as the forecast repair or replace cost *avoidances* due to installation of a countermeasure. Successful attack incidents are reduced.
- The **financial costs** are defined here as the forecast of the costs to procure the *countermeasure*, paid now, plus *the cost of its annual maintenance* that will be paid in the future.

Therefore, the ROIA is modeled as the following ratio:

(Forecast repair or replace cost *before* countermeasures) – (Forecast repair or replace cost *after* countermeasures)
Cost of countermeasures

Also, the actual ROIA is modeled as the following:

(Actual repair or replace cost *before* countermeasures) – (Actual repair or replace cost *after* countermeasures)
Cost of countermeasures

Forecasting Countermeasure Benefits

Let's forecast the ROIA of a hypothetical system needing four countermeasures for four vulnerabilities. Start by asking, "What is the likelihood of a significant spyware attack happening to a single computer that would cause a repair or replacement during a given year?" (which is the first vulnerability). We demonstrate assuming a five-year lifespan and a four percent discount rate for present value calculations[5].

The ROIA model is built into an Excel spreadsheet, with the Crystal Ball Monte Carlo Simulation[6] software added-in. Refer to Table 1 (extracted from the Excel spreadsheet) for a set of further assumptions. As shown in the table, there are seven degrees of attack likelihood, and frequencies are defined. For this demonstration, we forecast that the malware attack has a *Low* chance: happening at least once per year (Occurrences column) but on average 1.93 times per year (Mean column).

Note Figure 1 as we discuss how to compute the 1.93. We think that such malware-successful attacks will arrive at an individual computer in the same random way that cars *arrive* at highway toll booths—a *Poisson* arrival pattern (see Table 1). Crystal Ball requires a rate parameter for the Poisson. This is entered as 1.5, which is halfway between the 1 in Table 1's column 3 for a *Low* and the 2 for the *Medium*. The selected range has a *Low* value of 1 because we defined a *Low* as happening at least once per year. In theory, it could happen infinitely many times, so *plus infinity* is the high value. Given these parameters, Crystal Ball computes the average of this Poisson distribution as 1.93.

After forecasting the average (expected) number of occurrences of successful malware attacks per year, the cost to repair or replace equipment affected by those attacks needs to be forecasted. Table 2 is used as a guideline for assessing the criticality of each attack instance.

With this as a guideline, we forecast the cost to repair or replace on an individual basis for each type of successful attack (see Figure 2). For this demonstration, we model the criticality of a successful malware attack to be *Significant* and model the best-case repair or replace cost situation as \$20. The most likely case is \$150, and the worst case is \$400. This is a triangular distribution, with an average computed by Crystal Ball at \$190.

Table 3 (see next page) recaps this. For vulnerability number 1, the Internet service asset has a vulnerability of significant spyware attacks. It has a *Low* likelihood of happening, but if it happens the criticality is considered *Significant*. This should occur about 1.93 times annually per computer in our system, at an average cost of \$190 to

Criticality	Description
Insignificant	Will have almost no impact if the threat is realized.
Minor	Will have some minor effect on the asset value. Will not require any extra effort to repair or reconfigure the system.
Significant	Will result in some tangible harm, albeit only small and perhaps only noted by a few individuals or agencies. Will require some expenditure of resources to repair (e.g. *political embarrassment*).
Damaging	May cause damage to the reputation of system management, and/or notable loss of confidence in the system's resources or services. Will require expenditure of significant resources to repair.
Serious	May cause extended system outage, and/or loss of connected customers or business confidence. May result in the compromise of large amounts of government information or services.
Grave	May cause the system to be permanently closed, and/or be subsumed by another (secure) environment. May result in complete compromise of government agencies.

Table 2: *Criticality per Instance of Successful Attack*

Figure 1: *Poisson Distribution of Number of Malware Attacks per Year*

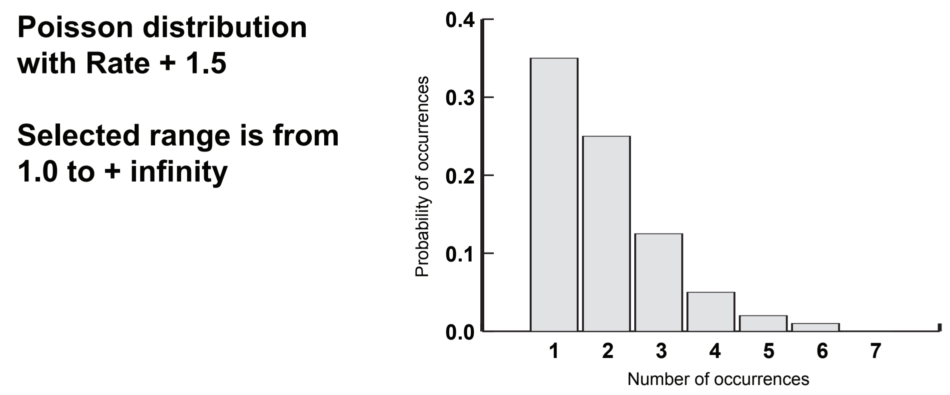
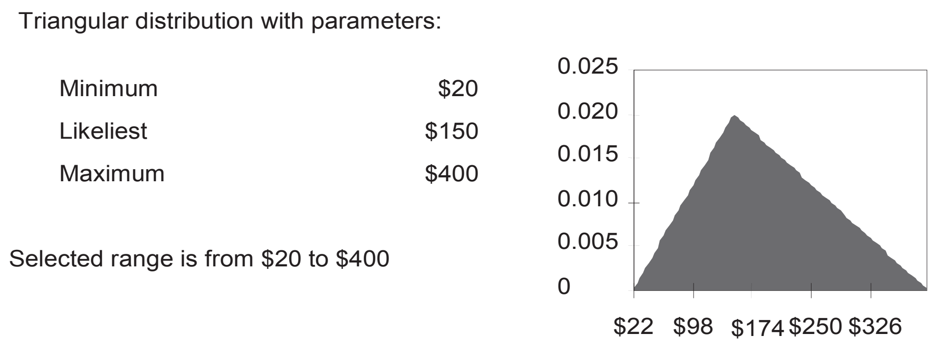


Figure 2: *Forecast Cost to Repair or Replace Due to a Successful Malware Attack*



No.	Asset	Vulnerability	"Before" Likelihood	Criticality	"Before" Number Occurrences per Year per Computer	Direct Cost per Incident	Number Computers	Agency Forecast Vulnerability Costs per Year "Before" Countermeasures Installed
1	Internet service	Significant spyware attack	Low	Significant	1.93	$190	100	$36,670
2	a	aaa	Medium	Insignificant	7.04	$37	100	$26,048
3	b	bbb	Low	Minor	1.93	$103	100	$19,879
4	c	ccc	Very Low	Damaging	1.42	$1,133	100	$160,886
.							**Total "Before" Vulnerability Costs ==>**	**$243,483**

Table 3: *Calculation of Expected Total "Before" Countermeasures' Installation Repair or Replace Cost*[7]

repair or replace the computer. For the 100-computer system, this amounts to an annual forecast average cost to repair or replace of $36,670.

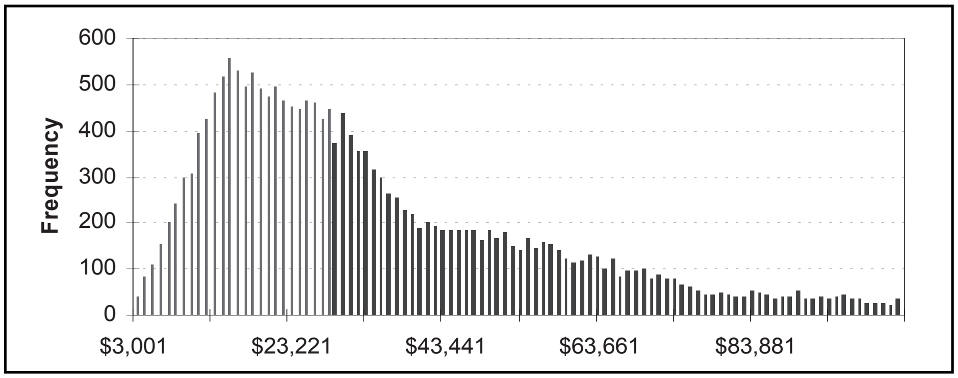
This calculation, however, is deterministic and does not account for the effect of the probability distributions. For example, although the average number of occurrences of successful attacks is 1.93, it could be 1 in a given year, or 2 in another year. Instead of multiplying the 1.93 *before* expected number of occurrences by the $190 direct cost per incident to repair or replace (and then by the 100 computers), we could—to get a better picture of what might actually happen—multiply the *before* occurrences distribution curve by the direct cost per incident distribution curve, and multiply that product by 100.

To forecast the expected cost *before* we buy the countermeasure, Crystal Ball selects a random number from the number of malware attacks probability distribution:

- This random number is converted into the *actual* number of times the threat occurs this year.
- Another random number is selected from the cost to repair or replace probability distribution, and this is converted into the *actual* repair or replace cost.
- These two values are multiplied together, and then multiplied by the number of computers (100).

This is repeated 20,000 times to produce a distribution curve for the annual cost to repair or replace (i.e., a Monte Carlo simulation run for 20,000 trials). Figure 3 shows a histogram plot of the outcomes.

Figure 3: *Forecast Vulnerability Costs for a Malware Attack "Before" Significant Spyware Countermeasure Installation*



The Monte Carlo simulation indicates that the possible annual cost to repair or replace all 100 computers ranges from about $3,000 to $84,000, with an average of about $28,782. This average value is where half of the area of the curve is to its left, and half is to its right, and that point can be read directly through Crystal Ball.

Assume that we now buy a countermeasure. To forecast the average cost to repair or replace *after* we buy the countermeasure, we multiply the cost to repair/replace by the number of times we expect it to occur and by 100 computers, as shown using Table 4.

For vulnerability number 1, the likelihood of a successful spyware attack *after* installation of the first countermeasure is modeled as *Very Low* but, if it happens, the criticality is considered *Significant*. This should occur 1.42 times annually per computer in a system, at an average cost of $190 to repair or replace the computer. For the 100-computer system, this amounts to an annual forecast average cost to repair or replace of $26,980.

As with the *before* costs, we determine the *after* costs distribution for this particular countermeasure using probabilistic methods. Figure 4 shows the *after* costs simulation results, and they are forecast to average $22,581 annually.

Each year's total deterministic benefit is calculated by subtracting its cost *after*

Table 4: *Calculation of Expected Total "After" Countermeasures' Installation Repair or Replace Cost*

No.	"After" Likelihood	Criticality	"After" Number Occurrences per Year per Computer	Direct Cost per Incident	Number Computers	Forecast Vulnerability Costs per Year "After" Countermeasures Installed
1	Very Low	Significant	1.42	$190	100	$26,980
2	Very Low	Insignificant	1.42	$37	100	$5,254
3	Negligible	Minor	0.25	$103	100	$2,575
4	Negligible	Damaging	0.25	$1,133	100	$28,325
					Total "After" Vulnerability Costs ==>	**$63,134**

countermeasures (Table 4, $63,134) from its total cost *before countermeasures* (Table 3, $243,483), or $180,349. Using a deterministic approach, we would multiply these totals by 5 (years) to obtain $901,745. However, using the probabilistic approach with the Monte Carlo simulation (see Figure 5), the average benefit (or cost avoidance) for those 5 years turns out to be $874,837.

Forecasting Countermeasure Costs

We now model the costs of the countermeasures. Here, there are four software countermeasure products installed. Each has an upfront purchase price cost, and each has annual maintenance. Refer to Table 5: Let's assume that these countermeasures will be *good* for five years each (this year and four subsequent years). The lower right corner cell is the sum of the five-year life span costs, or $98,200. This is known with certainty (by contract) and is not simulated.

Calculating the ROIA

The ROIA is now calculated by simulation. It is:

$$\frac{\text{(\$ Benefits Curve [Figure 5])}}{\text{(5 years of countermeasures costs)}}$$

The Figure 6 simulation (see next page) shows that it is possible that this program's ROIA could range from about -600 to about 1,900 percent. However, the *expected* ROIA in this *notional* example is 886 percent, and we are about 93 percent sure that the ROIA will be greater than 100 percent.

Net Present Value Calculation

The five-year ROIA forecast can be expressed in terms of *net present value*, which is an approach to comparing the worths of several alternate ways of allocating money.

For example, suppose that a person has $100 dollars. Let's look at two options on what they could do with that money:

- Option 1 might be to just put the money in their wallet; that allocation option has a *present value* of $100 because they could spend the $100 today.
- Option 2 might be to put the money in the bank, say, for one year at an interest rate of 4 percent; after one year, the investment would be worth $104. The money having a present value of $100 has an associated future value of $104. Which option has the most (financial) worth to this person? A financial analyst will say that the first option represents $100 of spending power today. Also, although the second option has $104 of spending power next year, by reverse engineering, the investment that $104 also represents, in theory, is $100 of spending power today. So the financial analyst will say that both ways of allocating money have the same purchasing power today. They both have the same net present value.

The ROIA model examines several financial allocations placed at different times in a five-year IA program. The theoretical purchasing power of those allocations today are calculated using net present value. That way the worth of these allocations can be forecast in advance. Or, after the five years are over and the actual results are known, then the actually realized net present value can be calculated.

For this simulation (shown in Figure 7, next page), the forecast net present value of this five-year IA program is $776,946.

Conclusions and Areas for Future Research

A quantitative forecast of an IA program's value is important to an organization. This model's basic paradigm is that at least a part of the financial ROIA can be quantitatively forecast as a measure of the effectiveness of countermeasures to possible system attacks. This can be formulated as the ratio of future cost avoidances due to those countermeasures to the cost of those countermeasures. This requires using probabilities of current and future successful attacks, costs of countermeasures to prevent or reduce future attacks, probable costs incurred as a result of successful attacks, and Monte Carlo simulations to obtain a distribution of forecast outcomes. The net present value of the IA

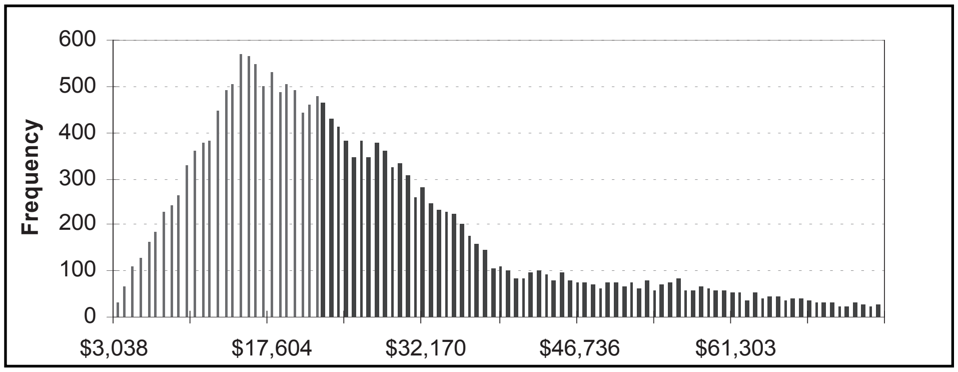


Figure 4: *Forecast Vulnerability Costs for a Malware Attack "After" Significant Spyware Countermeasure Installation*

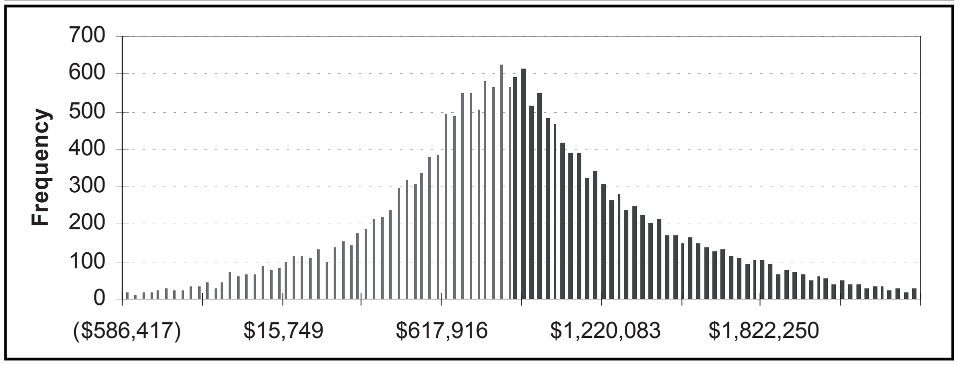


Figure 5: *Forecast Average Cost Avoidance for all Forecast Attacks "After" Countermeasures' Installations*

Table 5: *Actual Countermeasure Costs*

Counter Measures	Upfront Cost per Countermeasure	Recurring Annual Cost per Countermeasure Years 2 thru 5	Total Countermeasure Costs
Install anti-spyware software	$6,000	$600	$8,400
aaa	$20,000	$2,000	$28,000
bbb	$15,000	$1,500	$21,000
ccc	$10,000	$7,700	$40,800
	$51,000	**$11,800**	**$98,200**

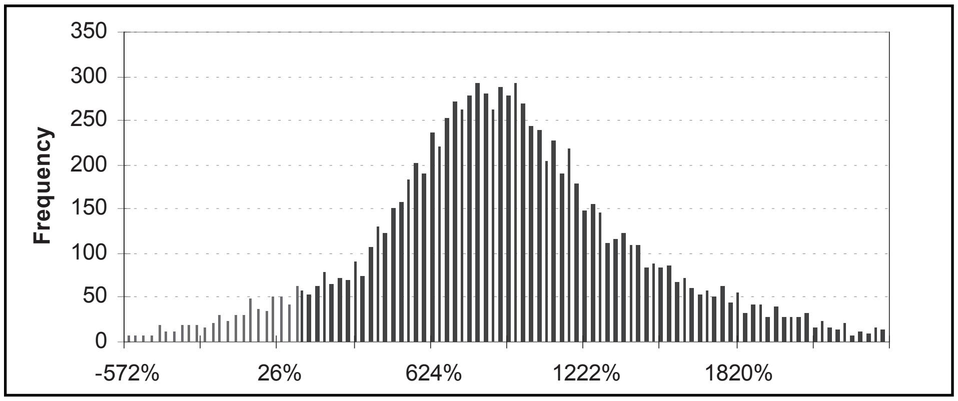


Figure 6: *Forecast Five-Year ROIA*

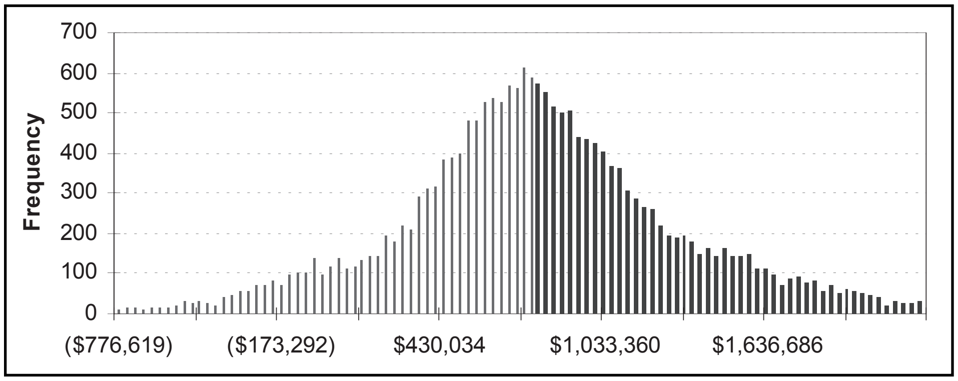


Figure 7: *Forecast Five-Year Net Present Value*

program can also be forecast.

It is also important to collect the data on actual cost avoidances as it arrives. The actuals can be used to build a knowledge base of cost/benefit information in improving future forecasting accuracy.

Future research might focus on ROIA in terms other than financial—such as the impact of compromised data. Which Balanced Scorecard perspective this might fall under, and how to quantify it, might be interesting and valued research.

Other research can include the impacts of risk mitigation. There is a standard deviation to the Monte Carlo simulation distribution curves, and the impact of new initiatives to the overall risk inherent in the IA countermeasures program could be forecast.◆

References

1. Kaplan, Robert S., and David P. Norton. The Balanced Scorecard: Translating Strategy into Action. Boston: Harvard Business School Press, 1996.
2. Government Chief Information Office, New South Wales (NSW) Department of Commerce, Australia. "ROSI Calculator." June 2004 <www.gcio.nsw.gov.au/library/guidelines/resolveuid/1549f99ec1ff7bcb8f7cb6cb8bceef8c>[8].

Notes

1. The views presented herein are solely those of the authors and do not represent the official opinions of the Defense Security Cooperation Agency.
2. This article is an abridgement of "A Model to Quantify the Return on Investment of Information Assurance" published in the *Defense Institute of Security Assistance Management (DISAM) Journal*, July 1, 2007. The authors thank the *DISAM Journal* for kind permission to provide this abridgement for CROSSTALK.
3. The spreadsheet used here, and the associated PowerPoint presentation, is available from the authors. All numbers are notional.
4. For our purposes, we changed the definitions of frequencies of occurrence (see column 2), and eventually modeled the frequencies using a Monte Carlo simulation based on Poisson distribution. The NSW modeled them using the *max freq p.a.* values as expected values deterministically (i.e., as constants in their equations, not varying values in Monte Carlo simulation equations).
5. The five-year lifespan is used here as an arbitrary time frame for illustration purposes. Some DoD IA financial analyses use a six-year time frame. These (and all other assumptions) can easily be modified, as appropriate.
6. Crystal Ball software is a leading spreadsheet-based software suite for predictive modeling, forecasting, Monte Carlo simulation, and optimization. All figures are established utilizing Crystal Ball Predictive Modeling Software.
7. The "aaa," "bbb," and "ccc" values in Table 3 and Table 5 represent general vulnerabilities and general countermeasures, respectively.
8. Model developed by Stephen Wilson. This reference is used with his and the NSW office's permission.

About the Authors

Ron Greenfield is the information assurance manager, Defense Security Cooperation Agency, Office of the Secretary of Defense. He is certified as an information system security officer, information system security professional, information system security manager, and personnel security background investigator.

Defense Security Cooperation Agency
201 12th ST South STE 203
Arlington, VA 22202
Phone: (703) 604-6579
Fax: (703) 602-7836
E-mail: ronald.greenfield@dsca.mil

Charley Tichenor, Ph.D., serves as an information technology operations research analyst for the DoD, Defense Security Cooperation Agency. He has a bachelor's degree in business administration from Ohio State University, an MBA from the Virginia Polytechnic Institute and State University, and a doctorate in business from Berne University.

Defense Security Cooperation Agency
210 12th ST South STE 203
Arlington, VA 22202
Phone: (703) 901-3033
Fax: (703) 602-7836
E-mail: charles.tichenor@dsca.mil

Letter to the Editor

Dear CrossTalk,

I would like to congratulate you on another superb year in 2008. The articles in the August anniversary issue (by Alistair Cockburn, Watts Humphrey, and Gerald Weinberg)—as well as other articles throughout the year—have given readers a diverse view on contemporary software development approaches.

Of course, to be fair, CrossTalk has been necessarily diverse. Since that famed Software Engineering Conference in Garmisch, Germany in 1968 (illuminated by Cockburn in *Good, Old Advice*), the field of software development has expanded and diversified at a rate incomparable to any other discipline.

Comparing software development to traditional engineering disciplines, my observation is that software developers have to be proficient in everything an engineer does—but in a much larger problem and solution space. The software engineer has to be a "super engineer" or—in the more commonly observed option—must be part of a "super team."

A key characteristic of software development (in the absence of "super humans") is the need for those super teams. This is supported by several CrossTalk articles I have read recently (including Cockburn's January 2009 article *"Spending" Efficiency to Go Faster*) that recognize functioning teams as important for success.

Another key characteristic might be the steady stream of innovation and new challenges.

In the early days, hardware resources were enormously expensive and most of the developer's attention was turned to the machine end of the software. It was even doubtful if software would ever be sold separately from hardware. This was the IBM age.

After that, actual software business flourished. This was the Microsoft age.

Nowadays, we see an abundance of hardware and network resources at an extremely low cost, resulting in software given away free to end-users and paid for by advertisement sales. This is the Google age. Most of the attention has now turned to the end-user of the software.

Any search for a sound and common theoretical basis is illusive in a field as diverse and diversifying as software development. The recipes for success, be it individual skills, team cohesion, or processes, must consequently be diverse and continually diversifying as well. Solutions for the problems of the past might turn out to become the problems of the present.

He who does not learn from history is doomed to repeat it; however, he who does only learn from history has no future.

—Gerold Keefer
Managing Director, Avoca, Ltd.
<gkeefer@avocallc.de>

Enforcing Static Program Properties in Safety-Critical Java Software Components

Dr. Kelvin Nilsen
Aonix

Using the Java language for the development of safety-critical code requires even more enforcement of static properties than is enforced by the traditional Java platform. This article examines style guidelines and describes development tools that enforce the guidelines in order to enable cost-effective certification of Java application code to DO-178B Level A and similar safety certification standards. These approaches eliminate the need for garbage collection, support safe and efficient modular composition of independently developed software components, and enable automatic analysis of an application's worst-case memory and CPU-time requirements.

Software deployed in safety-critical systems must achieve the highest standards of quality, must exhibit a high level of determinism, and must rely on minimal run time services to facilitate proof of the run time environment's correctness [1, 2]. Because of the rigorous certification requirements associated with safety-critical software, developers of safety-critical systems generally adopt styles of programming that are easier to certify but may be more difficult to program. For example, the safety-critical Java standard (JSR-302) offers stack memory allocation as an alternative to standard edition Java's garbage-collected heap [3, 4]. Although the algorithms for allocating and deallocating stack memory are very simple, efficient, and deterministic, reliance on stack memory introduces a different kind of problem. In particular, dangling pointers may be introduced if pointers to stack-allocated objects live longer than the objects they refer to. Certification of a safety-critical system needs to prove the absence of dangling pointers in addition to proving that each memory allocation request will be satisfied in a predictable amount of time.

The Real-Time Specification for Java (RTSJ) [5] eliminates dangling pointers to stack-allocated objects by enforcing the rule that no object allocated in an outer-nested stack frame may hold a pointer to any object allocated in an inner-nested stack frame. Enforcement of this rule is performed with a run time check every time an object's field is overwritten. According to this rule, a seemingly harmless statement like:

```
anObject.aField = aValue;
```

will throw an IllegalAssignmentError exception if aValue resides in a scope that is more inner-nested than the scope that holds anObject. Certification of a safety-critical application must prove that no assignments abort with this exception.

In the vernacular of computer science, a *static property* is a property that can be determined by analysis of a software program without (or before) running the program. For safety-critical software, all of the properties that are critical to its safe operation should be static properties. By the time the software is running as part of a safety-critical system, it is too late for a run time check to detect that a critical property has been violated.

There are two common approaches to verification of static properties. The approach most familiar to software engineers is a programming language type system [6, 7]. The second approach, broadly characterized as the use of static analysis, augments the analysis performed by the programming language type system. Table 1 highlights some of the differences between the two.

This article describes the implementation of a type system that enforces properties that are important to the development of safety-critical code using the Java language. The approaches are somewhat unique in that our implementation of the safety-critical Java type system borrows certain techniques that are more traditionally used in static analyzers. These techniques are not usually used in the implementation of the safety-critical Java type system. We make these techniques more efficient and precise by restricting the data-flow analysis to one method at a time. Among the important properties that are analyzed by the safety-critical type system, it can enforce that:

1. A method is written in a restrictive style allowing special development tools to automatically analyze the CPU time and the stack memory required to execute the method.
2. A method is known not to block its execution awaiting some condition that is to be satisfied by some other thread or by an external event.
3. A method does not copy its incoming arguments into state variables that would possibly persist beyond execution of the method itself.
4. The objects referenced from certain incoming arguments to a method are known to reside in a scope that encloses (surrounds) the scopes containing objects referenced from certain other arguments.

All of these properties are important attributes of real-time software systems. Property 1 is important when a programmer wants to know if it is appropriate to invoke a particular method from a context, such as a hardware interrupt handler or a hard real-time task that requires a reliable upper bound on the amount of CPU time and memory required to execute the method. Property 2 must be verified for methods that are invoked while holding certain kinds of priority ceiling locks. When temporary objects are passed as arguments to a method, property 3 establishes an assurance that a dangling pointer will not result as a side effect of the operations performed within the invoked method. Finally, the knowledge represented by property 4 makes it possible for a method to safely establish pointers from certain temporary objects (the ones that are known to have shorter lifetimes) to certain other temporary objects (those known to have longer lifetimes).

Safety-Critical Type Declarations

The safety-critical type system uses the meta-data annotation system introduced with Java 5.0 to associate safety-critical properties with specific software components[1]. Programmers use the type system of standard edition Java to specify, for example, that a particular method's argument is of HighResolutionTime. Using annotations to augment the standard edition type system, safety-critical Java developers use the safety-critical type system's @Scoped annotation to clarify, for exam-

Characteristic	Type System	Static Analyzer
Identification of Static Properties	The programmer inserts declarations to identify intent and the type system verifies that the code is consistent with the declared intent.	The static analyzer infers the intent from context and usage. The static analyzer may infer different intentions for the same code when used in different contexts. The static properties that will be inferred by the static analyzer are not easily recognized by the human review of source code.
Enforcement	By refusing to translate programs that contain type system errors, the compiler enforces the type system.	Programmers may decide not to run the static analyzer or may ignore its recommendations.
Precision	The programming language specification must precisely characterize exactly what constitutes a legal program.	The characterization of what will be understood by a static analyzer is much less precise. One vendor's static analyzer may reach very different conclusions than another's. Static analyzers may produce *false negatives*, stating that certain lines of code may violate desired properties even though an intelligent human analysis would prove that the code does not represent a problem. Static analyzers may produce *false positives*, concluding that a desirable property holds true when really it does not. This typically occurs when humans misconfigure the analysis in an attempt to reduce false negatives.
Efficiency	Because the compiler runs so frequently, type systems generally restrict themselves to properties that are easily and efficiently verified.	Whereas a compiler generally runs in seconds, a static analyzer often requires hours. Rather than focusing attention on each method or class in isolation, the typical static analyzer attempts to discover all of the contexts from which each method might be invoked, and it propagates static information known about each context into the execution of the method within that context.
Expressive Power	To facilitate efficient enforcement, the *vocabulary* for speaking about types is limited.	In theory, static analyzers can distinguish many more subtle nuances than a type system and can treat particular program components as having different properties when invoked from different contexts.

Table 1: *Comparison of the Two Common Approaches to Static Property Verification*

ple, that the argument may have been allocated in stack memory. The following method declaration illustrates this usage:

```
void  setDeadline(@Scoped
HighResolutionTime newDeadline);
```

The remainder of this section describes some of the annotations that are available to safety-critical developers using the safety-critical type system.

Resource Limitations

To indicate that a particular method must be implemented using a subset of the full Java language—that can be automatically analyzed by development tools to determine the worst-case CPU time and stack memory requirements—its declaration is accompanied by a @StaticAnalyzable annotation, as in the following code:

```
@StaticAnalyzable
void handleAsyncEvent() {
    // method body
}
```

The safety-critical Java type system enforces that all overriding methods also be @StaticAnalyzable. Furthermore, the type system enforces that any methods invoked from within a @StaticAnalyzable method are also declared @StaticAnalyzable, and it requires that the programmer provide special assertions to limit iteration counts and recursion depths, and to bound the sizes of any arrays or strings allocated within the method.

Non-Blocking Behavior

A special form of the @StaticAnalyzable annotation allows developers to state a requirement that the implementation of a particular method does not perform any blocking operations. Java annotations have associated attributes, with default values for each attribute. One of the attributes of @StaticAnalyzable is named enforce_non_blocking. Its default value is *true*. To specify that blocking is allowed, developers can override the default value, as in the following method declaration:

```
@StaticAnalyzable(enforce_non_blocking
= {false})
void waitForInput();
```

When declared (as shown), the stack memory usage and the total CPU time consumed by this method are bounded. However, since the method may block waiting for input, analysis of how long the method will execute depends on understanding when the input will become available.

Captive-Scoped Arguments

Certain incoming method arguments may be declared as @CaptiveScoped, meaning that the method promises to hold copies of those argument values only within its local variables or passed to other methods as @CaptiveScoped arguments. @Captive Scoped arguments can never be copied to instance or static fields. Thus, the invoker of a method knows that it can safely reclaim the memory associated with temporary objects, which are passed as @CaptiveScoped arguments as soon as the invoked method returns. The following declaration demonstrates use of the @CaptiveScoped annotation:

```
@CaptiveScopedThis void
reserve(@CaptiveScoped SizeEstimator
sizeIncrement);
```

Nesting Relationships of Stack-Allocated Arguments

In certain situations, the safety-critical Java type system understands that incoming temporary arguments have certain relative lifespan orderings. For example, the @Scoped arguments to an instance method of a reentrant-scope object are known to have a lifetime that is at least as long as the reentrant scope object itself. The safety-critical Java system organizes memory as a hierarchy of scopes. If one object is known to live as long as another, we say the first *encloses* the second. This terminology derives from the hierarchy of scopes within which the two objects reside. Outer-nested scopes enclose inner-nested scopes. The scopes are organized as a stack, so objects residing in outer-nested scopes live longer than objects residing in inner-nested scopes. Consider the following method declaration:

```
// Assume this method is associated with
// a @ReentrantScope class
@ScopedThis put(@Scoped Object
anObject);
```

At the invocation point of this method, the safety-critical Java type system enforces that the value assigned to the incoming anObject argument encloses the value assigned to the implicit this argument. Thus, within the implementation of the put() method, it is safe to assign anObject to a field of this. Since anObject lives at least as long as this, no dangling pointer will result when anObject's memory is reclaimed.

Data-Flow Analysis

Data-flow analysis consists of analyzing the flow of information within a software module. Traditionally, type systems do not perform data-flow analysis. Rather, data-flow analysis is an advanced technique performed by static analyzers that examine the flow of information throughout the entire program, including flow between methods. A unique characteristic of the data-flow analysis performed by the safety-critical type system is that it restricts its attention to the Java byte code one method at a time. This allows it to operate more efficiently, and it establishes a foundation upon which the results of static analysis can be fully deterministic, without false positives or false negatives, and without ambiguity from one vendor's implementation to the next. The following are the steps that comprise the data-flow analysis performed during enforcement of the safety-critical type system. Further detail on data-flow analysis techniques is available in reference [6].

1. The first step is to divide the method's code into independent basic blocks, with directed edges representing the possible control flows from one basic block to the next. A basic block is a sequence of instructions that is executed sequentially, without branches into or out of the code sequence.
2. For each basic block, identify the attribute information that is introduced by execution of that block (the *gen-set*) and the attributes that are superseded by execution of that block (the *kill-set*). In many analyses, the *gen-sets* and the *kill-sets* are described algorithmically rather than with discrete enumerations of elements.
3. Define the *join* functions for attribute information.
   a. For feed-forward attribute analysis, the *join* function is applied everywhere multiple control paths enter a given basic block from predecessor basic blocks.
   b. For feed-backward attribute analysis, the *join* function is applied everywhere multiple control paths leave a given basic block to the successor basic blocks.
4. For feed-forward attribute analysis, identify the initial set of attribute information based on safety-critical Java type annotations associated with the method's declaration.
5. For feed-backward attribute analysis, identify the initial set of attribute information based on safety-critical Java type annotations associated with the method's declaration.

"A unique characteristic of the data-flow analysis performed by the safety-critical type system is that it restricts its attention to the Java byte code one method at a time. This allows it to operate more efficiently ..."

6. Repeat until the inner loop executes without any further changes to previously computed attribute information. For each basic block in the method:
   a. For feed-forward attributes, compute the block's attribute information by joining the attribute information available from all predecessor blocks, removing the attribute information that is superseded by this block's *kill-set*, and adding the attribute information represented by this block's *gen-set*.
   b. For feed-backward attributes, compute the block's attribute information by merging the attribute information available from all successor basic blocks, removing the attribute information that is superseded by this block's *kill-set*, and adding the attribute information represented by this block's *gen-set*.
   c. For convenience in discussing execution of this algorithm, we speak of each basic block's *in-set* and *out-set*. The *in-set* represents the *join* of information flows into this basic block. The *out-set* represents the result of applying this block's *gen-set* and *kill-set* information to the *in-set* information. For feed-forward attribute analysis, the *in-set* information is associated with the start of the block and the *out-set* information is associated with the end of the block. For feed-backward attribute analysis, the *in-set* information is associated with the end of the block and the *out-set* information is associated with the beginning of the block.

Data-flow analysis problems guarantee termination by operating on a finite universe of possible attribute values. Thus, there is a maximum size for each *in-set* and *out-set*. Each iteration of the algorithm either leaves the *in-set* and *out-set* sizes unchanged, or at least one set expands. If the set sizes are unchanged, the algorithm has terminated.

Example Analysis of a Feed-Backward Attribute

Because of limitations built into the standard edition Java annotation system, it is not possible for developers to annotate their local variables. Thus, the safety-critical Java type system infers type information by examining how the variables are used within the method. If a local variable's value is ever assigned to a field variable or passed as an argument to a formal parameter that is declared @Captive Scoped, then the local variable must be treated as a *captive-scoped* variable. This situation is recognized by feed-backward analysis of data-flow, as illustrated by the following example.

Assume the following external method declarations:

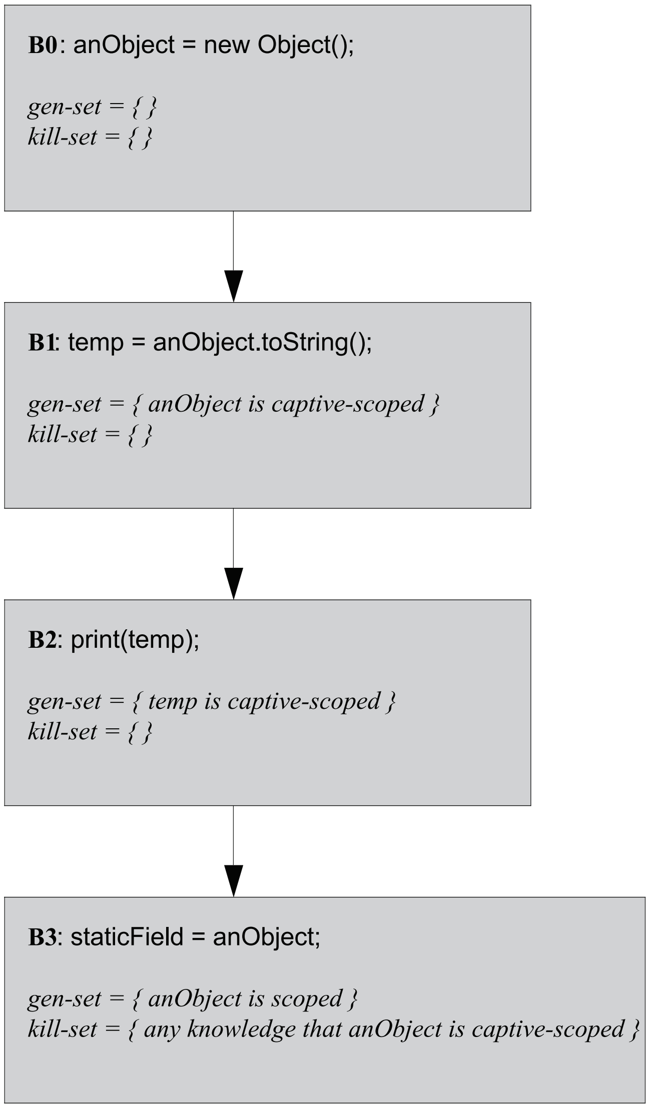
```
// Within class java.lang.Object
@CallerAllocatedResult
@CaptiveScopedThis String toString();

// Within the same class as the following
// method
@Scoped static Object staticField;
static void print(@CaptiveScoped String
arg);
```

Now, consider analysis of the following method:

```
[1] @Scoped static Object staticField;
[2] static void method() {
[3]      Object anObject = new Object();
[4]      print(anObject.toString());
[5]      staticField = anObject;
[6] }
```

This method allocates an Object, invokes the toString() method on this Object, printing the resulting String, and then assigning the value of anObject to the staticField variable. I will describe the analysis that allows the compiler to determine that the String returned from the Object.toString() method invocation on line 4 is allocated in this method's local scope and discarded upon return from the method. The same analysis detects that the Object allocated at line 3 must be allocated within the corresponding Class-Loader scope because the assignment on line 5 makes this Object reachable from the class variable named staticField.

The first step is to divide this method into basic blocks, computing the *kill-set* and *gen-set* for each. The results of this step are represented in Figure 1.



B0: anObject = new Object();

gen-set = { }
kill-set = { }

B1: temp = anObject.toString();

gen-set = { anObject is captive-scoped }
kill-set = { }

B2: print(temp);

gen-set = { temp is captive-scoped }
kill-set = { }

B3: staticField = anObject;

gen-set = { anObject is scoped }
kill-set = { any knowledge that anObject is captive-scoped }

Figure 1: *Basic Blocks Related By Control-Flow Edges*

Based on the information available in block **B1**, it appears that anObject is *captive-scoped*. This is because we invoke the toString() method on anObject, and this method is declared @CaptiveScopedThis. Based on the information available within block **B2**, it appears that the synthesized temp variable is *captive-scoped*. Note that the print() method expects a single *captive-scoped* String argument. In block **B3**, we discover that anObject must be a scoped variable because it is assigned to staticField, which is declared @Scoped. The safety-critical type system treats *captive-scoped* as a specialization of *scoped*. If a given variable is treated in different contexts as both *captive-scoped* and *scoped*, it concludes that the variable must be *scoped*. This is similar to the notion of *widening* in traditional type systems, which allow a single-precision floating point value to be assigned to a double-precision floating point value, but would not allow a double-precision floating point value to be assigned to a single-precision variable without an explicit type coercion. For this reason, the *kill-set* for **B3** removes the *captive-scoped* association for anObject.

For this attribute analysis, the *join* function represents the most conservative classification indicated by all subsequent uses of the variable. This same *join* behavior is applied when propagating usage information through a basic block. If one future usage indicates a variable is *scoped* when another indicates that it is *captive-scoped*, we treat the variable as *scoped* because that is the more conservative treatment. If any future usage indicates that a variable is not *scoped*, then we must treat the variable as *unscoped* even if certain other future usages treat the variable as *scoped* or *captive-scoped*. The *unscoped* attribute is the most conservative. A variable with the *unscoped* attribute is only allowed to reference *immortal* objects, which are allocated in the heap rather than stack memory. The RTSJ identifies such objects as immortal because there is no garbage collection and no command to reclaim the memory for an object previously allocated within the heap.

There is no specific scoping information represented by the annotations associated with this method's return result. Assume that we process the basic blocks in ascending numeric order. Remember that since we are performing a feed-backward analysis, the *in-set* is associated with the end of each block, and the *out-set* is associated with the start. After the first iteration through the basic blocks, we have the following information:

B0:*in-set = { }*
out-set = { }

B1:*in-set = { }*
out-set = { anObject is captive-scoped }

B2:*in-set = { }*
out-set = { temp is captive-scoped }

B3:*in-set = { }*
out-set = { anObject is scoped }

Propagating all of the available data-flow information to all basic blocks requires several additional iterations. After the second iteration, the data-flow associated with each block is the following:

B0:*in-set = { anObject is captive-scoped }*
out-set = { anObject is captive-scoped }

B1:*in-set = { temp is captive-scoped }* *out-set = { anObject is captive-scoped, temp is captive-scoped }*

B2: *in-set = { anObject is scoped }*
out-set = { temp is captive-scoped, anObject is scoped }

B3: *in-set = { }*
out-set = { anObject is scoped }

After the third iteration, we have:

B0: *in-set = { anObject is captive-scoped, temp is captive-scoped }*
out-set = { anObject is captive-scoped, temp is captive-scoped }

B1: *in-set = { anObject is scoped, temp is captive-scoped }*
out-set = { anObject is scoped, temp is captive-scoped }

B2: *in-set = { anObject is scoped }*
out-set = { temp is captive-scoped, anObject is scoped }

B3: *in-set = { }*
out-set = { anObject is scoped }

We need one more iteration to reach a fixed point. After, the fourth iteration, we discover the following:

B0: *in-set = { anObject is scoped, temp is captive-scoped }*
out-set = { anObject is scoped, temp is captive-scoped }

B1: *in-set = { anObject is scoped, temp is captive-scoped }*
out-set = { anObject is scoped, temp is captive-scoped }

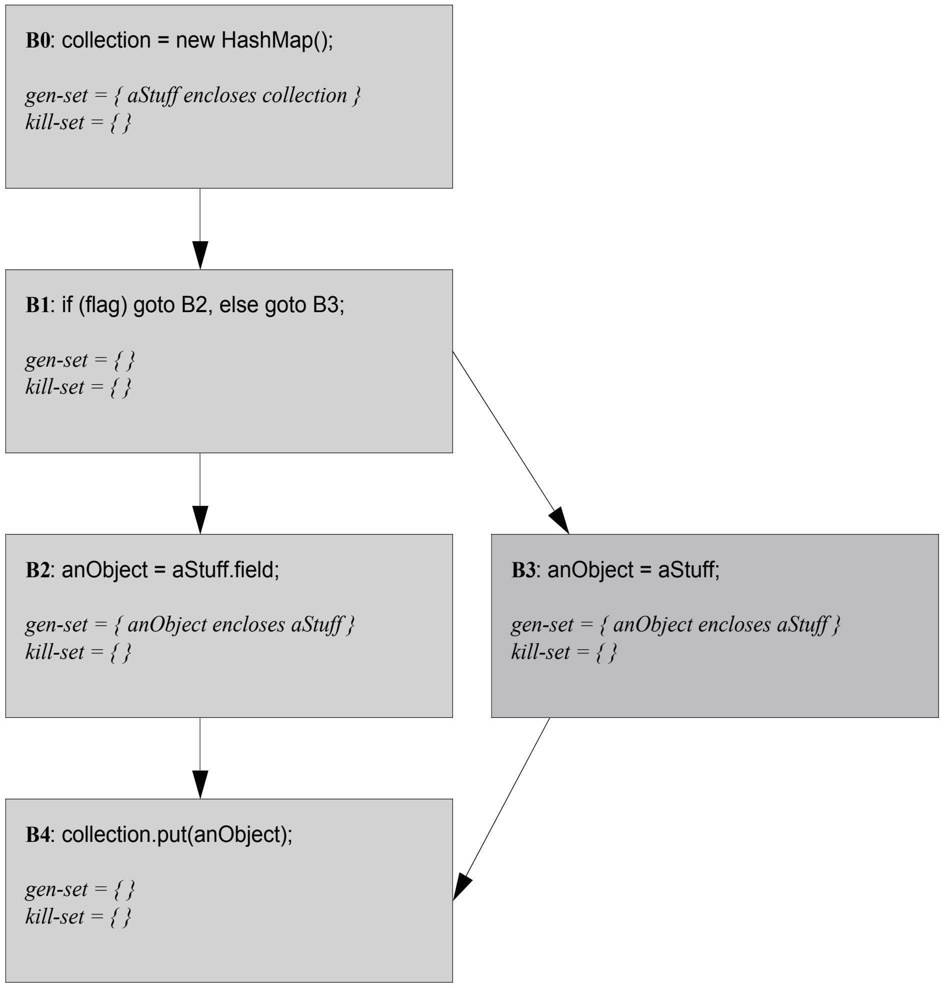
B2: *in-set = { anObject is scoped }*
out-set = { temp is captive-scoped, anObject is scoped }

B3: *in-set = { }*
out-set = { anObject is scoped }

If we were to iterate one more time through the basic blocks, there would be no further changes to our calculations of *in-sets* and *out-sets*. Thus, the data-flow analysis is done.

The safety-critical type system uses this information to determine that the *captive-scoped* temp String created implicitly at line 4 can be allocated in this method's local stack frame memory. Likewise, it determines that the *scoped* Object created at line 3 must be allocated in this class' ClassLoader scope because it must be referenced from one of the class's static variables.

Figure 2: *Basic Blocks Related By Control-Flow Edges*


B0: collection = new HashMap();
gen-set = { aStuff encloses collection }
kill-set = { }
B1: if (flag) goto B2, else goto B3;
gen-set = { }
kill-set = { }
B2: anObject = aStuff.field;
gen-set = { anObject encloses aStuff }
kill-set = { }
B3: anObject = aStuff;
gen-set = { anObject encloses aStuff }
kill-set = { }
B4: collection.put(anObject);
gen-set = { }
kill-set = { }

Example Analysis of a Feed-Forward Attribute

The question of whether one object resides in a scope that is nested external to the scope that holds another object is answered by a feed-forward analysis of data flow. We use the term *encloses* to describe this notion. For purposes of illustration, assume that the feed-backward analysis has already determined that the new HashMap object allocated at line 3 is to be allocated in this method's local stack frame memory. That knowledge becomes an input to the analysis described in the following program fragment:

```
[1] static void buildCollection(Boolean
    Flag, @Scoped Stuff aStuff) {
[2]     Object anObject;
[3]     HashSet collection = new
        HashMap();
[4]     if (flag)
[5]         an Object = aStuff.field;
[6]     else
[7]         an Object = aStuff;
[8]     collection.add(anObject);
[9] }
```

In this code, the new HashSet object created at line 3 is allocated in this method's local stack frame. Depending on the value of the incoming flag argument, we insert into the HashSet either a reference to the object named by the aStuff argument, or the object named by the field variable associated with the aStuff argument. The Stuff declaration (not shown) defines an instance field named field of type @Scoped Object. The HashSet class is declared with the @ReentrantScope annotation, and its add() method declares its single argument to be @Scoped. Given these declarations, the safety-critical Java compiler is required to prove that the argument to the HashSet.add() method resides in a scope that encloses the scope of the HashSet object itself. This is required because the add() method is going to create a reference from the HashSet object to the Object that is inserted into the set. The remainder of this section describes the analysis performed by the compiler to establish this relationship.

The first step is to divide this method

into basic blocks, computing the *kill-set* and *gen-set* for each. The output of this step is represented in Figure 2.

Note that block **B1** generates the knowledge that aStuff encloses collection. This is because any incoming *scoped* arguments necessarily reside in scopes that surround all locally allocated objects. Also note that block **B2** generates the knowledge that anObject encloses aStuff. This is because any field fetched from an object must necessarily reside in a scope that is visible from the object (i.e., that encloses the object). Otherwise, the code that originally assigned the field would have been disallowed. Similarly, block **B3** generates the same knowledge because, by definition, every scope encloses itself.

For this analysis, the *join* function computed for node N is the intersection of all *out-sets* associated with the predecessors of node N. In other words, the only information we know about the enclosure relationships between objects is information known on all incoming paths. If the information is only known upon exit from one of several predecessors to this block, the information may be true—but is not necessarily true upon entry to this particular basic block.

There is no specific object relationship information represented by the annotations associated with this method's incoming arguments; consequently, the *in-set* for block **B0** is empty. Assume that we process the basic blocks in ascending numeric order. After the first iteration through the basic blocks, we have the following information:

```
B0: in-set = { }
    out-set = { aStuff encloses collection }

B1: in-set = { aStuff encloses collection }
    out-set = { aStuff encloses collection }

B2: in-set = { aStuff encloses collection }
    out-set = { aStuff encloses collection,
                anObject encloses aStuff }

B3: in-set = { aStuff encloses collection }
    out-set = { aStuff encloses collection,
                anObject encloses aStuff }

B4: in-set = { aStuff encloses collection,
               anObject encloses aStuff }
    out-set = { aStuff encloses collection,
                anObject encloses aStuff }
```

If we iterate one more time through the basic blocks, there will be no further changes to our calculations of *in-sets* and *out-sets*. Thus, the data-flow analysis is done.

Block **B4** consists of the statement collection.put(anObject). The annotated API description for HashMap.put(), which is not shown, requires for every invocation that the argument to put() enclose the HashMap object that is the target of the put() invocation. The type system applies the transitive property on the relationships available within the *in-set* for block **B4**, thereby validating that the requirement for relative nesting of incoming arguments is satisfied. In other words, the safety-critical type system has proven that the invocation of HashMap.put() at line 8 is a legal invocation.

Conclusion

Standard edition Java provides the infrastructure that is required to augment the type system to speak of static properties that are relevant to safety-critical development. The augmented type system can be implemented by tools that run in combination with standard edition Java development tools by analyzing byte code. The benefits of this approach include leveraging mainstream economies of scale for much of the software and expertise associated with safety-critical development, exploiting the improved programming language features of Java in comparison with legacy languages like Ada, C, and C++, and providing an enhanced type system that focuses on properties of concern to safety-critical developers. All of this translates to reduced costs, improved longevity, and increased functionality for safety-critical software.◆

References

1. RTCA/DO-178B. "Software Considerations in Airborne Systems and Equipment Certification." 1 Dec., 1992.
2. Besnard, J., et al., Eds. Developing Software for Safety Critical Systems. Institute of Electrical and Electronics Engineers, July 1998.
3. Gosling, James, et al. The Java Language Specification. 3rd ed. Prentice Hall PTR, June 2005.
4. JSR-302: Safety-Critical Java Technology. Java Community Process <http://jcp.org/en/jsr/detail?id=302>.
5. Bollella, Gregory, et al. The Real-Time Specification for Java. Addison-Wesley Longman, 2000.
6. Aho, Alfred V., et al. Compilers: Principles, Techniques, and Tools. 2nd ed. Addison Wesley, Oct. 2007.
7. Nilsen, Kelvin. A Type System to Assure Scope Safety Within Safety-Critical Java Modules. Proc. of the 4th Annual Workshop on Java Technologies for Real-Time and Embedded Systems, ACM. Paris, France: 11-13 Oct. 2006.
8. The PERC Pico User Manual. Aonix. 19 Apr. 2008 <http://research.aonix.com/jsc/pico-manual.4-19-08. pdf>.

Note

1. The JSR-302 expert group of the Java Community Process is developing a specification for safety-critical development with the Java language. The team of experts, including the author of this article, has identified a number of static properties that should be assured for any Java software deployed in safety-critical systems, but has chosen not to standardize the mechanisms by which these properties are assured. This article describes the annotation system implemented for this purpose in a commercial product offered by the author's company [7, 8]. Because of space limitations, this article provides only an overview of the complete annotation system.

About the Author

Kelvin Nilsen, Ph.D., is the chief technology officer of Aonix, an international supplier of mission- and safety-critical software solutions. Nilsen oversees the design and implementation of the PERC real-time Java virtual machine along with other Aonix products, including ObjectAda compilers, development environment, libraries, and commercial off-the-shelf safety certification support. Nilsen's seminal research on the topic of real-time Java led to the founding of NewMonics (subsequently purchased by Aonix in 2003), a leader in advanced real-time virtual machine technologies to support real-time execution of Java programs. Nilsen has a bachelor's degree in physics from Brigham Young University as well as master's and doctorate degrees in computer science from the University of Arizona.

Aonix
5930 Cornerstone Court West
STE 250
San Diego, CA 92121
Phone: (801) 756-4821
Fax: (801) 756-4839
E-mail: kelvin@aonix.com

SSTC Systems & Software Technology Conference

"Technology: Advancing Precision"
20-23 April 2009 - Salt Lake City, Utah

Conference Registration Now Open

Register Now!

Visit WWW.SSTC-ONLINE.ORG to access:

- Complete Presentation Schedule
- Presentation Summaries
- Speaker Biographies
- Conference & Exhibit Registration
- Online Lodging Reservations

WHO SHOULD ATTEND:

- Acquisition Professionals
- Program/Project Managers
- Programmers
- Systems Developers
- Systems Engineers
- Process Engineers
- Quality and Test Engineers

www.sstc-online.org

Two, Four, Six, Eight! Software and Systems – Integrate!

Here I am, in yet another airport, writing a BackTalk column. The theme this month? Software and systems integration. And, of course, this reminds me of story.

I was lucky—my parents bought me a brand new 1973 Chevy Impala right out of high school[1]. The 1973 Chevy Impala, although a great car, came with a very basic AM/FM radio.

Keeping in touch with the times, I wanted to be "cool" and install a cassette player (contrary to stories, I was not old enough to have wanted an eight-track player). I saved up my dollars, and eventually went to a local auto parts shop and bought an in-dash AM/FM/cassette radio. It came with a complete book of instructions.

Really, how hard could it be? Well, the first instruction was to *remove the old radio*, and provided a helpful hint that perhaps *dropping the dashboard* was the way to go. So, armed with a set of borrowed-from-my-dad screwdrivers, I got to work. Eventually, stuff started to come off and I was able to see the mounting brackets for the old radio. Carefully, I removed it and isolated the wires.

I remember that there were about eight wires, almost all of them black. A few had some color-coded tags, but some did not. There was a socket in the back of the radio, but it had no labeling, and several of the wires had no obvious way to disconnect—they simply disappeared into a hole in the dashboard.

Surely, I thought, the new radio instructions would explain. As I read the instructions, I noted that the new radio had nine (not eight!) wires, each color-coded. I spent the better part of the day trying to match the old and new wires. I blew several fuses. I never could get one speaker to work. I ended up draining the battery, and my dad did not have a battery charger. I ended up rolling the car back, and using dad's car to jump mine.

Sundown came, and I finally gave up. Luckily, I had marked the old wires well, and I was able to reconnect the old radio, reinstall the dash[2], and return the new radio for a "full and prompt refund."

Years later, I bought a car[3] that came with only an AM/FM radio. I asked the dealer how much it cost to install an AM/FM radio with a six-disc CD player. The answer was less than $300, so I said "go for it!" When I was done with the hour-long "signing your life away" paperwork, I asked when I could bring back the car to change out the radio. The salesman laughed, and said "It's done already!" I was told (and then shown) that the new cars had a standard socket for all of their radios, and that the replacement was as simple as "Use special tool to remove old radio, insert new radio." Total time, five minutes!

Standardization. Agreement on interfaces. Planning the interfaces ahead of time. Testing them to make sure they work. Sounds easy, but it's not. You need a 50,000-foot view to see how the system will integrate. Programmers are too low on the food chain to see it. Most designers only understand a single system. True system architects are hard to find.

You see, integration takes time and planning and needs to be done *before* you start building the low-level modules of a system. Many large systems have a chief engineer, but his or her job is to understand the complex technical issues. You still need a chief architect to see how everything needs to fit together, and then design the system to work.

I have always taught that there are four phases of design: architecture, data, interface, and module. Most developers focus on the module design because it's understandable at a low level. Most software engineering processes and tools help with the data design. And, if you are using any type of CASE (Computer-Assisted Software Engineering), the interfaces are controlled. However, the really *big* picture is the architecture. I do not know of any really good tools that automate the overall systems and software integration. Maybe it's time for a CASSI (Computer-Assisted Software and Systems Integration) tool[4] to help create, design, manage, and help identify problem areas.

Which brings me back to my current wait in the airport. I am trying to get home from Seattle to Albuquerque. My first leg to Salt Lake City is already 90 minutes late. Since my layover is only an hour, I am guaranteed to miss my connection. I called the airline to rebook but was told I can't because their computers still show the flight out of Seattle as "on time." I point out that since I am calling (and am still on the ground in Seattle), there is no way it's "on time." The very nice and apologetic flight agent agrees, but says that until somebody officially lists the flight as "delayed," the computer "thinks" things are OK, and I cannot rebook a new connection for free.

Integration is hard. You *need* to plan for it. You *need* a system architect to help with the big picture, and plan the interfaces. You *need* a system architect to visualize, create, and then control the big picture—if you don't have a high-level integration plan, the probability of the software integrating properly is close to zero. And, like the airline, when problems occur, you *need* to recognize them early—and take remedial action. Don't keep listing your program as "on time" when you know there's a problem. It will only get worse.

—David A. Cook, Ph.D.
Principal Member of the Technical Staff,
The AEgis Technologies Group, Inc.
dcook@aegistg.com

Notes

1. Let me point out that I had a choice: go to college car-less, or get a car and go to college locally. I picked the car. I joined the Air Force within a year, so I obviously made the right choice.
2. Although, truth be told, until I got rid of the car in 1986, the speedometer cable rattled and occasionally the fuse to the dash light blew out. I am sure this was a normal occurrence, and had nothing to do with my reinstallation.
3. OK, it *was* a minivan, but driving a minivan *does not* automatically label you as "middle-aged."
4. And I can find no reference to this on the Internet, so if this becomes popular in the future, I've coined a new term!

CROSSTALK / 517 SMXS/MXDEA
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

PRSRT STD
U.S. POSTAGE PAID
Albuquerque, NM
Permit 737

SSE
Systems and Software Engineering

OUSD, Acquisition & Technology
Systems & Software Engineering

Systems and Software Engineering

Center of Excellence

Our Mission:

- Shape acquisition solutions and promote early technical planning
- Promote the application of sound systems and software engineering, developmental test and evaluation, and related technical disciplines across the Department's acquisition community and programs
- Raise awareness of the importance of effective systems engineering and drive the state-of-the-practice into program planning and execution
- Establish policy, guidance, best practices, education, and training in collaboration with academia, industry, and government communities
- Provide technical insight to program managers and leadership to support decision making

Learn more at: http://www.acq.osd.mil/sse/about.html

Director, Systems and Software Engineering
ODUSD(A&T)SSE
3090 Defense Pentagon
Room 3B938
Washington, DC 20301-3090

Phone: 703-695-7417

Email: ATL-SSE@OSD.MIL

CROSSTALK thanks the above organizations for providing their support.